

Progress Programming Handbook

© 2001 Progress Software Corporation. All rights reserved.

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

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

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

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

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

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

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

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

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

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

.

May 2001



Product Code: 4533
Item Number: 81101;9.1C

Contents

Preface . . . . . **xxxiii**
Purpose xxxiii
Audience xxxiii
Organization of This Manual xxxiii
Volume 1 xxxiii
Volume 2 xxxv
How to Use This Manual xxxvii
WebSpeed Considerations xxxvii
Typographical Conventions xxxviii
Syntax Notation xxxix
Example Procedures xliii
Progress Messages xlvi
Other Useful Documentation xlvii
Getting Started xlvii
Development Tools xlix
Reporting Tools l
4GL li
Database li
DataServers lii
SQL-89/Open Access lii
SQL-92 liii
Deployment liii
WebSpeed liv
Reference lv

1. Introducing the Progress 4GL **1–1**
1.1 What Is a 4GL? 1–2
1.1.1 A Powerful Application Builder 1–2
1.1.2 More Complex Applications Supported 1–3

1.1.3 Less Development Effort Required 1–4
1.1.4 Less Platform Knowledge Required 1–6
1.2 Progress—The Flexible and Complete 4GL 1–6
1.2.1 Flexibility 1–7
1.2.2 Consistency 1–7
1.2.3 Security 1–10
1.2.4 Interoperability 1–10
1.2.5 Connectivity 1–11
1.2.6 Portability 1–11
1.2.7 Localizability 1–12
1.3 Understanding Progress Syntax and Semantics 1–12
1.3.1 Elements of Progress Syntax 1–12
1.3.2 Blocks and Context 1–15
1.3.3 Block Context and Resource Scope 1–16
1.3.4 Compile-time Versus Run-time Code 1–19
1.4 Compile-time Versus Run-time Execution 1–26
1.4.1 Compile-time Execution 1–26
1.4.2 Run-time Execution 1–27

2. Programming Models 2–1
2.1 Procedure- and Event-driven Programming 2–2
2.2 Flow of Execution 2–2
2.2.1 Program Execution in a Procedure-driven Program 2–2
2.2.2 Program Execution in an Event-driven Program 2–5
2.3 Program Structures 2–8
2.3.1 Program Structure in a Procedure-driven Program 2–8
2.3.2 Program Structure in an Event-driven Program 2–9
2.4 When to Use the Procedure-driven and Event-driven Models 2–12

3. Block Properties 3–1
3.1 Blocks and Their Properties 3–2
3.1.1 REPEAT, FOR, and DO Blocks 3–2
3.1.2 Procedure Blocks 3–2
3.1.3 Trigger Blocks 3–3
3.1.4 EDITING Blocks 3–3
3.1.5 Block Properties 3–3
3.1.6 Looping 3–5
3.1.7 Record Reading 3–6
3.1.8 Frame Allocation 3–7
3.1.9 Record Scoping 3–10
3.1.10 Transactions 3–17
3.1.11 Error Processing 3–18
3.2 Procedure Properties 3–19
3.3 External Procedures 3–20

3.3.1 Procedure Filenames 3–20
3.3.2 Standard Execution Options 3–21
3.3.3 Examples 3–21
3.3.4 Procedure Parameters 3–23
3.3.5 Procedure Context and the Procedure Call Stack 3–23
3.3.6 Shared Objects 3–25
3.3.7 Procedure Handles 3–28
3.4 Internal Procedures 3–30
3.4.1 Internal Procedure Definition 3–30
3.4.2 Internal Procedure Execution 3–31
3.4.3 Internal Procedure Names 3–31
3.4.4 Standard Execution Options 3–31
3.4.5 Internal Procedures and Encapsulation 3–32
3.4.6 Shared and Local Context 3–33
3.4.7 Examples 3–35
3.4.8 Internal Procedures and the Call Stack 3–37
3.5 Persistent and Non-persistent Procedures 3–37
3.5.1 Advantages of Persistent Procedures 3–38
3.5.2 Creating Persistent Procedures 3–38
3.5.3 Working with Persistent Procedures 3–39
3.5.4 Deleting Persistent Procedures 3–41
3.5.5 Examples 3–42
3.5.6 Persistent Procedure Context and Shared Objects 3–44
3.5.7 Run-time Parameters 3–45
3.5.8 Transaction Management 3–45
3.5.9 Condition Management 3–46
3.5.10 CURRENT–LANGUAGE Changes 3–46
3.6 User-defined Functions 3–47
3.6.1 Locally Defined User-defined Functions 3–48
3.6.2 Externally Defined User-defined Functions 3–51
3.6.3 Remotely Defined User-defined Functions 3–52
3.7 Procedure Overriding 3–52
3.7.1 Feature Summary 3–53
3.7.2 Super Procedure Prototyping 3–55
3.7.3 Programming Example 3–57

4. Named Events 4–1
4.1 Feature Summary 4–2
4.2 Programming Example 4–3

5. Condition Handling and Messages 5–1
5.1 Progress Conditions 5–2
5.2 Rules About UNDO 5–3
5.2.1 Infinite Loop Protection 5–4

5.3 The ERROR Condition 5–6
5.3.1 The ERROR–STATUS System Handle 5–6
5.3.2 Default Handling 5–7
5.3.3 Overriding the Default Handling 5–10
5.4 The ENDKEY Condition 5–13
5.4.1 Default Handling 5–14
5.4.2 Overriding the Default Handling 5–17
5.5 The STOP Condition 5–18
5.5.1 Default Handling 5–18
5.5.2 Overriding the Default Handling 5–18
5.6 The QUIT Condition 5–19
5.6.1 Default Handling 5–19
5.6.2 Overriding the Default Handling 5–19
5.7 The END–ERROR Key 5–19
5.7.1 Distinguishing ERROR and ENDKEY 5–21
5.7.2 ENABLE/WAIT–FOR Processing 5–23
5.8 How Progress Handles System and Software Failures 5–24
5.9 Progress Messages 5–25
5.10 Application Messages 5–26

6. Handling User Input 6–1
6.1 The Keyboard and the Mouse 6–2
6.1.1 Key Mapping 6–2
6.1.2 Key Monitoring 6–3
6.1.3 Key Translation Functions 6–4
6.2 Key Codes and Key Labels 6–5
6.3 Key Functions 6–19
6.4 Changing the Function of a Key 6–25
6.5 Using Mouse Buttons and Events 6–27
6.5.1 Portable and Physical Buttons 6–27
6.5.2 Specifying Mouse Events 6–28
6.6 Telling Progress How to Continue Processing 6–29
6.7 Monitoring Keystrokes During Data Entry 6–35
6.7.1 Editing Blocks 6–35
6.7.2 User Interface Triggers 6–35
6.7.3 Using Editing Blocks and User Interface Triggers 6–36

7. Alternate I/O Sources 7–1
7.1 Understanding Input and Output 7–2
7.2 Changing the Output Destination 7–2
7.2.1 Output to Printers 7–3
7.2.2 Additional Options for Printing on Windows 7–4
7.2.3 Output to Files or Devices 7–6
7.2.4 Output to the Clipboard 7–6

7.2.5 Resetting Output to the Terminal 7–6
7.2.6 Sending Output to Multiple Destinations 7–7
7.2.7 Stream I/O vs. Screen I/O 7–10
7.2.8 A Printing Solution 7–11
7.3 Changing a Procedure's Input Source 7–14
7.3.1 Input From Files 7–14
7.3.2 Input from Devices, Directories, and the Terminal 7–15
7.3.3 Receiving Input from Multiple Sources. 7–16
7.3.4 Reading Input with Control Characters 7–18
7.4 Defining Additional Input/Output Streams 7–18
7.5 Sharing Streams Among Procedures 7–22
7.6 Summary of Opening and Closing Streams 7–24
7.7 Processes as Input and Output Streams (NT and UNIX only) 7–25
7.8 I/O Redirection for Batch Jobs 7–25
7.9 Reading the Contents of a Directory 7–26
7.10 Performing Code-page Conversions 7–28
7.11 Converting Non-standard Input Files 7–30
7.11.1 Using QUOTER to Format Data. 7–31
7.11.2 Importing and Exporting Data 7–37
7.11.3 Using the PUT Statement 7–39
7.11.4 Using the IMPORT Statement 7–40

8. The Preprocessor 8–1
8.1 &GLOBAL-DEFINE and &SCOPED-DEFINE Directives 8–2
8.1.1 Example Using &GLOBAL-DEFINE. 8–2
8.1.2 Preprocessor Name Scoping 8–3
8.2 &UNDEFINE Directive 8–5
8.3 &IF, &THEN, &ELSEIF, &ELSE, and &ENDIF Directives 8–6
8.4 DEFINED( ) Preprocessor Function 8–10
8.5 The &MESSAGE Directive 8–10
8.6 Referencing Preprocessor Names 8–11
8.7 Expanding Preprocessor Names 8–11
8.8 Using Built-in Preprocessor Names 8–14
8.8.1 Built-in Preprocessor Name References 8–14
8.8.2 Quoting Built-in Names and Saving {&SEQUENCE} Values . 8–16
8.9 Nesting Preprocessor References 8–17
8.10 Arguments to Include Files 8–18
8.11 Sample Application 8–18

9. Database Access 9–1
9.1 Database Connections 9–2
9.1.1 Connection Parameters 9–2
9.1.2 The CONNECT Statement. 9–3
9.1.3 Auto-connect 9–5

9.1.4 Multi-database Connection Considerations 9–5
9.1.5 Run-time Connection Considerations 9–5
9.1.6 Connection Failures and Disruptions 9–6
9.1.7 Progress Database Connection Functions 9–8
9.1.8 Conserving Connections vs. Minimizing Overhead. 9–9
9.2 Disconnecting Databases 9–10
9.3 Logical Database Names 9–11
9.4 Database Aliases 9–12
9.4.1 Creating Aliases in Applications 9–13
9.4.2 Compiling Procedures with Aliases. 9–14
9.4.3 Using Shared Record Buffers with Aliases 9–16
9.5 Data-handling Statements 9–16
9.6 Adding and Deleting Records 9–19
9.7 Defining a Set of Records to Fetch 9–19
9.8 Fetching Records 9–21
9.8.1 Order of Fetched Records. 9–21
9.8.2 Sample Record Fetches 9–22
9.8.3 ROWID and RECID Data Types 9–23
9.8.4 Results Lists 9–26
9.8.5 FIND Repositioning 9–31
9.9 Fetching Field Lists 9–32
9.9.1 Field List Benefits 9–32
9.9.2 Specifying Field Lists in the 4GL. 9–33
9.9.3 Avoiding Implied Field List Entries 9–35
9.9.4 Updating and Deleting with Field Lists 9–35
9.9.5 Cursor Repositioning and Field Lists 9–36
9.9.6 Field List Handling in Degenerate Cases 9–37
9.10 Joining Tables 9–38
9.10.1 Specifying Joins in the 4GL. 9–40
9.10.2 Using Inner Joins 9–43
9.10.3 Using Left Outer Joins. 9–45
9.10.4 Implementing Other Outer Joins 9–46
9.10.5 Mixing Inner and Left Outer Joins 9–47
9.11 The CONTAINS Operator, Word Indexes, and Word-break Tables 9–47
9.11.1 Understanding the CONTAINS Operator, Word Indexes, and Word-break Tables 9–48
9.11.2 Creating and Viewing Word Indexes. 9–48
9.11.3 Using Word-break Tables 9–49
9.11.4 Writing Queries Using the CONTAINS Operator 9–58
9.11.5 Word Indexing External Documents 9–61
9.11.6 Word Indexing Non-Progress Databases 9–62
9.12 Sequences 9–62
9.12.1 Choosing Between Sequences and Control Tables 9–63
9.12.2 Creating and Maintaining Sequences 9–67
9.12.3 Accessing and Incrementing Sequences 9–68

9.13 Database Trigger Considerations 9–72
9.13.1 Storing Triggers in an Operating System Directory 9–72
9.13.2 Storing Triggers in an R-code Library 9–73
9.14 Using the RAW Data Type 9–73
9.15 Multi-database Programming Techniques 9–75
9.15.1 Referencing Tables and Fields 9–75
9.15.2 Positioning Database References in an Application 9–77
9.15.3 Using the LIKE option 9–78
9.16 Creating Schema Cache Files 9–78
9.16.1 Building a Schema Cache File 9–78
9.16.2 Example Schema Cache File Creation 9–80
9.16.3 Using a Schema Cache File 9–82

10. Using the Browse Widget 10–1
10.1 Understanding Browse Widgets 10–2
10.2 Defining a Query for a Browse 10–3
10.2.1 Using Field Lists 10–4
10.2.2 Planning for the Size of the Data Set 10–4
10.3 Defining a Browse Widget 10–5
10.3.1 Defining a Read-only Browse 10–5
10.3.2 Defining an Updatable Browse 10–7
10.3.3 Defining a Single- or Multiple-select Browse 10–9
10.3.4 Using Calculated Fields 10–12
10.3.5 Sizing a Browse and Browse Columns 10–13
10.3.6 Searching Columns (Windows Only) 10–15
10.4 Programming with a Browse Widget 10–20
10.4.1 Browse Events 10–20
10.4.2 Refreshing Browse Rows 10–24
10.4.3 Repositioning Focus 10–24
10.4.4 Updating Browse Rows 10–25
10.4.5 Creating Browse Rows 10–27
10.4.6 Deleting Browse Rows 10–30
10.4.7 Locking Columns 10–31
10.4.8 Moving Columns 10–32
10.4.9 Overlaying Widgets on Browse Cells 10–34
10.4.10 Using Multiple Browse Widgets for Related Tables 10–37
10.4.11 Browse Style Options 10–38
10.5 Resizable Browse Widgets 10–40
10.5.1 Resizing the Browse 10–40
10.5.2 Moving the Browse 10–41
10.5.3 Resizing the Browse-column 10–41
10.5.4 Moving the Browse-column 10–42
10.5.5 Changing the Row Height 10–43
10.5.6 Additional Attributes 10–43

10.5.7 User Manipulation Events 10–44
10.5.8 Code Example. 10–45
10.6 Using Browse Widgets in Character Interfaces 10–46
10.6.1 Character Browse Modes 10–47
10.6.2 Control Keys 10–48
10.6.3 Functional Differences from the Windows Graphical Browse . 10–52

11. Database Triggers 11–1
11.1 Database Events 11–2
11.2 Schema and Session Database Triggers 11–3
11.2.1 Differences Between Schema and Session Triggers 11–3
11.2.2 Trigger Interaction 11–4
11.2.3 Schema Triggers. 11–4
11.2.4 CREATE Schema Trigger Example 11–7
11.2.5 DELETE Schema Trigger Example. 11–7
11.2.6 ASSIGN Schema Trigger Example 11–8
11.2.7 REPLICATION–CREATE Schema Trigger Example 11–8
11.2.8 REPLICATION–DELETE Schema Trigger Example 11–9
11.2.9 REPLICATION–WRITE Schema Trigger Example 11–9
11.3 Implementing Trigger-based Replication 11–10
11.3.1 4GL Support 11–10
11.3.2 Data Dictionary Support 11–11
11.3.3 Trigger-based Replication Procedure 11–11
11.4 Detecting and Capturing Data Collision 11–17
11.4.1 Create a Collision Log. 11–17
11.4.2 Analyze the Log Records 11–18
11.4.3 Replication Process Configuration 11–20
11.4.4 Session Triggers 11–21
11.4.5 FIND Session Trigger Example 11–24
11.4.6 WRITE Session Trigger Example 11–24
11.5 General Considerations 11–25
11.5.1 Metaschema Tables 11–25
11.5.2 User-interaction Code 11–25
11.5.3 FIND NEXT and FIND PREV 11–25
11.5.4 Triggers Execute Other Triggers 11–26
11.5.5 Triggers Can Start Transactions 11–26
11.5.6 Default Error Phrase 11–26
11.5.7 RETURN ERROR 11–26
11.5.8 Where Triggers Execute 11–26
11.5.9 Storing Trigger Procedures in Libraries 11–27
11.5.10 ESQL Access to Database Objects 11–27
11.5.11 Disabling Triggers for Dump/Load 11–27
11.5.12 SQL Considerations 11–27

12. Transactions 12–1
12.1 Introduction 12–2
12.2 Transactions Defined 12–2
12.3 All-or-nothing Processing 12–3
12.4 Understanding Where Transactions Begin and End 12–8
12.5 How Much to Undo 12–15
12.6 Controlling Where Transactions Begin and End 12–18
12.6.1 Making Transactions Larger 12–19
12.6.2 Making Transactions Smaller. 12–21
12.7 Transactions and Triggers 12–23
12.8 Transactions and Subprocedures 12–34
12.9 Transactions with File Input 12–36
12.10 Transactions and Program Variables 12–36
12.10.1 Transactions in Multi-database Applications 12–39
12.10.2 Transactions in Distributed Applications 12–40
12.11 Determining When Transactions Are Active 12–40
12.12 Progress Transaction Mechanics 12–41
12.12.1 Transaction Mechanics 12–41
12.12.2 Subtransaction Mechanics. 12–41
12.13 Efficient Transaction Processing 12–42

13. Locks 13–1
13.1 Applications in a Multi-user Environment 13–2
13.2 Sharing Database Records 13–3
13.2.1 Using Locks to Avoid Record Conflicts 13–4
13.2.2 How Progress Applies Locks 13–6
13.3 Bypassing Progress Lock Protections 13–8
13.4 How Long Does Progress Hold a Lock? 13–10
13.4.1 Locks and Transactions 13–11
13.4.2 Locks and Multiple Buffers 13–15
13.5 Resolving Locking Conflicts 13–16
13.6 Managing Locks to Improve Concurrency 13–20
13.7 Changing the Size of a Transaction 13–22
13.7.1 Making a Transaction Larger 13–23
13.7.2 Making a Transaction Smaller 13–24

14. Providing Data Security 14–1
14.1 Progress User ID and Password 14–2
14.1.1 User ID. 14–2
14.1.2 Password 14–2
14.2 Validating Progress User IDs and Passwords 14–3
14.3 Run-time Security 14–7
14.3.1 Checking for User IDs 14–7
14.3.2 Defining Activities-based Security Checking 14–10

15. Work Tables and Temporary Tables 15–1
15.1 Work Tables 15–2
15.1.1 Characteristics of Work Tables 15–2
15.1.2 Producing Categorized Reports 15–3
15.1.3 Using Work Tables to Collect and Manipulate Data 15–7
15.1.4 Using Work Tables for Complex Sorting 15–9
15.1.5 Using Work Tables for Cross-tab Reports 15–13
15.2 Temporary Tables 15–15
15.2.1 Characteristics of Temporary Tables 15–15
15.2.2 Defining a Temporary Table 15–16
15.2.3 Using Database, Temporary, and Work Tables 15–16
15.2.4 Similarities Between Temporary and Work Tables 15–19
15.2.5 Differences Between Temporary and Work Tables 15–19
15.2.6 Defining Indexes for Temporary Tables 15–22
15.2.7 Tailoring Temporary Table Visibility and Life Span 15–23
15.2.8 Temporary Tables As Parameters 15–25
15.2.9 Temporary Table Example 15–27
15.3 Dynamic Temporary Tables 15–29
15.3.1 Static and Dynamic Temp Tables 15–29
15.3.2 Creating a Dynamic Temp–table 15–29
15.3.3 Dynamic Buffers for Dynamic Temp–tables 15–33
15.3.4 Dynamic Temp–tables as Local and Remote Parameters 15–35
15.3.5 Error Handling and Messages 15–37

16. Widgets and Handles 16–1
16.1 Overview 16–3
16.1.1 Widget Types 16–3
16.2 Static Versus Dynamic Widgets 16–5
16.2.1 Creating Widgets 16–6
16.2.2 Using Widget Handles 16–6
16.2.3 Static Widgets 16–7
16.2.4 Dynamic Widgets 16–12
16.3 Handles 16–15
16.3.1 Widget Handles 16–15
16.3.2 Procedure Handles 16–16
16.3.3 Query, Buffer, and Buffer-field Handles 16–16
16.3.4 Server Handles 16–16
16.3.5 Component Handles 16–16
16.3.6 Transaction Handles 16–16
16.3.7 System Handles 16–17
16.3.8 Handle Management 16–19
16.4 Widget Attributes 16–20
16.4.1 Attribute References 16–21
16.4.2 Referencing Attribute Values 16–21

16.4.3 Setting Attribute Values 16–22
16.4.4 Attributes in Expressions 16–22
16.4.5 Attribute Example 16–23
16.5 Widget Methods 16–23
16.5.1 Method References 16–23
16.5.2 Invoking Methods 16–24
16.5.3 Methods in Expressions 16–24
16.5.4 Method Example 16–25
16.6 Widget Realization 16–26
16.6.1 Becoming Realized 16–26
16.6.2 Becoming Derealized 16–29
16.7 Resizing Widgets Dynamically 16–30
16.8 User-interface Triggers 16–31
16.8.1 User-interface Events 16–31
16.8.2 Trigger Definition 16–34
16.8.3 Trigger Execution 16–36
16.8.4 Trigger Scope 16–36
16.8.5 Reverting Triggers 16–39
16.8.6 Universal Triggers 16–40
16.8.7 Applying Events 16–42

17. Representing Data 17–1
17.1 Using the VIEW–AS Phrase 17–2
17.1.1 Multiple Representations for a Single Value 17–2
17.1.2 A Value's Default Representation 17–2
17.2 Fill-ins 17–3
17.2.1 Format and Size 17–3
17.2.2 Fill-in Attributes 17–5
17.2.3 Fill-in Fonts on Windows in Graphical Interfaces 17–6
17.3 Text 17–6
17.4 Sliders 17–9
17.4.1 Manipulating a Slider 17–10
17.4.2 Range and Size 17–10
17.4.3 Example Procedure 17–12
17.5 Toggle Boxes 17–14
17.6 Radio Sets 17–19
17.6.1 Defining a Radio Set 17–20
17.6.2 Example Procedures 17–21
17.7 Editors 17–24
17.7.1 Defining an Editor 17–24
17.7.2 Using an Editor 17–26
17.7.3 Attributes and Methods 17–27
17.7.4 Adding Functionality to an Editor 17–28
17.8 Selection Lists 17–30

17.8.1 Defining a Selection List 17–31
17.8.2 Scrolling 17–32
17.8.3 Character Mode 17–32
17.8.4 Example Procedure 17–33
17.8.5 Attributes and Methods 17–35
17.9 Combo Boxes 17–39
17.9.1 Defining a Combo Box 17–40
17.9.2 Working with Combo Boxes 17–42
17.9.3 Example Procedure 17–43
17.9.4 Attributes and Methods 17–45
17.10 Applying Formats When Displaying Widgets 17–49
17.10.1 Using the FORMAT Option 17–49
17.10.2 Character Formats 17–51
17.10.3 Integer and Decimal Formats 17–53
17.10.4 Logical Formats 17–57
17.10.5 Date Formats 17–58

18. Buttons, Images, and Rectangles 18–1
18.1 Buttons 18–2
18.1.1 Button Images 18–3
18.1.2 Default Buttons in Dialog Boxes 18–9
18.2 Images 18–9
18.3 Rectangles 18–15

19. Frames 19–1
19.1 What Is a Frame? 19–3
19.2 Why Progress Uses Frames 19–4
19.3 Static and Dynamic Frames 19–4
19.4 Frame Allocation 19–5
19.4.1 How Progress Allocates Frames 19–5
19.4.2 Default Frames 19–7
19.4.3 Specifying Default Frames 19–7
19.4.4 Controlling Frame Allocation 19–9
19.4.5 The Procedure Block 19–10
19.4.6 Block A 19–10
19.4.7 Block B 19–10
19.4.8 Block C 19–10
19.5 Types of Frames 19–11
19.5.1 Example Procedures 19–13
19.6 Frame Scope 19–15
19.6.1 Example Procedures 19–16
19.7 Triggers and Frames 19–17
19.8 Frame Flashing 19–18
19.9 Frame Families 19–19

19.9.1 Defining a Frame Family 19–22
19.9.2 Using Frame Families 19–22
19.10 Frame Services 19–26
19.10.1 Viewing and Hiding Frames 19–27
19.10.2 Advancing and Clearing Frames 19–34
19.10.3 Retaining Data During Retry of a Block 19–34
19.10.4 Creating Field-group Widgets 19–34
19.11 Using Shared Frames 19–35
19.12 Field-group Widgets 19–39
19.12.1 Field-group Types 19–41
19.12.2 Field-group Characteristics 19–42
19.12.3 Field-group Size 19–42
19.12.4 Field-group Position Relative to Frame 19–43
19.12.5 Z order and Field Groups 19–43
19.12.6 Field-group Visibility and Scrolling 19–44
19.12.7 Accessing Field Groups 19–46
19.13 Validation of User Input 19–49
19.13.1 Recommended Validation Techniques 19–49
19.13.2 The Rules of Frame-based Validation 19–50
19.13.3 Forcing Frame-wide Validation 19–50
19.13.4 Disabling Data Dictionary Validation on a Frame or Field 19–54

20. Using Dynamic Widgets 20–1
20.1 Managing Dynamic Widgets 20–2
20.1.1 Static Versus Dynamic Widget Management 20–2
20.1.2 Setting Up Dynamic Field-level Widgets 20–4
20.1.3 Setting Up Dynamic Frames 20–7
20.2 Managing Dynamic Widget Pools 20–9
20.2.1 Named Widget Pools 20–10
20.2.2 Unnamed Widget Pools 20–14
20.3 Creating and Using Dynamic Queries 20–15
20.3.1 Creating a Query Object 20–16
20.3.2 Using a Query Object 20–17
20.3.3 The Buffer Object 20–18
20.3.4 The Buffer-field Object 20–20
20.3.5 Error Handling 20–22
20.3.6 Dynamic Query Code Example 20–23
20.4 Creating and Using a Dynamic Browse 20–25
20.4.1 Creating the Dynamic Browse 20–26
20.4.2 Creating Dynamic Browse Columns 20–27
20.4.3 Dynamic Enhancements to the Static Browse 20–30

21. Windows 21–1
21.1 Multiple-window Pros and Cons 21–2

21.1.1 Windows Versus Dialog Boxes 21–3
21.1.2 Multiple Windows and Transactions 21–3
21.2 Window Attributes and Methods 21–4
21.3 Window Events 21–8
21.4 Creating and Managing Windows 21–8
21.4.1 Creating Window Families 21–9
21.5 Window-based Applications 21–14
21.5.1 Non-persistent Multi-window Management 21–15
21.5.2 Persistent Multi-window Management 21–16

22. Menus 22–1
22.1 Menu Types 22–2
22.1.1 Menu Bar. 22–2
22.1.2 Pop-up Menu. 22–3
22.2 Menu Hierarchy 22–4
22.3 Features for Menu Widgets 22–5
22.4 Defining Menus 22–5
22.4.1 Static Menu Bars. 22–5
22.4.2 Static Pop-up Menus 22–11
22.5 Nested Submenus 22–12
22.6 Enabling and Disabling Menu Items 22–13
22.7 Menu Toggle Boxes 22–15
22.8 Duplicating Menus 22–16
22.9 The MENU–DROP Event 22–18
22.10 Menu Item Accelerators 22–18
22.10.1 Defining Accelerators 22–18
22.10.2 Accelerators in Character Interfaces 22–20
22.11 Menu Mnemonics 22–21
22.12 Dynamic Menus 22–21
22.12.1 Properties of Dynamic Menus. 22–22
22.12.2 Dynamic Menu Bar 22–23
22.12.3 Querying Sibling Attributes 22–26
22.12.4 Example of a Dynamic Menu 22–27
22.12.5 Dynamic Pop-up Menus 22–29

23. Colors and Fonts 23–1
23.1 Making Colors and Fonts Available to an Application 23–2
23.1.1 Progress Default Colors 23–3
23.1.2 Progress Default Fonts 23–4
23.1.3 Application Colors and Fonts 23–4
23.2 Assigning Colors and Fonts to a Widget 23–4
23.3 Assigning Colors and Fonts to ActiveX Automation Objects and ActiveX Controls 23–7
23.4 Color and Font Inheritance 23–7

23.5 Color in Character Interfaces 23–8
23.5.1 Widget States and Color 23–8
23.5.2 Color Specification 23–9
23.6 Colors on Windows in Graphical Interfaces 23–9
23.6.1 Systems with the Palette Manager 23–9
23.6.2 Systems without the Palette Manager 23–10
23.7 Managing Colors and Fonts in Graphical Applications 23–10
23.8 Allowing the User to Change Colors and Fonts 23–11
23.8.1 Establishing Dynamic Colors 23–11
23.8.2 Color Dialog Box 23–12
23.8.3 Color Dialog Example 23–13
23.8.4 Saving a Modified Color 23–15
23.8.5 Font Dialog 23–15
23.8.6 Font Dialog Example 23–16
23.8.7 Saving a Modified Font 23–18
23.9 Accessing the Current Color and Font Tables 23–18
23.9.1 COLOR–TABLE Handle 23–18
23.9.2 FONT–TABLE Handle 23–20
23.10 Retrieving and Changing Color and Font Definitions 23–21
23.10.1 Changing Resource Definitions 23–21
23.10.2 Using GET–KEY–VALUE and PUT–KEY–VALUE 23–21
23.11 Managing Application Environments 23–23
23.11.1 Understanding Environment Management 23–23
23.11.2 Using Environment Management Statements 23–23
23.11.3 Managing Multiple Environments 23–24
23.12 Managing Color Display Limitations 23–26

24. Direct Manipulation 24–1
24.1 Attributes 24–2
24.2 Mouse Buttons on Windows 24–5
24.3 Selecting Widgets 24–6
24.3.1 Selecting by Pointing and Clicking 24–6
24.3.2 Box Selecting 24–7
24.3.3 Sets of Selected Widgets 24–8
24.4 Moving and Resizing Field-level Widgets 24–8
24.5 Moving and Resizing Frames 24–10
24.6 Custom Highlights 24–10
24.7 Interaction Modes 24–11
24.8 Direct-manipulation Events 24–11
24.8.1 General Characteristics 24–11
24.8.2 Order of Direct-manipulation Events 24–15
24.9 Grids 24–17

25. Interface Design 25–1
25.1 Progress Windows 25–2
25.1.1 Character-based Environment 25–2
25.1.2 Graphical Environment 25–4
25.1.3 Alert Boxes 25–7
25.2 Frame Characteristics 25–7
25.2.1 Using Frame Phrase Options 25–7
25.2.2 Setting Frame Attributes 25–9
25.2.3 Rules for Frame Phrases and Attributes 25–11
25.3 Frame Design Issues 25–11
25.3.1 Frame-level Design 25–11
25.3.2 Field- and Variable-level Design 25–13
25.3.3 FORM and DEFINE FRAME Statements 25–13
25.3.4 FRAME–ROW and FRAME–COL Options 25–20
25.3.5 Positioning Frames with FRAME–LINE 25–24
25.3.6 Positioning Frames with FRAME–DOWN 25–25
25.3.7 Scrolling Frames 25–27
25.3.8 Scrollable Frames 25–32
25.3.9 Strip Menus 25–34
25.3.10 Dialog Boxes 25–38
25.3.11 Frames for Nonterminal Devices 25–41
25.3.12 Multiple Active Frames 25–41
25.4 Tab Order 25–43
25.4.1 Tab Order for a Frame 25–43
25.4.2 Tab Order for a Frame Family 25–44
25.4.3 Tabbing and DATA–ENTRY–RETURN 25–48
25.5 Active Window Display 25–49
25.6 Three-dimensional Effects (Windows only; Graphical interfaces only) . . 25–49
25.6.1 THREE–D, Rectangles, and Images 25–51
25.6.2 THREE–D and Window Widgets 25–52
25.6.3 2D and 3D Widget Size Compatibility 25–52
25.7 Windows Interface Design Options 25–52
25.7.1 Incorporating Tooltip Details 25–52
25.7.2 Accessing the Windows Help Engine 25–53
25.7.3 Displaying the Small Icon Size 25–53

A. R-code Features and Functions A–1
A.1 R-code Structure A–2
A.1.1 Factors that Affect R-code Size A–3
A.1.2 R-code File Segment Layout A–4
A.2 R-code Libraries A–5
A.3 R-code Libraries and PROPATH A–6
A.4 R-code Execution A–7
A.4.1 Standard R-code Execution Environment A–7

A.4.2 Memory-mapped R-code Execution Environment A–11
A.4.3 R-code Directory Management A–13
A.4.4 R-code Execution Environment Statistics A–14
A.5 R-code Portability A–17
A.6 Code Page Compatibility A–18
A.7 Database CRCs and Time Stamps A–18
A.7.1 Time Stamp Validation A–19
A.7.2 CRC Validation A–19
A.7.3 CRC Calculation A–20
A.7.4 CRC Versus Time Stamp Validation A–21
A.8 R-code CRCs and Procedure Integrity A–24
A.8.1 Assigning CRCs to Schema Triggers A–24
A.8.2 Validating CRCs for Schema Triggers A–25
A.8.3 RCODE–INFO Handle A–25

Index Index–1

Figures

Figure 1–1: Measures of 4GL Efficiency 1–2
Figure 2–1: A Procedure-driven Application Calling the Operating System 2–3
Figure 2–2: Flow of Execution in a Procedure-driven Program 2–4
Figure 2–3: The Operating System Interacting with an Event-driven Application . . 2–5
Figure 2–4: Flow of Control in an Event-driven Program 2–6
Figure 3–1: Procedure Call Stack 3–24
Figure 3–2: Object Sharing Among External Procedures 3–26
Figure 6–1: Key Labels, Key Codes, and Key Functions 6–2
Figure 6–2: Progress Key Translation Functions 6–4
Figure 6–3: Key Codes 6–5
Figure 6–4: The p-gon1.p Procedure 6–32
Figure 7–1: The Unnamed Streams 7–2
Figure 7–2: Redirecting the Unnamed Output Stream 7–3
Figure 7–3: Multiple Output Destinations 7–7
Figure 7–4: The OUTPUT CLOSE Statement 7–9
Figure 7–5: Redirecting the Unnamed Input Stream 7–15
Figure 7–6: Multiple Input Sources 7–17
Figure 7–7: Multiple Output Streams Scenario 7–19
Figure 7–8: How QUOTER Prepares a File 7–33
Figure 7–9: Extracting QUOTER Input with SUBSTRING 7–35
Figure 8–1: Name Scoping 8–4
Figure 9–1: Data Movement 9–17
Figure 9–2: The Primary Data-handling Statements 9–18
Figure 9–3: Inner Joins 9–39
Figure 9–4: Left Outer Joins 9–42
Figure 9–5: Inner Join Example 9–44
Figure 9–6: Left Outer Join Example 9–46
Figure 9–7: Database Fields and Sequences Comparison 9–63
Figure 9–8: Sequences and Overlapping Transactions 9–66
Figure 9–9: Positioning Database References 9–77
Figure 10–1: Resizable Browse 10–46
Figure 10–2: Character Browse in Row Mode 10–47
Figure 10–3: Character Browse in Edit Mode 10–48
Figure 11–1: Cascading Replication 11–16
Figure 11–2: Example Replication Configuration 11–20
Figure 12–1: Transaction Undo Processing 12–3
Figure 12–2: Transaction Involving Two Tables 12–4
Figure 12–3: The Transaction Block for the p-txn11.p Procedure 12–35
Figure 13–1: Multi-user Update Scenario Without Locks 13–2
Figure 13–2: Multi-user Update Scenario with Locks 13–4
Figure 13–3: How Progress Applies Locks 13–7
Figure 13–4: Locks and Transactions 13–11
Figure 13–5: Transaction Size and Locks 13–22

Figure 13–6: Making Transactions Larger 13–23
Figure 13–7: Making Transactions Smaller 13–25
Figure 14–1: Sample Activity Permissions Entries 14–11
Figure 14–2: User with Permission to Run a Procedure 14–12
Figure 14–3: User Without Permission to Run a Procedure 14–13
Figure 14–4: Sample Security Record for Activity Permissions 14–16
Figure 14–5: Summary of Run-time Security Process 14–18
Figure 15–1: Processing a Table for a Categorized Report 15–4
Figure 15–2: Using Work Tables to Produce a Categorized Report 15–6
Figure 15–3: Maintaining Work Tables in Sorted Order 15–11
Figure 15–4: Comparing Two Sorting Methods 15–12
Figure 16–1: User-interface Display 16–3
Figure 16–2: A Widget Hierarchy 16–4
Figure 19–1: Frame 19–3
Figure 19–2: Frame Allocation 19–6
Figure 19–3: Controlling Frame Allocation 19–9
Figure 19–4: Types of Frames 19–11
Figure 19–5: Frame Scope 19–15
Figure 19–6: Frame Family (Windows Graphical Interface) 19–20
Figure 19–7: Frame Family (Character Interface) 19–21
Figure 19–8: Frame Family Viewing Demo 19–33
Figure 19–9: Shared Frames 19–35
Figure 19–10: Field-Group Allocation 19–41
Figure 21–1: Window Family 21–13
Figure 22–1: Window with a Menu Bar 22–2
Figure 22–2: Button with a Pop-up Menu 22–3
Figure 22–3: Menu Hierarchy 22–4
Figure 23–1: Windows Color Dialog Box 23–12
Figure 23–2: Windows Font Dialog Box 23–15
Figure 23–3: Changing Font 11 to Bold 8-Point Arial 23–17
Figure 24–1: Highlight Boxes 24–6
Figure 24–2: Selection Box 24–7
Figure 24–3: Drag Boxes 24–8
Figure 24–4: Resize Operation 24–9
Figure 24–5: Grid 24–17
Figure 24–6: Grid Enlargement 24–17
Figure 25–1: Character-based Window 25–2
Figure 25–2: Frame Family Tab Order 25–46
Figure A–1: R-code File Segment Layout A–5
Figure A–2: Standard R-code Execution Environment A–8
Figure A–3: Memory-mapped R-code Execution Environment A–12
Figure A–4: Progress Client Startup Options for –yd A–14
Figure A–5: Execution Buffer Map for –yd A–15
Figure A–6: Accessing the Session Sort File for –yd A–15
Figure A–7: Procedure Segment Information for –yd (Part 1) A–16

Figure A–8: Procedure Segment Information for –yd (Part 2) A–17

Tables

Table 3–1: Block Properties 3–3
Table 3–2: Starting Transactions 3–18
Table 3–3: Search Rules 3–53
Table 5–1: Progress Conditions 5–2
Table 6–1: Progress Key Input Precedence 6–3
Table 6–2: Key Codes and Key Labels 6–6
Table 6–3: Alternate Key Labels 6–17
Table 6–4: Progress Key Functions 6–19
Table 6–5: Actions You Assign to Keys 6–26
Table 6–6: Mouse Buttons on Windows 6–27
Table 7–1: Printing Options for Windows 7–12
Table 7–2: Using Streams 7–24
Table 7–3: Quoter Examples 7–32
Table 8–1: Preprocessor Expressions 8–7
Table 8–2: Preprocessor Operators 8–7
Table 8–3: Functions Allowed in Preprocessor Expressions 8–9
Table 8–4: Built-in Preprocessor Name References 8–14
Table 9–1: Connection Failure Behavior 9–6
Table 9–2: Progress Database Functions 9–8
Table 9–3: Record Fetching 9–21
Table 9–4: Results Lists for Specific Query Types 9–27
Table 9–5: Is the Dot a Word Delimiter? 9–50
Table 9–6: Word Delimiter Attributes 9–50
Table 9–7: Comparison of Sequences and Control Tables 9–65
Table 9–8: Sequence Statements and Functions 9–68
Table 10–1: Browse and Query Interaction 10–11
Table 10–2: Row Mode Control Keys 10–49
Table 10–3: Edit Mode Control Keys 10–50
Table 11–1: Replication 4GL Elements 11–10
Table 11–2: Replication Changes Log Table Schema 11–12
Table 11–3: Collision Log Table Schema 11–18
Table 12–1: Starting Transactions and Subtransactions 12–17
Table 13–1: When Progress Releases Record Locks 13–13
Table 14–1: Values to Use for ID Lists 14–9
Table 15–1: Database, Temporary, and Work Tables 15–17
Table 15–2: Temporary Table Options, Visibility, and Life Span 15–24
Table 15–3: Identifying Sender and Receiver Parameters 15–26
Table 16–1: System Handles 16–17
Table 16–2: Realizing Widgets 16–26
Table 16–3: Derealizing Widgets 16–29
Table 16–4: Mouse Events 16–32
Table 16–5: Other Progress Events 16–33
Table 16–6: Trigger Scopes 16–36

Table 17–1:	Default Display Formats	17–51
Table 17–2:	Character Display Format Examples	17–51
Table 17–3:	Numeric Display Format Examples	17–55
Table 17–4:	Logical Display Format Examples	17–57
Table 17–5:	Date Display Format Examples	17–58
Table 18–1:	Using NO–FILL and Color with Rectangles	18–15
Table 19–1:	Determining Frame Type	19–12
Table 20–1:	Static Versus Dynamic Widget Management	20–2
Table 23–1:	Progress Default Colors	23–3
Table 23–2:	Colors Used in Character Interfaces	23–8
Table 24–1:	Direct-manipulation Attributes	24–2
Table 24–2:	Direct-manipulation Mouse Events	24–5
Table 25–1:	Maximized Window Attributes	25–5
Table 25–2:	Frame Border Attributes	25–6
Table A–1:	R-code Segments	A–2
Table A–2:	Metaschema Fields in CRC Calculation	A–20

Examples

p-hello1.p 1–13
p-hello2.p 1–13
p-blocks.p 1–14
p-block1.p 1–15
p-clock1.p 1–18
p-ctrt0.p 1–20
p-ctrt1.p 1–24
p-ctrt2.p 1–25
p-procd.p 2–8
p-eventd.p 2–10
p-bkchp.p 3–5
p-bkchp2.p 3–5
p-bkchp3.p 3–6
p-bkchp4.p 3–6
p-bkchp5.p 3–6
p-bkchp6.p 3–7
p-bkchp7.p 3–8
p-borfrm.p 3–9
p-strscp.p 3–11
p-stscp2.p 3–11
p-wkscp.p 3–12
p-wkscp1.p 3–12
p-wkscp2.p 3–13
p-wkspc3.p 3–13
p-wkstr.p 3–14
p-frref1.p 3–14
p-frref2.p 3–15
p-bkchpa.p 3–17
p-exprc1.p 3–22
p-exprc2.p 3–22
p-share1.p 3–27
p-share2.p 3–27
p-share3.p 3–27
p-intprc.p 3–35
p-inprc1.p 3–36
p-exprc3.p 3–37
p-persp1.p 3–42
p-persp2.p 3–43
p-udf1.p 3–48
p-udf2.p 3–50
p-udfdef.p 3–51
p-udf3.p 3–51
p-pomain.p 3–57

p-podrvr.p 3–58
p-posupr.p 3–59
p–nedrvr.p 4–3
p–nepub.p 4–3
p–nesub1.p 4–4
p–nesub2.p 4–4
p–itundo.p 5–5
p–itund2.p 5–5
p–errst2.p 5–7
p-error.p 5–8
p-txn6.p 5–9
p-error2.p 5–11
p-error3.p 5–12
p–txn7.p 5–13
p-txn8.p 5–14
p–impeof.p 5–16
p–txn9.p 5–17
p–stop.p 5–18
p–error4.p 5–19
p-error4.p 5–22
p–error5.p 5–23
p–retstr.p 5–26
p–retval.p 5–26
p-action.p 6–25
p-intro.p 6–29
p-gon1.p 6–30
p-keys.p 6–31
p-gon2.p 6–34
p-kystka.p 6–36
p-kystkb.p 6–37
p-kystk.p 6–38
p-kystk2.p 6–39
p-kystk3.p 6–40
p-kystk4.p 6–41
p-io.p 7–2
p-iop2.p 7–3
p-wgetls.p 7–4
p-wchpr.p 7–5
p-chgot2.p 7–7
p-out.p 7–9
_osprint.p 7–13
p-datf1.d 7–14
p-chgin.p 7–15
p-io3.p 7–16
p-chgin2.p 7–17

p-dfstr.p 7–20
p-sstrm.p 7–22
p-dispho.p 7–22
p-osdir.p 7–27
p-osfile.p 7–27
p-codpag.p 7–29
p-datf1.d 7–30
p-datf12.d 7–32
p-datf12.q 7–33
p-datf13.d 7–34
p-chgin3.p 7–34
p-datf13.q 7–34
p-datf14.d 7–35
p-chgin4.p 7–36
p-datf14.q 7–36
p-datf15.d 7–36
p-chgin5.p 7–37
p-export.p 7–38
p-datf16.d 7–38
p-exprt2.p 7–38
p-datf17.d 7–39
p-putdat.p 7–39
p-datf18.d 7–39
p-import.p 7–40
p-imprt2.p 7–40
p-datf12.d 7–41
p-impun1.p 7–41
p-datf13.d 7–41
p-impun2.p 7–42
p-datf14.d 7–42
p-impun3.p 7–43
main.p 8–4
include.i 8–4
p-proc.p 8–13
p-incl.i 8–13
p-main.p 8–17
p-inclde.i 8–17
p-editrc.p 8–19
p-editrc.i 8–20
p-cust.f 8–26
parm3.pf 9–4
topproc.p 9–4
subproc.p 9–4
p-infor.p 9–9
mainprc1.p 9–10

subproc1.p 9–10
subproc2.p 9–10
alias1.p 9–13
alias2.p 9–14
dispcust.p 9–14
main2.p 9–16
makebuf.p 9–16
disp.p 9–16
p-fldls1.p 9–34
p-rldls2.p 9–34
p-join1.p 9–44
p-join2.p 9–45
p-mainproc.p 9–77
db1proc.p 9–77
db2proc.p 9–77
p-schcs1.p 9–81
p-schcs2.p 9–82
p-br01.p 10–5
p-br02.p 10–7
p-br03.p 10–10
p-br04.p 10–12
p-br05.p 10–14
p-br06.p 10–17
p-br07.p 10–21
p-br08.p 10–26
p-br09.p 10–28
p-br11.p 10–30
p-br12.p 10–31
p-br13.p 10–32
p-br14.p 10–35
p-br15.p 10–37
p-rszbrw.p 10–45
crcust.p 11–7
p-dltcst.p 11–7
p-ascust.p 11–8
p-sestrg.p 11–24
p-trans.p 12–2
p-txn1.p 12–5
p-txn2.p 12–9
p-txn3.p 12–10
p-check.p 12–13
p-check2.p 12–14
p-txn3a.p 12–15
p-txn5.p 12–15
p-txn3a.p 12–19

p-txn4.p 12–19
p-check.p 12–21
p-txn10.p 12–21
p-check.p 12–22
p-txn12.p 12–23
p-txn13.p 12–25
p-txn14.p 12–29
p-txn11.p 12–34
p-txn11a.p 12–34
p-var.p 12–36
p-tnchp.p 12–38
p-nord2.p 12–40
p-lock1.p 13–5
p-lock2.p 13–5
p-lock3.p 13–7
p-lock6.p 13–8
p-lock7.p 13–9
p-lock4.p 13–11
p-lock12.p 13–15
p-lock4.p 13–16
p-lock5.p 13–16
p-lock6.p 13–19
p-lock13.p 13–21
p-txn3a.p 13–22
p-txn4.p 13–23
p-txn10.p 13–25
_login.p 14–4
p-passts.p 14–6
p-csmnu3.p 14–7
p-adcus2.p 14–8
p-adcust.p 14–8
p-adcus3.p 14–9
p-prmsn.p 14–11
p-adcus4.p 14–14
p-delcs2.p 14–14
p-itlst2.p 14–15
p-rept6.p 14–15
p-chkprm.i 14–16
p-secadm.p 14–17
p-secupd.p 14–17
p-wrk1.p 15–3
p-wrk2.p 15–5
p-wrk3.p 15–8
p-wrk4.p 15–10
p-wrk3.p 15–12

p-wrk4.p 15–12
p-wrk5.p 15–14
p-ttnodb.p 15–21
p-tmptbl.p 15–28
p-ttdyn2.p 15–34
p-passdb.p 15–36
p-getdb.p 15–37
p-dybuts.p 16–14
p-method.p 16–25
p-trig1.p 16–34
p-says.p 16–35
p-pers1.p 16–38
p-rev.p 16–40
p-utrig.p 16–41
p-apply.p 16–42
p-repos3.p 16–43
p-fillin.p 17–4
p-slider.p 17–12
p-tbox.p 17–15
p-tbox2.p 17–18
p-radio1.p 17–21
p-radio2.p 17–22
p-edit.p 17–25
p-edit2.p 17–28
p-sel1.p 17–33
p-sel2.p 17–36
p-combo1.p 17–43
p-combo2.p 17–46
p-ftchp2.p 17–49
p-but1.p 18–2
p-arrows.p 18–5
p-but2.p 18–7
p-ldimg.p 18–12
p-frm1.p 19–3
p-frm2.p 19–6
p-frm3.p 19–7
p-frm4.p 19–7
p-form.p 19–8
p-deffrm.p 19–8
p-frm5.p 19–9
p-frm6.p 19–11
p-frm7.p 19–13
p-frm8.p 19–13
p-frm9.p 19–14
p-frm10.p 19–16

p-frm11.p 19–16
p-frm12.p 19–17
p-frm13.p 19–18
p-frm14.p 19–18
p-frmwin.p 19–19
p-fof1.p 19–22
p-fof2.p 19–24
p-form1.p 19–27
p-form2.p 19–28
p-hide.p 19–28
p-ovrlay.p 19–30
p-dicust.p 19–35
p-updord.p 19–35
p-dicust.i 19–35
p-dicust.i 19–36
p-dicust.p 19–37
p-updord.p 19–38
p-frmval.p 19–53
p-dynfl1.p 20–6
p-dynfrm.p 20–8
p-dybut2.p 20–11
p-dybut3.p 20–12
p-unname.p 20–15
p-qryob1.p 20–19
p-qryob2.p 20–21
p-fnlqry.p 20–23
p-dybrw1.p 20–27
p-dybrw2.p 20–28
p-dybrw3.p 20–29
p-stbrw1.p 20–31
p-wina.p 21–6
p-wow1.p 21–11
p-win1.p 21–15
p-perwn1.p 21–17
p-perwn2.p 21–20
p-perwn3.p 21–26
p-bar.p 22–9
p-popup.p 22–11
p-menu.p 22–14
p-popup2.p 22–17
p-dymenu.p 22–27
p-dypop.p 22–29
p-clrfnt.p 23–5
p-cdial1.p 23–13
p-fdial1.p 23–16

p-dirman.p 24–4
p-manual.p 24–10
p-wbars.p 25–3
p-frm15.p 25–7
p-frm16.p 25–9
p-frm17.p 25–11
p-form4.p 25–15
p-frm19.p 25–17
p-form3.p 25–17
p-back.p 25–19
p-frrow.p 25–21
p-frrow1.p 25–22
p-frrow2.p 25–23
p-frline.p 25–24
p-frdown.p 25–25
p-scroll.p 25–27
p-scrlab.p 25–32
p-strip.p 25–34
p-chsmnu.p 25–36
p-diagbx.p 25–39
p-frm18.p 25–42
p-foftab.p 25–47
p-threed.p 25–50
p-trload.p A–27

Preface

Purpose

This book provides the 4GL programmer with in-depth information about various programming topics for the Progress 4GL. Use it along with the *Progress Language Reference* as a comprehensive guide and reference to programming with Progress. For information on Progress/SQL, see the *Progress Embedded SQL-89 Guide and Reference*.

Audience

This book is intended for Progress developers and anyone who needs to read and understand Progress code. The reader should have the basic knowledge of Progress provided by the *Progress Language Tutorial for Windows* and *Progress Language Tutorial for Character*.

Organization of This Manual

This manual consists of two volumes, Volume 1 and Volume 2. The contents of each volume is as follows:

Volume 1

Chapter 1, "Introducing the Progress 4GL"

Progress is a flexible and complete 4GL. This chapter describes the foundations and essential features of the Progress 4GL, and introduces its basic syntax and semantics.

Chapter 2, "Programming Models"

The Progress 4GL supports two basic programming models: the procedure-driven model and the event-driven model. This chapter describes each model, explains their differences, and suggests how to choose one or a mixture of these models for your application development.

Chapter 3, "Block Properties"

Progress is a block-structured language. This chapter describes the types of blocks Progress supports and the properties of each. This includes a description of the techniques for data and functional encapsulation in the 4GL. This chapter also describes user-defined functions and procedure overriding.

Chapter 4, "Named Events"

Describes how to use Progress named events, and how Progress procedures can create, subscribe to, and unsubscribe to them.

Chapter 5, "Condition Handling and Messages"

Describes how Progress handles unusual conditions such as errors and application termination. This chapter also discusses Progress messages.

Chapter 6, "Handling User Input"

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

Chapter 7, "Alternate I/O Sources"

Describes how your Progress application can handle I/O to and from operating system files and special devices.

Chapter 8, "The Preprocessor"

Describes the features and capabilities of the Progress preprocessor, a tool for conditional 4GL compilation.

Chapter 9, "Database Access"

Describes the Progress 4GL statements that allow you to read and write data in a database. This chapter also describes techniques for browsing database data on a display.

Chapter 10, "Using the Browse Widget"

Describes how to design and interact with browse widgets and the queries they represent.

Chapter 11, “Database Triggers”

Describes how you can provide blocks of code, in the database or in your application, that execute when specific database events occur.

Chapter 12, “Transactions”

Describes statements and techniques that Progress provides to manage updates to a database, including the ability to undo and retry updates in a reliable sequence.

Volume 2

Chapter 13, “Locks”

Describes how Progress uses record locks to manage multi-user access to a database.

Chapter 14, “Providing Data Security”

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

Chapter 15, “Work Tables and Temporary Tables”

Describes two types of memory-resident table structures you can use within a Progress application to temporarily store data.

Chapter 16, “Widgets and Handles”

Surveys the types of user-interface components (widgets) and system objects that Progress supports, and the 4GL techniques for interacting with them, including attributes, methods, and user-interface triggers.

Chapter 17, “Representing Data”

Describes the widgets that you can use to represent data for variables and database fields: fill-ins, text, editors, sliders, toggle boxes, radio sets, selection lists, and combo boxes. This chapter also describes Progress display formats.

Chapter 18, “Buttons, Images, and Rectangles”

Describes how to use push button, image, and rectangle widgets in your applications.

Chapter 19, “Frames”

Describes how Progress manages frame widgets and how you can interact with frames and frame families in your applications.

Chapter 20, “Using Dynamic Widgets”

Describes dynamic (run-time defined) widgets and compares the 4GL techniques for managing static (compile-time defined) and dynamic widgets. This chapter explains how to include dynamic widgets in your application and manage groups of dynamic widgets using widget pools.

Chapter 21, “Windows”

Describes how Progress provides windows to an application and how you can manage windows and window families with the 4GL. This chapter includes basic examples of typical multi-window applications.

Chapter 22, “Menus”

Describes how you can use menu bars, pull-down menus, and pop-up menus in your applications.

Chapter 23, “Colors and Fonts”

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

Chapter 24, “Direct Manipulation”

Describes how to make widgets selectable, movable, and resizable.

Chapter 25, “Interface Design”

Surveys the techniques for laying out and managing widgets within a frame or window.

Appendix A, “R-code Features and Functions”

Describes the structure and management of Progress r-code, the executable code into which Progress compiles 4GL procedures. This appendix also describes techniques for tuning r-code size and performance, and the use of time stamps and cyclic redundancy checks (CRCs) to maintain r-code and database integrity.

How to Use This Manual

The two volumes of this manual treat programming topics in depth. For information on basic Progress 4GL concepts and how to start programming in Progress, see the *Progress Language Tutorial for Windows* and *Progress Language Tutorial for Character* for your user-interface environment. For information on the standard tools for developing Progress applications, see the *Progress Basic Development Tools* manual.

This manual treats programming topics in depth. For information on basic Progress 4GL concepts and how to start programming in Progress, see the *Progress Language Tutorial for Windows* and *Progress Language Tutorial for Character* for your user-interface environment. For information on the standard tools for developing Progress applications, see the *Progress Basic Development Tools* manual.

WebSpeed Considerations

If you are reading this manual to learn more about SpeedScript programming, note that the following chapters and sections do not apply to developing WebSpeed applications:

Volume 1

Chapter 1, "Introducing the Progress 4GL": The "Compile-time Versus Run-time Execution" section does not apply.

Chapter 3, "Block Properties": The "EDITING Blocks" section does not apply.

Chapter 5, "Condition Handling and Messages": Only the "Progress Conditions," "Rules About UNDO," "The ERROR Condition," and "The STOP Condition" sections apply.

Chapter 6, "Handling User Input"

Chapter 7, "Alternate I/O Sources": Only the "Understanding Input and Output," "Changing the Output Destination," ("Output to Files or Devices" and "Stream I/O vs. Screen I/O"), "Changing a Procedure's Input Source" ("Reading Input with Control Characters"), "Defining Additional Input/Output Streams," "Sharing Streams Among Procedures," and "Summary of Opening and Closing Streams" sections apply.

Chapter 10, "Using the Browse Widget"

Volume 2

Chapter 16, "Widgets and Handles": Only the material on system handles, attributes, and methods applies. See *Pocket WebSpeed* for a list of SpeedScript elements.

Chapter 17, "Representing Data"

Chapter 18, "Buttons, Images, and Rectangles"

Chapter 19, "Frames"

Chapter 20, “Using Dynamic Widgets”

Chapter 21, “Windows”

Chapter 22, “Menus””

Chapter 23, “Colors and Fonts”

Chapter 24, “Direct Manipulation”

Chapter 25, “Interface Design”

Typographical Conventions

This manual uses the following typographical conventions:

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

The following typographical conventions are used to represent keystrokes:

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

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

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

  CTRL–X

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

  ESCAPE H
  ESCAPE CURSOR–LEFT

Syntax Notation

The syntax for each component follows a set of conventions:

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

  In this example, ACCUM is a keyword:

  SYNTAX

  ```
  ACCUM aggregate expression
  ```

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

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

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

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

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

  SYNTAX

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

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

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

  SYNTAX

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

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

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

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

  SYNTAX

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

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

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

  SYNTAX

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

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

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

  SYNTAX

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

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

  SYNTAX

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

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

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

  SYNTAX

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

  In this example, you must specify MESSAGE, then at least one of *expression* or SKIP, but any additional number of *expression* or SKIP is allowed:

  SYNTAX

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

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

SYNTAX

```
{ include-file
  [ argument | &argument-name = "argument-value" ] ... }
```

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

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

SYNTAX

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

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

SYNTAX

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

Example Procedures

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

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

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

Accessing Files in Procedure Libraries on Windows Platforms

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

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

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

Extracting Source Files from Procedure Libraries on Windows Platforms

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

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

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

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

```
MKDIR prodoc
```

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

```
MKDIR prodoc\langref
```

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

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

PROLIB extracts all examples into `prodoc\langref`.

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

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

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

Extracting Source Files from Procedure Libraries on UNIX Platforms

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

1 ♦ Run the PROENV utility:

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

Running proenv sets the DLC environment variable to the directory where you installed Progress (by default, /usr/dlc). The proenv utility also adds the DLC environment variable to your PATH environment variable and adds the bin directory (PATH=%DLC%;%DLC%\bin;%PATH%).

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

```
mkdir prodoc
```

3 ♦ Create the proghand directory under prodoc:

```
mkdir prodoc/proghand
```

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

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

PROLIB extracts all examples into prodoc\langref.

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

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

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

Progress Messages

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

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

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

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

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

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

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

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

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

1 ♦ Start the Progress Procedure Editor:

```
install-dir/dlc/bin/pro
```

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

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

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

Other Useful Documentation

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

Getting Started

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

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

Progress Installation and Configuration Guide Version 9 for UNIX

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

Progress Installation and Configuration Guide Version 9 for Windows

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

Progress Version 9 Product Update Bulletin

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

Progress Application Development Environment — Getting Started (Windows only)

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

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

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

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

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

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

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

Progress Startup Command and Parameter Reference

A reference manual that describes the Progress startup commands and parameters in alphabetical order.

Welcome to Progress (Hard copy only)

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

Development Tools

Progress ADM 2 Guide

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

Progress ADM 2 Reference

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

Progress AppBuilder Developer's Guide (Windows only)

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

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

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

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

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

Progress Debugger Guide

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

Progress Help Development Guide (Windows only)

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

Progress Translation Manager Guide (Windows only)

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

Progress Visual Translator Guide (Windows only)

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

Reporting Tools

Progress Report Builder Deployment Guide (Windows only)

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

Progress Report Builder Tutorial (Windows only)

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

Progress Report Builder User's Guide (Windows only)

A guide for generating reports with the Progress Report Builder.

Progress Results Administration and Development Guide (Windows only)

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

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

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

4GL

Building Distributed Applications Using the Progress AppServer

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

Progress External Program Interfaces

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

Progress Internationalization Guide

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

Progress Language Reference

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

Database

Progress Database Design Guide

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

Progress Database Administration Guide and Reference

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

DataServers

Progress DataServer Guides

These guides describe how to use the DataServers to access non-Progress databases. They provide instructions for building the DataServer modules, a discussion of programming considerations, and a tutorial. Each DataServer has its own guide, for example, the *Progress DataServer for ODBC Guide*, the *Progress DataServer for ORACLE Guide*, or the *Progress/400 Product Guide*.

MERANT ODBC Branded Driver Reference

The Enterprise DataServer for ODBC includes MERANT ODBC drivers for all the supported data sources. For configuration information, see the MERANT documentation, which is available as a PDF file in *installation-path*\odbc. To read this file you must have the Adobe Acrobat Reader Version 3.1 or higher installed on your system. If you do not have the Adobe Acrobat Reader, you can download it from the Adobe Web site at: http://www.adobe.com/prodindex/acrobat/readstep.html.

SQL-89/Open Access

Progress Embedded SQL-89 Guide and Reference

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

Progress Open Client Developer's Guide

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

Progress SQL-89 Guide and Reference

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

SQL-92

Progress Embedded SQL-92 Guide and Reference

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

Progress JDBC Driver Guide

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

Progress ODBC Driver Guide

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

Progress SQL-92 Guide and Reference

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

Deployment

Progress Client Deployment Guide

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

Progress Developer's Toolkit

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

Progress Portability Guide

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

WebSpeed

Getting Started with WebSpeed

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

WebSpeed Installation and Configuration Guide

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

WebSpeed Developer's Guide

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

WebSpeed Version 3 Product Update Bulletin

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

Welcome to WebSpeed (Hard copy only)

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

Reference

Pocket Progress (Hard copy only)

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

Pocket WebSpeed (Hard copy only)

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

1

Introducing the Progress 4GL

The Progress Fourth Generation Language (4GL) represents the culmination of several foundation technologies integrated into a single programming environment. These technologies include:

- A block-structured syntax with control structures (including blocks, procedures, and functions) that let you encapsulate flexible and powerful application objects.
- An integrated front-end manager featuring an event-driven architecture that you can use to create and manage flexible user interfaces for both graphical and character environments.
- An integrated back-end manager that includes both a native Progress Relational Database Management System (RDBMS) and a DataServer facility that you can use to access many third-party RDBMSs and flat file systems.
- A compiled intermediate code, executed by a run-time interpreter, that is both efficient and portable across a range of platforms and application environments.

The following sections include an overview of how Progress integrates these technologies and covers the following topics:

- What is a 4GL?
- Progress—the flexible and complete 4GL
- Understanding Progress syntax and semantics
- Compile-time versus run time-execution

1.1 What Is a 4GL?

Before describing the Progress 4GL, it is helpful to understand what a 4GL is. The Progress 4GL is an application development language that allows you to solve a range of application problems with much less effort than earlier language technologies.

1.1.1 A Powerful Application Builder

Figure 1–1 shows the relative power of the four basic generations of computer languages, using three measures of development efficiency.

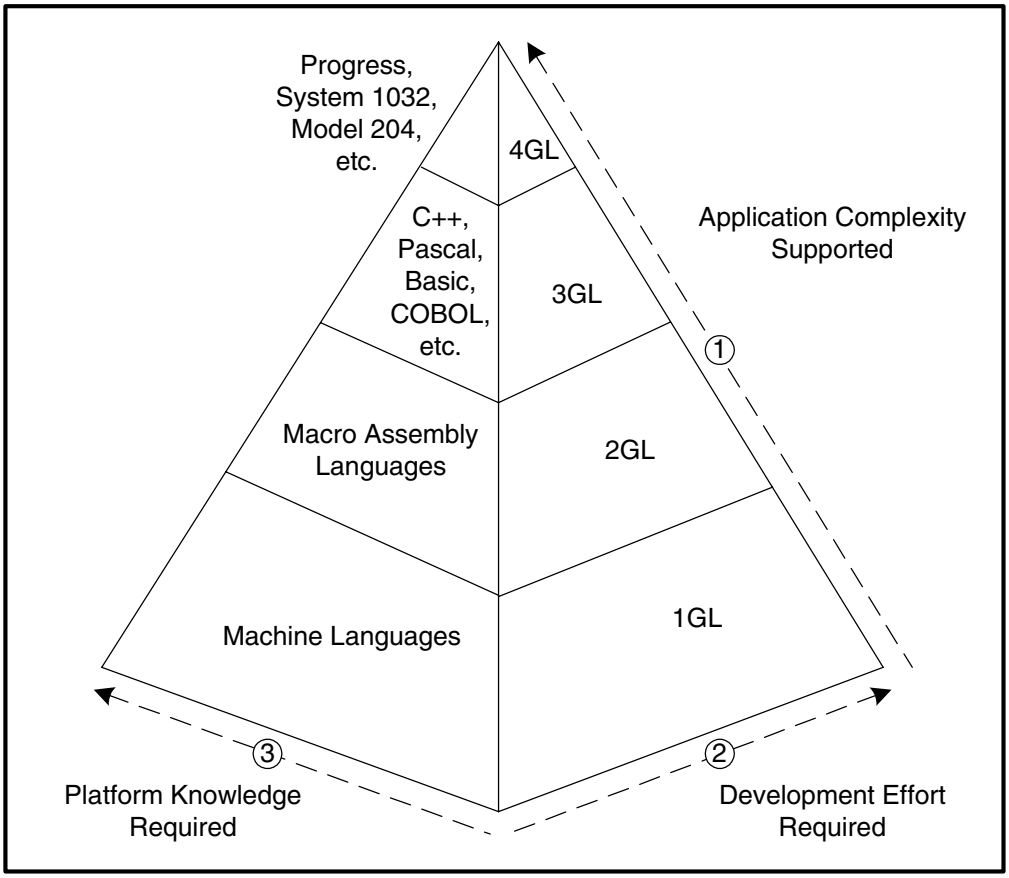


Figure 1–1: Measures of 4GL Efficiency

Each successive generation of language tools:

- Provides greater capacity to handle ever more complex applications
- Requires less development effort per application
- Requires less knowledge and management of the platform where an application is developed and deployed

The more a tool set supports these three indicators, the more effectively a developer can deliver application solutions.

1.1.2 More Complex Applications Supported

A 4GL supports complex applications by providing application-oriented language features. These are features that reflect the application problem rather than the capabilities of a particular machine.

For example, where an earlier generation language might allow you to see data only as a stream of bytes or characters, a 4GL presents data with equal ease as a byte stream or as related collections of objects in a database. Similarly, where a less powerful language provides only basic arithmetic and logical operations, a 4GL also provides application-oriented operations for record reading and writing, database navigation, searching, sorting, time and date conversion, and user interface management, to name just a few.

To support these more complex operations efficiently, reliably, and with a high degree of flexibility, a modern 4GL like Progress provides a run-time manager. This run-time manager maintains a consistent environment for developing and deploying applications, whether single platform or multi-platform, local or distributed.

1.1.3 Less Development Effort Required

A powerful 4GL provides a basic set of operations that do a lot of work with little developer intervention. At the same time, the language allows a fine degree of control over the basic operations it provides. This allows the developer to satisfy a variety of design requirements using the same basic set of tools.

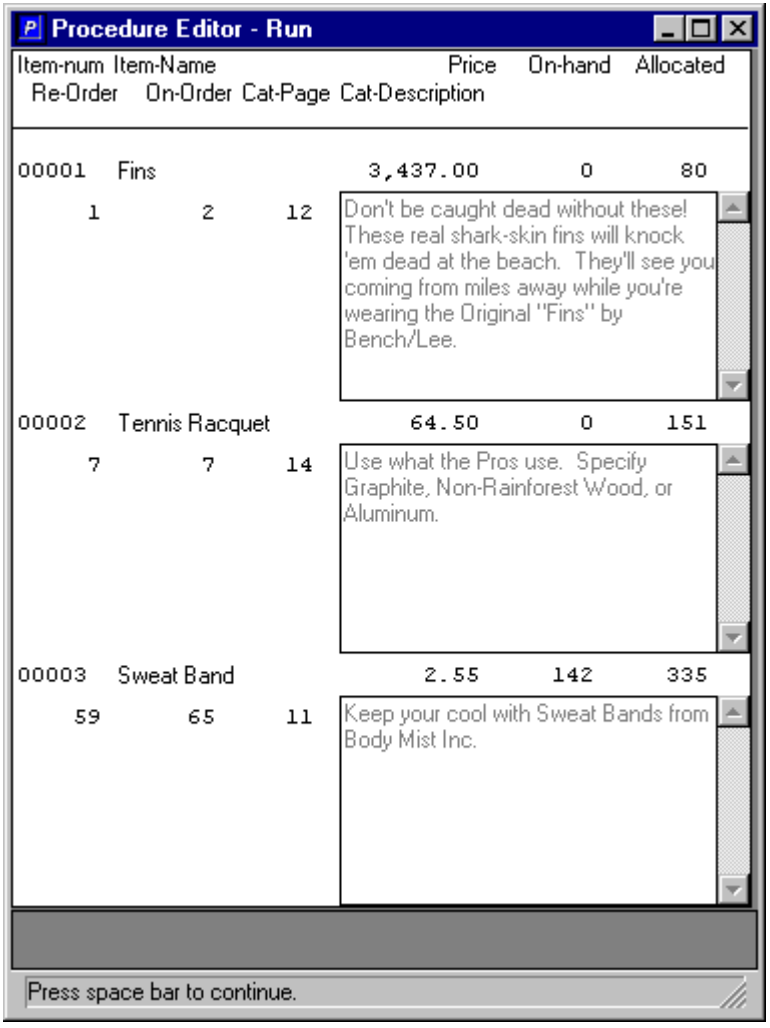
To support an application with a single window, a 3GL like C or COBOL must perform many different input/output operations and keep track of a variety of processing conditions for each operation. In a 4GL like Progress, the same basic set of operations can manage a single application window or many windows, depending on the requirements of the application and the type of user interface environment in which it runs.

For example in Progress, one of the simplest, yet most powerful, tools is the FOR statement. This statement provides iterative record reading and display management capabilities so powerful that you can write an entire application with it.

This is one such classic Progress application:

```
FOR EACH Item: DISPLAY Item. END.
```

This statement reads each record from a database table named Item in the sports database provided with Progress. It displays all the fields of each record in a default window frame complete with column labels. Every time the frame fills up, it automatically prompts the user to display the next frame:



From this simple start, there are dozens of ways to modify this application to select and combine data or change its appearance on the display. Compared with this single FOR statement, to develop this application in a 3GL requires several hundred lines of code.

1.1.4 Less Platform Knowledge Required

The consistent run-time environment of a 4GL like Progress insulates the developer from many platform management concerns. There is little or no need to learn the compilers, linkers, and debuggers of the environment where the application is developed and deployed.

Along with offering the developer greater independence from the platform on which they work, the self-contained run-time environment of a 4GL is often portable across many platforms. Progress is the most flexible and complete of these run-time 4GLs, because it both provides a common language to interact with many platforms and optimizes that language to interact with many of the unique features of each platform.

1.2 Progress—The Flexible and Complete 4GL

The Progress 4GL excels in seven main areas, all supported within the language:

- **Flexibility** — Rapid application development and testing
- **Consistency** — Integration of user interface, database, and logic with a single language
- **Security** — Multi-user access control and data management
- **Interoperability** — Interfaces to and from external and remote programs and systems
- **Connectivity** — Access to databases in local and client/server configurations
- **Portability** — Development and deployment on multiple platforms
- **Localizability** — Development and deployment support for national languages

The following sections describe these major language areas and where to find more information on them. Later sections in this chapter describe the syntax and structure of a 4GL application and how to build one.

1.2.1 Flexibility

Progress is an interpreted language with a run-time compiler to rapidly build and test applications. With the help of the Application Development Environment (ADE), most of which is built using the 4GL, you can compile and run applications on the fly. The compiler generates an efficient run code (*r-code*) that the Progress run-time interpreter can execute immediately, without additional operating system intervention (no machine language generation or linkage required).

You can also compile and execute 4GL routines directly from within your application. Thus, changes to the code and even the scope of an application can often be made in a matter of minutes. See the "Compile-time Versus Run-time Execution" section in this chapter for information on how to compile and execute 4GL applications. See Appendix A, "R-code Features and Functions." for a description of the structure and utility of Progress r-code.

1.2.2 Consistency

The Progress 4GL supports all front-end, back-end, and distributed application logic with a single block-structured language that includes the following features:

- Consistent data model
- Consistent user interface model
- Encapsulation
- Common event handler
- Choice of programming models
- Integrated preprocessor
- Default processing
- Integrated application help

Consistent Data Model

The 4GL presents a consistent model for external (database) and internal (memory) data management. That is, you can work with both database fields and variables using similar data manipulation and user interface techniques. Progress also protects variables and database fields at run time using similar update and data consistency mechanisms (transactions). Progress transactions also support multi-user database access with the help of a flexible lock manager. For information on how Progress manages variable and field values in the user interface, see Chapter 17, "Representing Data." For information on how Progress manages database records and program variables in transactions, see Chapter 12, "Transactions." For information on the 4GL elements you can use to coordinate multi-user database transactions, see Chapter 13, "Locks."

The 4GL supports implicit access to database records and fields using record-reading blocks (including the FOR statement described earlier). You can also read database records and fields explicitly using record-reading statements that access either single or multiple records (queries) at a time. This includes B-tree indexing and support for joins within a single database or between the tables of multiple databases. For information on Progress 4GL mechanisms for accessing databases, see Chapter 9, "Database Access."

Progress also uses the same data model (with appropriate variations) to access non-Progress databases and file systems with DataServers. For more information on the 4GL variations required to access DataServers, see the Progress DataServer guide for each DataServer.

The 4GL also supports memory-resident temporary tables. You define temporary tables much like groups of variables. They are, in fact, variables grouped into memory-resident database records. Thus with temporary tables, Progress allows you to define arrays of records and access them with the same 4GL interface used to access database records. For information on how to use these memory-resident tables, see Chapter 15, "Work Tables and Temporary Tables."

Single-User Interface Model

The 4GL uses a single internal model (with some variations) for both graphical and character user interfaces. This model represents the user interface as related hierarchies of widgets, where a *widget* is any single user interface object that you can define completely in the 4GL.

This single widget model supports several different visualizations, depending on the operating environment. In graphical environments, like Windows, widgets are 4GL-aware controls that include such objects as windows, frames, selection lists, fill-ins, and database browse widgets. In character-environments, the 4GL supports many of the same widgets in character form. See Chapter 10, "Using the Browse Widget," and most of the remaining chapters starting with Chapter 16, "Widgets and Handles," for information on using 4GL widgets.

Encapsulation

Using basic external and internal procedure blocks, Progress provides an encapsulation mechanism that allows you to create 4GL-based objects. That is, you can package data, widgets, and code into a single unit that your application can access through a controlled interface, much like encapsulated object modules in C and some other languages.

You create and manage encapsulated objects at run-time by executing external procedure files persistently. See Chapter 3, "Block Properties," for information on external and internal procedures, persistent procedures, and user-defined functions. See also Chapter 21, "Windows," which shows one way you can use persistent procedures to manage a user interface.

An ADE application-building tool, the AppBuilder, includes an implementation of encapsulated objects, Progress SmartObjects. These include a variety of objects that you can combine to build many types of database applications using the current version of the Progress Application Development Model (ADM). For more information, see the *Progress AppBuilder Developer's Guide*.

Common Event Handler

The 4GL language provides a common event handler for both user interface and database events, which are programmable using triggers. This event handler allows your application to manage all supported graphical and character user interfaces using similar trigger routines. The same mechanism helps to streamline the coding of data integrity rules by trapping various database read and write events. For information on how to use database triggers, see Chapter 11, "Database Triggers." See also Chapter 6, "Handling User Input," and Chapter 16, "Widgets and Handles," for information on how to work with user interface triggers.

Choice Of Programming Models

With block control structures and event triggers, you can choose from a range of program models to implement your applications, from purely procedure-driven to purely event-driven. See Chapter 2, "Programming Models," which compares the two basic procedure-driven and event-driven program models, and how they can be used in the 4GL. See Chapter 3, "Block Properties," for information on the basic block control structures of the 4GL.

Integrated Preprocessor

The Progress 4GL requires no preprocessing steps separate from compilation for any aspect of 4GL application development. However, there is an integrated 4GL preprocessor for conditional compilation to help your application adapt to varying environmental requirements. For information on this facility, see Chapter 8, "The Preprocessor."

Default Processing

A rich set of overridable defaults allows your applications to be as simple or as complex as user requirements demand. These defaults allow you to develop useful applications with as little as one line of code, or as many lines as you need to manage complex user interfaces and multi-user database queries and transactions. This includes default error-handling capabilities available from the statement to the application level. For information on the error-handling facility and its defaults, see Chapter 5, "Condition Handling and Messages." Other defaults are described as they apply to each 4GL feature.

Integrated Application Help

The 4GL provides access to a variety of application help facilities, depending on your application's user interface. For applications running in a graphical user interface (GUI), you can build a complete hypertext on-line help system using the Progress Help compiler. Through the 4GL, you can also provide context-sensitive access to this help system.

For applications running in any user interface (character or GUI), you can provide field-level help. Field-level help automatically prompts the user when they give input focus to a field. You can specify the prompt (or help string) for each database field directly in the Progress Data Dictionary or using 4GL help strings.

For more information on both on-line help and field-level help, see the *Progress Help Development Guide*.

1.2.3 Security

Progress provides mechanisms to maintain application and database security for multi-user applications. You can authenticate user IDs and passwords and check them for activity-based access rights. For information on how to establish and validate Progress user IDs and passwords, see Chapter 14, "Providing Data Security."

1.2.4 Interoperability

Progress offers a rich variety of interfaces to systems and applications outside Progress. The most basic of these is access to alternate I/O sources, such as printers, operating system sequential files, and pipes. For more sophisticated applications, Progress provides 4GL access to a variety of external program interfaces on supported operating systems, including:

- A host language call interface to C language functions
- System clipboards
- UNIX named pipes

- Windows dynamic link libraries
- Windows dynamic data exchange
- ActiveX Automation support
- ActiveX control container support

For information on the basic system access capabilities of the 4GL, see Chapter 7, "Alternate I/O Sources." For more information on the supported external program interfaces, see the *Progress External Program Interfaces* manual.

The 4GL also includes a related, but separate, SQL implementation. This is especially effective when accessed as Progress Embedded SQL-89 (ESQL) from a host language application that seeks to read and write to a Progress database. You can precompile Progress/ESQL-89 queries into Progress r-code for more efficient execution from the host language application. For more information on Progress/SQL-89, see the *Progress SQL-89 Guide and Reference*, and for more information on Progress/ESQL-89, see the *Progress Embedded SQL-89 Guide and Reference*. You can also access Progress from a Microsoft Open Database Connectivity (ODBC) application using the MERANT Progress ODBC Driver. For more information on this driver, see the MERANT documentation. For more information on how this driver works with Progress, see the *Progress SQL-89 Guide and Reference*.

1.2.5 Connectivity

Progress supports a distributed application environment and a client/server database environment. Through the Progress 4GL and utilities, you can establish and manage connections to local and remote databases, and local and distributed application logic. For information on designing, developing, deploying, and running distributed Progress applications, see *Building Distributed Applications Using the Progress AppServer*. For information on connecting to local and remote Progress databases, see Chapter 9, "Database Access." For information on using the Progress 4GL to connect to Progress DataServers (which allow Progress applications to access certain non-Progress databases), see the Progress DataServer guides.

1.2.6 Portability

Progress applications and r-code are portable across operating environments, with certain platform restrictions due to differences in combinations of machines, user interfaces, and DataServers. In general, platform-specific features that you implement in the 4GL usually compile on different development platforms. However, you might not be able to execute the r-code without error on platforms where the features are not supported.

One use of the 4GL preprocessor is to allow you to implement features supported by different platforms in a single application. You can then conditionally compile these features based on the platform where you choose to deploy an instance of the application. However, even where different platforms support the same 4GL features, they might not execute the same generated r-code. That is, you might have to compile your application on some of the platforms where you plan to run it. See Appendix A, "R-code Features and Functions," for information on the r-code portability rules for different machines, user interfaces, and DataServers.

1.2.7 Localizability

One of the unique features of Progress is a high level of international language support. This includes the flexibility to localize a single application for multiple national languages, or even cultural regions. Progress even provides support for native double-byte application development and deployment for East Asian languages, such as Chinese and Japanese. For more information on language support, see the *Progress Internationalization Guide*.

1.3 Understanding Progress Syntax and Semantics

The following sections describe some general features of Progress syntax and semantics. Some of these features are more fully described in later chapters.

1.3.1 Elements of Progress Syntax

Progress is a block-structured, but statement-oriented language. That is, much of the behavior of a Progress application depends on how statements are organized into blocks. However, the basic executable unit of a Progress application is the statement.

Statements

This is the basic syntax of a Progress application:

SYNTAX

```
statement { . | : } [ statement { . | : } ...
```

Thus, a Progress 4GL application consists of one or more statements. Each *statement* consists of a number of words and symbols entered in a statement-specified order and terminated by a period (.) or a colon (:), depending on the type of statement.

This is a one-statement application that displays “Hello, World!” in a message alert box:

p-hello1.p

```
MESSAGE "Hello, World!" VIEW-AS ALERT-BOX.
```

Comments

A Progress application can also contain non-executable comments wherever you can put white space (except in quoted strings). Each comment begins with “/*” and terminates with “*/”, and you can nest comments within other comments:

p-hello2.p

```
/*/* Simple Application */*/

MESSAGE /* Statement Keyword */ "Hello, World!" /* String */
    VIEW-AS ALERT-BOX /* Options */ . /*/* Period Terminates */
                                    -^--------^-    Statement */
```

Blocks

In Progress, a *block* is a sequence of one or more statements, including any nested blocks, that share a single context. A *context* consists of certain resources that a block of statements share. The content of this shared context depends on the type of block and its relationship to other blocks. The sample procedure, `p-blocks.p`, shows a typical layout of blocks in a procedure:

p-blocks.p

```
/* BEGIN EXTERNAL PROCEDURE BLOCK */
DEFINE BUTTON bSum LABEL "Sum Customer Balances".
DEFINE VARIABLE balsum AS DECIMAL.
DEFINE FRAME A bSum.

ON CHOOSE OF bSum IN FRAME A DO:   /* BEGIN Trigger Block */
    RUN SumBalances(OUTPUT balsum).
    MESSAGE "Corporate Receivables Owed:"
        STRING(balsum, "$>,>>>,>>>,>>9.99") VIEW-AS ALERT-BOX.
END.                                /* END Trigger Block */

ENABLE bSum WITH FRAME A.
WAIT-FOR WINDOW-CLOSE OF FRAME A.

PROCEDURE SumBalances:  /* BEGIN Internal Procedure Block */
DEFINE OUTPUT PARAMETER balance-sum AS DECIMAL INITIAL 0.

    FOR EACH customer FIELD (Balance):  /* BEGIN Iterative Block */
        balance-sum = balance-sum + Balance.
END.                                    /* END Iterative Block */

END.                    /* END Internal Procedure Block */
/* END EXTERNAL PROCEDURE BLOCK */
```

In response to choosing a button, this procedure calculates the sum of all customer balances in the sports database and displays it in a message alert box.

The most basic block is the procedure, and the most basic procedure is an external procedure—a file containing one or more statements—because this is the smallest unit that Progress can compile separately. Thus, `p-hello2.p` is an external procedure with one statement, and `p-blocks.p` is an external procedure with several statements. An external procedure block is also the only type of block that requires no special syntax to define it. Progress always defines an external procedure block, by default, when the procedure executes.

You must begin all other types of blocks with appropriate header statements. Header statements, such as the DO, FOR, and PROCEDURE statements shown in `p-blocks.p`, are typically terminated with a colon, although you can use a period. You generally must terminate the block begun with a header statement with an END statement.

1.3.2 Blocks and Context

The context of a block generally lies within the code defined by the beginning and end of the block. For an external procedure, the block beginning and end is the first and last statement in the file. For any other block, it is the block's header and END statement. The context of an external procedure includes all data, widgets, triggers, and internal procedures that it defines. This is the only type of block that can define trigger and internal procedure blocks within its context. Thus, the external procedure context is often called the main block of the procedure. For example, the variable, balsum, is defined in the main block of `p-blocks.p` and can be modified by all statements and blocks defined within `p-blocks.p`, such as the trigger block. However, the internal procedure, SumBalances, defines an output parameter, balance-sum, that can only be modified within the context of its own block.

In general, any data or widgets that you define within the context of a procedure or trigger block, are available only to the statements of that procedure or trigger block. However, any data or widgets that you define within other types of blocks are actually part of the nearest enclosing procedure or trigger context, not the context of the block where they are defined.

For example, this procedure calculates the sum of customer balances using a variable, balance-sum, defined within a FOR block:

p-block1.p

```
FOR EACH Customer FIELDS (Balance):
    DEFINE VARIABLE balance-sum AS DECIMAL INITIAL 0.
    balance-sum = balance-sum + Balance.
END.

MESSAGE "Corporate Receivables Owed:"
        STRING(balance-sum, "$>,>>>,>>>,>>9.99") VIEW-AS ALERT-BOX.
```

However, balance-sum is actually assigned to the outer procedure context, so the MESSAGE statement, outside the FOR block, can also reference it.

Variable definitions are always assigned to the nearest enclosing procedure or trigger context, because Progress maintains the run-time values for variables in a single stack frame for the entire procedure or trigger. (Progress implements trigger blocks in much the same way as procedure blocks.) Thus, when a procedure A calls another procedure B, all of procedure A's variable values are saved for the return from procedure B. However, when Progress executes a DO, FOR, or other type of block, no prior variable values are saved before the block executes. Likewise, there is no separate data context to maintain variables defined within these blocks. Therefore, Progress maintains the variables defined within these blocks as though they were defined directly in the procedure context.

1.3.3 Block Context and Resource Scope

The context of some blocks also helps determine the scope of certain resources. On the other hand, the scope of other resources might have little to do with the context in which you initially define them. *Scope* is really the duration that a resource is available to an application. Scope can very depending on the resource and the conditions of application execution.

In general, the scope of resources created at *compile time* (when Progress compiles your application) is determined at compile time; the scope of resources created at *run time* (when Progress executes your application) is determined at run time. (See the "Compile-time Versus Run-time Execution" section.) The scope of a resource begins when the resource is instantiated (created in your application) and ends when the resource is destroyed (removed from your application).

For example, a FOR statement defines the scope of any database buffer that it implicitly defines for record reading. The scope of such a record buffer is identical to the context of the FOR block, because the buffer is deleted when the FOR block finishes with it. For example, in `p-block1.p`, the scope of the Customer buffer ends when the FOR block completes. Although the MESSAGE statement following the FOR block can access balance-sum, it can no longer access the Balance field for any record read into the Customer buffer by the FOR block.

Unscoped Resources

On the other hand, this procedure fragment does not even compile, precisely because the FOR block defines the scope of FRAME A. It appears that FRAME A is defined and available in the procedure context. However, the DEFINE FRAME statement does not scope the frame that it defines. Instead, the frame is scoped by the context of its first use, in this case the FOR block. As a result, the frame is actually deleted at run time when the FOR block completes. Progress knows this and thus prevents this fragment from compiling:

```
DEFINE FRAME A Name Balance WITH 10 DOWN.

FOR EACH Customer FIELDS (Name Balance) WITH FRAME A:
    DEFINE VARIABLE balance-sum AS DECIMAL INITIAL 0.
    balance-sum = balance-sum + Balance.
    DISPLAY Name Balance.
    DOWN.
END.

FIND FIRST Customer.
DISPLAY Name Balance WITH FRAME A.
```

The solution is to either use a FORM statement to scope the frame to the procedure block, or define a separate frame for the second reference.

Dynamic Resources

In general, dynamic resources are resources that you implicitly or explicitly create and delete at run time. Record buffers that a FOR statement implicitly defines are dynamic buffers: Progress creates them when the FOR block executes and deletes them when its execution completes. Frames scoped to a FOR block are dynamic in the same way. The whole context of a procedure is dynamic: Progress creates its local data and user interface resources when you call the procedure and deletes them when the procedure returns or otherwise goes out of scope.

However, Progress allows you to create and delete many dynamic resources explicitly. These include external procedure contexts (persistent procedures) and most user interface widgets. In general, the scope of a dynamic resource lasts from the time you create it to the time you delete it or when the Progress client session ends, which ever occurs first. When a procedure block completes execution and its context goes out of scope, this does not affect the scope of any dynamic resources that you have created in that context. The completion of the procedure can only affect whether the resources are still accessible from the context that remains.

Progress allows you to define handles to most dynamic resources. These handles are variables that point to the resources you have created. You must ensure that handles to these resources are always available to your application until you delete the resources. If you create a dynamic widget within a procedure and do not delete it before the procedure completes, the widget remains in scope within your Progress session. If you also lose the original handles to these widgets when the procedure ends, you might not be able to access them again. If you cannot access them, you cannot delete them. In effect, they become memory lost to Progress and the system (that is, a memory leak), as in `p-clock1.p`.

p-clock1.p

```
DEFINE VARIABLE show-time AS LOGICAL.
DEFAULT-WINDOW:HIDDEN = TRUE.
REPEAT:
    MESSAGE "Choose:  OK      to see the current time" SKIP
            "         Cancel  to stop clock"
        VIEW-AS ALERT-BOX BUTTONS OK-CANCEL UPDATE show-time.
    IF show-time THEN RUN get-time.
                 ELSE QUIT.
END.
PROCEDURE get-time:
    DEFINE VARIABLE clock-handle AS WIDGET-HANDLE.
    DEFINE VARIABLE current-time AS CHARACTER FORMAT "x(8)".
    DEFINE FRAME clock SKIP(.5) SPACE(1) current-time SKIP(.5)
        WITH THREE-D NO-LABELS TITLE "Current Time" CENTERED.
    current-time:SENSITIVE = TRUE.
    CREATE WINDOW clock-handle ASSIGN
        HEIGHT = FRAME clock:HEIGHT
        WIDTH  = FRAME clock:WIDTH
        TITLE  = "Clock"
        MESSAGE-AREA = FALSE
        STATUS-AREA = FALSE
        ROW = 1
        COLUMN = 1.
    CURRENT-WINDOW = clock-handle.
    current-time = STRING(TIME, "HH:MM:SS").
    DISPLAY current-time WITH FRAME clock.
    PAUSE 5 NO-MESSAGE.
    clock-handle:HIDDEN = TRUE.
END.
```

This example calls a procedure, get-time, to display the current time for five seconds on request. When get-time executes, it creates a new window for the clock and hides that window before returning. However, the handle to the window, clock-handle, is part of the get-time context and is lost when the procedure returns. Thus, each time get-time returns, it makes another window's worth of memory unavailable for use until Progress exits. The correct way for get-time to manage its clock window is to use the DELETE WIDGET statement to explicitly delete the window before returning.

For more information on blocks, context, and scoping, see Chapter 3, "Block Properties." For more information on frame scope, see Chapter 19, "Frames" For more information on managing dynamic widgets, see Chapter 20, "Using Dynamic Widgets".

1.3.4 Compile-time Versus Run-time Code

Like many languages, the Progress 4GL includes two basic types of code:

- Compile-time, sometimes known as nonexecutable code
- Run-time, sometimes known as executable code

However, as an interpretive language, Progress syntax combines compile-time and run-time components in many more ways than a compiled language like C. The flexibility of this syntax helps implement the rich variety of overridable defaults that characterizes the Progress 4GL.

Compile-time Code

Certain statements exist only to generate r-code when Progress compiles them. These are *compile-time* statements. That is, they create static data and user interface resources (widgets) that the run-time statements can reference and modify, but not destroy, during execution.

Run-time Code

Run-time statements use the static resources created by compile-time statements, but can also create, use, and destroy dynamic resources at run time. That is, run-time statements include statements that interact with static resources, dynamic resources, or both. Many run-time statements also include compile-time options. These are options that generate resources at compile time that are later used by the same statements at run time.

Which Code Is Which?

How can you identify the type of code you are using? There are some general features of 4GL syntax that provide the clues. The following procedure that computes factorials, `p-ctrt0.p`, illustrates many of these features.

Compile-time Syntax Elements

Most compile-time code consists of the following syntax elements:

- Compile-time statements, including the FORM statement and all other 4GL statements that begin with the DEFINE keyword:

p-ctrt0.p

```
DEFINE VARIABLE xvar AS INTEGER VIEW-AS FILL-IN.
DEFINE VARIABLE yvar AS INTEGER VIEW-AS SLIDER MAX-VALUE 16 MIN-VALUE 1.
DEFINE VARIABLE zvar AS INTEGER VIEW-AS FILL-IN.
DEFINE VARIABLE fact-compute AS LOGICAL.
DEFINE BUTTON bOK LABEL "OK".
DEFINE BUTTON BCancel LABEL "Cancel" AUTO-GO.

FORM SKIP (.05)
    "Enter upper factorial to compute:" yvar FORMAT ">>" SKIP(.05)
    "Factorial of" xvar FORMAT ">>" "=" zvar SKIP(.05)
    bOK bCancel
WITH FRAME zFrame NO-LABELS.
```

- The non-executable components of block header statements and END statements:

```
ON CHOOSE OF bOK DO:
    ASSIGN  yvar
            zvar = 1.
    DO xvar = 1 TO yvar WITH FRAME zFrame:
            .
            .
            .
    END.
END.
ENABLE yvar bOK bCancel WITH FRAME zFrame.
WAIT-FOR GO OF FRAME zFrame.
&MESSAGE This procedure has {&LINE-NUMBER} lines.
```

- Options and phrases associated with compile-time statements, wherever they appear:

```
ON CHOOSE OF bOK DO:
    ASSIGN  yvar
            zvar = 1.
    DO xvar = 1 TO yvar WITH FRAME zFrame:
        zvar = xvar * zvar.
        DISPLAY xvar zvar FORMAT ">,>>>,>>>,>>>" WITH FRAME zFrame.
        MESSAGE "Compute next factorial" VIEW-AS ALERT-BOX
            QUESTION BUTTONS YES-NO UPDATE fact-compute.
        IF NOT fact-compute THEN LEAVE.
    END.
END.
ENABLE yvar bOK bCancel WITH FRAME zFrame.
WAIT-FOR GO OF FRAME zFrame.
&MESSAGE This procedure has {&LINE-NUMBER} lines.
```

NOTE: The compile-time variable references, xvar, zvar, and yvar, in the ASSIGN, DISPLAY, and ENABLE statements reference the static widgets (FILL-IN or SLIDER) that represent the variable values at run time. Thus, Progress distinguishes variables from their representation. (See also the following "Run-time Syntax Elements" section.)

- Literal expressions (constants):

```
ON CHOOSE OF bOK DO:
    ASSIGN  yvar
            zvar = 1.
    DO xvar = 1 TO yvar WITH FRAME zFrame:
        zvar = xvar * zvar.
        DISPLAY xvar zvar FORMAT ">,>>>,>>>,>>>" WITH FRAME zFrame.
        MESSAGE "Compute next factorial" VIEW-AS ALERT-BOX
            QUESTION BUTTONS YES-NO UPDATE fact-compute.
        IF NOT fact-compute THEN LEAVE.
    END.
END.
```

- Preprocessor directives and definitions:

```
ENABLE yvar bOK bCancel WITH FRAME zFrame.
WAIT-FOR GO OF FRAME zFrame.
&MESSAGE This procedure has {&LINE-NUMBER} source lines.
```

Run-time Syntax Elements

Most run-time code consists of the following syntax elements:

- All statements other than compile-time statements:

```
ON CHOOSE OF bOK DO:
    ASSIGN  yvar
            zvar = 1.
    DO xvar = 1 TO yvar WITH FRAME zFrame:
        zvar = xvar * zvar.
        DISPLAY xvar zvar FORMAT ">,>>>,>>>,>>>" WITH FRAME zFrame.
        MESSAGE "Compute next factorial" VIEW-AS ALERT-BOX
            QUESTION BUTTONS YES-NO UPDATE fact-compute.
        IF NOT fact-compute THEN LEAVE.
    END.
END.
ENABLE yvar bOK bCancel WITH FRAME zFrame.
WAIT-FOR GO OF FRAME zFrame.
&MESSAGE This procedure has {&LINE-NUMBER} lines.
```

- The options and phrases associated with run-time statements, including the executable components of block header statements, **except** those options and phrases that are also associated with compile-time statements (such as Format and Frame phrases):

```
ON CHOOSE OF bOK DO:
    ASSIGN  yvar
            zvar = 1.
    DO xvar = 1 TO yvar WITH FRAME zFrame:
        zvar = xvar * zvar.
        DISPLAY xvar zvar FORMAT ">,>>>,>>>,>>>" WITH FRAME zFrame.
        MESSAGE "Compute next factorial" VIEW-AS ALERT-BOX
            QUESTION BUTTONS YES-NO UPDATE fact-compute.
        IF NOT fact-compute THEN LEAVE.
    END.
END.
ENABLE yvar bOK bCancel WITH FRAME zFrame.
WAIT-FOR GO OF FRAME zFrame.
&MESSAGE This procedure has {&LINE-NUMBER} lines.
```

The block header statements of iterative blocks (DO, FOR, and REPEAT blocks) all have executable components. These executable components monitor and enforce the iteration conditions (when the block starts and stops executing) and select any database data that are available to the block. For more information, see Chapter 3, “Block Properties.”

- Assignment statements and non-literal expressions (variables, functions, attributes, and methods):

```
ON CHOOSE OF bOK DO:
    ASSIGN  yvar
            zvar = 1.
    DO xvar = 1 TO yvar WITH FRAME zFrame:
        zvar = xvar * zvar.
        DISPLAY xvar zvar FORMAT ">,>>>,>>>,>>>" WITH FRAME zFrame.
        MESSAGE "Compute next factorial" VIEW-AS ALERT-BOX
            QUESTION BUTTONS YES-NO UPDATE fact-compute.
        IF NOT fact-compute THEN LEAVE.
    END.
END.
ENABLE yvar bOK bCancel WITH FRAME zFrame.
WAIT-FOR GO OF FRAME zFrame.
&MESSAGE This procedure has {&LINE-NUMBER} lines.
```

NOTE: Progress distinguishes variables from their data representation. However, the references to yvar, xvar, and zvar in the ASSIGN and DISPLAY statements reference both the variables and their static widget representations because Progress moves data between them implicitly at run time. Progress makes this distinction explicit for variables and dynamic widgets because you must move data between them explicitly at run time. For more information on the relationship between widgets and data, see Chapter 16, "Widgets and Handles"

Progress also allows run-time expressions to qualify some compile-time options. See the example in the "How Compile-time Code and Run-time Code Interact" section.

How Compile-time Code and Run-time Code Interact

Because the Progress 4GL is a run-time interpreted language, it can combine compile-time and run-time code in a number of interesting and powerful ways.

For example, as noted earlier, some run-time statements can also include compile-time options. Thus, you can define a static frame in which to display data using a FORM statement, then add options to that static frame definition using Frame phrase options in subsequent run-time statements, such as FOR and DISPLAY.

In example procedure p-ctrt1.p, the data fields, frame type, and title for frame alpha are all defined at compile time and in three different statements:

p-ctrt1.p

```
DEFINE VARIABLE iter AS INTEGER.
DEFINE FRAME alpha Customer.Name Customer.Phone.

iter = 10.
MESSAGE "iter =" iter VIEW-AS ALERT-BOX.
FOR EACH Customer WITH FRAME alpha iter DOWN:
    iter = 5.
    DISPLAY name phone balance WITH TITLE "Customer Balances".
END.
MESSAGE "iter =" iter VIEW-AS ALERT-BOX.
```

This procedure also illustrates a compile-time option (DOWN) qualified by a run-time expression (iter). The DOWN option specifies the type of frame for frame alpha at compile time. However, the number of iterations (rows of data displayed) in the down frame are determined at run time. This is possible, because certain design-oriented widget options (attributes) can actually be modified up to the point that the widget is *realized* (ready for viewing by the window system). Progress realizes frame alpha, a static widget, at the point of its first use, in the DISPLAY statement.

Another interesting thing about this procedure is that the iter value that qualifies the DOWN option is set **after** the DOWN option appears in the code:

```
FOR EACH Customer WITH FRAME alpha iter DOWN:
    iter = 5.
           .
           .
           .
```

This is possible because Progress knows where the run-time iteration value for the DOWN option comes from at run time, and uses the last value set before the frame is realized in the DISPLAY statement.

NOTE: If the value for a run-time expression is unknown (?), Progress uses whatever default applies to the qualified compile-time option. So, in the example, if iter is not assigned a value before its use with the DOWN option, Progress uses the DOWN option default. In this case, the alpha frame displays as many rows as the frame can fit in its window.

A powerful example of the interaction between compile-time and run-time code is the use of the VALUE option in a number of run-time statements. In example procedure p-ctrt2.p, the VALUE option allows you to use a run-time expression (proc-name[proci]) to provide a compile-time object name:

p-ctrt2.p

```
DEFINE VARIABLE proci AS INTEGER.
DEFINE VARIABLE whand AS WIDGET-HANDLE.
DEFINE VARIABLE proc-name AS CHARACTER EXTENT 3
    INIT ["p-join2.p", "p-foftab.p", "p-wow1.p"].

CREATE WINDOW whand. CURRENT-WINDOW = whand.
MESSAGE "These are STATIC procedure executions." VIEW-AS ALERT-BOX.
RUN p-join2.p.
DELETE WIDGET whand.

CREATE WINDOW whand. CURRENT-WINDOW = whand.
RUN p-foftab.p.
DELETE WIDGET whand.

CREATE WINDOW whand. CURRENT-WINDOW = whand.
RUN p-wow1.p.
DELETE WIDGET whand.

DO proci = 1 TO 3:
    CREATE WINDOW whand. CURRENT-WINDOW = whand.
    IF proci = 1 THEN
        MESSAGE "These are DYNAMIC procedure executions."
            VIEW-AS ALERT-BOX.
    RUN VALUE(proc-name[proci]).
    DELETE WIDGET whand.
END.
```

In the RUN statement, the object name is the name of a procedure to execute. Procedure p-ctrt2.p thus shows how the same three procedures can be executed using static compile-time object names or using object names evaluated by the VALUE option at run time.

NOTE: The three procedures executed by p-ctrt2.p, including p-join2.p, p-foftab.p, and p-wow1.p appear in later chapters of this manual and are also available on-line. For more information on accessing the on-line versions, see the preface of this manual.

Further examples of compile-time/run-time interactions appear throughout this manual, especially in the chapters on blocks, widgets, and interface design.

1.4 Compile-time Versus Run-time Execution

In Progress you execute a procedure using the RUN statement. However, you can use two basic techniques to compile and execute a procedure:

- **Compile-time execution** — You can compile the procedure file on the fly when you execute it using the RUN statement. In this case, Progress locates the procedure file, compiles it to a temporary r-code file, then executes the temporary r-code file.
- **Run-time execution** — You can generate an r-code file from the procedure file using the COMPILE statement or the Application Compiler in the ADE. Then, when you execute the procedure with the RUN statement, Progress locates and executes the r-code file directly.

Both techniques use the PROPATH environment variable settings. That is, both the COMPILE and RUN statements locate procedures using PROPATH. (For more information on the PROPATH environment variable, see the *Progress Client Deployment Guide*.) How the RUN statement executes a procedure (compile-time or run-time execution) depends on how you set up your environment and when you choose to compile the procedure. Each technique provides different capabilities and advantages.

1.4.1 Compile-time Execution

The RUN statement performs compile-time execution when Progress cannot find an existing r-code file that corresponds to the specified procedure. Because r-code is interpreted, there is no need to link an additional machine executable; the RUN statement can thus execute the procedure immediately after compiling it.

Compile-time execution has two main uses:

- Rapid prototyping and testing
- Dynamic 4GL Code Generation

Rapid Prototyping and Testing

Rapid prototyping allows you to quickly complete a code and test cycle for a procedure or a whole application without any concern for compile-time or run-time optimizations. After you have filled in your application architecture, you can then fine tune it using the Application Compiler and other Progress tools and utilities.

Dynamic 4GL Code Generation

Dynamic code generation is an advanced technique that you can see at work in the Progress AppBuilder. From within any application that includes the Progress compiler, you can write 4GL code fragments to a file, execute the file as a procedure, and remove the file, all under application control.

1.4.2 Run-time Execution

Compile-time execution can save time in development. However, to improve application performance, the first step is to ensure that it is running in r-code form.

Thus, run-time execution has two main uses:

- Application fine tuning and deployment
- Optimized dynamic code generation

Application Fine Tuning and Deployment

This is the technique most often used during final construction and deployment of an end-user application. Executing an r-code file is generally much faster than compile-time executing a procedure source file. After you have completed all prototyping and testing, you might fine tune each procedure using the Application Compiler. (For more information on the Application Compiler, see the *Progress Basic Development Tools* (Character only) book, and in graphical interfaces, the on-line help for the Application Compiler.) You might then use the compiler to generate all the r-code files for your application and tune its final performance using database administration utilities, such as PROMON, to help establish optimum values for session startup and database connection parameters. For more information on application deployment, see the *Progress Client Deployment Guide* and the *Progress Developer's Toolkit*.

Optimized Dynamic Code Generation

If your application includes the Progress compiler to generate and execute code dynamically, you can increase performance using the COMPILE statement. This allows you to save semi-permanent r-code files between sessions that execute much more quickly from startup. It is also a more secure way to generate dynamic code, because you can remove the procedure source files during each session, leaving only the r-code. The Application Compiler also uses this technique to compile procedures for applications. For more information on the COMPILE statement, see the *Progress Language Reference*.

2

Programming Models

This chapter describes two general programming models that Progress supports and describes how to determine which model best suits your development effort. These models are as follows:

- Procedure-driven programming
- Event-driven programming

2.1 Procedure- and Event-driven Programming

Procedure-driven programming, which you can think of as traditional programming, defines the programming process as the development of procedures that explicitly direct the flow of data and control.

Event-driven programming defines the programming process as the development of procedures that respond to the flow of data and control as directed by the user, program, or operating system.

These programming models differ in flow of execution and structure. In addition, each model works best with a particular programming environment. The following sections describe these models in more detail.

2.2 Flow of Execution

The difference between the procedure-driven and event-driven models is primarily one of program flow. In a typical procedure-driven program, the execution of program statements is predetermined and controlled by the programmer. It generally runs from the beginning of the source file to the end. The programmer codes any branches or loops. In a typical event-driven program, program execution is largely determined by the system. First the programmer sets up a series of actions (triggers) to be executed in response to specific events. Then the system waits for user input to initiate these events. Finally, the system responds to these events, thereby controlling program execution.

2.2.1 Program Execution in a Procedure-driven Program

The flow of execution in a procedure-driven program is generally top-down and linear. User interaction with the program is limited; the user can choose from a few preprogrammed tasks. Each of these tasks consists of several actions that have been prearranged by the programmer. The sequence of these actions is also determined by the programmer; once a user selects a task, the application leads the user step-by-step through the actions required to complete the task. As actions are performed, the program might provide the user with status messages.

In the procedure-driven model, the application initiates calls to the operating system for user input when the program is ready. Figure 2–1 shows this process.

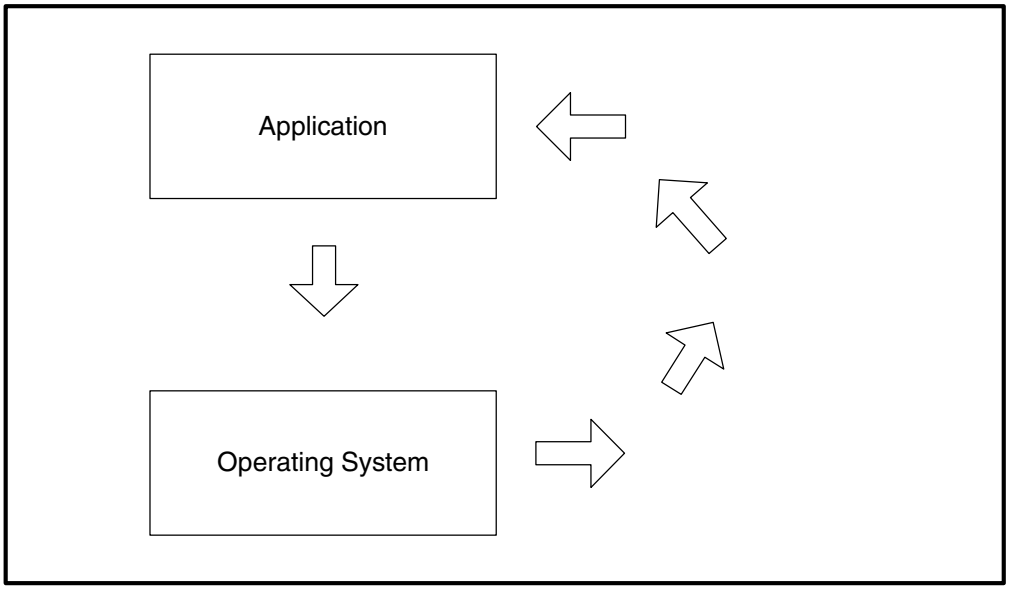


Figure 2–1: A Procedure-driven Application Calling the Operating System

Because movement from procedure to procedure is preprogrammed, the programmer is said to determine the flow of execution. The program performs only one process at a time. Generally, execution flows from the beginning of the source code to the end. The program may contain code loops, branches, and jumps that alter this flow, but it is the programmer who must plan for and manage the conditions for such alterations.

Figure 2–2 shows the flow of execution in a typical procedure-driven program.

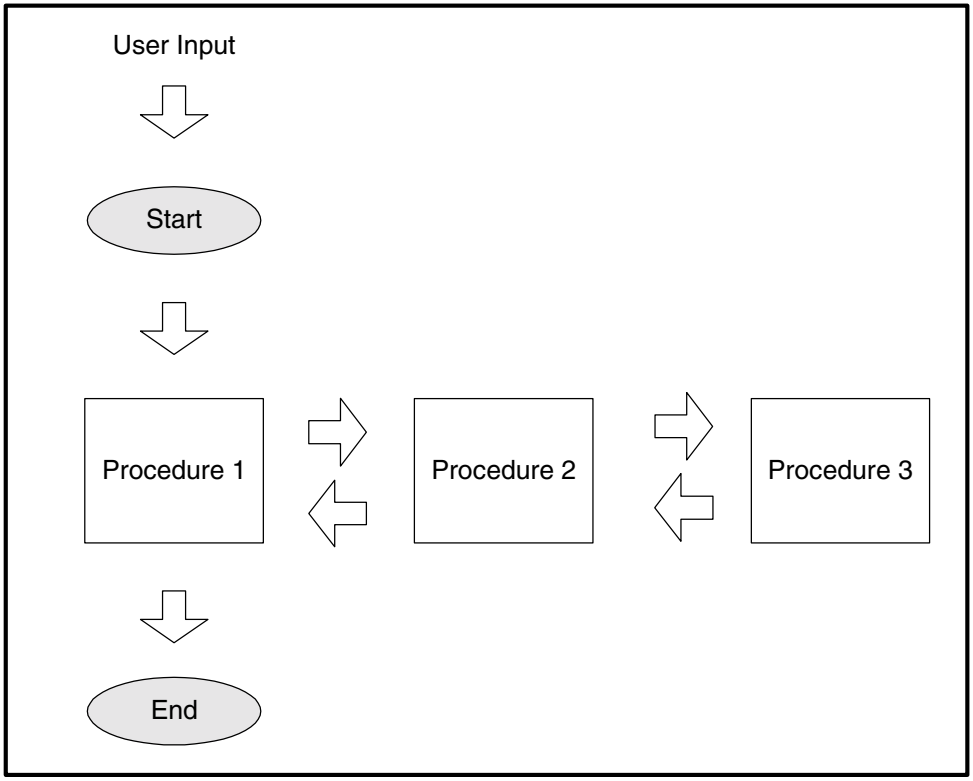


Figure 2–2: Flow of Execution in a Procedure-driven Program

Program execution typically begins at a unique starting point, such as when the user issues a RUN statement. Execution continues line by line through any number of procedures. With the exception of programmed branches to other procedures, program execution is sequential, both within the program and within each procedure. Program execution stops when it reaches the end point in the program. Because branching and looping behavior is tightly controlled, you can trace the flow of execution in a procedure-driven program.

2.2.2 Program Execution in an Event-driven Program

The flow of execution in an event-driven program is primarily determined by the system and, to some extent, by the user. Event-driven programs respond to input by repeatedly checking for events. An *event* is any type of interaction a user or operating system can have with an application. Events can occur as a result of user action or program code, or they can be triggered by the system.

In the event-driven model, virtually all actions supported by the application are available to the user at all times. The user can interact with the program freely, choosing from many simple tasks and combining them in any order to achieve a desired goal. Also, the user can usually change or undo an action easily, without disrupting the current task. Instead of just displaying status messages, event-driven applications display the effects of actions as they occur.

In the event-driven model, the operating system sends information on many kinds of events to the application. The application responds to and takes action on only those events that it has been preprogrammed to recognize and ignores any other events. Figure 2–3 shows this process.

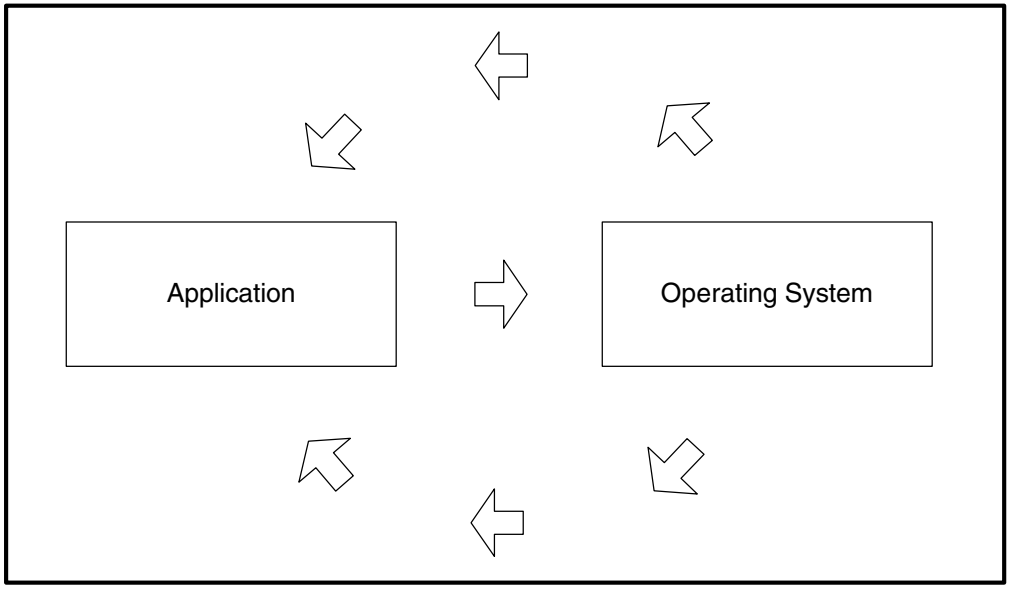


Figure 2–3: The Operating System Interacting with an Event-driven Application

Because it triggers events, the system is said to determine the flow of execution. The programmer sets up a series of triggers to be executed in response to events. Each *trigger* is a programmed response, a self-contained block of code that performs a specific task. Once these triggers are set up, the system waits for actions that initiate another sequence of events. This sequence of events determines which of the predefined triggers are executed and in what order. The programmer does not generally define the order in which these triggers are executed.

Figure 2–4 shows the flow of execution in a typical event-driven program:

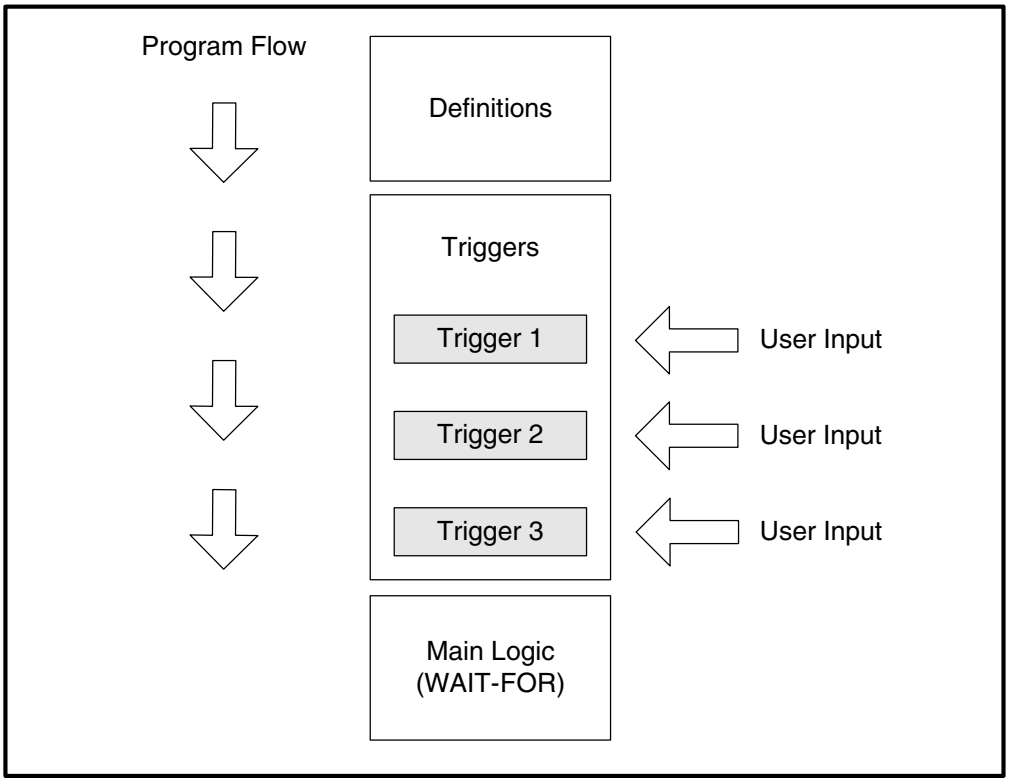


Figure 2–4: Flow of Control in an Event-driven Program

Program execution starts at the top of the program and proceeds sequentially to an input-blocking point where the program waits for events. During program initialization, the program sets up the user interface: it registers definitions for the functional objects of the user interface (buttons, frames, and triggers). Then the program enters a WAIT-FOR state. While a program is in a WAIT-FOR state, the program is said to be blocking for input. In this state, the user interacts with the functional components of the interface to initiate events. When the user performs an action that initiates a specific event, an event handler or trigger code block executes.

The main purpose of the WAIT-FOR condition is to control dismissing the interface. While a program is in this state, the user can interact with user interface objects, based on what their labels indicate about their associated functions, to initiate events. The user initiates an event by performing an action that triggers an event, usually with the keyboard or mouse. When there are no more events to trigger, the user can either cancel the transaction or initiate the event that satisfies the WAIT-FOR condition. After dismissing the interface, program control returns to the interface that called it, if any.

Progress supports user interface triggers that allow users to associate a task (large or small) with an action, such as choosing a button or menu item, or entering a field. In addition, Progress supports the normal control structures required for procedure-driven programming as well as flexible control structures (ON, ENABLE, DISABLE, WAIT-FOR, and PROCESS EVENTS statements) for event-driven programming.

In an event-driven application, complex sequences of actions are not preprogrammed. Only the specific individual actions are preprogrammed. Each of these actions might be associated with an action that takes place in response to a menu item, button, or combination of key presses.

Event-driven programming is rooted in interrupt-driven, real-time operating system technology. With interrupt-driven processing, the operating system can interact with the user and allow the user to access the machine during program execution. Requests to the operating system (interrupts) are acted on immediately. Events, therefore, resemble operating system or real-time interrupts, where *real-time* refers to the requirement that the software coordinate its activities with the real-world environment, in this case the user.

When you are interrupted while performing a task, you typically stop what you are doing, record where you are, take care of the interruption, and later return to your original task. Interrupt handling in computers provides for a similar process to occur in relation to the execution of a program. When an interrupt signal occurs within a computer system, the computer's central processing unit:

1. Stops executing the current program or procedure and saves its context (WAIT-FOR).

2. Starts executing another program or procedure (trigger).

Thus, when a user initiates an event, it sends an interrupt signal to the operating system. This signal then triggers a real-time response (it applies an event) in the program.

2.3 Program Structures

Another difference between procedure-driven programs and event-driven programs is in their structure. Procedure-driven programs consist of structured code blocks, while event-driven programs primarily contain blocks of trigger code.

2.3.1 Program Structure in a Procedure-driven Program

The basic structure of a procedure-driven program is as follows.

1. Test for a condition.

2. Execute a block of code.

3. Control the return.

The following example of a simple procedure-driven program using a FOR EACH loop illustrates this structure:

p-procd.p

```
FOR EACH customer:
  DISPLAY cust-num name phone WITH 6 DOWN.
  PROMPT-FOR phone.
  IF INPUT phone NE phoneyl
  THEN DO:
     MESSAGE "Phone Number changed".
     BELL.
  END.

  ASSIGN phone.
END.
```

These notes help explain the code:

- The FOR EACH statement starts an iterating block and finds one customer record in the database on each iteration.
- The DISPLAY statement moves the customer data from the database to the screen buffer and displays six occurrences of the Cust-Num, Name, and Phone fields in an unnamed default frame.
- The PROMPT-FOR statement requests user input for the Phone field and then places it in the screen buffer.
- The IF . . . THEN statement includes an INPUT statement and a DO block. The code instructs Progress to compare the current value in the Phone field to its value in the database. If the two values are not the same, Progress displays the message "Phone Number changed", the terminal beeps, and program execution drops down to the next line, the ASSIGN statement. If the two values are identical, Progress skips the DO block and moves directly to the ASSIGN statement.
- The ASSIGN statement instructs Progress to move the new Phone data into the database.
- When all the customer records have been read, the program ends.

2.3.2 Program Structure in an Event-driven Program

An event-driven program is a collection of triggered responses. The basic structure is as follows.

1. Definitions — Define and enable the user interface objects that comprise its interface.
2. Triggers — Define the behavior that should occur in response to events applied to these user interface objects. The code within a trigger block tells the system to watch for a condition, execute a block of code, and control the return.
3. Main Logic — Establish a WAIT-FOR state to allow the user to interact with the user interface objects and then dismiss the user interface when the WAIT-FOR condition is satisfied. The WAIT-FOR statement typically tests for the end condition.

The following example is a simple event-driven program using the WAIT-FOR statement:

p-eventd.p

```
DEFINE BUTTON btn_1 LABEL "Next".
DEFINE BUTTON btn_2 LABEL "Prev".

FIND FIRST customer.
DISPLAY cust-num name phone SKIP (0.5)
      btn_2 TO 40 SPACE btn_1
      SKIP (0.5)
    WITH FRAME f SIDE-LABELS.

ON CHOOSE OF btn_1 DO:
  FIND NEXT customer NO-ERROR.
  IF NOT AVAILABLE customer THEN DO:
      FIND LAST customer.
      BELL.
  END.
  DISPLAY cust-num name phone WITH FRAME f.
END.

ON CHOOSE OF btn_2 DO:
  FIND PREV customer NO-ERROR.
  IF NOT AVAILABLE customer THEN DO:
      FIND FIRST customer.
      BELL.
  END.
  DISPLAY cust-num name phone WITH FRAME f.
END.

ON LEAVE OF phone DO:
  IF INPUT customer.phone NE customer.phone THEN DO:
    ASSIGN customer.phone.
    BELL.
    MESSAGE "Phone Number changed."
      VIEW-AS ALERT-BOX INFORMATION.
  END.
END.

ENABLE phone btn_2 btn_1 WITH FRAME f.
WAIT-FOR GO OF FRAME f.
```

These notes help explain the code:

- The two DEFINE BUTTON statements define two static buttons, btn_1 and btn_2, and label them “Next” and “Prev”. These labels provide visual cues to the user that relate to each button’s function.

- The next two statements instruct Progress to find the first customer record in the database and display the Cust-Num, Name, and Phone data in frame f, labeling each field to the left and center.

- The first ON statement defines trigger code that Progress executes each time the user applies a CHOOSE event to btn_1, the Next button. This CHOOSE behavior applies only to btn_1.

  NOTE: At run time, the CHOOSE event in this example is applied by clicking btn_1. Other programming techniques for applying a CHOOSE event include using a 4GL statement or a keyboard accelerator.

  The trigger code instructs Progress to find the next customer record and omit any error messages. In the next line, Progress tests whether a customer record was found. If not, Progress finds the last customer record and the terminal beeps. Finally, Progress displays the contents of the Cust-Num, Name, and Phone fields in frame f and ends processing in the block.

- The next ON statement defines trigger code that Progress executes each time the user applies a CHOOSE event to btn_2, the Prev button. This CHOOSE behavior applies only to btn_2.

  The trigger code instructs Progress to find the previous customer record and omit any error messages. In the next line, Progress tests whether a customer record was found. If not, Progress finds the first customer record and the terminal beeps. Finally, Progress displays the contents of the Cust-Num, Name, and Phone fields in frame f and ends processing in the block.

- The last ON statement defines trigger code that Progress executes each time the user applies a LEAVE event in the Phone field. This LEAVE behavior applies only to the Phone field.

  The trigger code instructs Progress to compare the current value in the Phone field to its value in the database. If the two values are not the same, Progress moves the new Phone data into the database, the terminal beeps, an information box appears and displays the message “Phone Number changed”, and processing ends in this block.

- The ENABLE statement places the Phone field on the screen with the Prev and Next buttons in frame f. (When Progress executes this statement, frame f is viewed on the screen.)
- The WAIT-FOR statement establishes a WAIT-FOR condition for frame f. When the user applies a GO event, the WAIT-FOR condition is satisfied.

2.4 When to Use the Procedure-driven and Event-driven Models

Choosing a programming model has a lot to do with the environment in which your application runs. Progress supports both character and window interfaces for several operating system environments.

Multi-tasking windowing systems, such as Microsoft Windows, offer many graphical interface objects that are more intuitive for end users. In graphical interfaces, Progress supports the use of colors, fonts, and direct manipulation (selecting, moving, and resizing) of screen objects by the user. Attributes and methods for these objects provide visual feedback to the user. In this environment, the user has access to multiple windows and can use a mouse and keyboard to enter complex input to the program. Graphical interface objects are designed to receive events from the operating system. Thus, if you plan to use these objects, you should use the event-driven model.

Character interfaces, which Windows and UNIX provide, supply a few user interface objects, including fields, text, and primitive graphics. In this environment, the user has access to only one window at a time and uses the keyboard to enter input to the program. The program processes the input behind the scenes and displays the final results as another stream of characters on the user's screen. In general, you should use the procedure-driven model whenever a simple character interface is all your application requires.

Progress offers event-driven capabilities for both graphical and character environments. Specifically, Progress provides versions of the graphical user interface objects for character interfaces through its windowing system. Your application might combine elements of both event-driven and procedure-driven design.

3

Block Properties

The Progress language is *block structured*, letting you group together parts of your procedures into blocks. Blocks are useful in two significant ways:

- You can use blocks to apply a se1est of characteristics to a group of statements rather than applying the characteristics on a statement-by-statement basis.
- Progress provides a wide range of automatic processing services for blocks.

This chapter describes:

- The different kinds of blocks and their properties
- Procedure properties
- External procedures
- Internal procedures
- Persistent and non-persistent procedures
- User-defined functions
- Procedure overriding

3.1 Blocks and Their Properties

Progress provides the following types of blocks:

- REPEAT blocks
- FOR blocks
- DO blocks
- Procedure blocks
- Trigger blocks
- EDITING blocks

3.1.1 REPEAT, FOR, and DO Blocks

REPEAT, FOR, and DO blocks consist of a block header statement (REPEAT, FOR, or DO) followed by zero or more statements terminated by an END statement. Each of these blocks can contain any other type of block, except a procedure and trigger block. REPEAT, FOR, and DO blocks execute wherever they appear in a procedure or trigger block.

3.1.2 Procedure Blocks

There are two types of procedure blocks: external and internal. An *external procedure* block consists of a text file containing zero or more statements that you can compile into a unit and that you can execute by name using the RUN statement. An external procedure is the most basic block and can contain all other types of statements and blocks.

An *internal procedure* block consists of a PROCEDURE statement followed by one or more statements terminated by an END statement that you can execute by name using a RUN statement. An internal procedure is always compiled as part of an external procedure. An internal procedure cannot contain other internal procedure blocks, but it can contain any other type of statement or block that an external procedure can contain. For more information on procedure blocks, see the "Procedure Properties" section.

3.1.3 Trigger Blocks

A trigger block consists of a single statement, or zero or more statements starting with DO and ending with END, that defines a database trigger, user-interface trigger, or procedure trigger. It appears as part of a user-interface widget or trigger definition, and can contain all other types of blocks except procedure blocks. A trigger block executes in response to an event, depending on the type of trigger. For more information on database triggers, see Chapter 11, "Database Triggers." For more information on user-interface triggers, see Chapter 16, "Widgets and Handles." For information on procedure triggers, see the "External Procedures" section in this chapter.

3.1.4 EDITING Blocks

An EDITING block appears as the EDITING-phrase part of an input-blocking statement, such as PROMPT–FOR, SET, or UPDATE. It consists of the EDITING block header followed by zero or more statements terminated by an END statement. This block executes once for each key stroke received during execution of the input statement, and thus allows the application to manage the input for each field one key stroke at a time.

NOTE: The EDITING block is maintained primarily for compatibility with earlier versions. Event-driven applications generally use user interface triggers to manage user input. For more information on EDITING blocks and user interface triggers, see Chapter 6, "Handling User Input."

3.1.5 Block Properties

Progress provides a different set of default processing services for each type of block. Table 3–1 shows what properties Progress associates with each type of block. Implicit properties are those that come by default. Explicit properties are those that you can associate with the block by adding Progress keywords, phrases, or statements to the block definition. The properties in this table represent the basic properties associated with the type of block. Progress provides additional properties depending on the block type.

Table 3–1: Block Properties *(1 of 2)*

Property	REPEAT		FOR		DO		Procedure or Trigger	
	Implicit	Explicit	Implicit	Explicit	Implicit	Explicit	Implicit	Explicit
Looping	YES	WHILE TO/BY	YES[5]	WHILE TO/BY	NO	WHILE TO/BY	NO	NO

Table 3–1: Block Properties *(2 of 2)*

Property	REPEAT		FOR		DO		Procedure or Trigger	
Record reading	NO	NO	YES	Record Phrase	NO	NO	NO	NO
Frame scoping	YES	WITH FRAME	YES	WITH FRAME	NO	WITH FRAME	YES	NO
Record scoping	YES	FOR	YES	NO	NO	FOR	YES[1]	NO
UNDO	YES	TRANS ACTION ON ERROR	YES	TRANS ACTION ON ERROR	NO	TRANS ACTION ON ERROR	YES	N0
ERROR processing	YES	ON ERROR	YES	ON ERROR	NO	ON ERROR	YES	NO
ENDKEY processing	YES	ON ENDKEY	YES	ON ENDKEY	NO	ON ENDKEY	YES	NO
STOP processing	YES[2]	ON STOP	YES[2]	ON STOP	NO	ON STOP	YES	NO
QUIT processing	YES[2]	ON QUIT	YES[2]	ON QUIT	NO	ON QUIT	YES	NO
System transaction	YES[2]	TRANS ACTION[3]	YES[2]	TRANS ACTION[3]	NO	TRANS ACTION[3] ON ERROR[2]	YES[2]	NO
Parameter Passing	NO	NO	NO	NO	NO	NO	NO	YES[1]

1 A record is scoped to the containing .p unless you explicitly define the buffer in a trigger or internal procedure

2 Only if the block contains database updates or reads with exclusive locks and there is no higher level transaction active.

3 Only for procedure blocks.

4 Only for procedure blocks.

5 Using the EACH option.

Scope is one of the basic properties of all blocks. *Scope* is the duration that a resource is available to an application, and blocks determine the scope of certain resources. For example, records and certain types of frame definitions are automatically scoped to certain blocks (see Table 3–1). Although you can place them in other blocks, local data and widget definitions are always scoped to the smallest enclosing procedure block (internal or external). Similarly, the compiler accepts a trigger or internal procedure block in most other blocks. However, trigger definitions are scoped to the enclosing trigger or procedure block (internal or external) in which they are defined, and internal procedure definitions are always scoped to the enclosing external procedure block in which they are defined. Thus, an internal procedure defined in a REPEAT block is actually defined only once (not for the number of times the block repeats), and is available for execution anywhere in the external procedure.

The following sections explain the looping, record reading, frame scoping, and record scoping block properties. For information on the UNDO, ERROR, ENDKEY, STOP, and QUIT properties, see Chapter 5, "Condition Handling and Messages." For information on transactions, see Chapter 12, "Transactions."

3.1.6 Looping

Looping, or iteration, is one of the automatic processing services that Progress provides for blocks. REPEAT blocks and FOR EACH blocks iterate automatically and you can force a DO block to iterate by using a DO WHILE, or a DO *variable* = *expression1* TO *expression2* [BY *k*]. The following code fragments show iterations using these block statements:

p-bkchp.p

```
/* This REPEAT block loops through its statements until the user
   presses the END-ERROR (F4) key.                                */

REPEAT:
  PROMPT-FOR customer.cust-num.
  FIND customer USING cust-num.
  DISPLAY customer WITH 2 COLUMNS.
END.
```

p-bkchp2.p

```
/* This FOR EACH block reads the records from the customer
   table one at a time, processing the statements in the
   block for each of those records.                       */

FOR EACH customer:
  DISPLAY customer WITH 2 COLUMNS.
END.
```

REPEAT and FOR EACH blocks iterate automatically. You can make a DO block iterate by using the WHILE option or the TO option in the DO block header:

p-bkchp3.p

```
/* This block loops through its statements for the first 5 customers. */

DEFINE VARIABLE i AS INTEGER.

DO i = 1 TO 5:
  FIND NEXT customer.
  DISPLAY customer WITH 2 COLUMNS.
END.
```

3.1.7 Record Reading

The FOR EACH block is the only block statement that automatically reads records each time it iterates. The following examples demonstrate this:

p-bkchp4.p

```
/* This FOR EACH block reads the records from the customer
   table one at a time, processing the statements in the
   block for each of those records.                      */

FOR EACH customer:
  DISPLAY customer WITH 2 COLUMNS.
END.
```

You can be more specific about which records you want to read by using a record phrase in the FOR EACH block header:

p-bkchp5.p

```
/* The record-spec (made up of a simple WHERE phrase in
   this example) tells the FOR EACH block to read only
   those records from the customer file that have a
   credit-limit greater than 50,000. */

FOR EACH customer WHERE credit-limit > 50000:
    DISPLAY customer WITH 2 COLUMNS.
END.
```

3.1.8 Frame Allocation

Each statement that displays data or prompts you to enter data uses a frame for display or input of that data. A procedure can use many different frames. Progress uses these guidelines to determine which frame a particular statement uses to display data:

- If the statement explicitly names a frame, the statement uses that frame to display data.
- If the statement does not explicitly name a frame, Progress displays the data in the default frame for the block in which the statement occurs.

Progress automatically allocates a frame for REPEAT, FOR EACH, triggers, and procedure blocks that display data. If you do not use the Frame phrase with these blocks, Progress creates an *unnamed* frame. The unnamed frame is the default frame for the block. If you name a frame in the block header, that frame becomes the default frame for the block and Progress does not allocate an unnamed frame for that block. Progress scopes the frame named in the block header statement to that block if it is the first block to reference the frame.

If you do not specify a default frame for a DO block in the block header, the default frame for the DO block is the default frame for the nearest enclosing FOR EACH, REPEAT, trigger, or procedure block, or the nearest enclosing DO block that explicitly names a frame.

Progress creates separate frames for DO blocks only if you specify a frame phrase in the DO statement. If the frame phrase contains the FRAME keyword and names a new frame, then Progress creates a separate frame and that frame is the default frame for the DO block. If the frame phrase does not contain the FRAME keyword, then Progress creates a new unnamed frame and that becomes the default frame for the DO block.

This procedure demonstrates frame allocation for the procedure block and a REPEAT statement:

p-bkchp6.p

```
/* The default frame for the procedure block is an unnamed
   frame.The first DISPLAY statement uses a different frame,
   aaa.The default frame for the REPEAT block is bbb.The
   PROMPT-FOR statement does not use the default frame for the
   block;instead it uses frame aaa.The DISPLAY statement uses
   the default frame for the REPEAT block.                    */

DISPLAY "Customer Display" WITH FRAME aaa.
REPEAT WITH FRAME bbb:
  PROMPT-FOR customer.cust-num WITH FRAME aaa.
  FIND customer USING cust-num.
  DISPLAY customer WITH 2 COLUMNS.
END.
```

In this procedure, Progress allocates frame aaa to the procedure block. This frame is not the default frame for the procedure block. If a subsequent statement in the procedure block displayed data without naming a frame, Progress would create an unnamed frame and it would be the default frame for the procedure block. As it stands, the procedure block has no default frame. The default frame for the procedure block is always an unnamed frame.

The default frame for the REPEAT block is frame bbb. Since the DISPLAY statement contained within the REPEAT block does not specify a frame, it uses the default frame bbb.

Frame Scope

The *scope* of a frame is the outer-most block in which you reference that frame. (Hiding a frame or setting its VISIBLE attribute does not scope the frame; nor does the DEFINE FRAME statement.) The following procedure demonstrates this:

p-bkchp7.p

```
/* The scope of frame aaa is the procedure block because
   that is the first block that references frame aaa.    */

DISPLAY "Customer Display" WITH FRAME aaa.
REPEAT:
  PROMPT-FOR customer.cust-num WITH FRAME aaa.
  FIND customer USING cust-num.
  DISPLAY customer WITH 2 COLUMNS.
END.
```

Progress scopes every frame to a block. Scoping a frame to a block does not make that frame the default frame for the block. Progress automatically scopes frames to REPEAT, FOR EACH, and procedure blocks. Progress only scopes frames to DO blocks if you indicate a new frame in a frame phrase (DO WITH FRAME).

Frame Services

Progress provides these services for each of the frames scoped to a block:

- Advancing
- Hiding and viewing
- Clearing
- Retaining previously entered data during RETRY of a block

Progress determines when to provide these services based on the block to which it scopes the frame. For more information on frame scoping, see Chapter 19, “Frames.”

Borrowed Frames

Any trigger or internal procedure can reference frames that are scoped to the containing procedure. A user interface trigger for a frame or field-level widget automatically borrows the frame for that widget from the containing procedure. This holds true even if the frame is the default (unnamed) frame. This allows you to reference the field value within the trigger without specifying the frame. You can borrow other frames by referencing them within the trigger or internal procedure. For example, the following procedure contains several instances of borrowed frames:

p-borfrm.p

```
FORM
   Customer.Name LABEL "Search String"
   WITH FRAME prompt-frame.

FORM
   Customer.Name Customer.Credit-limit
   WITH FRAME upd-frame.

ON NEXT-FRAME OF Customer.Name IN FRAME prompt-frame
   DO:
      FIND Customer USING Customer.Name.
      DISPLAY Customer.Name Customer.Credit-limit WITH FRAME upd-frame.
      ENABLE Customer.Credit-limit WITH FRAME upd-frame.
   END.

ON LEAVE OF Customer.Credit-limit
   ASSIGN Customer.Credit-limit.

STATUS INPUT "Press " + KBLABEL("NEXT-FRAME") + " to search; " +
             KBLABEL("END-ERROR") + " to exit".

ENABLE Customer.Name WITH FRAME prompt-frame.

WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

In `p-borfrm.p`, a LEAVE trigger is defined for the Credit–limit field in upd–frame. Because the trigger borrows that frame, you do not have to qualify the field reference in the ASSIGN statement with the frame name. The ON statement for the NEXT–FRAME trigger of the Name field explicitly references prompt–frame. Therefore, Progress assumes that frame when you reference the Name field in the FIND statement. The DISPLAY and ENABLE statements explicitly borrow the upd–frame from the outer procedure.

3.1.9 Record Scoping

When Progress reads a record from the database, it stores that record in a record buffer. The scope of the record is the portion of the procedure where that record buffer is active. Record scope determines when Progress clears the record from the buffer, when it writes the record to the database, and how long a record lock is in effect.

Generally, the scope of a record is the smallest enclosing block that encompasses all references to the record. That is, the record is active until the block ends. However, there are exceptions to this rule depending on whether the record is weak scoped, strong scoped, or introduced by a free reference.

The exceptions are as follows:

- A *strong-scoped reference* occurs for buffers that are referenced in the header of a REPEAT FOR or DO FOR block. Any reference to that buffer within the REPEAT FOR or DO FOR block is a strong-scoped reference.
- A *weak-scoped reference* occurs for buffers that are referenced in the header of a FOR EACH or PRESELECT EACH block. Any reference to that buffer within the FOR EACH or PRESELECT block is a weak-scoped reference.
- All other references to records are *free references*.

The general scoping rules are as follows:

- If you have a strong-scoped reference to a buffer, you cannot reference that buffer in a containing block. The buffer is always scoped to the strong-scope block. You can however, have two sequential blocks that each contain strong-scoped references to the same buffer.
- If you have a weak-scoped reference to a buffer, you can reference that same buffer outside the weak-scope block. This raises the scope of the buffer to the outer block. You cannot, however, nest two weak-scope blocks for the same buffer. Also, a weak-scope block cannot contain any free references to the same buffer.
- If you have a free reference to a buffer, Progress tries first to scope that buffer to the nearest enclosing block with record scoping properties. However, other references to the same buffer outside that block cause Progress to raise the scope to a higher block.

The following example contains two DO FOR blocks that each contain a strong-scoped reference to the customer buffer:

p-strscp.p

```
DO:

   DO FOR customer:                                  }
      FIND FIRST customer.                           }  Strong scope
      DISPLAY cust-num name WITH FRAME a.            }
   END.

   DO FOR customer:                                  }
      FIND NEXT customer.                            }  Strong scope
      DISPLAY cust-num name WITH FRAME b.            }
   END.

END.
```

In `p-strscp.p`, the customer buffer is scoped to each of the DO FOR blocks. Because strong-scoped blocks cannot be nested, you **cannot** change the outer DO to a DO FOR customer.

You **cannot** make a free reference to the customer buffer outside the DO FOR block. For example, the following code does not compile:

p-stscp2.p

```
/* This code cannot compile. */

DO:
   DO FOR customer:                  }
      FIND FIRST customer.           }  Strong scope
      DISPLAY cust-num name.         }
   END.

   FIND NEXT customer. ————— Invalid free reference
END.
```

If you try to compile `p-stscp2.p`, you get a compile error.

The following example contains two FOR EACH blocks that each contain a weak-scoped reference to the customer buffer:

p-wkscp.p

```
DO:
   FOR EACH customer:                                  } Weak scope
      DISPLAY cust-num name WITH FRAME a.              }
   END.

   FOR EACH customer:                                  } Weak scope
      DISPLAY cust-num name WITH FRAME b.              }
   END.
END.
```

In `p-wkscp.p`, the customer buffer is scoped to each of the FOR EACH blocks. However, because these blocks apply a weak scope, you can reference the buffer outside these blocks and Progress automatically enlarges the scope. For example, the following procedure contains a weak-scope reference and a free reference to the customer buffer:

p-wkscp1.p

```
FOR EACH customer:                                     }
   DISPLAY cust-num name WITH FRAME a.                 }
END.                                                   }  Scope raised to
                                                       }  procedure
FIND FIRST customer.                                   }
DISPLAY customer.cust-num name WITH FRAME b.           }
```

In `p-wkscp1.p`, the scope of the customer buffer is referenced in the FOR EACH block, but a free reference also occurs outside the FOR EACH block. The scope is raised to the procedure block.

You cannot nest two blocks that apply a weak scope to the same buffer. For example, the following code does not compile:

p-wkscp2.p

```
/* This code does not compile. */

DEFINE VARAIBLE saved-rep LIKE customer.sales-rep.

DO:

   FOR EACH customer:
      DISPLAY cust-num name WITH FRAME a.
      saved-rep = customer.sales-rep.
      FOR EACH customer:
         IF customer.sales-rep = saved-rep
         THEN DISPLAY cust-name WITH FRAME a.
         DOWN WITH FRAME a.
      END.
   END.

END.
```

Weak scope

Nesting weak scope not allowed

You can however, enclose a weak-scoping block within a strong-scoping block, as in the following example:

p-wkspc3.p

```
DO FOR customer:
   FOR EACH customer:
      DISPLAY cust-num name WITH FRAME a.
   END.
END.
```

Weak scope

Scope raised to DO FOR block

In `p-wkscp3.p`, the scope of the customer buffer is raised from the weak-scope FOR EACH block to the strong-scope DO FOR block.

You can also have sequential strong-scope and weak-scope blocks for the same record, as in the following example:

p-wkstr.p

```
DO FOR customer:                    } Strong scope
    FIND FIRST customer.
   DISPLAY cust-num name.
END.

FOR EACH customer:                  } Weak scope
   DISPLAY cust-num name.
END.
```

Note that the scope is not raised in `p-wkstr.p`. Instead, the DO FOR and FOR EACH blocks each scope the customer record separately. A reference to the customer buffer outside these blocks would be invalid and would cause a compiler error.

Any reference that is neither a strong-scope reference nor a weak-scope reference is a free reference. For example, the following code contains two free references to the customer buffer:

p-frref1.p

```
REPEAT:                                  }
     FIND NEXT customer.                 }
     DISPLAY cust-num name WITH FRAME a. }
     LEAVE.                              } Scope raised to procedure
END.                                     }
                                         }
FIND NEXT customer.                      }
DISPLAY cust-num name WITH FRAME b.      }
```

In `p-frref1.p`, the first free reference occurs within the REPEAT block. Progress initially tries to scope the customer buffer to that block. However, because another free reference occurs after the REPEAT block, Progress raises the scope to the procedure.

The following example also contains two free references to the customer buffer. However, in this case each reference is enclosed in a REPEAT block:

p-frref2.p

```
REPEAT:                                          }
     FIND NEXT customer.                         }
     DISPLAY cust-num name WITH FRAME a.         }
     LEAVE.                                      }
END.                                             }  Scope raised to procedure
                                                 }
REPEAT:                                          }
     FIND NEXT customer.                         }
     DISPLAY cust-num name WITH FRAME b.         }
END.                                             }
```

In this case, Progress still raises the scope to the procedure block. This means that the first iteration of the second REPEAT block finds the second customer record.

To determine the scope of a block, compile your procedure using the LISTING option. For example, to compile the `p-frref2.p` procedure, issue this command:

```
COMPILE p-frref2.p LISTING p-frref2.out.
```

The p-frref2.out argument specifies the file in which you want Progress to store the listing information. You can use any valid filename. When you view this file, you see the following screen:

```
p-frref2.p                              PROGRESS(R) Page 1

{} Line Blk
-- ---- ---
     1    1 DO:
     2    2  REPEAT:
     3    2     FIND NEXT customer.
     4    2     DISPLAY cust-num name WITH FRAME a.
     5    2     LEAVE.
     6    1  END.
     7    1
     8    2  REPEAT:
     9    2     FIND NEXT customer.
    10    2     DISPLAY cust-num name WITH FRAME b.
    11    1  END.
    12    1
    13      END.

p-frref2.p                               PROGRESS(R) Page 2

     File Name       Line Blk. Type Tran            Blk. Label
-------------------- ---- --------- ---- --------------------------------
p-frref2.p              0 Procedure No
    Buffers: sports.Customer

p-frref2.p              1 Do        No
p-frref2.p              2 Repeat    No
    Frames:  a

p-frref2.p              8 Repeat    No
    Frames:  b
```

This listing shows that the buffer sports.customer is scoped to the procedure block.

Record scoping provides several services. The scope of a record tells Progress:

- When to write the record to the database. Progress automatically writes a modified record to the database at the end of the record scope, as shown in this procedure.

p-bkchpa.p

```
/* On each iteration of the loop, the FOR EACH statement reads a
   single record from the database into the record buffer. The scope
   of the record is the FOR EACH block because that is the outermost
   block that references the record. At the end of the record scope,
   which is the end of the iteration that uses that record, Progress
   writes the record to the database.                                */

FOR EACH customer:
  UPDATE customer WITH 2 COLUMNS.
END.
```

- When to release a record for use by other users in a multi-user system. Progress releases a record at the end of each iteration of the block to which the record is scoped. Whether the record will then be available for other users depends on the type of lock on the record and on whether a transaction is still active. See Chapter 12, "Transactions" and Chapter 13, "Locks" for more details.

- When to validate the record against any unique index and mandatory field criteria defined in the Dictionary.

- When to reinitialize the current position within an index. Progress maintains this position for each index used within a record's scope. The position is initialized upon entry to the block to which the record is scoped.

- How to resolve references to unqualified field names. Within a record's scope, you can use field names (without specifying the table) to refer to fields that do not share a name with fields in other tables in the database. You must include the table name when you reference fields of the same name from two or more tables.

3.1.10 Transactions

A *transaction* is a set of changes to the database that is either executed completely or leaves no modification to the database. A *subtransaction* is nested in a transaction and encompasses all activity within one iteration of the following kinds of blocks: DO ON ERROR, FOR EACH, procedure, REPEAT, and triggers. Table 3–2 shows which kinds of blocks start transactions and subtransactions.

Table 3–2: Starting Transactions

Type of Block	Transaction Not Active	Transaction Active
DO TRANSACTION FOR EACH TRANSACTION REPEAT TRANSACTION Any DO ON ENDKEY, DO ON ERROR, FOR EACH, REPEAT, or procedure block that contains statements that modify the database or that read records using an EXCLUSIVE–LOCK.	Starts a transaction	Starts a subtransaction
Any FOR EACH, REPEAT, or procedure block that does not contain statements that modify the database or read records using an EXCLUSIVE–LOCK.	Does not start a transaction	Starts a subtransaction

For more information on transactions, see Chapter 12, "Transactions."

When an error occurs, Progress performs differently depending on the context. Progress's error processing allows you to accept default action depending on the type of error, or to define the error behavior yourself. The following section discusses error processing.

3.1.11 Error Processing

Progress's error processing depends on what type of error is generated. Progress recognizes the following errors:

- A procedure-generated error
- A predefined error key
- A predefined **ENDKEY** key
- The **END–ERROR** key
- A system or software failure
- The **STOP** key

Progress performs default processing when it encounters these errors. When Progress encounters a procedure error or the user presses a predefined error key, Progress performs an UNDO, RETRY. When the user presses a predefined endkey, Progress performs an UNDO, LEAVE. When the user presses END–ERROR, Progress performs an UNDO, LEAVE if the user has not entered any keyboard events, and an UNDO, RETRY if they have. When the user presses **STOPS**, Progress undoes the current transaction and retries the startup procedure. When the system fails, Progress undoes any partially completed transactions for all users on the system.

For more information on how Progress handles errors and how you can handle errors within the 4GL, see Chapter 5, "Condition Handling and Messages."

3.2 Procedure Properties

The procedure is the fundamental block in Progress. All types of procedures, external and internal, have certain common properties:

- They can contain all other types of blocks.
- You can execute them by name using the RUN statement.
- They can accept run-time parameters for input or output.
- They can define their own data and user interface environment (context) with restrictions depending on the type of procedure.
- They can share data and widgets defined in the context of another procedure, with restrictions depending on the type of procedure.
- They can execute recursively. That is, they can run themselves in a new context.

You can nest most blocks within a procedure. For example, you can include a DO, FOR, or REPEAT block within another DO, FOR, or REPEAT block. The number of times that you can nest blocks depends on your setting of the Nested Blocks (–nb) startup parameter. The default depth is 50. For more information on –nb, see the *Progress Startup Command and Parameter Reference*.

Depending on the type of procedure, you can define any number of local objects within its context, including buffers, temporary tables, queries, variables, I/O streams, frames, and many other widgets. You can reference each one of these objects within the procedure after it is defined. In addition, you can share or pass many of these objects among procedures in different ways.

The actual content of a procedure's context and how it is managed depends on whether the procedure is an external or internal procedure, and on how you define and execute it.

3.3 External Procedures

There are actually two types of external procedures: non-persistent and persistent. Most of this section applies to both. For information on persistent procedures and how they differ from non-persistent procedures, see the "Persistent and Non-persistent Procedures" section.

The *external procedure* is the most basic procedure. It can contain a single 4GL statement, many statements, or it can be an empty file. Its main characteristic is that it is a block of code that can be created, stored, and compiled separately from all other procedures as an operating system file. As such, you can run call an external procedure from any other procedure by specifying its filename in a RUN statement:

SYNTAX

```
RUN filename [ run-options ]
```

3.3.1 Procedure Filenames

You can specify *filename* using any of these forms:

- The unquoted name of the procedure file.
- VALUE(*char–expression*), where *char–expression* is any character expression equal to the procedure filename.
- An unquoted reference to an r-code library member. For more information on r-code libraries and how to reference them, see the *Progress Client Deployment Guide*.

For example, this statement runs the procedure file `p-exprc2.p`, listed later in this section:

```
RUN p-exprc2.p.
```

NOTE: When Progress encounters a RUN statement, it first searches for an available internal procedure whose name matches *filename*. If it cannot find a matching internal procedure, it then tries to find the external procedure. For more information on internal procedures, see the "Internal Procedures" section.

Because an external procedure is an operating system file, the maximum length and composition of an external procedure filename depends on your operating system. By convention, procedure source files end with a .p or .w extension. The compiled versions of procedure files, r-code files, **must** end with a .r extension. If you specify the name of a source file, Progress tries to find and execute the corresponding r-code file before compiling and executing the source file.

Progress also imposes additional limits to promote compatibility between versions of operating systems like UNIX. For more information on the length and composition of external procedure names, see the section on Progress limits in the *Progress Client Deployment Guide*.

3.3.2 Standard Execution Options

The *run–options* of the RUN statement can include run-time parameters, compile-time arguments, and the NO–ERROR option for handling run-time errors. For information on the NO–ERROR option, see Chapter 5, "Condition Handling and Messages."

Run-time parameters include comma-separated data items enclosed in parentheses. Each data item is preceded by INPUT, OUTPUT, INPUT–OUTPUT, or BUFFER, corresponding to the type of parameter defined by the DEFINE PARAMETER statement in the procedure. In general, the data items passed in the RUN statement must match, in data type and order, the corresponding parameters defined in the called procedure. For more information on run-time parameters, see the examples and the paragraphs on procedure parameters that follow.

Compile-time arguments consist of any space-separated strings that you want to pass to the procedure at compile time. When executed with compile-time arguments, Progress automatically recompiles the procedure, substituting each argument for {*n*} in the 4GL, where *n* is an integer specifying the ordinal position of the argument in the RUN statement, starting at 1. Compile-time arguments must follow any run-time parameters in the statement. You can only specify compile-time arguments when running external procedures.

3.3.3 Examples

This is an external procedure that displays the names of all the customers in the sports database, sorted by name.

```
FOR EACH customer BY name: DISPLAY name.
```

NOTE: Any statement or group of statements that you enter in a Procedure Editor buffer become an external procedure when you save or execute the buffer. Also, all the procedure files saved by AppBuilder are external procedures. For information on the

NOTE: Procedure Editor, see the *Progress Basic Database Tools* (Character only) manual, and in graphical interfaces, the on-line help for the Procedure Editor. For information on the AppBuilder, see the *Progress AppBuilder Developer's Guide*.

This is an external procedure that returns, as an output parameter, the name of a customer in the sports database when given a customer number as an input parameter. If the specified customer does not exist, the procedure returns the unknown value (?) as the customer name:

p-exprc1.p

```
DEFINE INPUT PARAMETER pcust-num LIKE customer.cust-num.
DEFINE OUTPUT PARAMETER pname LIKE customer.name INITIAL ?.

FOR EACH customer:
    IF customer.cust-num = pcust-num THEN DO:
        pname = customer.name.
        RETURN.
    END.
END.
```

This external procedure calls `p-exprc1.p` with a user-specified customer number and displays the resulting customer name in a message. Note that `p-exprc2.p` is a top-level, application-driven procedure:

p-exprc2.p

```
DEFINE VARIABLE vcust-num LIKE customer.cust-num INITIAL 0.
DEFINE VARIABLE vname LIKE customer.name.

DO WHILE NOT vcust-num = ?:
    SET vcust-num LABEL "Customer Number" WITH SIDE-LABELS.
    IF NOT vcust-num = ? THEN DO:
        RUN p-exprc1.p (INPUT vcust-num, OUTPUT vname).
        MESSAGE "Customer" vcust-num "is" vname + ".".
    END.
END.
```

For more information on the application- and event-driven models, the two basic techniques for writing Progress applications, see Chapter 2, "Programming Models."

3.3.4 Procedure Parameters

A *procedure parameter* passes a value to a procedure or returns a value from a procedure at run time. To define a parameter for a procedure, you specify the DEFINE PARAMETER statement in the procedure block. This statement can appear anywhere before you first reference the parameter in the procedure. The required information you specify in the DEFINE PARAMETER statement includes:

- The name you use to reference the parameter in the procedure.
- How the parameter is used: for input, output, both input-output, or to pass a buffer.
- The data type of the parameter.

Input parameters pass data to a procedure when it is called, and output parameters return data from the called procedure to the calling procedure when the called procedure completes execution. Buffers are always passed for input-output. In general, the data items passed in the RUN statement must match, in data type and order, the corresponding parameters defined in the called procedure.

In the previous examples, the RUN statement in `p-exprc2.p` passes vcust–num and vname to the parameters pcust–num and pname, respectively in `p-exprc1.p`. For more information on procedure parameters, see the *Progress Language Reference*.

3.3.5 Procedure Context and the Procedure Call Stack

In the previous examples, when `p-exprc2.p` calls `p-exprc1.p`, `p-exprc2.p` waits on the procedure call stack until `p-exprc1.p` completes execution and returns. The *procedure call stack* is an internal memory structure that Progress uses to keep track of procedure context and scope. When one procedure calls another, Progress saves the context of the calling procedure on the call stack. The called procedure then establishes the new current context. When the called procedure returns, Progress re-establishes the context of the calling procedure as the current context.

NOTE: Triggers also execute and interact with the call stack like procedures. When a trigger executes in response to an event, its context becomes the current context. For more information, see Chapter 11, “Database Triggers.”

Figure 3–1 shows how the procedure call stack maintains procedure context and scope in a Progress client. It shows an application, `your-app.p`, running from the Procedure Editor, which is also on the stack. As each procedure calls the next, its context is saved on the stack.

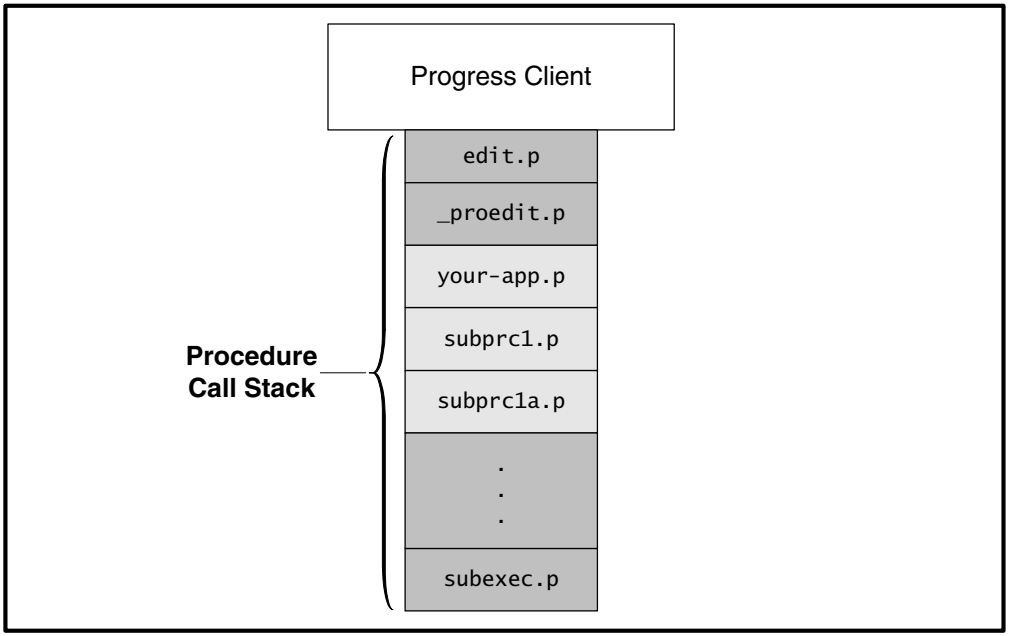


Figure 3–1: Procedure Call Stack

Progress can save any number of procedures on the call stack (that is, you can nest any number of procedure calls), depending on the size of each saved context and the size of the call stack. The size of the context depends on the size and number of definitions created and maintained by the procedure. The size of the call stack depends on the setting of the Nested Blocks (–nb) startup parameter and the Stack Size (–s) startup parameter (in 1K units). The default for –nb is 50 and the default for –s is 40. For more information on –nb and –s, see the *Progress Startup Command and Parameter Reference*.

NOTE: There is a distinction between procedure scope and context. While the procedure context is the actual environment in which a procedure executes, the procedure scope is the duration that its context remains available to the application. For example, in Figure 3–1, while the context of `your-app.p` and `subprc1.p` might be invisible to each other, `subprc1.p` remains within the scope of `your-app.p`.

3.3.6 Shared Objects

In general, without procedure parameters, the context of a called external procedure is invisible to the calling procedure. For example, `p-exprc2.p` (in the previous examples) cannot directly access the customer buffer defined in `p-exprc1.p`. Also for external procedures, the context of the calling procedure is generally invisible to the called procedure. In other words, a procedure's context is generally local to itself. However, external procedures can pass information and resources among them using shared objects (data, streams, or user interface widgets).

Scoped Versus Global

There are two kinds of shared objects: scoped and global. A *scoped* shared object is accessible from the point of creation only for the scope of the procedure that first defines it. A *global* shared object is accessible from the point of creation until the end of the Progress session. Thus a called procedure can reference a scoped shared object that is created by a calling procedure (or any other procedure still on the call stack) prior to the call. **Any** procedure can access a global shared object after it is created.

NOTE: Any scoped shared objects created **or** referenced in the context of a persistent procedure remain available as long as the procedure persists. For more information, see the "Persistent and Non-persistent Procedures" section.

Figure 3–2 shows how you can reference scoped and global shared objects outside of the procedure context in which they are created. (By definition, a procedure can reference any shared object created in its own context.) The Main Procedure starts out by creating scoped shared object A. It subsequently calls Subprocedure 1, which creates global shared object B and scoped shared object C. Subprocedure 1 calls Subprocedure 2 and the Main Procedure subsequently calls Subprocedure 3.

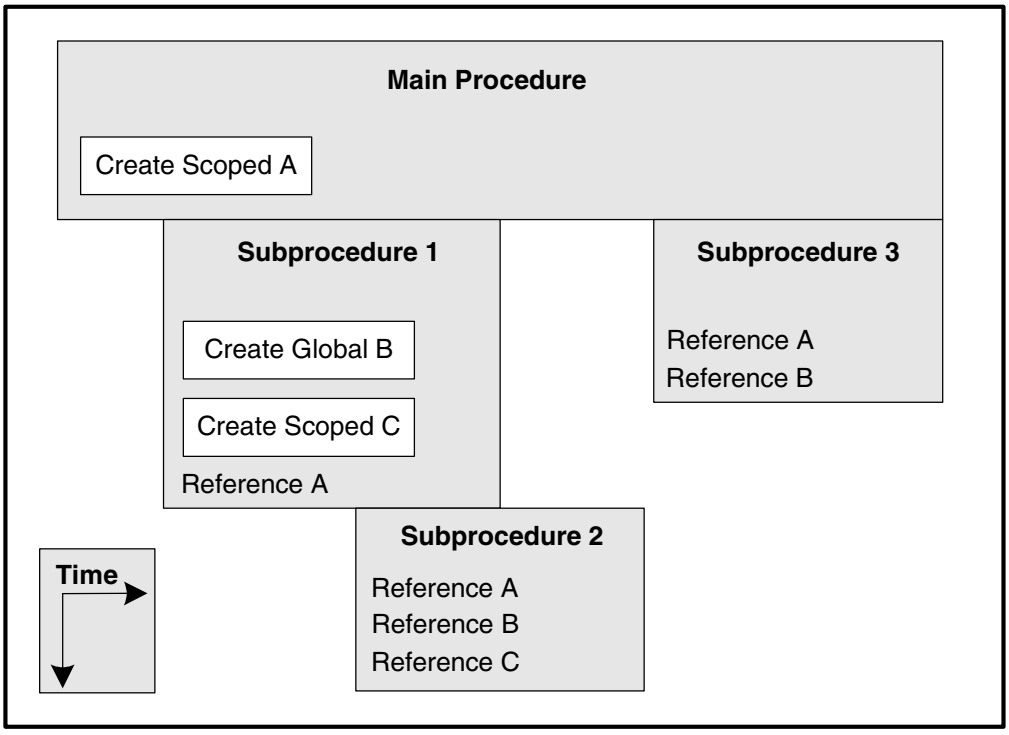


Figure 3–2: Object Sharing Among External Procedures

All the procedures can reference A because it is created in the Main Procedure, before any subprocedures are called. All procedures **except** the Main Procedure can reference B even though it is created in a subprocedure, because B is global. Note that the Main Procedure cannot reference global B, because when the Main Procedure is compiled, B has not yet been created. Subprocedure 2 is the only procedure that can reference all of shared objects A, B, and C because it is called within the scope of scoped objects A and C. Subprocedure 3 can only reference A and B, because C is out of scope.

Thus, a scoped shared object is accessible to any procedure, as long as the initial procedure that creates the object is still in scope. A global shared object is accessible to any procedure as long as the initial procedure that creates the object has executed prior to the first reference to the object, whether or not the initial procedure is still in scope.

Defining and Referencing

Any object defined using the NEW SHARED option in its DEFINE statement creates a scoped shared object. Any object defined using the NEW GLOBAL SHARED option creates a global shared object. Any object defined using the SHARED option (without NEW or GLOBAL) creates a reference to a previously created scoped or global shared object. Currently, the only objects that can be globally shared are variables and streams. For example, the following procedures define and use both scoped and global shared variables:

p-share1.p

```
DEFINE NEW SHARED VARIABLE sharedvar AS CHARACTER INITIAL "Shared".
RUN p-share2.p.
RUN p-share3.p.
```

p-share2.p

```
DEFINE NEW GLOBAL SHARED VARIABLE globvar AS CHARACTER INITIAL "Global".
DEFINE SHARED VARIABLE sharedvar AS CHARACTER.
DISPLAY sharedvar + " in p-share2.p." FORMAT "x(22)" WITH NO-LABELS.
```

p-share3.p

```
DEFINE SHARED VARIABLE sharedvar AS CHARACTER.
DEFINE SHARED VARIABLE globvar AS CHARACTER.
DISPLAY sharedvar + " in p-share3.p." FORMAT "x(22)" SKIP
        globvar + " in p-share3.p." FORMAT "x(22)" WITH NO-LABELS.
```

Procedure `p-share1.p` defines a scoped shared variable (sharedvar) that `p-share2.p` and `p-share3.p` can reference, and `p-share2.p` defines a global variable (globvar) that `p-share3.p` can reference. Note that `p-share1.p` cannot reference globvar because `p-share1.p` compiles and references any variable before `p-share2.p` can define it.

For information on shared buffers, see Chapter 9, “Database Access,” in this manual, and on shared streams, see Chapter 7, “Alternate I/O Sources.” Any other Progress objects that include the NEW and SHARED keywords in their definition can also be shared. For more information, see the DEFINE statement for the objects in the *Progress Language Reference*.

3.3.7 Procedure Handles

A *procedure handle* is a value that identifies the context of an external procedure. Every active external procedure has a procedure handle associated with it. The THIS–PROCEDURE system handle returns the current procedure handle value as a HANDLE data type from within any executing block of an external procedure.

NOTE: A system handle, like a function, returns the handle to a Progress widget, procedure, or other system object. For more information on system handles, see Chapter 16, "Widgets and Handles."

Procedure handles have attributes that you can use to store and retrieve information on the particular procedure. These attributes are especially useful for working with persistent procedures. For information on the attributes of a procedure handle, see the THIS–PROCEDURE System Handle reference entry in the *Progress Language Reference*. For information on referencing attributes, see Chapter 16, "Widgets and Handles," in this manual. For information on using procedure attributes with persistent procedures, see the "Persistent and Non-persistent Procedures" section.

Procedure Triggers

You can also use the THIS–PROCEDURE handle to define and execute local procedure triggers and internal procedures from anywhere in your application. A *procedure trigger* is a block of code defined with the ON statement that executes in response to an event applied to the specified procedure handle. For more information on internal procedures and how to execute them using procedure handles, see the "Internal Procedures" section.

This is the basic syntax to define a procedure trigger:

SYNTAX

```
ON event-list OF procedure-handle block
```

The *event–list* can be any comma-separated list of Progress event names you choose, from keyboard to high-level widget events. For lists of keyboard events see Chapter 6, "Handling User Input." For lists of all other events, see the *Progress Language Reference*. The *procedure–handle* can specify any valid procedure handle value, but you generally use the THIS–PROCEDURE system handle for the current procedure. The *block* can be a single statement or a DO block.

NOTE: Progress processes events for procedure triggers with no interpretation. Although you must use a Progress-supplied event name to specify a procedure event, the name is used only to identify the trigger *block* for execution.

You can store and pass the *procedure–handle* to other external procedures using parameters, shared variables, and shared temporary table fields defined with the HANDLE data type. As long as the specified external procedure remains in scope, other procedures can use this handle to execute procedure triggers and internal procedures defined in the specified procedure.

NOTE: A procedure handle value remains valid only as long as the specified procedure is in scope. Once the procedure goes out of scope, the specified value is invalid and can be reused. You can use the VALID–HANDLE function and TYPE procedure handle attribute to test whether a procedure handle value is valid. The TYPE attribute must return "PROCEDURE" to filter out any re-use as a widget handle.

Applying Events to Procedure Triggers

You can execute a procedure trigger anywhere in an application using the APPLY statement with this syntax:

SYNTAX

```
APPLY event TO procedure-handle
```

The *event* can be a character expression that evaluates to any event name specified in the *event–list* of a trigger for *procedure–handle*. The *procedure–handle* must be a valid procedure handle value for which the specified trigger has been defined.

This example is an external procedure that defines and executes its own procedure trigger, displaying the message "Hi!". The phandle variable is for illustration. You can also use THIS–PROCEDURE directly in the ON and APPLY statements:

```
DEFINE VARIABLE phandle AS HANDLE.
phandle = THIS-PROCEDURE.
ON RETURN OF phandle MESSAGE "Hi!".
APPLY "RETURN" TO phandle.
```

Your decision whether to use procedure triggers or internal procedures to execute code in an external procedure context is design dependent. In general, internal procedures provide the greatest flexibility, because you can reference them by any name you define and you can pass them run-time parameters. Internal procedures can also map directly to Windows dynamic link library (DLL) functions.

3.4 Internal Procedures

An *internal procedure* is a block of code defined within an external procedure that you can execute by name within the context of the containing external procedure. Internal procedures allow you to modularize your code and reduce the number of external files in an application. A single external procedure (.p or .w) can contain any number of internal procedures.

NOTE: There are actually two types of internal procedure: 4GL and DLL internal procedures. This section applies mainly to 4GL internal procedures. DLL internal procedures provide access to DLL functions on Windows. While DLL internal procedures execute different code than 4GL internal procedures, you call them by name in the same way as any 4GL internal procedure. For more information on defining and using DLL internal procedures, see the *Progress External Program Interfaces* manual.

3.4.1 Internal Procedure Definition

This is syntax to define an internal 4GL procedure block:

SYNTAX

```
PROCEDURE name :
   block
END [ PROCEDURE ] .
```

You can terminate the PROCEDURE statement with a period or a colon. The maximum length of an internal procedure *name* is context dependent. In general, it is not limited by operating system or the 32 character maximum for variable names. For more information on internal procedure name length and composition, see the *Progress Client Deployment Guide*.

The *block* can contain one or more of any Progress statement except another PROCEDURE statement. The *block* can also be empty.

NOTE: You cannot define an internal procedure within another. Only an external procedure can contain internal procedures.

3.4.2 Internal Procedure Execution

You execute an internal procedure using the RUN statement with this syntax:

SYNTAX

```
RUN proc-name [ IN handle ] [ run-options ]
```

You can invoke the RUN statement for an internal procedure at the following points in an application:

- From the containing external procedure.
- From any internal procedure (including itself) in the containing external procedure.
- From any other external procedure using the IN *handle* option, where *handle* specifies a containing external procedure that is in scope. The IN *handle* option is described later in this section.

3.4.3 Internal Procedure Names

You can specify *proc–name* using any of these forms:

- The unquoted name of the internal procedure as specified in the PROCEDURE statement.
- VALUE(*char–expression*), where *char–expression* is any character expression equal to the name of the internal procedure as specified in the PROCEDURE statement.

In general, when Progress encounters a RUN statement, it first searches the available internal procedures for *proc–name*. If it cannot find the specified internal procedure and the RUN statement does not use the IN option, Progress tries to find an external procedure in either r-code or source code form that matches *proc–name*. For more information on external procedures and r-code, see the "External Procedures" section.

3.4.4 Standard Execution Options

The *run–options* can include the same run-time parameters and NO–ERROR option, as in external procedures. However, you cannot pass compile-time arguments {*n*} to an internal procedure because internal procedures are always compiled as part of the containing external procedure. Thus, you must specify any compile-time arguments used in an internal procedure definition when you call the containing external procedure that defines it. For more information on compile-time arguments, see the "External Procedures" section.

3.4.5 Internal Procedures and Encapsulation

The IN *handle* option allows you to execute an internal procedure that is defined in any external procedure context. The *handle* can be any field, variable, parameter, system handle, or attribute expression that returns a valid procedure handle. This capability promotes code re-use and encapsulation. *Encapsulation* is a technique of modular design where you hide (encapsulate) related data and functionality from other external procedures and provide all access to the hidden data through internal procedures defined by the hidden context. It also allows you to break up a larger external procedure into several smaller ones, especially if you encounter r-code segment limits (see Appendix A, "R-code Features and Functions"). For more information on procedure handles, see the "External Procedures" section.

Valid Procedure Handles

To be valid, the procedure handle must identify one of the following external procedures that also defines the internal procedure you want to execute:

- The currently executing procedure or any procedure waiting on the call stack.
- A persistent procedure.

In short, the procedure handle must identify the context of a procedure that is still in scope when you execute the internal procedure. For more information on procedure context, procedure scope, and the procedure call stack, see the "External Procedures" section.

NOTE: You can use the VALID–HANDLE function and the TYPE procedure handle attribute to test whether a procedure handle identifies an active procedure context. VALID–HANDLE returns TRUE if the specified procedure context is in scope. The TYPE attribute must also return "PROCEDURE" to filter out valid widget handles.

For information on persistent procedures and how to obtain procedure handles for them, see the "Persistent and Non-persistent Procedures" section. Otherwise, your procedure handle must be set to the THIS–PROCEDURE system handle within the external procedure that contains the internal procedure you want to execute.

Condition Handling

Internal procedures executed with the IN *handle* option raise ERROR on the RUN statement for the following conditions:

- The *handle* value is invalid.
- The specified internal procedure does not exist in the procedure specified by *handle*.
- Run-time parameters do not match the internal procedure definition.

You can trap these ERROR conditions using the NO–ERROR option of the RUN statement.

3.4.6 Shared and Local Context

Internal procedures are part of the context of the containing external procedure. An internal procedure can access anything in the containing procedure's context, including shared objects and run-time parameters passed to the containing procedure. Also, like external procedures, the local context of an internal procedure is generally invisible to the procedure that calls it. In other words, while the containing procedure's context is visible to the internal procedure's context, the local context of the internal procedure is invisible to the containing procedure.

While an internal procedure can access shared objects defined in the containing procedure, you cannot introduce any shared object definitions in the internal procedure itself. For example, the statement DEFINE NEW SHARED FRAME results in a compiler error.

In addition, you cannot define local or shared streams, temporary tables, or work tables in an internal procedure. However, you can define local variables, queries, user-interface widgets, and buffers within an internal procedure.

Progress scopes buffers defined in an internal procedure to that procedure. When the procedure ends, you can no longer access those buffers. An internal procedure can also reference buffers from a containing procedure.

Progress follows these steps to resolve buffer references:

1 ♦ Progress searches for a defined buffer that satisfies the reference, looking first in the local procedure. If it does not find the defined buffer, it searches for it in the containing procedure.

2 ♦ If Progress cannot find an explicitly defined buffer, it looks for other references to the same implicit buffer. Progress looks first in the local procedure. If it does not find an implicit reference, it searches for an implicit reference in the containing procedure.

3 ♦ If Progress cannot find an existing implicit reference to the same buffer, it tries to create a new implicit buffer. Progress scopes the new buffer to the procedure block of the containing procedure file.

An internal procedure has its own frame list and can reference by name any frame already named in the containing procedure. Progress resolves references to named frames by first looking in the internal procedure, then looking through the frames named in the containing procedure. If it finds none, the internal procedure gets a new local frame. A frame scoped to the containing procedure and used in the internal procedure is a *borrowed* frame. The internal procedure can add fields to a borrowed frame but cannot use a local program variable in any of the frame's expressions. This is because Progress might evaluate a borrowed frame's expressions outside the scope of the internal procedure.

3.4.7 Examples

The following external procedure defines and calls three internal procedures. The external procedure `p-intprc.p` runs the internal procedure getcust for each customer number you enter. The internal procedure getcust runs getord and getord runs getitem. The result is that you see the order–line and item information if the customer exists and has orders:

p-intprc.p

```
DEFINE BUFFER cust_buf FOR customer.
DEFINE BUFFER ord_buf FOR order.

DEFINE VARIABLE t-i-cust-num LIKE customer.cust-num.
DEFINE VARIABLE t-i-item-num LIKE item.item-num.

REPEAT:
    UPDATE t-i-cust-num.
    RUN getcust.
END.

PROCEDURE getcust:
   FIND FIRST cust_buf WHERE cust_buf.cust-num = t-i-cust-num NO-ERROR.
   IF NOT AVAILABLE cust_buf
   THEN DO:
      MESSAGE "Customer not found.".
      RETURN ERROR.
   END.
   ELSE DO:
      FIND FIRST ord_buf OF cust_buf NO-ERROR.
      IF AVAILABLE ord_buf
      THEN RUN getord.
      ELSE MESSAGE "No orders for customer.".
   END.
END PROCEDURE. /* getcust */

PROCEDURE getord:
   FOR EACH order-line OF ord_buf:
      DISPLAY order-line WITH FRAME foo.
      t-i-item-num = order-line.item-num.
      RUN getitem.
   END.
END PROCEDURE. /* getord */

PROCEDURE getitem:
   FIND FIRST item WHERE item.item-num = t-i-item-num.
   DISPLAY cust_buf.cust-num cust_buf.name item.item-num item.item-name
      WITH FRAME foo1.
END PROCEDURE. /* getitem */
```

The next two procedures duplicate the functionality of p-intprc.p by separating the internal procedures and the update loop into separate external procedures. The procedure defining the internal procedures, p-inprc1.p passes its procedure handle to p-exprc3.p. Thus, p-exprc3.p runs getcust in p-inprc1.p:

p-inprc1.p

```
DEFINE BUFFER cust_buf FOR customer.
DEFINE BUFFER ord_buf FOR order.
DEFINE VARIABLE t-i-item-num LIKE item.item-num.

RUN p-exprc3.p (INPUT THIS-PROCEDURE).

PROCEDURE getcust:
   DEFINE INPUT PARAMETER t-i-cust-num LIKE customer.cust-num.
   FIND FIRST cust_buf WHERE cust_buf.cust-num = t-i-cust-num NO-ERROR.
   IF NOT AVAILABLE cust_buf
   THEN DO:
      MESSAGE "Customer not found.".
      RETURN ERROR.
   END.
   ELSE DO:
      FIND FIRST ord_buf OF cust_buf NO-ERROR.
      IF AVAILABLE ord_buf
      THEN RUN getord.
      ELSE MESSAGE "No orders for customer.".
   END.
END PROCEDURE. /* getcust */

PROCEDURE getord:
   FOR EACH order-line OF ord_buf:
      DISPLAY order-line WITH FRAME foo.
      t-i-item-num = order-line.item-num.
      RUN getitem.
   END.
END PROCEDURE. /* getord */

PROCEDURE getitem:
   FIND FIRST item WHERE item.item-num = t-i-item-num.
   DISPLAY cust_buf.cust-num cust_buf.name item.item-num item.item-name
      WITH FRAME foo1.
END PROCEDURE. /* getitem */
```

p-exprc3.p

```
DEFINE INPUT PARAMETER db-procs AS HANDLE.
DEFINE VARIABLE t-i-cust-num LIKE customer.cust-num.

REPEAT:
   UPDATE t-i-cust-num.
   RUN getcust IN db-procs (INPUT t-i-cust-num).
END.
```

3.4.8 Internal Procedures and the Call Stack

Note that, like external procedures, Progress moves internal procedures on and off the procedure call stack as they call other procedures. For more information on the procedure call stack, see the "External Procedures" section.

3.5 Persistent and Non-persistent Procedures

An external procedure can be either non-persistent or persistent. A *non-persistent procedure* creates and maintains its context only until it returns from execution. In other words, the context of a non-persistent procedure remains in scope only until the RUN statement that executes it completes. An external procedure is non-persistent by default.

A *persistent procedure* creates its context when it executes and then maintains that context after it returns until the end of the Progress session, or until it is explicitly deleted. In other words, the context of a persistent procedure remains in scope after the RUN statement that executes it completes until you remove it. Thus, as long as its context is in scope, the triggers and internal procedures of a persistent procedure remain available for execution by your application. An external procedure creates a persistent context when you execute it using a RUN statement with the PERSISTENT option.

NOTE: If you run an application that creates persistent procedures from an ADE tool, that tool (for example, the Procedure Editor or User Interface Builder) removes all instances of persistent procedures still created when the application terminates.

3.5.1 Advantages of Persistent Procedures

Persistent procedures promote modular application design and development by more easily allowing you to distribute functionality among several procedures. For example, you might build a persistent procedure that provides access to a database through a set of local buffers that it otherwise hides from the rest of the application. Or your persistent procedure might create and manage its own windows while allowing independent access (non-modal access) to other windows in your application. Aside from helping to manage functionality, the additional modularity also helps to avoid hitting r-code segment limits.

Thus, creating a persistent procedure whose context you access through internal procedures provides the most effective means to achieve encapsulation in Progress. For more information on using internal procedures to promote encapsulation, see the "Internal Procedures" section.

You can also use a persistent procedure to create multiple versions of the same context. Each time you call a persistent procedure, it creates a separate instance of its context. Your application (and user) can access and manage each context independently from the others. If your persistent procedure manages its own windows, you can use this feature to provide some (but not all) of the capabilities of the Microsoft multiple document interface (MDI).

For more information on using persistent procedures to manage multiple windows, see Chapter 21, "Windows."

3.5.2 Creating Persistent Procedures

To make an external procedure persistent, execute it using the PERSISTENT option of the RUN statement. This is an *instantiating RUN statement*, because it both executes the external procedure and creates a persistent instance of its context:

SYNTAX

```
RUN filename PERSISTENT [ SET handle ] [ run-options ]
```

The *filename* and *run-options* are the same as for any external procedure (see the "External Procedures" section).

The *handle* is a field or variable defined with the HANDLE data type. The RUN statement sets *handle* to a procedure handle value that identifies the current context of the persistent procedure. You can use this procedure handle to invoke procedure triggers or internal procedures defined in the persistent procedure context. For information on procedure handles, see the "External Procedures" section.

For example, this code fragment creates two persistent procedures from `db-tools.p` and `dde-tools.p`, and assigns the procedure handles to hdb–tools and hdde–tools, respectively:

```
DEFINE VARIABLE hdb-tools AS HANDLE.
DEFINE VARIABLE hdde-tools AS HANDLE.
DEFINE VARIABLE ldone AS LOGICAL INITIAL FALSE.
DEFINE BUFFER cust-buf FOR customer.

RUN db-tools.p PERSISTENT SET hdb-tools.
RUN dde-tools.p PERSISTENT SET hdde-tools (INPUT "Customer Activity").

RUN get-first-customer IN hdb-tools (BUFFER cust-buf).

DO WHILE NOT ldone:
    RUN add-to-spread-sheet IN hdde-tools (BUFFER cust-buf).
    RUN get-next-customer IN hdb-tools (BUFFER cust-buf, OUTPUT ldone).
END.
```

Assume that these two procedures interact with the sports database (`db-tools.p`) and a spreadsheet application using Windows Dynamic Data Exchange (DDE) (`dde-tools.p`). The fragment then runs the internal procedures within these persistent procedures to add customer information to a spreadsheet named Customer Activity. (For more information on using DDE, see the *Progress External Program Interfaces* manual.)

This application fragment shows how you can use persistent procedures to set up modular environments that encapsulate processing details from the rest of an application.

3.5.3 Working with Persistent Procedures

If you do not specify *handle* when you create a persistent procedure, you can obtain the handle for the procedure using the SESSION system handle. Progress maintains all handles to persistent procedure instances in a chain that you can access. The SESSION handle has two attributes, FIRST–PROCEDURE and LAST–PROCEDURE, that provide access to the first and last persistent procedures (respectively) that are created. These attributes return procedure handles that you can use to reference attributes of each persistent procedure context. For a complete list of these attributes, see the THIS–PROCEDURE System Handle reference entry in the *Progress Language Reference*.

For example, using the NEXT–SIBLING attribute with the FIRST–PROCEDURE handle value or the PREV–SIBLING attribute with the LAST–PROCEDURE handle value, you can follow the entire chain of persistent procedure handles to locate the one you want. Note that PREV–SIBLING of FIRST–PROCEDURE and NEXT–SIBLING of LAST–PROCEDURE specify invalid handles to terminate the chain. You can check the validity of any procedure handle value in the chain using the VALID–HANDLE function. It returns TRUE for a valid value and FALSE for an invalid value.

The PRIVATE–DATA attribute is especially helpful in identifying a particular persistent procedure in the chain. When initially created with the RUN statement, a persistent procedure can set this attribute to a unique character value that helps identify its context to other procedures. You can use shared data to help manage the values for this private data. Note that the FILE–NAME attribute might not be sufficient to identify a specific persistent procedure instance, because you can create multiple instances of the same persistent procedure.

Other attributes of procedure handles that help manage persistent procedures include the PERSISTENT, CURRENT–WINDOW, and INTERNAL–ENTRIES attributes. When used with the THIS–PROCEDURE handle, the PERSISTENT attribute returns TRUE if the current procedure is being run persistently. This is useful if you want to write a procedure to run either persistently or non-persistently, providing different code options for each mode of execution.

The CURRENT–WINDOW attribute allows you to assign a unique current window to a procedure context that never changes as long as the procedure remains in scope or until you reset it. This is especially useful for persistent procedures that maintain their own windows. For more information, see Chapter 21, “Windows.”

The INTERNAL–ENTRIES attribute returns a comma-separated list of all the internal procedures defined within the specified external procedure. This is helpful to identify the functional capabilities (internal procedures) provided to your application by each type of persistent procedure that you create. The GET–SIGNATURE method returns the type and parameters of a specified internal procedure.

For example, this code fragment uses several attributes to select persistent procedures that participate in the application user interface, and makes their current windows visible:

```
DEFINE VARIABLE hcurrent AS HANDLE.
DEFINE VARIABLE hnext AS HANDLE.
DEFINE VARIABLE hwindow AS WIDGET-HANDLE.

hcurrent = SESSION:FIRST-PROCEDURE.
DO WHILE VALID-HANDLE(hcurrent):
    IF hcurrent:PRIVATE-DATA = "USER INTERFACE" THEN DO:
        hwindow = hcurrent:CURRENT-WINDOW.
        hwindow:VISIBLE = TRUE.
    END.
    hcurrent = hcurrent:NEXT-SIBLING.
END.
```

For more information on attribute references, see Chapter 16, “Widgets and Handles.”

3.5.4 Deleting Persistent Procedures

You can delete an instance of a persistent procedure using the DELETE PROCEDURE statement:

SYNTAX

```
DELETE PROCEDURE handle [ NO-ERROR ]
```

This statement removes the procedure context specified by *handle*, a procedure handle to the context of a persistent procedure. If *handle* identifies a non-persistent procedure context or is otherwise invalid, the statement returns the ERROR condition. The NO–ERROR option allows you to ignore or trap statement ERROR conditions as you want.

When this statement completes execution, all resources defined in the specified procedure context are returned to the system and the procedure handle value ceases to be valid. The procedure handle is also removed from the session chain of persistent procedures. Any local buffers associated with the persistent context are disconnected and their data validated, possibly firing write triggers. If the validation fails, the DELETE PROCEDURE statement returns the ERROR condition and pends the deletion until all buffer validations succeed and any write triggers fire.

Note that if you execute the DELETE PROCEDURE statement in the context of the procedure you are deleting, the delete pends until execution leaves the persistent context and returns to the context of the calling external procedure. This happens when you execute the statement in a trigger or internal procedure that is defined within the persistent procedure.

3.5.5 Examples

These examples illustrate several features of persistent procedures. The application returns a customer name from the sports database when you enter a customer number. The main event-driven procedure, `p-persp1.p`, creates a persistent procedure from `p-persp2.p` and stores its handle in hdb-tools:

p-persp1.p

```
DEFINE VARIABLE hdb-tools AS HANDLE.
DEFINE VARIABLE out-message AS CHARACTER FORMAT "x(60)".
DEFINE VARIABLE vcust-num LIKE customer.cust-num INITIAL 0.
DEFINE VARIABLE vname LIKE customer.name.

DEFINE BUTTON bcancel LABEL "Cancel Database Access".

DEFINE FRAME CustFrame
    vcust-num LABEL "Customer Number"
    vname LABEL "Customer Name"
    bcancel
WITH SIDE-LABELS.

ON RETURN OF vcust-num IN FRAME CustFrame DO:
    ASSIGN vcust-num.
    RUN get-cust-name IN hdb-tools (INPUT vcust-num, OUTPUT vname).
    DISPLAY vname WITH FRAME CustFrame.
END.

ON CHOOSE OF bcancel IN FRAME CustFrame DO:
    DEFINE VARIABLE hcurrent AS HANDLE.
    DEFINE VARIABLE hnext AS HANDLE.

        hcurrent = SESSION:FIRST-PROCEDURE.
        DO WHILE VALID-HANDLE(hcurrent):
        hnext = hcurrent:NEXT-SIBLING.
        IF hcurrent:PRIVATE-DATA = "DATABASE" THEN DO:
            RUN destroy-context IN hcurrent (OUTPUT out-message).
            MESSAGE out-message.
        END.
        hcurrent = hnext.
    END.
    APPLY "WINDOW-CLOSE" TO CURRENT-WINDOW.
END.

RUN p-persp2.p PERSISTENT SET hdb-tools.

ENABLE ALL WITH FRAME CustFrame.
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

p-persp2.p

```
THIS-PROCEDURE:PRIVATE-DATA = "DATABASE".

PROCEDURE get-cust-name:
    DEFINE INPUT PARAMETER pcust-num LIKE customer.cust-num.
    DEFINE OUTPUT PARAMETER pname LIKE customer.name.

    FIND customer WHERE customer.cust-num = pcust-num NO-ERROR.
    IF NOT AVAILABLE(customer) THEN
        pname = ?.
    ELSE
        pname = customer.name.
END PROCEDURE.

PROCEDURE destroy-context:
    DEFINE OUTPUT PARAMETER out-message AS CHARACTER FORMAT "x(60)".
    DELETE PROCEDURE THIS-PROCEDURE.
    out-message = THIS-PROCEDURE:PRIVATE-DATA + " module deleted.".
END PROCEDURE.
```

This persistent procedure provides two internal procedures to the application, get–cust–name and destroy–context. The get–cust–name procedure performs the database access for the application, and the destroy–context procedure deletes the context of the persistent procedure from the application.

Note that while `p-persp1.p` controls the application, `p-persp2.p` provides all the code to manage itself, in this case to delete its own context. If you write an application with many persistent procedures, each one can provide a common set of internal procedures with the same name and interface, such as delete–context. Thus, the delete–context in one persistent procedure might operate differently than that in another, but they all can be called in the same context of the controlling procedure.

In fact, `p-persp1.p` illustrates this generic procedure capability when you choose the bcancel button, by looping through the persistent procedure chain to delete the `p-persp2.p` context. You can create any number of different persistent procedures with the PRIVATE–DATA attribute set to "DATABASE," and with many additional variations on the theme.

NOTE: Persistent procedures that manage their own user interface often create and manage their own widget pools. For more information, see Chapter 21, “Windows.”

3.5.6 Persistent Procedure Context and Shared Objects

While the context of a persistent procedure remains available to your application, there are certain capabilities that persistent procedures do not allow:

- **Static dialog boxes** — You cannot maintain static dialog boxes in a persistent procedure. Any static dialog boxes that a persistent procedure defines go out of scope when the instantiating RUN statement returns to the calling external procedure. For more information, see Chapter 21, "Windows."
- **Shared streams or frame, browse, and menu widgets** — You cannot create or access shared streams or shared frame, browse, or menu widgets in a persistent procedure. If you do, the instantiating RUN statement does not create the procedure context and returns with the ERROR condition.
- **DISABLE TRIGGERS Statement** — You cannot execute the DISABLE TRIGGERS statement anywhere while executing code in a persistent procedure. Otherwise, Progress raises the ERROR condition at that point.

All other widgets and shared objects are allowed in a persistent procedure.

Scoped Object Sharing

In persistent procedures, scoped object sharing works differently than in non-persistent procedures. When a persistent procedure references a shared object, the scoped shared object stays in scope as long as the persistent procedure remains in scope. In other words, a shared object can go out of scope only when no persistent procedure remains that references it. For more information on scoped shared objects, see the "External Procedures" section.

Persistent Shared Objects and the Call Stack

Note that after a persistent procedure is created, it no longer appears on the procedure call stack. Only its triggers and internal procedures go on the call stack when they execute other procedures.

Normally Progress resolves scoped shared objects by looking for them on the call stack. However, if the trigger or internal procedure of a persistent procedure calls an external procedure that references scoped shared objects, Progress resolves these objects by looking for them in the context of the persistent procedure **as well as** on the call stack. For more information on the procedure call stack, see the "External Procedures" section.

3.5.7 Run-time Parameters

Any persistent procedure created using a RUN statement invoked with INPUT, OUTPUT, or INPUT–OUTPUT run-time parameters continues to have access to these parameters after the RUN statement completes execution. The output values are returned to the caller at the time that the procedure returns from its instantiation. However the parameters remain available to the persistent procedure context just like local variables.

Buffer parameters work differently in persistent procedures. During execution of the instantiating RUN statement, a buffer is passed by reference and any new value returned, just as for a non-persistent procedure. However, after the procedure returns from creating its persistent context, Progress creates a copy of the buffer parameter and resets its cursors as an initially defined local buffer for use in the persistent context. Thus, to give the persistent procedure context continued access to a buffer from the calling (instantiating) context, both the calling and persistent context must define a common shared buffer.

NOTE: Run-time parameters passed to the internal procedures of a persistent procedure function no differently than if passed to any other internal or non-persistent procedure. For more information, see the "External Procedures" section.

3.5.8 Transaction Management

Transactions behave for persistent procedures in exactly the same way as for non-persistent procedures. If a transaction (or subtransaction) begins during execution of an instantiating RUN statement, the transaction is **not** held open pending deletion of the procedure context. Thus, the transaction is accepted or rejected by the time the instantiating procedure returns to its caller.

The same is true for any trigger or internal procedure defined in the persistent procedure context that opens a transaction or subtransaction. If you want to undo work done by a trigger or internal procedure defined by a persistent procedure, you must enclose it within a transaction bound entirely by the context of the trigger or internal procedure block. Otherwise, the transaction completes when the trigger or internal procedure returns and cannot be undone. Thus, persistent procedures encourage the use of atomic transactions no larger than the context of an encapsulated trigger or internal procedure.

For more information on transaction management, see Chapter 12, "Transactions."

UNDO Processing

Any database work subject to UNDO that also creates a persistent procedure can be undone. However, the persistent procedure context remains persistent. In other words, UNDO does not delete any persistent procedures. Otherwise, all records that can be disconnected for UNDO during execution of a non-persistent procedure, can also be disconnected during instantiation of a persistent procedure.

Schema Changes and Schema Locks

If your application makes a change to the schema of a database, and another procedure tries to access a table in that database, Progress automatically restarts the application and any persistent procedures created during the session are deleted.

If a persistent procedure accesses a database table during its instantiating execution, the schema share lock that it obtains remains in effect until the procedure is deleted.

3.5.9 Condition Management

In general, if a STOP or QUIT condition or a RETURN ERROR occurs on an instantiating RUN statement, the executed procedure returns unscoped as a non-persistent procedure and no context is created for it. For more information on condition management, see Chapter 5, "Condition Handling and Messages."

STOP and QUIT Conditions with Persistent Procedures

If a STOP or QUIT condition is not handled by a corresponding ON STOP or ON QUIT phrase, Progress performs the standard STOP or QUIT action and deletes all persistent procedure instances created in the session. Also, any system error such as a hard **CTRL–C**, that raises an untrappable STOP condition, initiates the same STOP processing.

Database Disconnection

Normally, when a database disconnects due to a network error or execution of the PROSHUT utility, Progress raises an untrappable STOP condition and unwinds the procedure call stack to the point above which no procedures reference the disconnected database. At that point Progress converts the untrappable STOP condition to a trappable STOP condition. Before unwinding the call stack, Progress also deletes any persistent procedures that reference the disconnected database.

3.5.10 CURRENT–LANGUAGE Changes

If the CURRENT–LANGUAGE variable changes, all persistent procedure instances continue to function using the text segment with which they were created. CURRENT–LANGUAGE changes affect only inactive external procedures (external procedures that are neither persistent nor active on the procedure call stack). For more information on languages and translation, see the *Progress Internationalization Guide* manual.

3.6 User-defined Functions

Progress 4GL user-defined functions let your application define a logical rule or transformation once, then apply the rule or transformation with absolute consistency an unlimited number of times. For example, if you are developing a weather application that converts temperatures between Celsius (C) and Fahrenheit (F), you can write two user-defined functions, one for converting from C to F and one for converting from F to C. Then, whenever your application must convert a temperature in either direction, you merely reference one of the user-defined functions.

The definition of a user-defined function can reside in the procedure that references it, in a procedure external to the procedure that references it, or in a procedure remote to the procedure that references it. In other words, a user-defined function can be defined *locally*, *externally*, or *remotely*.

User-defined functions follow the rules of scope and visibility that internal procedures follow. For more information, see the section "Shared and Local Context."

Some restrictions apply to user-defined functions:

- You cannot define a work table, a temporary table or any shared variable, object or stream within a user-defined function.
- A user-defined function cannot contain any input-blocking statements, such as UPDATE or PROMPT–FOR, where the program waits for events.

The examples in this section use a user-defined function that accepts a number of type INTEGER, multiplies the number by two, and returns the result. The examples reside on-line in `%DLC%\src\prodoc\langref` on Windows and in `$DLC/src/prodoc/langref` on UNIX.

This section covers the following topics:

- Locally-defined user-defined functions
  - Defining them
  - Referencing them
  - Referencing them before defining them
- Externally-defined user-defined functions
- Remotely-defined user-defined functions

3.6.1 Locally Defined User-defined Functions

The following procedure defines doubler(), then references it three times:

p-udf1.p

```
/* Demonstrates defining and referencing a user-defined function */

/* Define doubler() */
FUNCTION doubler RETURNS INTEGER (INPUT parm1 AS INTEGER).
    RETURN (2 * parm1).
END FUNCTION.

/* Reference doubler() */
DISPLAY  "doubler(0)=" doubler(0) skip
  "doubler(1)=" doubler(1) skip
  "doubler(2)=" doubler(2) skip.
```

Defining User-defined Functions

The FUNCTION statement defines doubler(), a user-defined function that accepts an integer, multiplies it by two, and returns the result.

The FUNCTION statement specifies the name of the function (doubler), the data type the function returns (INTEGER), and the following information on the single parameter: its mode (INPUT), its name within the definition (parm1), and its data type (INTEGER).

The function's logic resides in the RETURN statement—specifically, in the expression that follows the RETURN keyword. The logic is so simple (taking a number and multiplying it by two) that it does not need separate statements. If the logic were more complex and needed separate statements, they would reside between the FUNCTION statement and the RETURN statement.

The END FUNCTION statement ends the block that the FUNCTION statement begins. If you omit END FUNCTION and run the procedure, nothing appears, because Progress includes the DISPLAY statement within the user-defined function, which is probably not what you want. Progress Software Corporation recommends that when you define a user-defined function, you conclude the block with the END FUNCTION statement.

Referencing User-defined Functions

The DISPLAY statement references doubler() three times:

1. Passing doubler() the literal 0
2. Passing doubler() the literal 1
3. Passing doubler() the literal 2

The previous example defines the user-defined function by using the FUNCTION statement. Here is its syntax:

SYNTAX

```
FUNCTION function-name [ RETURNS ] data-type
  [ ( param [ , param ] ... ) ]
  { FORWARD | [ MAP [ TO ] actual-name ] IN proc-handle }
```

For *function–name*, substitute the name of the function. For *data–type*, substitute the data type the function returns. For *param*, which stands for parameter, substitute the following syntax:

SYNTAX

```
{
  {   [ INPUT ] | OUTPUT | INPUT-OUTPUT }
      param-name AS data-type
    |   { [ INPUT ] | OUTPUT | INPUT-OUTPUT }
        TABLE FOR temp-table-name [ APPEND ]
    |  BUFFER buffer-name FOR database-table-name
}
```

The preceding syntax shows three ways of specifying a parameter, which correspond to the three types of parameter that user-defined functions allow: *scalar* (elementary data item), temporary table, and database table buffer. In the top third of the syntax diagram, which describes scalar parameters, note how each parameter has a mode (INPUT, OUTPUT, or INPUT–OUTPUT), a name, and a data type.

For more information on the FUNCTION statement, see its reference entry in the *Progress Language Reference*.

Referencing User-defined Functions Before Defining Them

Progress lets a procedure reference a user-defined function before defining it, as long as the procedure *declares* (preannounces) the user-defined function first. In other words, a procedure can declare, reference, and define a user-defined function, in that order. The declaration describes the user-defined function just enough to enable the Progress compiler, which reads procedure files from top to bottom, to resolve references to it.

NOTE: These declarations are sometimes called *forward declarations*.

To declare a user-defined function, code a FUNCTION statement that includes:

- The name of the function
- The data type that the function returns
- For each parameter, the mode and data type
- The FORWARD keyword

Here are some additional notes on declaring user-defined functions:

- A function declaration does not need the END FUNCTION statement.
- If you declare a function, when you subsequently define the function, you need specify only the function name, the return type, and the logic.

The following procedure declares, references, and defines the user-defined function doubler(). Note that the references occur before the definition:

p-udf2.p

```
/* Forward-declares, references, and defines a user-defined function */

/* Forward declare doubler() */
FUNCTION doubler RETURNS INTEGER (INPUT parm1 AS INTEGER) FORWARD.

/* Reference doubler() */DISPLAY "doubler(0)=" doubler(0).
DISPLAY "doubler(1)=" doubler(1).
DISPLAY "doubler(2)=" doubler(2).

/* Define doubler() */FUNCTION doubler RETURNS INTEGER.
    RETURN (2 * parm1).
END FUNCTION.
```

3.6.2 Externally Defined User-defined Functions

Progress lets procedures reference user-defined functions whose definitions reside in external procedures. The following example, which consists of two procedures, illustrates this.

The first procedure, p-udfdef.p, defines doubler():

p-udfdef.p

```
/* Defines user-defined function doubler() */

FUNCTION doubler RETURNS INTEGER (INPUT parm1 AS INTEGER).
  RETURN (2 * parm1).
END FUNCTION.
```

The second procedure, p-udf3.p, runs the first procedure persistently, declares doubler(), references it, and deletes the persistent procedure—in that order:

p-udf3.p

```
/* references an externally-defined user-defined function */

/* define items */
DEFINE VARIABLE myhand AS HANDLE.
DEFINE VARIABLE mystr  AS CHARACTER FORMAT "x(20)".

/* forward declare doubler() */
FUNCTION doubler RETURNS INTEGER (INPUT parm1 AS INTEGER) IN myhand.

/* run the procedure that doubler() */
RUN src\prodoc\langref\p-udfdef.p PERSISTENT SET myhand.

/* reference doubler() */
DISPLAY  "doubler(0)=" doubler(0) skip
        "doubler(1)=" doubler(1) skip
        "doubler(17)=" doubler(17) skip.

/* delete the procedure that defines doubler */
DELETE PROCEDURE(myhand).
```

3.6.3 Remotely Defined User-defined Functions

Progress lets procedures reference user-defined functions whose definitions reside in *remote* procedures. (A remote procedure is a procedure that runs in a Progress AppServer on a remote machine.) Remotely-defined user-defined functions have one restriction: they cannot contain buffer parameters.

NOTE: If a remote procedure defines a user-defined function that contains one or more buffer parameters, the local procedure cannot reference that user-defined function. The remote procedure, however, can, because to the remote procedure, the user-defined function is local.

Remotely-defined user-defined functions are identical to externally-defined user-defined functions, except that the RUN statement uses the ON SERVER option, and that buffer parameters are not allowed.

For an example of a remotely-defined user-defined function, see the example of the externally-defined user-defined function in the previous section, and in the RUN statement, add the ON SERVER option. For an actual example, see *Building Distributed Applications Using the Progress AppServer*.

3.7 Procedure Overriding

Progress supports a mechanism for overriding Progress internal procedures and user-defined functions. This accomplishes the following goals:

- Implementing a basic object-oriented programming principle of great value in designing Progress components that build on standard behavior.
- Replacing the dispatch mechanism, which the ADM of previous versions of Progress implemented in 4GL code. The new mechanism makes overriding procedures faster, cleaner, and completely transparent to callers, who can now invoke potentially multiple layers of application behavior by using an ordinary RUN statement.

Another term for this behavior is *dynamic procedure scoping*. Progress dynamically scopes the name space in which Progress searches for a procedure someone wishes to run, and extends that name space to more than one running procedure instance.

This section covers the following topics:

- Feature summary
- Super procedure prototyping
- Programming example

NOTE: For a complete description of the 4GL elements described in this section, see the *Progress Language Reference*.

3.7.1 Feature Summary

Progress procedure overriding allows the following:

- The programmer can write different versions of an internal procedure, all with the same name, inserting each into a different procedure file. The same applies to user-defined functions.
- The programmer can make a procedure file a *super procedure* of the local procedure file or of the current Progress session by using the ADD–SUPER–PROCEDURE method (and can reverse this action by using the REMOVE–SUPER–PROCEDURE method). When the local procedure file invokes an internal procedure or user-defined function, Progress searches for it using well-defined search rules.

Table 3–3: Search Rules

Step	Location
1	Local procedure (the procedure with the original RUN statement)
2	Super procedures of the local procedure file
3	Current Progress session
4	External procedures

For more information on the search rules, see the ADD–SUPER–PROCEDURE( ) Method reference entry in the *Progress Language Reference*.

- An internal procedure can invoke the super version of itself by using the RUN SUPER statement. A user-defined function can do the same by using the SUPER function.

  NOTE: When a super procedure executes a RUN SUPER statement, you can specify that Progress continue to search the super procedure chain of the local procedure instead of the super procedure chain of the current super procedure. This behavior is controlled by an optional parameter of the ADD–SUPER PROCEDURE method. For more information, see the *Progress Language Reference*.

- The programmer can get a list of the super procedure handles associated with a procedure file or with the current Progress session by using the SUPER–PROCEDURES attribute. For each procedure handle, the program can use the INTERNAL–ENTRIES attribute to determine the internal procedures and user-defined functions the corresponding procedure file contains. For each internal procedure and user-defined function, the program can determine the signature by using the GET–SIGNATURE method.

  NOTE: An internal procedure or user-defined function can exclude itself from the INTERNAL–ENTRIES attribute's list by defining itself using the PRIVATE option. For more information on the PRIVATE option, see the PROCEDURE Statement and FUNCTION Statement reference entries in the *Progress Language Reference*.

- The super version of an internal procedure (or user-defined function) can get a handle to the procedure file of the original version by using the TARGET–PROCEDURE system handle. Similarly, the super version can get a handle to the procedure file of the original invocation (RUN statement or user-defined function invocation) by using the SOURCE–PROCEDURE system handle. TARGET–PROCEDURE and SOURCE–PROCEDURE thus allow access to the attributes and objects associated with the original procedure files.

- The programmer can write prototypes for internal procedures and user-defined functions invoked in the local procedure file and implemented in super procedures. When a prototype appears, the Progress compiler stores its information so that the INTERNAL–ENTRIES attribute and GET–SIGNATURE method can access it.

  For more information on super procedure prototyping, see the "Super Procedure Prototyping" subsection of this section.

3.7.2 Super Procedure Prototyping

A procedure file can include a prototype—as in "function prototype"—for each internal procedure and user-defined function invoked by (or in) the procedure file and implemented (at run time) in a super procedure. At compile time, Progress adds information from each prototype to the procedure file's INTERNAL–ENTRIES attribute, which lists the procedure file's internal procedures and user-defined functions. At run time, the programmer can examine the INTERNAL–ENTRIES attribute, and for each routine on the list, can invoke the GET–SIGNATURE method to get the routine's signature.

NOTE: If the definition of an internal procedure or user-defined function uses the PRIVATE option, the routine does not appear in the INTERNAL–ENTRIES attribute of a procedure file other than the original, and the routine's signature is not returned by the GET–SIGNATURE method of a procedure file other than the original.

Prototyping Internal Procedures

To write a prototype for an internal procedure whose implementation resides (at run time) in a super procedure, use the PROCEDURE statement with the IN SUPER option. The syntax is as follows:

SYNTAX

```
PROCEDURE proc-name IN SUPER:
```

proc-name

The name of the internal procedure.

For each parameter, code a DEFINE PARAMETER statement the regular way.

If your program includes such a prototype, GET–SIGNATURE(*proc–name*) returns the signature in the following format:

```
PROCEDURE,,parameters
```

Prototyping User-defined Functions

To write a prototype for a user-defined function whose implementation resides (at run time) in a super procedure, use the FUNCTION statement with the IN SUPER option. The syntax is as follows:

SYNTAX

```
FUNCTION function-name RETURNS return-type
  [ ( parameters ) ] IN SUPER
```

function–name

The name of the user-defined function.

return–type

The data type of the item the function returns. For a list of data types, see the reference entry for the FUNCTION statement in the *Progress Language Reference*.

parameters

The parameters of the user defined function. For the parameter syntax, see the reference entry for the FUNCTION statement in the *Progress Language Reference*.

If you include the prototype in your program, compile the program, examine the INTERNAL–ENTRIES attribute of the super procedure, and invoke the GET–SIGNATURE method for the function, GET–SIGNATURE returns the signature in the following format:

```
FUNCTION,return-type,parameters
```

Prototyping Routines Implemented IN SUPER and Locally

A program might define a user-defined function IN SUPER and also implement the function locally. The local implementation might invoke its super version (by using the SUPER function), or might override it entirely. The programmer might write two prototypes, one FORWARD and the other IN SUPER. The Progress compiler permits this, and also verifies that the parameters of the prototype match those of the local definition.

Similarly, a program might define an internal procedure IN SUPER and also implement the internal procedure locally. (In this case, the program contains only one prototype, since internal procedures do not normally have prototypes.) The Progress compiler permits this.

In both cases, at run time, Progress invokes the local implementation.

3.7.3 Programming Example

The following example consists of three procedure files: a main routine, a driver, and a third procedure file that becomes a super procedure of the driver.

The main routine, procedure file `p-pomain.p`, runs the driver procedure persistently:

p-pomain.p

```
/* p-pomain.p */
DEFINE VARIABLE h AS HANDLE.
DEFINE VARIABLE a AS CHARACTER.
FUNCTION sample2 RETURNS CHARACTER (INPUT-OUTPUT a AS CHARACTER) IN h.

RUN p-podrvr.p PERSISTENT SET h.
RUN sample1 IN h (INPUT-OUTPUT a).
MESSAGE a VIEW-AS ALERT-BOX.
a = "".
MESSAGE sample2(a) VIEW-AS ALERT-BOX.
```

The driver, procedure file p-podrvr.p, runs the third procedure file persistently, makes it a super procedure of itself, defines the internal procedure sample1, and defines the user-defined functions sample2, GetPartName, and SetPartName:

p-podrvr.p

```
/* p-podrvr.p */
FUNCTION SetPartName RETURNS INTEGER (INPUT a AS CHARACTER) FORWARD.
DEFINE VARIABLE h AS HANDLE.
DEFINE VARIABLE localPartName AS CHARACTER.

/* Add a super procedure */
RUN p-posupr.p PERSISTENT SET h.
THIS-PROCEDURE:ADD-SUPER-PROCEDURE (h).
SetPartName("2000 Calendar").

PROCEDURE sample1:
    DEFINE INPUT-OUTPUT PARAMETER a AS CHARACTER.
    a = a + "proc: Part name is: ".
    /* Invoke procedure sample1 in the super procedure. */
    RUN SUPER (INPUT-OUTPUT a).
END PROCEDURE.

FUNCTION sample2 RETURNS CHARACTER (INPUT-OUTPUT a AS CHARACTER).
    a = a + "func: Part name is: ".
    /* Invoke function sample2 in the super procedure. */
    SUPER (INPUT-OUTPUT a).
    RETURN a.
END FUNCTION.

FUNCTION GetPartName RETURNS CHARACTER ():
    RETURN localPartName.
END FUNCTION.

FUNCTION SetPartName RETURNS INTEGER (INPUT partname AS CHARACTER):
    localPartName = partname.
END FUNCTION.
```

The third procedure file, p-posupr.p, defines a new version of the internal procedure sample1 and a new version of the user-defined function sample2:

p-posupr.p

```
/* p-posupr.p */
DEFINE VARIABLE h AS HANDLE.
FUNCTION GetPartName RETURNS CHARACTER () IN H.

PROCEDURE sample1:
    DEFINE INPUT-OUTPUT PARAMETER a AS CHARACTER.
    h = TARGET-PROCEDURE.
    a = a + GetPartName().
    MESSAGE "TARGET-PROCEDURE is:" TARGET-PROCEDURE:FILE-NAME
        VIEW-AS ALERT-BOX.
    MESSAGE "SOURCE-PROCEDURE is:" SOURCE-PROCEDURE:FILE-NAME
        VIEW-AS ALERT-BOX.
END PROCEDURE.

FUNCTION SAMPLE2 RETURNS CHARACTER (INPUT-OUTPUT a AS CHARACTER):
    h = TARGET-PROCEDURE.
    a = a + GetPartName().
    RETURN a.
END.
```

To start the example, run p-pomain.p.

4

Named Events

Progress supports an event mechanism which lets you use named events in 4GL programs. Named events provide several benefits:

- First, they let a procedure or widget notify one or more other procedures that some event has occurred, without requiring the procedures to know about each other or to maintain lists of handles to each other. In short, named events decouple communications. This lets you develop applications from components that you build independently and assemble flexibly.
- Second, they replace the *notify* procedure, which the ADM of previous versions of Progress implemented in 4GL code. By contrast, the ADM of the current version of Progress implements named events in 4GL code. This makes applications that use the current ADM and SmartObjects cleaner, faster, and easier to debug.

NOTE: Progress named events are completely different from the key function, mouse, widget, and direct manipulation events. For more information on direct manipulation events, see the *Progress Language Reference*.

The chapter covers the following topics:

- Feature summary
- Programming example

NOTE: For a complete description of the 4GL language elements mentioned in this chapter, see the *Progress Language Reference*.

4.1 Feature Summary

Progress named events allow the following:

- Any procedure or widget within the current Progress session can generate a named event by using the PUBLISH statement.
- Any procedure within the current Progress session can subscribe to a named event by using the SUBSCRIBE statement. Each subscriber contains an internal procedure that Progress runs when the named event occurs. By default, the name of this internal procedure (the *local internal procedure*) is the name of the event. The subscriber can specify a different name by using the SUBSCRIBE statement's RUN–PROCEDURE option.

NOTE: A widget can publish, but not subscribe to, a named event.

- A potential subscriber can find out about named events of interest by using the PUBLISHED–EVENTS attribute.
- Subscribers can cancel subscriptions by using the UNSUBSCRIBE statement.
- Named events can pass parameters and thus have signatures (the number of parameters, and the data type and mode (INPUT, OUTPUT, etc.) of each).

NOTE: Progress supports named events within single Progress sessions only. In other words, a procedure in one session cannot subscribe to a named event created in another session. You can, however, build a Progress procedure that functions as a message broker, routing named events among procedures or widgets within a single Progress session.

4.2 Programming Example

The following example consists of four procedure files: a driver, a publisher, and two subscribers. The driver, p-nedrvr.p, runs the publisher and the two subscribers persistently, and then subscribes to the event NewCustomer on behalf of the second subscriber:

p–nedrvr.p

```
/* p-nedrvr.p  */
DEFINE VARIABLE hPub  AS HANDLE.
DEFINE VARIABLE hSub1 AS HANDLE.
DEFINE VARIABLE hSub2 AS HANDLE.

DEFINE BUTTON bNewCust LABEL "New Customer".
DEFINE BUTTON bQuit LABEL "Quit".

RUN p-nepub.p PERSISTENT set hPub.
RUN p-nesub1.p PERSISTENT set hSub1 (hPub).
RUN p-nesub2.p PERSISTENT set hSub2.

/* Subscribe to event NewCustomer on behalf of subscriber 2 */
SUBSCRIBE PROCEDURE hSub2 TO "NewCustomer" IN hPub.

FORM bNewCust bQuit WITH FRAME x.

ENABLE ALL WITH FRAME x.

ON CHOOSE OF bNewCust RUN NewCust in hPub.

WAIT-FOR CHOOSE OF bQuit OR WINDOW-CLOSE OF CURRENT-WINDOW.
```

The publisher, p-nepub.p, publishes the event NewCustomer:

p–nepub.p

```
/* p-nepub.p */
PROCEDURE NewCust:
   DEFINE VARIABLE name AS CHARACTER INITIAL "Sam".
   /* Let subscriber know new customer */
   PUBLISH "NewCustomer" (INPUT name).
END PROCEDURE.
```

The first subscriber, p-nesub1.p, subscribes to the event NewCustomer:

p–nesub1.p

```
/* p-nesub1.p */
DEFINE INPUT PARAMETER hPub AS HANDLE.
SUBSCRIBE TO "NewCustomer" IN hPub.

PROCEDURE NewCustomer:
  DEFINE INPUT PARAMETER name AS CHAR.
  MESSAGE "Subscriber 1 received event NewCustomer concerning " name
    VIEW-AS ALERT-BOX.
END.
```

The second subscriber, p-nesub2.p, already subscribed to the event NewCustomer, cancels all subscriptions:

p–nesub2.p

```
/* p-nesub2.p */
PROCEDURE NewCustomer:
  DEFINE INPUT PARAMETER name AS CHAR.
  MESSAGE "Subscriber 2 received event NewCustomer concerning " name
    VIEW-AS ALERT-BOX.
  /* This subscriber receives the first event, then removes itself */
  UNSUBSCRIBE TO ALL.
END.
```

To start the example, run the driver, p-nedrvr.p.

5

Condition Handling and Messages

Progress recognizes four special conditions:

- The ERROR condition
- The ENDKEY condition
- The STOP condition
- The QUIT condition

Progress defines a default response to each of these conditions, but you can override these defaults within your application. This chapter details how Progress handles each of these conditions and how you can change that handling. It also covers:

- How Progress responds to system and software errors
- Progress messages

5.1 Progress Conditions

Progress handles the conditions listed in Table 5–1. Conditions may be raised by specific 4GL statements, procedure errors, or keyboard events.

Table 5–1: Progress Conditions

Condition	Primary Causes	Default Processing	To Override Default Processing
ERROR	RETURN ERROR executed by a trigger or run procedure. OR Procedure error, such as creating a duplicate entry in a unique index or performing a FIND, FIND FIRST, or FIND LAST for a nonexistent record. OR The user presses **ERROR**.	Undo and retry the nearest REPEAT, FOR EACH, or procedure block. Within a database trigger, undo and return ERROR.	ON ERROR phrase
ENDKEY	The user presses **ENDKEY**. OR A FIND NEXT or FIND PREV fails. OR Reach the end of an input file.	Undo and leave the nearest REPEAT, FOR EACH, or procedure block.	ON ENDKEY phrase
STOP	STOP statement executed. OR The user presses **STOP.**	Undo the current transaction and retry the startup procedure (or return to the Editor).	ON STOP phrase
QUIT	QUIT statement executed.	Undo the current transaction and return to the operating system (or to the Editor). On an AppServer, terminate the session and return to the Progress client.	ON QUIT phrase

For each condition, you may want to accept the default Progress processing. In many situations this provides the functionality you would want. However, you can also override the default behavior.

5.2 Rules About UNDO

Within an ON ERROR, ON ENDKEY, ON STOP, or ON QUIT phrase, you can specify a block to be undone. This is the syntax for the UNDO option:

SYNTAX

```
UNDO [ label1 ]
  [    , LEAVE  [ label2 ]
     | , NEXT   [ label2 ]
     | , RETRY  [ label1 ]
     | , RETURN [ ERROR | NO-APPLY ] [ return-string ]
]
```

Within this syntax, you can name the block to be undone, *label1*. The block you name must be the current block or a block that contains the current block. If you omit *label1,* Progress determines the block to undo. It chooses the innermost containing block with the error property. The following blocks have the error property:

- FOR
- REPEAT
- Procedure blocks
- DO blocks with the TRANSACTION keyword or ON–ERROR phrase

Optionally, you can specify the action Progress takes after undoing the block:

- LEAVE that block or an enclosing block.
- Perform the NEXT iteration of the the block or an enclosing block.
- RETRY the current iteration of the block.
- RETURN to the calling procedure. If you are within a user-interface trigger, you can return NO–APPLY to cancel the action. In any other context, you can return ERROR to raise the ERROR condition in the caller. You can also pass a return string back to the caller. The caller can read that string by invoking the RETURN–VALUE function.

The default action is to retry the block that was undone.

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

5.2.1 Infinite Loop Protection

If a statement within a block forces an undo and retry of the block, an infinite loop is possible. That is, it is possible that whatever caused the block to be undone and retried the first time (whether it is a condition such as ERROR or an explicit UNDO statement), may recur during the retry. If Progress detects that such an infinite loop would occur, then it does not retry the block. Instead it performs the NEXT iteration of a FOR EACH block or iterating DO block; it performs a LEAVE on a REPEAT block, procedure block, trigger block, or non-iterating DO block.

Progress detects that an infinite loop would occur if the following conditions are true:

- The block does not reference any statements that block for user input (such as the UPDATE or WAIT–FOR statements) before the error occurs.
- The block does not reference the RETRY function before the error occurs.

If the block contains statements that block for input, then Progress assumes the user can prevent an infinite loop by supplying different input. If the block contains a reference to the RETRY function then Progress assumes that you have done your own programming to avoid infinite loops.

For example, the following code specifies that the DO block should be retried if the FIND statement produces an error. However, this would lead to an infinite loop (with i always equal to 1) if the specified record does not exist:

p–itundo.p

```
DEFINE VARIABLE i AS INTEGER.

DO i = 1 TO 10 ON ERROR UNDO, RETRY:
   DISPLAY i.
   FIND Customer WHERE Cust-num = 999.
END.
```

If you run this procedure, Progress performs a NEXT rather than a RETRY when an error occurs. Therefore, you see the displayed value of i iterate from 1 to 10. After the tenth error, the procedure ends. Thus, the infinite loop is avoided.

In the following example, the RETRY function is referenced before the statement that causes the error. Therefore, Progress does not detect that an infinite loop occurs on retry:

p–itund2.p

```
DEFINE VARIABLE i AS INTEGER.

DO i = 1 TO 10 ON ERROR UNDO, RETRY:

   IF RETRY
   THEN MESSAGE "Retrying DO loop.".

   DISPLAY i.
   FIND Customer WHERE Cust-num = 999.
END.
```

If you run this code, Progress enters an infinite loop in which i is always 1. Because the RETRY function is referenced before the error, Progress assumes that you are handling the infinite loop problem. You can fix this by changing the ON ERROR phrase to perform a NEXT or LEAVE rather than a RETRY. Alternately, you can perform some processing when RETRY is true that changes the value of i.

5.3 The ERROR Condition

There are three common ways a procedure can generate the ERROR condition:

- The RETURN ERROR statement is executed.
- The user presses **ERROR**.
- A statement cannot be executed properly. For example, the procedure may try to create a duplicate entry in a unique index or find a record that does not exist.

5.3.1 The ERROR–STATUS System Handle

Many Progress statements support the NO–ERROR option. If you specify this option, the ERROR condition is not generated by that statement. Instead, if the statement cannot execute properly, execution continues with the next statement. You can subsequently check whether an error occurred by examining the attributes of the ERROR–STATUS system handle.

The ERROR–STATUS handle contains information on the last statement executed with the NO–ERROR option. You can determine whether Progress attempted to raise the error condition in that statement by reading the value of the ERROR attribute of ERROR–STATUS. To retrieve the error messages and error numbers, first read the value of the NUM–MESSAGES attribute, then use the methods ERROR–STATUS:GET–MESSAGE(*n*) and ERROR–STATUS:GET–NUMBER(*n*). For more information on these methods, see the *Progress Language Reference* section on the ERROR–STATUS System Handle.

The ERROR–STATUS:ERROR attribute remains true until the completion of the next statement with the NO–ERROR option.

The following program attemps to retrieve a record from the customer table with NO–ERROR. If an error occurs, you are prompted to display the message:

p–errst2.p

```
DEFINE VARIABLE i AS INTEGER NO-UNDO.

FORM WITH FRAME bbb.

REPEAT WITH FRAME bbb:
   PROMPT-FOR customer.cust-num WITH FRAME aaa.
   FIND customer USING cust-num NO-LOCK NO-ERROR.
   IF ERROR-STATUS:ERROR THEN DO:
     MESSAGE ERROR-STATUS:NUM-MESSAGES
         " errors occurred during conversion." SKIP
         "Do you want to view them?"
       VIEW-AS ALERT-BOX QUESTION BUTTONS YES-NO
       UPDATE view-errs AS LOGICAL.
     IF view-errs THEN
       DO i = 1 TO ERROR-STATUS:NUM-MESSAGES:
         MESSAGE ERROR-STATUS:GET-NUMBER(i)
                 ERROR-STATUS:GET-MESSAGE(i).
       END.
     END.
     ELSE
       DISPLAY customer WITH 2 COLUMNS.
END.
```

5.3.2 Default Handling

When the ERROR condition occurs anywhere outside of a trigger block, Progress follows these steps:

1. Looks for the closest block with the error property.

2. Undoes and retries that block.

FOR EACH blocks, REPEAT blocks, and procedure blocks automatically have the error property. This means that each of these blocks implicitly has the ON ERROR UNDO, RETRY phrase attached to it. The following two examples are the same:

```
FOR EACH customer:
```

```
FOR EACH customer ON ERROR UNDO, RETRY:
```

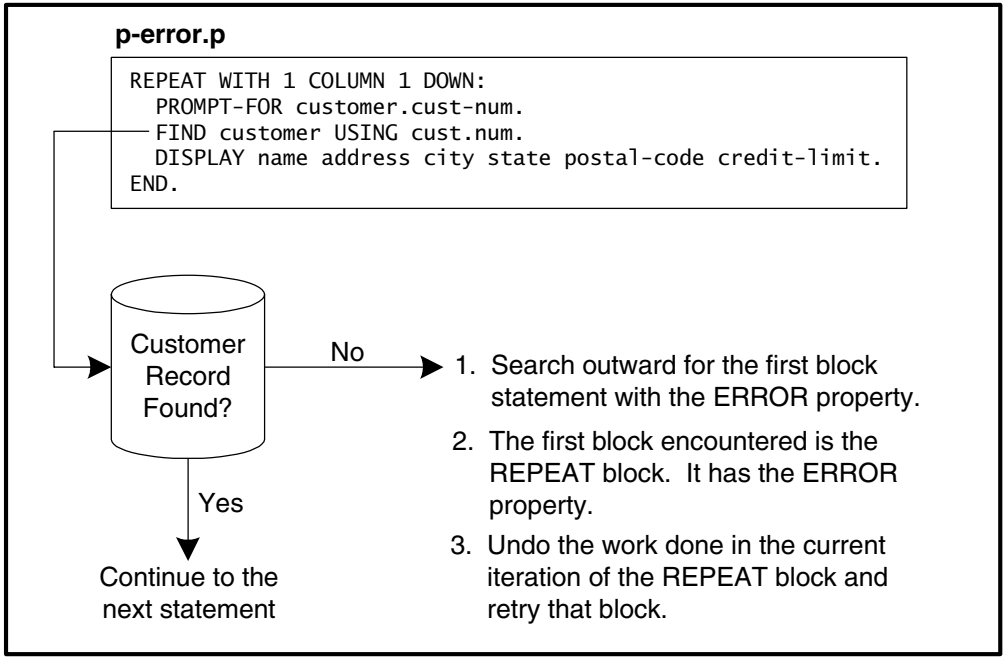
A DO block has the error property if its header contains the TRANSACTION keyword or the ON ERROR phrase.

When the ERROR condition occurs within a database trigger block, Progress follows these steps:

1. Undoes the trigger block.
2. Returns ERROR.

The following example shows how Progress handles a processing error:

p-error.p



If Progress is not handling a specific processing error the way you want, you can override automatic error handling by handling the error yourself.

Progress has no predefined **ERROR** key. However, you can define any control key or other special key as the **ERROR** key. Then, when the user presses that key at an input prompt, the procedure does error processing.

The next example demonstrates what Progress does when the user presses a defined **ERROR** key:

p-txn6.p

```
ON F9 ERROR.

REPEAT:
  INSERT order WITH 2 COLUMNS.
  FIND customer OF order.
  REPEAT:
    CREATE order-line.
    order-line.order-num = order.order-num.
    DISPLAY order-line.order-num.
    UPDATE line-num order-line.item-num qty.
    FIND item OF order-line.
    order-line.price = item.price.
    UPDATE order-line.price.
  END.
END.
```

Transaction

Subtransaction

The ON statement defines **F9** as the **ERROR** key. That is, when the user presses **F9**, Progress raises the **ERROR** condition, searches for the closest block that has the error property and undoes and retries that block.

NOTE: If you do not have an **F9** key on your keyboard, you can modify this procedure to name another key, such as **F2** or **PF2**. Or, if your keyboard has no F or PF keys, specify ON CTRL–W ERROR to assign ERROR to the **CTRL–W** keys.

When you run this procedure, the INSERT statement displays a screen similar to this:

Order-num:	1005	Cust-Num:	0
Ordered:	07/14/94	Shipped:	/ /
Promised:	07/28/94	Carrier:	
Instructions:		PO:	daniel0714941005
Terms:	Net30	Sales-Rep:	

If you press **F9**, Progress follows these steps:

1. Raises the ERROR condition.
2. Searches outward for the closest block that has the error property. REPEAT blocks have the error property so Progress stops at the REPEAT block.
3. Undoes all work done in the current iteration of the REPEAT block and retries that iteration of the REPEAT block.

In this case, Progress undoes and retries the transaction block.

If you run the `p-txn6.p` procedure again, and press **F9** during the order–line block, Progress undoes the subtransaction and retries that iteration of the inner REPEAT block.

5.3.3 Overriding the Default Handling

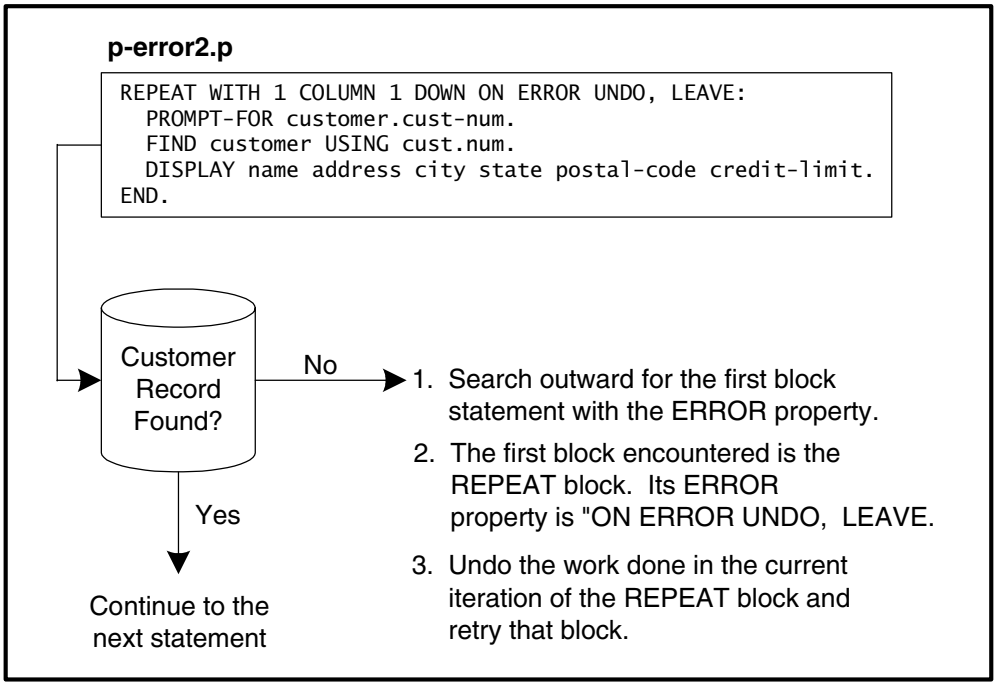
When an error occurs, you might not want Progress to perform the default error processing. Rather than undoing and retrying the nearest block with the error property, you might want Progress to take some other action. For example, you might want Progress to leave the block or try the next iteration of the block. On the other hand, you might want Progress to undo a smaller amount of work. You have two techniques to change the error handling:

- Override the default error processing for a block by coding an explicit ON ERROR phrase on the nearest REPEAT, FOR EACH, or procedure block.
- Change the amount of work to be undone by adding the ON ERROR phrase to a DO block (thus giving the DO block the error property) or specifying a different block in the ON ERROR phrase.

Take the procedure `p-error.p`, shown earlier, as an example. In this procedure, if the FIND statement fails, Progress undoes and retries that iteration of the REPEAT block. Suppose that you want to change that error processing to the following: if the FIND statement fails, undo the current iteration, and **leave** the REPEAT block. This is the modified `p-error.p` procedure.

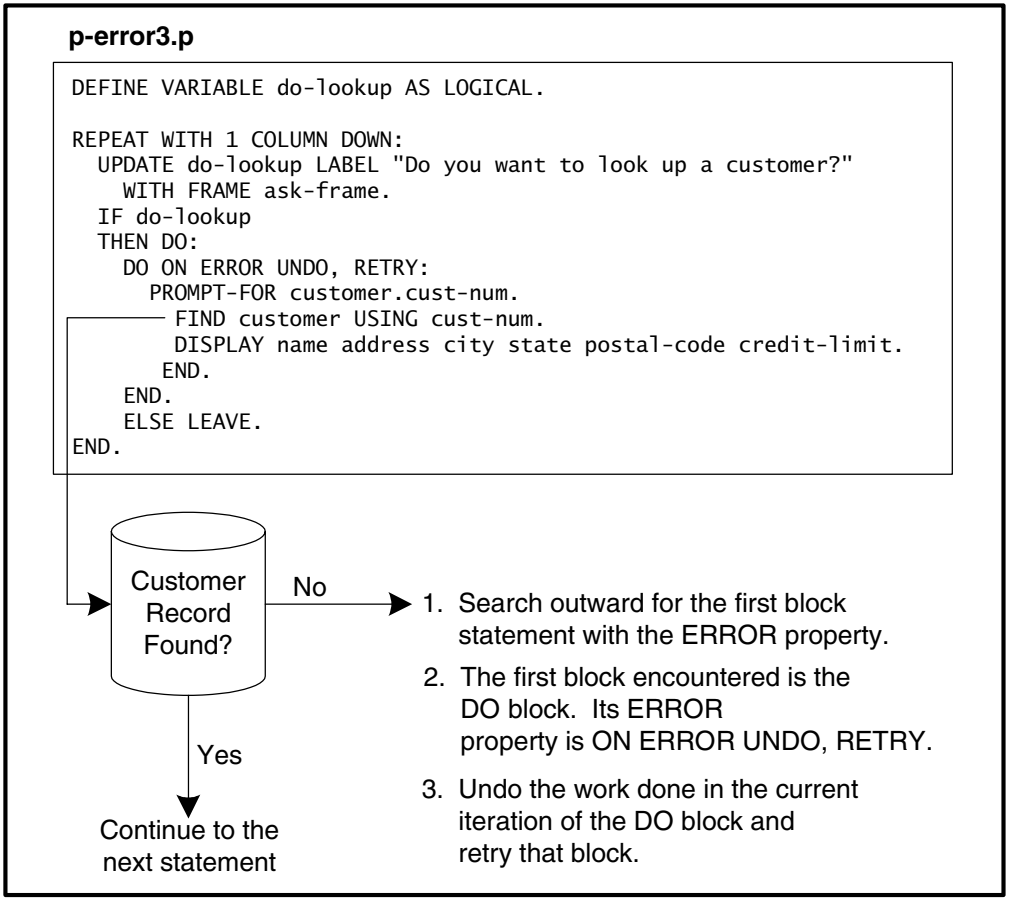
In `p-error2.p`, the explicit ON ERROR phrase changes the handling of the ERROR condition:

p-error2.p



In the following example, the ON ERROR phrase is added to a DO block to change the amount of work to be undone:

p-error3.p



Because the DO block has the error property, only the DO block is undone when an error occurs. Therefore, the UPDATE statement is not retried; instead, execution continues from the PROMPT–FOR statement.

In p-error3.p, you made Progress undo less than the default amount of work. Suppose you wanted Progress to undo **more** than the default amount of work. For example, in p-txn6.p, shown earlier, if you make an error while entering an order–line (you pressed **F9**), you want to undo all the work done on the current order. To do this, you add a label to the outer REPEAT block and reference that label in the ON ERROR phrase of the inner REPEAT block:

p–txn7.p

```
ON F9 ERROR.

o-block:
REPEAT:
  INSERT order WITH 2 COLUMNS.
  FIND customer OF order.
  o-l-block:
  REPEAT ON ERROR UNDO o-block, RETRY o-block:
    CREATE order-line.
    order-line.order-num = order.order-num.
    DISPLAY line-num order-line.item-num qty.
    SET line-num order-line.item-num qty.
    FIND item OF order-line.
    order-line.price = item.price.
    UPDATE order-line.price.
  END.
END.
```

Here, the inner REPEAT block explicitly says that, in the event of an error, Progress should undo not the current REPEAT block but the outer o–block REPEAT block. In addition, it tells Progress to retry the o–block block. This error processing applies to any kind of an error. That is, it happens if you press **F9**, or if there is some sort of processing error such as the FIND statement's is unable to locate a record.

5.4 The ENDKEY Condition

The ENDKEY condition occurs when you press **ENDKEY** when input is enabled. It also occurs if an attempt to find a record fails or you reach the end of an input stream.

Progress does not have a predefined **ENDKEY** on the keyboard. However, you can use the ON statement to assign a key as **ENDKEY**, or you can define any control (**CTRL**) key or other special key as **ENDKEY**. Then, when the user presses that key, the ENDKEY condition is raised. Also, the **END–ERROR** key behaves like **ENDKEY** for the first I/O operation in a block.

NOTE: Do not confuse the **ENDKEY** with the **END** key. They are completely unrelated.

5.4.1 Default Handling

When the ENDKEY condition occurs, Progress follows these steps:

1. Looks for the closest block that has the endkey property.

2. Undoes and leaves that block.

FOR EACH blocks, REPEAT blocks, triggers, and procedure blocks have the endkey property by default. This means that each of these blocks implicitly has the ON ENDKEY UNDO, LEAVE phrase attached to it. The following two examples are the same:

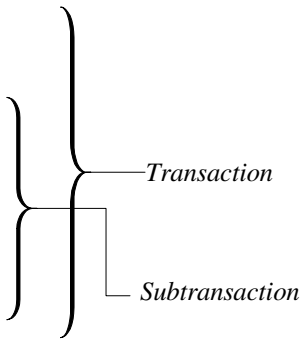
```
FOR EACH customer:
```

```
FOR EACH customer ON ENDKEY UNDO, LEAVE:
```

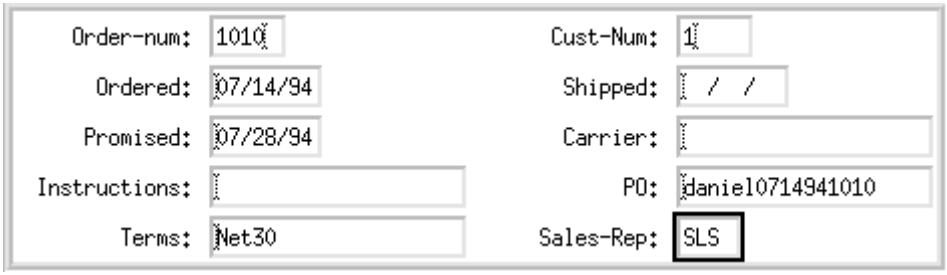
This example shows how Progress handles **ENDKEY**:

p-txn8.p

```
ON F9 ENDKEY.

o-block:
REPEAT:
  INSERT order WITH 2 COLUMNS.
  FIND customer OF order.
  o-l-block:
  REPEAT:
    CREATE order-line.
    order-line.order-num = order.order-num.
    DISPLAY order-line.order-num.
    UPDATE line-num order-line.item-num qty.
    FIND item OF order-line.
    order-line.price = item.price.
    UPDATE order-line.price.
  END.
END.
```



The ON statement defines **F9** as the **ENDKEY**. That is, when the user presses **F9**, Progress raises the ENDKEY condition. Progress searches for the closest block that has the ENDKEY property and undoes and leaves that block:

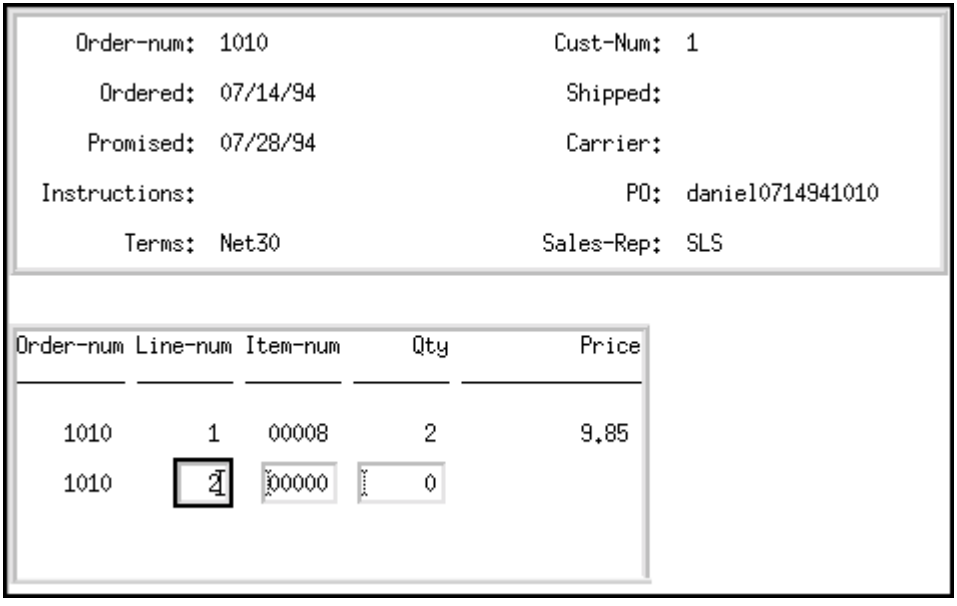


Before entering a sales rep value, press **F9**. Progress follow these steps:

1. Raises the ENDKEY condition.

2. Searches outward for the closest block that has the ENDKEY property. REPEAT blocks have the ENDKEY property, so Progress stops at the outer REPEAT block.

3. Undoes all work done in the current iteration of the REPEAT block and leaves that REPEAT block. Since there are no statements to process after the END statement of the REPEAT block, Progress returns you to the Procedure Editor:



Press **F9** while entering the second order–line. Progress follows these steps:

1. Raises the ENDKEY condition.

2. Searches outward for the closest block that has the ENDKEY property. REPEAT blocks have the ENDKEY property so Progress stops at the inner REPEAT block.

3. Undoes all work done in the current iteration of the REPEAT block and leaves that iteration of the REPEAT block.

4. After leaving the inner REPEAT block, the procedure continues with the next statement after the END of the inner block, encounters the END of the outer REPEAT block, and does the next iteration of the outer block.

Note that this means that the order record and first order–line record that you created are committed to the database. Only the second order–line record is undone.

If you are importing data from an operating system file, the ENDKEY condition occurs when you reach the end of the file. For example, `p-impeof.p` takes advantage of this:

p–impeof.p

```
INPUT FROM customer.d.

REPEAT:
   CREATE customer.
   IMPORT customer.
END.
```

Assume that `customer.d` contains three records. On the third pass through the REPEAT loop, `p-impeof.p` creates a customer record and reads data from the last line from the file into that record. On the next pass through the loop, the procedure creates another customer record. However, the IMPORT statement fails because it has reached the end of the file. This causes the ENDKEY condition. The default behavior for ENDKEY undoes the current iteration of the REPEAT loop and therefore undoes the record creation.

5.4.2 Overriding the Default Handling

You can override the default ENDKEY processing by adding the ON ENDKEY phrase to a block. Suppose you wanted Progress to undo more than the current iteration of the inner REPEAT block. That is, if you press **F9** while entering an order–line, you want to undo all the work done on the current order and leave the procedure together:

p–txn9.p

```
ON F9 ENDKEY.

o-block:
REPEAT:
  INSERT order WITH 2 COLUMNS.
  FIND customer OF order.
  O-1-block:
  REPEAT ON ENDKEY UNDO o-block, RETRY o-block:
    CREATE order-line.
    order-line.order-num = order.order-num.
    DISPLAY order-line.order-num.
    UPDATE lin-num order-line.item-num qty.
    FIND item OF order-line.
    price = cost.
    UPDATE price.
  END.
END.
```

Here, the inner REPEAT block explicitly states that in the event of the ENDKEY condition, Progress does not undo the current REPEAT block but the outer o–block REPEAT block. In addition, it tells Progress to retry the o–block REPEAT block.

5.5 The STOP Condition

The STOP condition occurs when a STOP statement is executed or when you press **STOP** when input is enabled (or anytime in a character interface). Progress also raises the STOP condition when an unrecoverable system error occurs; for example, when a database connection is lost or a run procedure is not found. This stop key is usually mapped as follows:

- **CTRL–BREAK** (Windows)
- **CTRL–C** (UNIX)

5.5.1 Default Handling

When the STOP condition occurs, by default Progress follows these steps.

1. Undoes the current transaction.
2. If the application was started from a Progress tool such as the Procedure Editor, it returns to that tool; otherwise, it reruns the startup procedure.

5.5.2 Overriding the Default Handling

The default handling for the STOP condition is almost always appropriate. However, you can override the default handling by adding the ON STOP phrase to a REPEAT, FOR EACH, or DO statement:

p–stop.p

```
FOR EACH Customer ON ERROR UNDO, LEAVE
                  ON STOP UNDO, RETRY:
   IF RETRY
   THEN DO:
      MESSAGE "The STOP condition has occurred." SKIP
              "Do you wish to continue?" VIEW-AS ALERT-BOX QUESTION
              BUTTONS yes-no UPDATE continue-ok AS LOGICAL.
      IF NOT continue-ok
      THEN UNDO, LEAVE.
   END.

   UPDATE Customer.
END.
```

In `p-stop.p`, the ON ERROR and ON STOP phrases cause the FOR EACH loop to retry if and only if the STOP condition occurs. When this happens, the procedure displays an alert box that lets you decide whether to continue.

5.6 The QUIT Condition

The QUIT condition occurs when a QUIT statement is executed.

5.6.1 Default Handling

When the QUIT condition occurs, by default Progress follows these steps:

1. Commits the current transaction.
2. If the application was started from the Procedure Editor or User Interface Builder, returns to that tool; otherwise, returns to the operating system.

NOTE: If you run a procedure that consists of only a QUIT statement from the Procedure Editor, Progress exits the Procedure Editor.

5.6.2 Overriding the Default Handling

The default handling of the QUIT condition is almost always appropriate. However, by using the ON QUIT phrase in a REPEAT, FOR EACH, or DO statement, you can (cause undo work to be undone) or override the rest of the default handling.

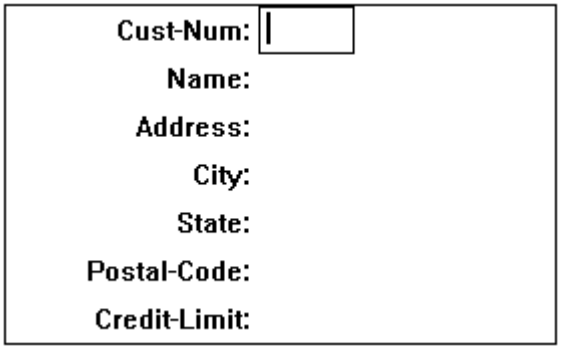
5.7 The END–ERROR Key

The **END–ERROR** key (usually **F4** in a character interface, and **ESCAPE** on Windows in graphical interfaces) is so called because sometimes Progress treats it as **ERROR** and other times treats it as **ENDKEY**:

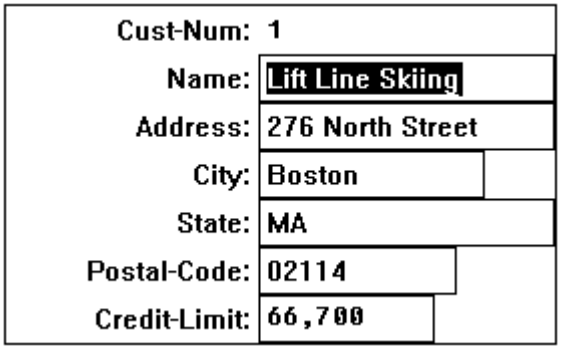
p–error4.p

```
REPEAT WITH 1 COLUMN 1 DOWN:
  PROMPT-FOR customer.cust-num.
  FIND customer USING cust-num.
  UPDATE name address city state postal-code credit-limit.
END.
```

This code runs the following procedure:



At this point, you are at the PROMPT–FOR statement in the procedure. Press **END–ERROR**. The procedure returns you to the Procedure Editor. Run the procedure again, entering a 1 for Cust num and press **GO**:



At this point you are at the UPDATE statement in the procedure. Press **END–ERROR**. The procedure returns you to the PROMPT–FOR statement and lets you enter another customer number.

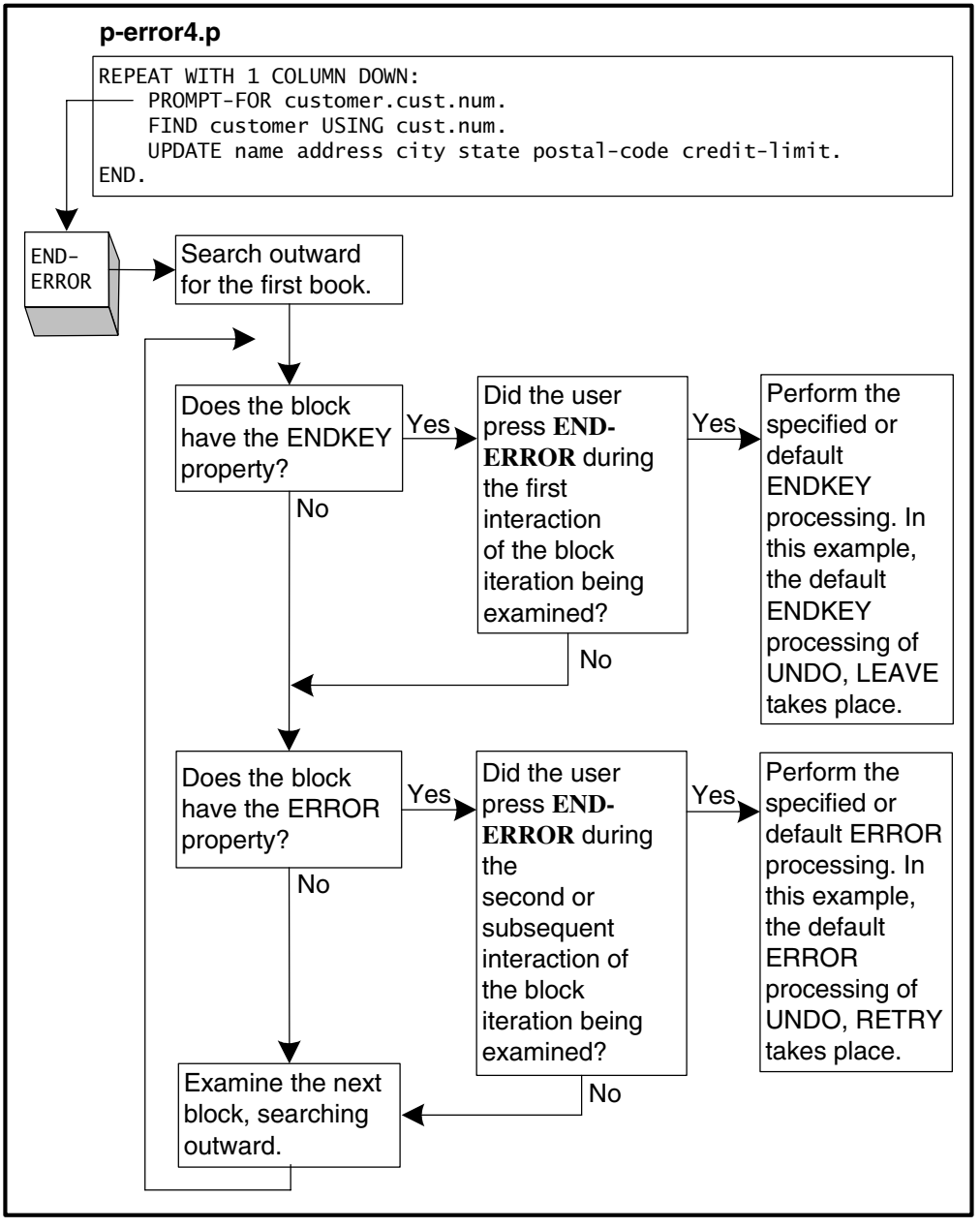
5.7.1 Distinguishing ERROR and ENDKEY

Progress treats the same **END–ERROR** key differently under different circumstances. This is why:

- If you press **END–ERROR** when you are on the first screen interaction (the first statement that gives you an opportunity to enter data) during a block iteration, it is probably because you decided you do not want to complete the task you started. In this case, Progress undoes whatever you have done in the current iteration of the block and leaves the block. That is, it acts as an **ENDKEY** key.
- If you press **END–ERROR** on the second or later interaction of a block iteration, it is probably because you made a mistake or want to retry the work you have done. In this case, Progress undoes whatever you have done and retries. That is, it acts as an **ERROR** key.

Run the `p-error4.p` procedure again. The following figure shows what happens when you press **END–ERROR:**

p-error4.p



In determining whether the user pressed **END–ERROR** during the first or subsequent interaction in a block iteration, Progress does not count any keyboard entries to continue after a pause.

5.7.2 ENABLE/WAIT–FOR Processing

The **END–ERROR** key modifies its behavior yet again depending on whether input is blocked by a WAIT–FOR statement or any other input blocking statement, such as UPDATE. The `p-error5.p` procedure shows the difference:

p–error5.p

```
DEFINE VARIABLE a AS CHARACTER.
DEFINE VARIABLE b AS CHARACTER.
FORM a b WITH FRAME X 2 COLUMNS.

SESSION:APPL-ALERT-BOXES = TRUE.
ON END-ERROR ANYWHERE DO:
  MESSAGE KEYFUNCTION(LASTKEY) "raised".
END.
MESSAGE "Entering UPDATE Block".
DO ON ERROR UNDO, RETRY ON ENDKEY UNDO, LEAVE WITH FRAME X:
  IF RETRY THEN MESSAGE "Retrying UPDATE Block".
  UPDATE a WITH FRAME X.
  MESSAGE "After UPDATE a".
  UPDATE b WITH FRAME X.
  MESSAGE "After UPDATE b".
END.
MESSAGE "Entering ENABLE/WAIT-FOR Block".
DO ON ERROR UNDO, RETRY ON ENDKEY UNDO, LEAVE WITH FRAME X:
  IF RETRY THEN MESSAGE "Retrying ENABLE/WAIT-FOR Block".
  DISABLE ALL.
  ENABLE a.
  WAIT-FOR GO OF FRAME X.
  MESSAGE "After ENABLE/WAIT-FOR a".
  DISABLE ALL.
  ENABLE b.
  WAIT-FOR GO OF FRAME X.
  MESSAGE "After ENABLE/WAIT-FOR b".
END.
```

Within the DO block of UPDATE statements, **END–ERROR** works as explained earlier. That is, for an UPDATE statement, **END–ERROR** on field a acts like **ENDKEY** (leaving the block) and on field b acts like **ERROR** (retrying the block and displaying the RETRY message).

However, in a similar DO block, where input is enabled with the ENABLE statement and blocked with the WAIT–FOR statement, **END–ERROR** always works like **ENDKEY.** It always leaves the block when entered for field a or field b. Progress treats **END–ERROR** differently in this case, because the WAIT–FOR statement is designed to control input in a more random and

less modal fashion than the UPDATE statement. In an event-driven interface, the triggers for each widget are generally responsible for handling error input for that widget. Thus, **END–ERROR** as **ENDKEY**. provides a consistent way to exit the interface from any input widget.

5.8 How Progress Handles System and Software Failures

Following a system hardware or software failure (a crash), Progress undoes partially completed transactions for all users. This is the fundamental difference between transactions and subtransactions: Progress undoes a partially completed transaction after a system failure, including work done in any complete or incomplete subtransactions encompassed within the transaction.

Also, if the user presses **STOP**, Progress undoes the current transaction and returns control to the startup procedure if one is still active, or to the Editor.

If Progress loses a database connection (for example, because a remote server failed), client processing might continue. In this case, Progress does the following:

- Raises the STOP condition. For this special instance of the STOP condition you cannot change the default processing. Any ON STOP phrases are ignored.
- Deletes any persistent procedures that reference the disconnected database.
- Progress undoes blocks beginning with the innermost active block and working outward. It continues to undo blocks until it reaches a level above all references to tables ore sequences in the lost database.
- Progress changes to the normal STOP condition. From this point, any further ON STOP phrases you have coded are used.
- Progress continues to undo blocks until it reaches an ON STOP phrase. If no ON STOP phrase is reached, it undoes all active blocks and restarts the top level procedure.

5.9 Progress Messages

Progress and applications written under Progress display several types of messages to inform you of both routine and unusual occurrences:

- Execution messages inform you of errors encountered while Progress is running a procedure (for example, a record with a specified index field value could not be found).
- Compile messages inform you of errors found while Progress is reading and analyzing a procedure prior to running it (for example, a reference to a table name that is not defined in the database).
- Startup messages inform you of unusual conditions detected while Progress is getting ready to execute (for example, an invalid startup parameter was entered).

After it displays a message, Progress proceeds in one of several ways:

- Continues execution, subject to the error processing actions which you give, or are assumed, as part of your procedure. This is the usual action taken following execution messages.
- Returns to the Procedure Editor (or other tool from which you ran the procedure) so that you can correct an error in a procedure. This is the usual action taken following compiler messages.
- Halts processing of a procedure and returns immediately to the Editor or other tool.
- Terminates the current session.

Access Progress Help (press **HELP**) if you do not understand a message. If you encounter an error message that terminates Progress, the message ends with a number in parentheses. Carefully note that number before restarting Progress. In all other cases, Progress keeps track of messages that it displayed recently.

5.10 Application Messages

Within an application you might want to pass a message from one procedure back to the procedure that called it. For example, if a subprocedure encounters an error, you might want to pass a message about the error back to the caller. You can pass such as message on the RETURN statement. For example, `p-retstr.p` tries to find a customer record in the database and returns either the customer name or a message indicating why the record is not available:

p–retstr.p

```
DEFINE PARAMETER BUFFER cust-buf FOR customer.
DEFINE INPUT PARAMETER  cnum     AS INTEGER.

FIND cust-buf WHERE cust-num = cnum NO-ERROR.

IF NOT AVAILABLE cust-buf
THEN DO:
    IF LOCKED cust-buf
    THEN RETURN "Record is locked.".
    ELSE RETURN "Record not found.".
END.

RETURN cust-buf.name.
```

The calling procedure can read a returned message by using the RETURN–VALUE function. For example, `p-retval.p` runs `p-retstr.p` and displays the returned value in a frame:

p–retval.p

```
DEFINE VARIABLE i AS INTEGER.

FORM
  customer.cust-num customer.name
  WITH FRAME name-frame DOWN.

FIND FIRST customer.

DISPLAY cust-num name WITH FRAME name-frame.

DO i = cust-num + 1 TO CURRENT-VALUE(next-cust-num):
   DOWN WITH FRAME name-frame.
   RUN p-retstr.p (BUFFER customer, i).
   DISPLAY i @ customer.cust-num
           RETURN-VALUE @ customer.name WITH FRAME name-frame.
END.
```

This procedure displays all customer numbers from the first to the current value of the next–cust–num sequence. For each number it displays either the customer name or a message indicating the status of that record.

NOTE: If you have a procedure which does not end with the RETURN statement, the value in RETURN–VALUE will be the value of the last executed RETURN statement. RETURN–VALUE is not cleared if there is no RETURN statement. If no value was reurned by the most recently executed RETURN statement, RETURN–VALUE returns an empty string (“”).

For more information on the RETURN statement and the RETURN–VALUE function, see the *Progress Language Reference*.

6

Handling User Input

This chapter explains how you can monitor user input from the keyboard or mouse. It discusses:

- The keys you normally use when running a Progress procedure and how Progress handles each of those keys.
- How you can redefine the functions of different keys.
- How Progress handles mouse buttons, and how you can monitor mouse events.
- How you can monitor specific keystrokes within a procedure.

This chapter, focuses on data that you enter from the keyboard. Chapter 7, "Alternate I/O Sources," explains how to write procedures that accept input from a file.

6.1 The Keyboard and the Mouse

Progress accepts user input from the keyboard and, where available, from the mouse. Although the keys on different keyboards vary somewhat, Progress defines several hundred standard key codes that map to common key labels (or sequences). Some of these codes are also mapped to special functions by Progress or the windowing system. As shown in Figure 6–1, the **F2** key label maps to key code 302. On most systems, this key code also maps to the Progress **GO** key function.

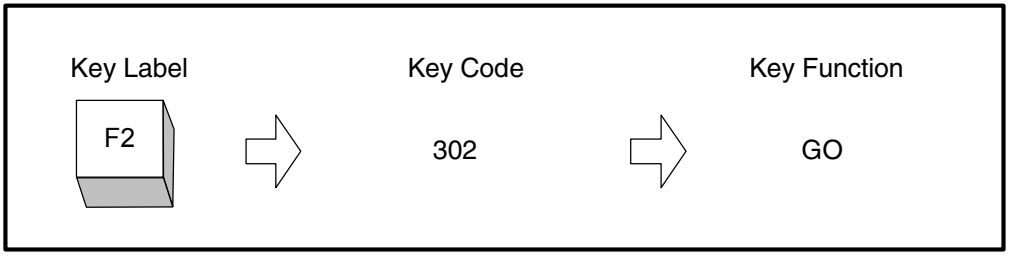


Figure 6–1: Key Labels, Key Codes, and Key Functions

Progress also defines a four-button mouse model. Each of the four portable buttons maps to a physical button (possibly with a modifying key) on your mouse. For example, the portable SELECT button usually maps to the left button on a two- or three-button mouse. The mouse buttons generate input events that map to key labels. However, the way Progress handles mouse input is different from the keyboard because of the way the mouse generates events—sending both press (down) and release (up) signals. Progress provides access either to the down or up event or to combinations of these inputs as a single event. For more information on mouse buttons, events, and how to monitor them, see the "Using Mouse Buttons and Events" section.

6.1.1 Key Mapping

Your environment determines what key sequences on your keyboard map to each Progress key function. Your environment might reside in the registry (Windows only) or in an initialization file. Examples of initialization files are the `progress.ini` file (on Windows) and the PROTERMCAP file (on UNIX). For more information on environments, see the chapter on colors and fonts in this book and the chapter on user interface environments in the *Progress Client Deployment Guide*.

6.1.2 Key Monitoring

Often, an application does not need to be aware of specific keystrokes or mouse actions by the user because the normal functionality provided by Progress and the window system is sufficient. This is frequently true when you use the procedure-driven programming model. However, you might want to define a special behavior to occur when the user presses a specific key or mouse button. For example, every time the user presses a certain key, you might want to display a message or take some action. This is called *monitoring* the user's keystrokes. Progress provides a set of language constructs to do this, including user interface triggers. For more information, see the"Monitoring Keystrokes During Data Entry" section.

During a session, Progress responds to key code events according to a precedence that depends on the current user interface and the application design. Table 6–1 shows the general order of precedence for Progress key code events.

Table 6–1: Progress Key Input Precedence

Precedence (High to Low)	Key Code Event
8	In the accelerator list for the current window
7	In the event list for an active user interface trigger
6	An active popup menu key
5	An active menu mnemonic key
4	An active field-level (button) mnemonic key (Windows only)
3	A special Progress internal key (built-in key function, like GO)
2	A key associated with a key function in the PROTERMCAP file (UNIX only)
1	A standard input key ("A", "B", and so on)

Thus, when the user presses a key, Progress first checks to see if it corresponds to an active accelerator in the current window. If not, Progress checks for a user interface trigger for that key, then whether the key displays a popup menu, and so on. If the key corresponds to no other function, Progress processes it as data input, with any required character validation.

6.1.3 Key Translation Functions

Progress key codes, key labels, and key functions are interrelated, and Progress provides a set of built-in functions to convert from one to another. Figure 6–2 shows how the conversions work, and which functions to use for each conversion.

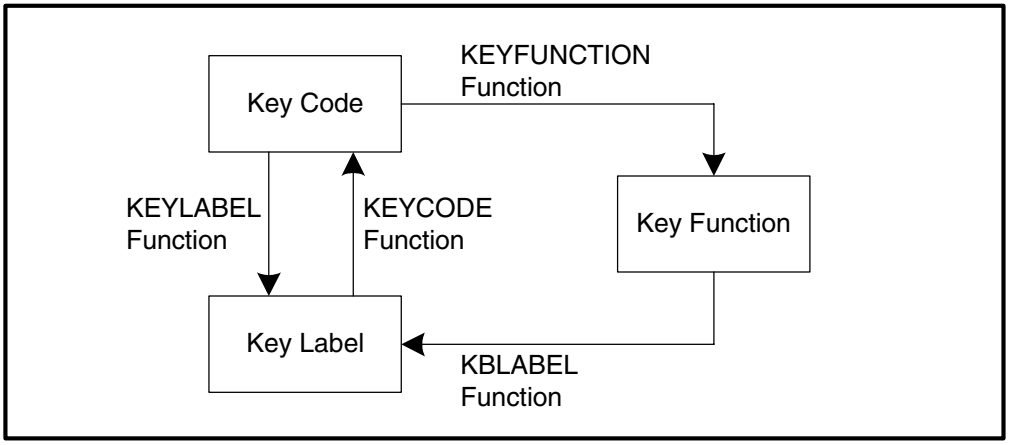


Figure 6–2: Progress Key Translation Functions

You can use the KEYCODE function to determine the key code for a particular key. For example, the following function returns the value 127, which is the integer code assigned to **DEL**. Use the KEYLABEL function to determine the keyboard label for a particular key code.

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

For example, the following function returns the value **CTRL–G:**

```
KEYLABEL(7).
```

See the *Progress Language Reference* for more information on the KBLABEL, KEYCODE, KEYFUNCTION, and KEYLABEL functions.

6.2 Key Codes and Key Labels

All Progress keys have both a code and a label. The key code is a Progress internal identifier for the key. Progress stores all key codes in 16-bit words. The structure of a key code is as follows:

- Bits 0–8 — Unique code assigned to the key
- Bit 9 — The **SHIFT** key modifier (512)
- Bit 10 — The **ALT** or **ESC** key modifier (1024)
- Bit 11 — The **CTRL** key modifier (2048)
- Bits 12–15 — Reserved bits

This structure allows Progress to respond to a wide variety of key combinations. For example, you can write a trigger that fires when the user presses **CTRL–SHIFT–ALT–F1.** Figure 6–3 shows how Progress generates key codes when different keys are pressed.

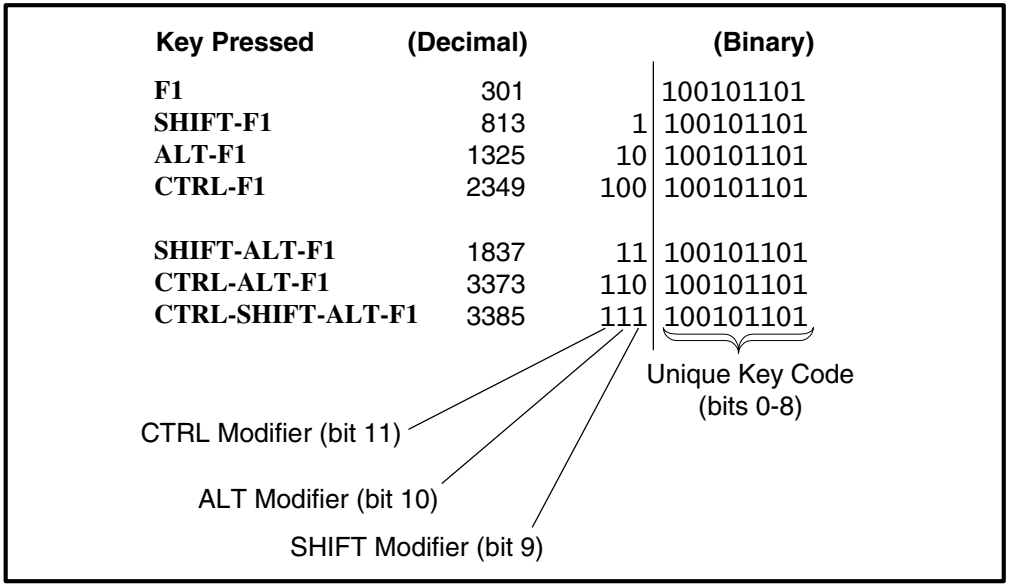


Figure 6–3: Key Codes

Progress uses this scheme wherever possible. However, some keys are assigned key codes that do not fit into this scheme. The following rules determine what key codes are outside of this scheme:

- If you press **CTRL** plus an alphabetic character or the (@, \,], ^, or _ special characters), Progress turns the CTRL key modifier off and maps to the appropriate character code. For example, **CTRL–A** is mapped to keycode 1, which accords with the ASCII standard (and therefore most 7–bit character code pages).
- If you press **SHIFT** plus a code page character, Progress turns off the SHIFT key modifier and maps to the appropriate character code. For example, **SHIFT–A** is equivalent to A, and A is mapped to keycode 65, which accords with the ASCII standard (and therefore most 7–bit character code pages). Also, many code page characters (for example, the comma) do not have uppercase equivalents; therefore, the SHIFT key modifier is turned off.
- If you press **ALT** plus an alphabetic character, Progress maps to the uppercase alphabetic characters. For example, **ALT–A** and **ALT–a** are both mapped to keycode 1089. The keycode 1089 corresponds to the keycode 65 (A) with the ALT key modifier turned on (65 + 1024 = 1089).

Progress provides more than one key label for certain key codes. The key labels in Table 6–2 are the preferred labels, but you can also use the labels in Table 6–3.

Table 6–2: Key Codes and Key Labels *(1 of 12)*

Key Code	Key Label
0	**CTRL–@**
1 through 7	**CTRL–A** through **CTRL–G**
8	**BACKSPACE**
9	**TAB**
10 through 12	**CTRL–J** through **CTRL–L**
13	**ENTER** (Windows) **RETURN** (UNIX)
14 through 26	**CTRL–N** through **CTRL–Z**
27	**ESC**

Table 6–2: Key Codes and Key Labels *(2 of 12)*

Key Code	Key Label
28	**CTRL–**
29	**CTRL–]**
30	**CTRL–^**
31	**CTRL–_**
32 through 126	Corresponding character
127	**DEL**
128 through 255	Corresponding extended character
300 through 399	**F0** through **F99**
400 through 499	**PF0** through **PF99**
501	**CURSOR–UP**
502	**CURSOR–DOWN**
503	**CURSOR–RIGHT**
504	**CURSOR–LEFT**
505	**HOME**
506	**END**
507	**PAGE–UP**
508	**PAGE–DOWN**
509	**BACK–TAB**
510	**INS**
511	**HELP–KEY**
512	**DELETE**
513	**EXECUTE**

Table 6–2: Key Codes and Key Labels *(3 of 12)*

Key Code	Key Label
514	**PAGE**
515	**FIND**
516	**INS–LINE**
517	**DEL–LINE**
518	**LINE–ERASE**
519	**PAGE–ERASE**
520	**SHIFT–BACKSPACE**
521	**SHIFT–TAB**
522	**EXIT**
535	**ERASE**
536	**WHITE**
537	**BLUE**
538	**RED**
539	**RESET**
541	**CTRL–BREAK**
551 through 560	**U1** through **U10**
609	**MOUSE–SELECT–UP**
610	**MOUSE–MOVE–UP**
611	**MOUSE–MENU–UP**
612	**MOUSE–EXTEND–UP**
617	**MOUSE–SELECT–DOWN**
618	**MOUSE–MOVE–DOWN**

Table 6–2: Key Codes and Key Labels *(4 of 12)*

Key Code	Key Label
619	**MOUSE–MENU–DOWN**
620	**MOUSE–EXTEND–DOWN**
625	**MOUSE–SELECT–CLICK**
626	**MOUSE–MOVE–CLICK**
627	**MOUSE–MENU–CLICK**
628	**MOUSE–EXTEND–CLICK**
640	**MOUSE–MOVE**
649	**MOUSE–SELECT–DBLCLICK**
650	**MOUSE–MOVE–DBLCLICK**
651	**MOUSE–MENU–DBLCLICK**
652	**MOUSE–EXTEND–DBLCLICK**
812 through 911	**SHIFT–F0** through **SHIFT–F99**
912 through 1011	**SHIFT– PF0** through **SHIFT–PF99**
1013	**SHIFT–CURSOR–UP**
1014	**SHIFT–CURSOR–DOWN**
1015	**SHIFT–CURSOR–RIGHT**
1016	**SHIFT–CURSOR–LEFT**
1017	**SHIFT–HOME**
1018	**SHIFT–END**
1019	**SHIFT–PAGE–UP**
1020	**SHIFT–PAGE–DOWN**
1021	**SHIFT–BACK–TAB**

Table 6–2: Key Codes and Key Labels *(5 of 12)*

Key Code	Key Label
1022	**SHIFT–INS**
1023	**SHIFT–HELP–KEY**
1024	**ALT–CTRL–@**
1025 through 1031	**ALT–CTRL–A** through **ALT–CTRL–G**
1032	**ALT–BACKSPACE**
1033	**ALT–TAB**
1034 through 1036	**ALT–CTRL–J** through **ALT–CTRL–L**
1037	**ALT–RETURN**
1038 through 1050	**ALT–CTRL–N** through **ALT–CTRL–Z**
1051	**ALT–ESC**
1052	**ALT–CTRL–**
1053	**ALT–CTRL–]**
1054	**ALT–CTRL–^**
1055	**ALT–CTRL–_**
1057 through 1150	**ALT**–plus corresponding character
1151	**ALT–DEL**
1152 through 1279	**ALT**–plus corresponding extended character
1324 through 1423	**ALT–F0** through **ALT–F99**
1424 through 1523	**ALT–PF0** through **ALT–PF99**
1525	**ALT–CURSOR–UP**
1526	**ALT–CURSOR–DOWN**
1527	**ALT–CURSOR–RIGHT**

Table 6–2: Key Codes and Key Labels *(6 of 12)*

Key Code	Key Label
1528	**ALT–CURSOR–LEFT**
1529	**ALT–HOME**
1530	**ALT–END**
1531	**ALT–PAGE–UP**
1532	**ALT–PAGE–DOWN**
1533	**ALT–BACK–TAB**
1534	**ALT–INS**
1535	**ALT–HELP–KEY**
1536	**ALT–DELETE**
1537	**ALT–EXECUTE**
1538	**ALT–PAGE**
1539	**ALT–FIND**
1540	**ALT–INS–LINE**
1541	**ALT–DEL–LINE**
1542	**ALT–LINE–ERASE**
1543	**ALT–PAGE–ERASE**
1544	**ALT–SHIFT–BACKSPACE**
1545	**ALT–SHIFT–TAB**
1546	**ALT–EXIT**
1549 through 1558	**ALT–U1** through **ALT–U10**
1559	**ALT–ERASE**
1560	**ALT–WHITE**

Table 6–2: Key Codes and Key Labels *(7 of 12)*

Key Code	Key Label
1561	**ALT–BLUE**
1562	**ALT–RED**
1563	**ALT–RESET**
1565	**ALT–CTRL–BREAK**
1575 through 1584	**ALT–U1** through **ALT–U10**
1663	**SHIFT–ALT–DEL**
1664 through 1791	**SHIFT–ALT–**plus corresponding extended character
1836 through 1935	**SHIFT–ALT–F0** through **SHIFT–ALT–F99**
1936 through 2035	**SHIFT–ALT–PF0** through **SHIFT–ALT–PF99**
2037	**SHIFT–ALT–CURSOR–UP**
2038	**SHIFT–ALT–CURSOR–DOWN**
2039	**SHIFT–ALT–CURSOR–RIGHT**
2040	**SHIFT–ALT–CURSOR–LEFT**
2041	**SHIFT–ALT–HOME**
2042	**SHIFT–ALT–END**
2043	**SHIFT–ALT–PAGE–UP**
2044	**SHIFT–ALT–PAGE–DOWN**
2045	**SHIFT–ALT–BACK–TAB**
2046	**SHIFT–ALT–INS**
2047	**SHIFT–ALT–HELP–KEY**
2056	**CTRL–BACKSPACE**
2057	**CTRL–TAB**

Table 6–2: Key Codes and Key Labels *(8 of 12)*

Key Code	Key Label
2075	**CTRL–ESC**
2080 through 2111	**CTRL**–plus corresponding character
2171 through 2174	**CTRL**–plus corresponding character
2175	**CTRL–DEL**
2209 through 2303	**CTRL**–plus corresponding extended character
2348 through 2447	**CTRL–F0** through **CTRL–F99**
2448 through 2547	**CTRL–PF0** through **CTRL–PF99**
2549	**CTRL–CURSOR–UP**
2550	**CTRL–CURSOR–DOWN**
2551	**CTRL–CURSOR–RIGHT**
2552	**CTRL–CURSOR–LEFT**
2553	**CTRL–HOME**
2554	**CTRL–END**
2555	**CTRL–PAGE–UP**
2556	**CTRL–PAGE–DOWN**
2557	**CTRL–BACK–TAB**
2558	**CTRL–INS**
2559	**CTRL–HELP–KEY**
2560	**CTRL–DELETE**
2561	**CTRL–EXECUTE**
2562	**CTRL–PAGE**
2563	**CTRL–FIND**

Table 6–2: Key Codes and Key Labels *(9 of 12)*

Key Code	Key Label
2564	**CTRL–INS–LINE**
2565	**CTRL–DEL–LINE**
2566	**CTRL–LINE–ERASE**
2567	**CTRL–PAGE–ERASE**
2568	**CTRL–SHIFT–BACKSPACE**
2569	**CTRL–SHIFT–TAB**
2570	**CTRL–EXIT**
2573	**CTRL–SHIFT–RETURN**
2583	**CTRL–ERASE**
2584	**CTRL–WHITE**
2585	**CTRL–BLUE**
2586	**CTRL–RED**
2587	**CTRL–RESET**
2559 through 2608	**CTRL–U1** through **CTRL–U10**
2860 through 2859	**CTRL–SHIFT–F0** through **CTRL–SHIFT–F99**
2960 through 3059	**CTRL–SHIFT–PF0** through **CTRL–SHIFT–PF99**
3061	**CTRL–SHIFT–CURSOR–UP**
3062	**CTRL–SHIFT–CURSOR–DOWN**
3063	**CTRL–SHIFT–CURSOR–RIGHT**
3064	**CTRL–SHIFT–CURSOR–LEFT**
3065	**CTRL–SHIFT–HOME**
3066	**CTRL–SHIFT–END**

Table 6–2: Key Codes and Key Labels *(10 of 12)*

Key Code	Key Label
3067	**CTRL–SHIFT–PAGE–UP**
3068	**CTRL–SHIFT–PAGE–DOWN**
3069	**CTRL–SHIFT–BACK–TAB**
3070	**CTRL–SHIFT–INS**
3071	**CTRL–SHIFT–HELP–KEY**
3104 through 3135	**CTRL–ALT–**plus corresponding character
3163	**CTRL–ALT–[**
3164	**CTRL–ALT–**
3165	**CTRL–ALT–]**
3166	**CTRL–ALT–^**
3167	**CTRL–ALT–_**
3168	**CTRL–ALT–‘**
3195 through 3198	**CTRL–ALT–**plus corresponding character
3199	**CTRL–ALT–DEL**
3233 through 3327	**CTRL–ALT–**plus corresponding extended character
3372 through 3471	**CTRL–ALT–F0** through **CTRL–ALT–F10**
3472 through 3571	**CTRL–ALT–PF0** through **CTRL–ALT–PF99**
3573	**CTRL–ALT–CURSOR–UP**
3574	**CTRL–ALT–CURSOR–DOWN**
3575	**CTRL–ALT–CURSOR–RIGHT**
3576	**CTRL–ALT–CURSOR–LEFT**
3577	**CTRL–ALT–HOME**

Table 6–2: Key Codes and Key Labels *(11 of 12)*

Key Code	Key Label
3578	CTRL–ALT–END
3579	CTRL–ALT–PAGE–UP
3580	CTRL–ALT–PAGE–DOWN
3581	CTRL–ALT–BACK–TAB
3582	CTRL–ALT–INS
3583	CTRL–ALT–HELP–KEY
3585	CTRL–ALT–EXECUTE
3586	CTRL–ALT–PAGE
3587	CTRL–ALT–FIND
3588	CTRL–ALT–INS–LINE
3589	CTRL–ALT–DEL–LINE
3590	CTRL–ALT–LINE–ERASE
3591	CTRL–ALT–PAGE–ERASE
3592	CTRL–SHIFT–ALT–BACKSPACE
3593	CTRL–SHIFT–ALT–TAB
3594	CTRL–ALT–EXIT
3597	CTRL–SHIFT–ALT–RETURN
3607	CTRL–ALT–ERASE
3608	CTRL–ALT–WHITE
3609	CTRL–ALT–BLUE
3610	CTRL–ALT–RED
3611	CTRL–ALT–RESET

Table 6–2: Key Codes and Key Labels *(12 of 12)*

Key Code	Key Label
3623 through 3632	**CTRL–ALT–U1** through **CTRL–ALT–U10**
3884 through 3983	**CTRL–SHIFT–ALT–F0** through **CTRL–SHIFT–ALT–F99**
3994 through 4083	**CTRL–SHIFT–ALT–PF0** through **CTRL–SHIFT–ALT–PF99**
4085	**CTRL–SHIFT–ALT–CURSOR–UP**
4086	**CTRL–SHIFT–ALT–CURSOR–DOWN**
4087	**CTRL–SHIFT–ALT–CURSOR–RIGHT**
4088	**CTRL–SHIFT–ALT–CURSOR–LEFT**
4089	**CTRL–SHIFT–ALT–HOME**
4090	**CTRL–SHIFT–ALT–END**
4091	**CTRL–SHIFT–ALT–PAGE–UP**
4092	**CTRL–SHIFT–ALT–PAGE–DOWN**
4093	**CTRL–SHIFT–ALT–BACK–TAB**
4094	**CTRL–SHIFT–ALT–INS**
4095	**CTRL–SHIFT–ALT–HELP–KEY**

Table 6–2 shows the key labels returned for each key code by the KEYLABEL function. In addition to these key labels, some key codes have alternative labels. Although the KEYLABEL function does not return these values, if you can pass any of these labels to the KEYCODE function, the corresponding key code is returned. Table 6–3 lists the alternate key labels.

Table 6–3: Alternate Key Labels *(1 of 2)*

Key Code	Alternate Key Label
7	**BELL**
8	**BS**

Table 6–3: Alternate Key Labels *(2 of 2)*

Key Code	Alternate Key Label
10	**LINEFEED, LF**
12	**FORMFEED, FF**
13	**RETURN** (Windows) **ENTER** (UNIX)
27	**ESCAPE**
127	**CANCEL**
501	**UP, UP–ARROW**
502	**DOWN, DOWN–ARROW**
503	**RIGHT, RIGHT–ARROW**
504	**LEFT, LEFT–ARROW**
505	**ESC–H**
507	**PGUP, PREV–PAGE, PREV–SCRN**
508	**PGDN, NEXT–PAGE, NEXT–SCRN**
509	**SHIFT–TAB** (UNIX), **BACK–TAB** (Windows)
510	**INSERT, INS–CHAR, INS–C, INSERT–HERE**
512	**DEL–CHAR, DELETE–CHAR, DEL–C**
516	**INS–L, LINE–INS**
517	**DEL–L, LINE–DEL**

6.3 Key Functions

Table 6–4 shows the keys that Progress maps to each key function in each user interface. Note that graphical interfaces that have full mouse control do not require as many navigation keys. Note also that in character interfaces, the precise mapping depends on the terminal type, and that UNIX in particular provides many terminal types. Table 6–4 lists possible mappings.

NOTE: If you enter **CTRL–ALT–SHIFT–F1** from a Windows Progress client, graphical or character, Progress displays a window that tells what Progress version you are running.

Table 6–4: Progress Key Functions *(1 of 6)*

Key Function	Key Label		
	Windows Graphical Interface	UNIX Character Interface	Windows Character Interface
ABORT	–	CTRL–\	–
APPEND–LINE	–	CTRL–A	CTRL–A
BACK–TAB	SHIFT–TAB	CTRL–U	CTRL–U SHIFT–TAB
BACKSPACE	BACKSPACE	BACKSPACE CTRL–H DEL–CHAR	BACKSPACE CTRL–H
BELL	CTRL–G BELL	BELL	–
BLOCK	–	CTRL–V	CTRL–V
BOTTOM–COLUMN	–	ESC–CTRL–B	–
BREAK–LINE	–	ESC–B	–
CANCEL–PICK	–	ESC–CTRL–X	–
CHOICES	–	ESC–BACKSPACE ESC–CTRL–H	–
CLEAR	–	F8 CTRL–Z	F8 CTRL–Z

Table 6–4: Progress Key Functions *(2 of 6)*

Key Function	Key Label		
	Windows Graphical Interface	UNIX Character Interface	Windows Character Interface
CLOSE	F8	ESC–Z F8	CTRL–ALT–Z F8
COMPILE	SHIFT–F2	ESC–P	CTRL–ALT–P
COPY	–	F11 ESC–C	F11 CTRL–ALT–C
CURSOR–DOWN	CURSOR–DOWN	CURSOR–DOWN CTRL–J	CURSOR–DOWN CTRL–J
CURSOR–LEFT	CURSOR–LEFT	CURSOR–LEFT CTRL–O	CURSOR–LEFT CTRL–O
CURSOR–RIGHT	CURSOR–RIGHT	CURSOR–RIGHT CTRL–L	CURSOR–RIGHT CTRL–L
CURSOR–UP	CURSOR–UP	CURSOR–UP CTRL–K	CURSOR–UP CTRL–K
CUT	CTRL–X	F10 ESC–X	F10 CTRL–ALT–X
DEFAULT–POP–UP	SHIFT–F10	ESC–U	SHIFT–F4 CTRL–ALT–U
DELETE–CHARACTER	DEL	DEL DELETE	DELETE
DELETE–COLUMN	–	ESC–CTRL–Z	–
DELETE–END–LINE	–	ESC–K	CTRL–ALT–K
DELETE–FIELD	–	ESC–CTRL–D	–
DELETE–LINE	–	CTRL–D	CTRL–D
DELETE–WORD	–	ESC–D	CTRL–ALT–D
EDITOR–BACKTAB	–	CTRL–B	CTRL–B

Table 6–4: Progress Key Functions *(3 of 6)*

Key Function	Key Label		
	Windows Graphical Interface	UNIX Character Interface	Windows Character Interface
EDITOR–TAB	–	CTRL–G TAB	CTRL–G TAB
END	END	END ESC–.	END
END–ERROR	ESC	F4 CTRL–E	F4 ESC CTRL–E
ENTER–MENUBAR	ALT	F3 PF3 ESC–M	F3 ALT
EXIT	–	ESC–Q	CTRL–ALT–Q
FIND	CTRL–F	CTRL–F	CTRL–F
FIND–NEXT	F9	ESC–F	CTRL–ALT–F
FIND–PREVIOUS	SHIFT–F9	ESC–I	CTRL–ALT–I
GET	F3	F5 ESC–O	F5 CTRL–ALT–O
GO	F2	F1 CTRL–X	F1 CTRL–X
GOTO	CTRL–G	ESC–G	CTRL–ALT–G
HELP	F1	ESC–?	–
HOME	HOME	ESC–, ESC–H	HOME
INSERT–COLUMN	–	ESC–CTRL–N	–
INSERT–FIELD	–	ESC–CTRL–G	–
INSERT–FIELD–DATA	–	ESC–CTRL–F	–

Table 6–4: Progress Key Functions *(4 of 6)*

Key Function	Key Label		
	Windows Graphical Interface	UNIX Character Interface	Windows Character Interface
INSERT–FIELD–LABEL	–	ESC–CTRL–E	–
INSERT–MODE	INSERT	F9 CTRL–T	INSERT F9 CTRL–T
LEFT–END	HOME	ESC–CURSOR–LEFT	ALT–CURSOR–LEFT
MAIN–MENU	–	ESC–RETURN ESC–CTRL–M	–
MOVE	–	ESC–CTRL–V	–
NEW	SHIFT–F3	ESC–N	CTRL–ALT–N
NEW–LINE	–	CTRL–N	CTRL–N
NEXT–ERROR	–	ESC–E	–
NEXT–FRAME	F6	ESC–TAB ESC–CTRL–I	–
NEXT–WORD	–	CTRL–W	–
OPEN–LINE–ABOVE	–	ESC–L	CTRL–ALT–L
OPTIONS	–	ESC–CTRL–O	–
PAGE–DOWN	PAGE–DOWN PGDN NEXT–PAGE NEXT–SCRN	ESC–CURSOR– DOWN	PAGE–DOWN
PAGE–LEFT	–	ESC–W	–
PAGE–RIGHT	–	ESC–Y	–

Table 6–4: Progress Key Functions *(5 of 6)*

Key Function	Key Label		
	Windows Graphical Interface	UNIX Character Interface	Windows Character Interface
PAGE–UP	PAGE–UP PGUP PREV–PAGE PREV–SCRN	ESC–CURSOR–UP	PAGE–UP
PASTE	CTRL–V	F12 ESC–V	F12 CTRL–ALT–V
PICK	–	ESC–CTRL–P	–
PICK–AREA	–	ESC–CTRL–W	–
PICK–BOTH	–	ESC–CTRL–Q	–
PREV–FRAME	SHIFT–F6	ESC–CTRL–U	CTRL–SHIFT–TAB
PREV–WORD	–	CTRL–P	CTRL–P
PUT	F6	F6 ESC–S	F6
RECALL	–	F7 CTRL–R	–
REPLACE	–	ESC–R	–
REPORTS	–	ESC–CTRL–A	–
RESUME–DISPLAY	–	CTRL–Q	–
RETURN	ENTER RETURN CTRL–M	RETURN CTRL–M	ENTER CTRL–M
RIGHT–END	–	ESC–CURSOR–RIGHT	ALT–CURSOR–RIGHT
SAVE–AS	SHIFT–F6	ESC–A	CTRL–ALT–A
SCROLL–LEFT	–	ESC–CTRL–L	–
SCROLL–MODE	–	ESC–T	CTRL–ALT–T

Table 6–4: Progress Key Functions *(6 of 6)*

Key Function	Key Label		
	Windows Graphical Interface	UNIX Character Interface	Windows Character Interface
SCROLL–RIGHT	–	ESC–CTRL–R	–
SETTINGS	–	ESC–CTRL–@	–
STOP	CTRL–BREAK	CTRL–C	CTRL–C
STOP–DISPLAY	–	CTRL–S	–
TAB	TAB	TAB CTRL–I	TAB CTRL–I
TOP–COLUMN	–	ESC–CTRL–T	–
UNIX–END	–	CTRL–\	–

6.4 Changing the Function of a Key

You can globally change Progress key functions by modifying your environment. For more information on modifying your environment, see the chapter on colors and fonts in this book and the chapter on user interface environments in the *Progress Client Deployment Guide*.

Although Progress has defined functions for several of the keys on your keyboard, you can redefine those keys to perform other functions within an application. In addition, you can assign functions to any of the other keyboard keys.

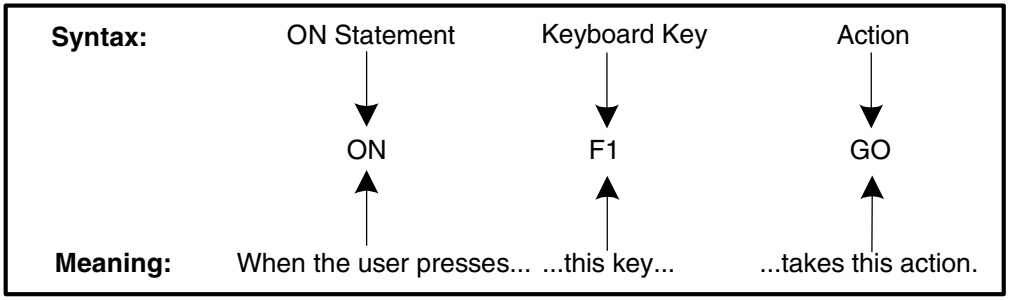
Suppose the user is accustomed to pressing **F2** to get help information and **F1** to signal that they are finished entering data. Although Progress usually treats **F2** as GO and **F1** as HELP, you can switch the functions of those keys as demonstrated in the following procedure:

p-action.p

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

FOR EACH customer:
   DISPLAY cust-num.
   UPDATE Name credit-limit sales-rep.
END.
```

In this procedure, the ON statements redefine the function of **F1**, **F2**, and **CTRL–X**. Run this procedure, then press **F1**. You can see that **F1** now performs the GO function, normally performed by **F2**. If you press **CTRL–X**, Progress rings the terminal bell. The new key definitions are in effect for the duration of the session, unless you redefine them. Also, on UNIX, any key label you use in an ON statement must have an entry in the PROTERMCAP file for your terminal.

When you use the ON statement, you use the name of the key whose function you are redefining, followed by the action you want to take when the user presses that key.



The `p-action.p` procedure uses three actions: GO, HELP, and BELL. There are many other actions you can use when redefining the function of a key. Table 6–5 lists these actions.

Table 6–5: Actions You Assign to Keys

Action	Default Key
BACKSPACE	**BACKSPACE**
BACK–TAB	**SHIFT–TAB, CTRL–U, CODE–TAB**
BELL	–
CLEAR	**F8, CTRL–Z, CODE–Z**
CURSOR–UP	**×, CTRL–K**
CURSOR–DOWN	**±, CTRL–J**
CURSOR–LEFT	**?, CTRL–H**
CURSOR–RIGHT	**?, CTRL–L**
DELETE–CHARACTER	**DEL**
ENDKEY	–
END–ERROR	**F4, CTRL–E, CODE–E, ESC** (Windows)
ERROR	–
GO	**F2, CTRL–X, CODE–X**
HELP	**F1, HELP, CTRL–W, CODE–W**
HOME	**HOME, ESC–H** (UNIX)
INSERT–MODE	**F3, INSERT, CTRL–T, OVERTYPE**
RECALL	**F7, CTRL–R, CODE–R**
RETURN	**RETURN**
STOP	**CTRL–BREAK** (Windows), **CTRL–C** (UNIX)
TAB	**TAB, CTRL–I**

6.5 Using Mouse Buttons and Events

Progress uses a logical (portable) model to reference mouse button input. This ensures that your application works in whatever operating environment it runs. Progress also provides access to the physical mouse input of your operating environment if you need it. Both forms of input are available as events in a manner similar to keyboard events.

6.5.1 Portable and Physical Buttons

Progress supports four portable mouse buttons:

- SELECT — You can select or choose an object by pointing to it and clicking this mouse button. Any previously selected object becomes deselected.
- EXTEND — You can toggle the selection state of an object by pointing to it and clicking this button. This has no effect on any other object; therefore, you can use the EXTEND button to toggle selection individually on more than one object.
- MENU — If an object has an associated pop-up menu, you can activate that menu by clicking this button.
- MOVE — By pressing and holding this button you can drag an object on the screen.

Although Progress supports four buttons, the standard mouse used with Windows has only two buttons. Therefore, some physical mouse buttons have double functions, or, you must use control keys with one or more buttons. Table 6–6 shows the mappings between the Progress portable mouse buttons and the physical mouse buttons on Windows.

Table 6–6: Mouse Buttons on Windows

Portable Button	Windows
SELECT	LEFT mouse button
EXTEND	**CTRL** with LEFT mouse button
MENU	RIGHT mouse button
MOVE	LEFT mouse button

Progress supports two main classes of mouse events—portable and three-button events. You can use portable mouse events to associate triggers with logical actions of any mouse. You can use the three-button mouse events to associate triggers with specific physical actions of a three-button mouse. The names of the portable mouse events come from the mouse key labels

listed in Table 6–6 (for example, MOUSE–SELECT–CLICK). They also correspond to the names of the portable mouse buttons used to generate them. The names of the three-button mouse events correspond to the physical buttons that generate them on a three-button mouse (for example, LEFT–MOUSE–CLICK). For a complete description of the names and functions of the portable and three-button mouse events, see the *Progress Language Reference*.

Both portable and three-button mouse events divide into two subclasses—low-level and high-level mouse events. Low-level mouse events are generated by the simplest mouse button actions, while high-level events are generated by more complex actions.

6.5.2 Specifying Mouse Events

When choosing the events to associate with a trigger, use portable mouse events whenever possible. If you use these events, Progress automatically maps the event to the mouse key the native window system uses to perform the same event. For example, if Progress supports a future window system that uses the right button as the selection button, the MOUSE–SELECT–CLICK event will automatically map to the right button for that system, while still mapping to the left button for Windows.

Portable and Three-Button Event Priority

If you use three-button mouse events, they take priority over any corresponding portable event. For example, if you define a MOUSE–SELECT–CLICK and a LEFT–MOUSE–CLICK trigger for the same widget, Progress only fires the three-button trigger. (In this case, it only fires the LEFT–MOUSE–CLICK trigger.)

Low-level and High-level Events

The low-level category consists of those events triggered by mouse button motion in a single direction, such as down or up. The high-level category consists of those events triggered by more complex mouse button motion, such as click or double-click. For Progress, corresponding low-level and high-level mouse events are analogous to equivalent key label and key function events. Like key label and key function events, or three-button and portable mouse events, low-level mouse events take priority over corresponding high-level mouse events. For example, if you define a MOUSE–SELECT–UP and MOUSE–SELECT–CLICK trigger on the same widget, only the MOUSE–SELECT–UP trigger executes.

Like portable events, use high-level mouse events exclusively whenever possible. If you must use low-level events, do not mix low-level and high-level event triggers on the same widget. The processing of both classes of events on the same widget can lead to unintended results.

While Progress always recognizes corresponding low-level and high-level events, it executes only one trigger for both. (The same is true for corresponding key label and key function events, as well as portable and three-button mouse events). Progress looks first for a trigger on a

low-level event. If there is no trigger on a low-level event, Progress looks for a trigger on a high-level event. Once it finds and fires a trigger, it waits for the next event. However, each graphic interface may recognize and pass high-level events to Progress after recognizing a different combination of low-level events. For example, one system might recognize a double-click on a down and another system might recognize the double-click on an up mouse event. This causes mixed event sequences to differ significantly from interface to interface.

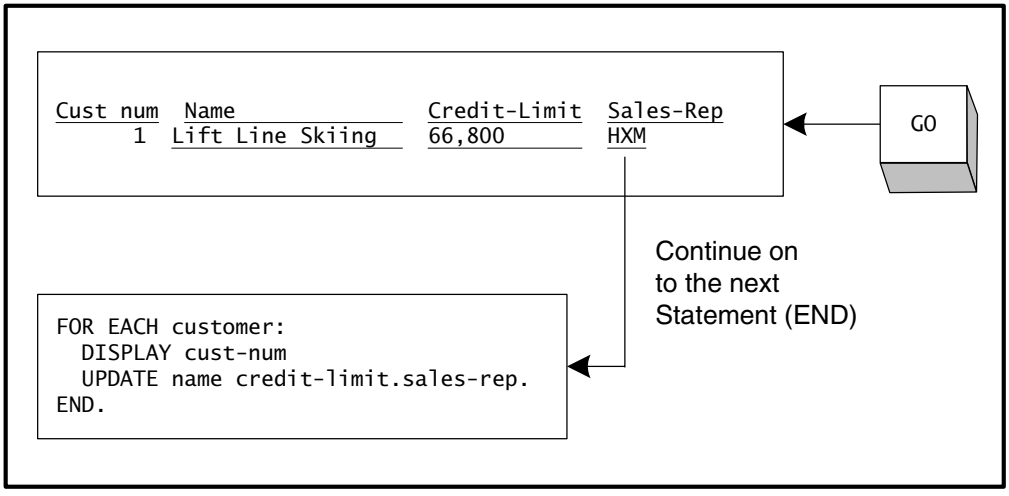
Thus, much like using pixels for frame and window layout, the mixing of low- and high-level mouse events creates portability problems between graphic interfaces. Furthermore, there is no guarantee that the expected event sequences in the same graphic interface might not change in future versions of that interface.

6.6 Telling Progress How to Continue Processing

When you modify a field or variable, pressing **GO** tells Progress to accept the data in all the modified fields and variables in the current statement and to go on to the next statement in the procedure.

As the following figure shows, if you press **GO** while the cursor is in the name, credit–limit, or sales–rep fields, Progress continues on to the next statement in the procedure, which is END. The procedure then returns to the beginning of the FOR EACH loop.

p-intro.p



You might want to define function keys that tell Progress to continue processing data a certain way. You can use the GO–ON phrase with the SET or UPDATE statement to do this, as shown in the following procedure:

p-gon1.p

```
DISPLAY "You may update each customer." SKIP
        "After making your changes, press one of:" SKIP(1)
        KBLABEL("GO") + "  - Make the change permanent" FORMAT "x(40)"
           SKIP
        KBLABEL("END-ERROR") + "  - Undo changes and exit" FORMAT "x(40)"
           SKIP
        "F8  - Undo changes and try again" SKIP
        "F10 - Find next customer" SKIP
        "F12 - Find previous customer"
WITH CENTERED FRAME ins.

FIND FIRST Customer.upd-loop:
REPEAT:
   UPDATE Cust-num Name Address Address2 City St
        GO-ON(F8 F10 F12) WITH 1 DOWN CENTERED.

   CASE LASTKEY:
      WHEN KEYCODE("F8") THEN
         UNDO upd-loop, RETRY upd-loop.
      WHEN KEYCODE("F10") THEN
         FIND NEXT Customer.
      WHEN KEYCODE("F12") THEN
         FIND PREV Customer.
   END CASE.
END.
```

In this example, if the user presses **F8**, **F10**, or **F12** while updating the customer data, the procedure immediately goes on to the next statement in the procedure. Let's take a closer look at this procedure.

Any key you can press while running a Progress procedure has a code, a function, and a label associated with it. The code of a key is an integer value that Progress uses to identify that key. For example, the code of **F1** is 301. The function of a key is the work that Progress does when you press the key. For example, the function of the **F1** key may be HELP. The label of a key is the actual label that appears on the keyboard key. The label of the **F1** key is F1.

As shown earlier, you can use the KEYLABEL, KEYCODE, KEYFUNCTION, and KBLABEL functions to convert key labels, key codes, and key functions. In addition to these functions, the LASTKEY function returns the key code of the last key pressed.

You can use the functions described in this table to monitor the keys being pressed, as in this example:

p-keys.p

```
REPEAT:
  DISPLAY "Press any key".
  READKEY.
  DISPLAY LASTKEY LABEL "Key Code"
          KEYLABEL(LASTKEY) LABEL "Key Label"
          KEYFUNCTION(LASTKEY) LABEL "Key Function"
                               FORMAT "x(12)".
  IF KEYFUNCTION(LASTKEY) = "end-error" THEN LEAVE.
END.
```

Run procedure `p-keys.p` to see how the different keys you press translate into key codes, key labels, and key functions.

Now, run the `p-gon1.p` procedure. You see a screen similar to the one shown in the following figure:

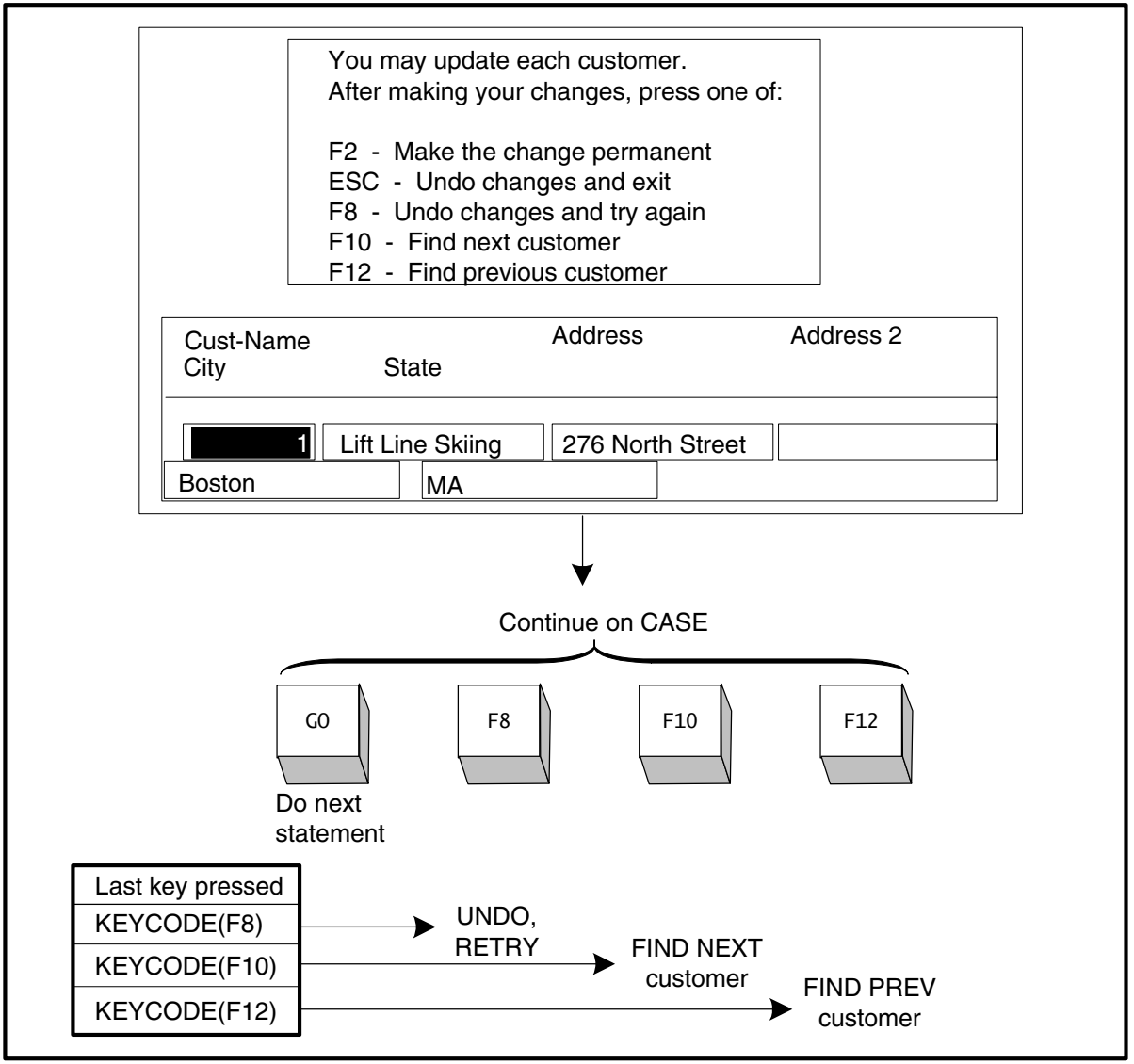


Figure 6–4: The p-gon1.p Procedure

While updating the customer information, press **F9**, **F10**, **F11**, or use either of the standard techniques to signal the end of data entry, Progress goes on to the next statement in the procedure. If you press any other key, Progress does not continue on to the next statement in the procedure, but instead performs the data entry operation associated with that key. If you press **END–ERROR**, Progress performs the default ENDKEY processing of UNDO, LEAVE. See "Condition Handling and Messages," for more information on ENDKEY processing.

If Progress does continue on to the next statement in the procedure, the CASE statement determines the action to take by checking the value of the last key pressed.

The procedure p-gon2.p shows how you can achieve the same functionality in an event-driven application:

p-gon2.p *(1 of 2)*

```
FORM
   Customer.Cust-num Customer.Name Customer.Address
   Customer.Address2 Customer.City Customer.State
   WITH CENTERED FRAME upd-frame.

DISPLAY "You may update each customer." SKIP
        "After making your changes, press one of:" SKIP(1)
        KBLABEL("GO") + "  - Make the change permanent" FORMAT "x(40)"
           SKIP
        KBLABEL("END-ERROR") + "  - Undo changes and exit" FORMAT "x(40)"
           SKIP
        "F8  - Undo changes and try again" SKIP
        "F10 - Find next customer" SKIP
        "F12 - Find previous customer"
        WITH CENTERED FRAME ins.

ON F8 ANYWHERE
   DO:
     DISPLAY Customer.Cust-num Name Address City State
        WITH CENTERED FRAME upd-frame.
   END.

ON F10 ANYWHERE
   DO:
     APPLY "GO" TO FRAME upd-frame.
     FIND NEXT Customer.
     DISPLAY Cust-num Name Address Address2 City State
        WITH CENTERED FRAME upd-frame.
   END.

ON F12 ANYWHERE
   DO:
     APPLY "GO" TO FRAME upd-frame.
     FIND PREV Customer.
     DISPLAY Cust-num Name Address Address2 City State
        WITH CENTERED FRAME upd-frame.
   END.
```

p-gon2.p *(2 of 2)*

```
ON GO OF FRAME upd-frame
   ASSIGN FRAME upd-frame Customer.Cust-num Name Address Address2
          City State.

ENABLE ALL WITH FRAME upd-frame.
FIND FIRST Customer.
APPLY "F8" TO FRAME upd-frame.

WAIT-FOR END-ERROR OF FRAME upd-frame.
```

Use the ANYWHERE option of the ON statement to set up triggers that execute no matter which widget has input focus. In `p-gon2.p`, the ANYWHERE option is used to assign triggers to function keys.

NOTE: In a larger program, you must be careful of the scope of the ANYWHERE trigger.

6.7 Monitoring Keystrokes During Data Entry

Progress provides two methods to monitor keystrokes—editing blocks and user interface triggers.

6.7.1 Editing Blocks

Prior to Progress Version 7 and user interface triggers, editing blocks were the only method available to monitor individual keystrokes. An editing block is part of an UPDATE, SET, or PROMPT–FOR statement that allows the programmer to read and process each keystroke individually. Editing blocks are used for two purposes:

- To enable a few special keys in one or more fields while allowing normal data entry.
- To disallow normal data entry in one or more fields and instead allow the use of only a few special keys to change the value.

Each type of editing block can now be replaced with user interface triggers.

6.7.2 User Interface Triggers

Progress allows you to access all keystrokes as events. You can intercept these events using the ON statement or the trigger phrase of any statement that declares a user interface widget. For each intercepted event, you can provide a user interface trigger to implement any Progress action. This action can add to, and often replace, the default action associated with the event.

All Progress key labels and key functions are valid events. You can specify a keyboard event in the ON statement or trigger phrase by:

- Using the key label name, such as f2.
- Using the key function name, such as go.

Key label events take priority over corresponding key function events. For example, if you specify a trigger for the TAB key function event and another trigger on the same widget for the CTRL–I key label event, only the CTRL–I trigger executes.

Where a key label event corresponds to a key function, use the key function event whenever possible. Key functions are more portable. If you must reference key label events, be sure not to also reference corresponding key function events for the same widget.

6.7.3 Using Editing Blocks and User Interface Triggers

This procedure uses an editing block to enable a special key while allowing normal data entry:

p-kystka.p

```
DEFINE FRAME cust-frame
     Customer.Cust-num SKIP Customer.name SKIP Customer.address SKIP
     Customer.address2 SKIP Customer.city Customer.state SKIP
     Customer.sales-rep HELP "To see a list of values, press F6."
     WITH DOWN SIDE-LABELS.

REPEAT WITH FRAME cust-frame:
   PROMPT-FOR Customer.Cust-num.
   FIND Customer USING Cust-num.
   UPDATE Name Address Address2 City State Customer.Sales-rep
     EDITING:
        READKEY.
        IF FRAME-FIELD = "sales-rep" AND LASTKEY = KEYCODE("F6")
        THEN DO:
             FOR EACH Salesrep:
                DISPLAY Salesrep.Sales-rep
                   WITH NO-LABELS 9 DOWN COLUMN 60 ROW 5.
             END.
        END.
        ELSE DO:
             APPLY LASTKEY.
        END.
     END.
END.
```

This procedure uses an EDITING block on the UPDATE statement. This means that the EDITING block processes every keystroke the user enters while that UPDATE statement is executing. Within the EDITING block, the READKEY statement reads a keystroke from the user. The associated key code is automatically stored as LASTKEY. The procedure uses LASTKEY and the FRAME–FIELD function to determine whether the key is **F6**, and whether it was entered from within the Sales–rep field. If so, it displays a list of all known sales reps. If the key is not **F6**, or the current field is not the Sales–rep field, then the APPLY statement is executed. This causes Progress to handle the keystroke normally.

This procedure uses a user interface trigger in place of the EDITING block to achieve the same functionality:

p-kystkb.p

```
DEFINE FRAME cust-frame
     Customer.Cust-num SKIP Customer.name SKIP Customer.address SKIP
     Customer.address2 SKIP Customer.city Customer.state SKIP
     Customer.sales-rep HELP "To see a list of values, press F6."
     WITH DOWN SIDE-LABELS.

ON F6 OF Customer.Sales-rep
   DO:
      FOR EACH salesrep:
        DISPLAY Salesrep.Sales-rep
            WITH NO-LABELS 9 DOWN COLUMN 60 ROW 5.
      END.
   END.

REPEAT WITH FRAME cust-frame:
   PROMPT-FOR Customer.Cust-num.
   FIND Customer USING Cust-num.
   UPDATE Name Address Address2 City State Sales-rep.
END.
```

Use a trigger in place of the EDITING block to produce simpler, cleaner code. It would be easier to rewrite this code to be more event driven. First, replace the UPDATE statement with ENABLE, WAIT–FOR, and ASSIGN. You then might remove the REPEAT block and define a button that the user can select to move to the next Customer record. You might also replace the display of sales reps with a selection list or browse widget.

The following procedure uses an EDITING block to disallow normal data entry on a field and allow only a special key:

p-kystk.p

```
DEFINE FRAME cust-frame
   Customer.Cust-num SKIP Customer.Name SKIP Customer.Address SKIP
   Customer.Address2 SKIP Customer.City Customer.State SKIP
   Customer.Sales-rep HELP "Use the space bar to scroll the values."
   WITH SIDE-LABELS.

REPEAT WITH FRAME cust-frame:
   PROMPT-FOR Customer.Cust-num.
   FIND Customer USING Cust-num.
   UPDATE Name Address City State Sales-rep
       EDITING:
          READKEY.
          IF FRAME-FIELD <> "Sales-rep"
          THEN DO:
             APPLY LASTKEY.
          END.
          ELSE DO:
             IF LASTKEY = KEYCODE(" ")
             THEN DO:
                FIND NEXT Salesrep NO-ERROR.
                IF NOT AVAILABLE Salesrep
                THEN FIND FIRST Salesrep.

                DISPLAY Salesrep.Sales-rep @ Customer.Sales-rep.
             END.
             ELSE IF LOOKUP(KEYFUNCTION(LASTKEY),
                            "TAB,BACK-TAB,GO,END-ERROR") > 0
                  THEN APPLY LASTKEY.
                  ELSE BELL.
          END.
       END.
END.
```

Like `p-kystka.p`, this procedure uses an EDITING block on the UPDATE statement. In the EDITING block, it uses READKEY, LASTKEY, and FRAME–FIELD to obtain and analyze a keystroke. If the keystroke is not in the Sales–rep field, it is processed normally. Within the Sales–rep field, only the spacebar is treated specially and only the TAB, BACK–TAB, GO, and END–ERROR key functions are treated normally. If the user types any other key within the field, the terminal bell sounds. When the user presses the spacebar in the Sales–rep field, the value of that field and the Sales–region field change.

The following procedure uses triggers to accomplish the same thing:

p-kystk2.p

```
DEFINE FRAME cust-frame
     Customer.Cust-num SKIP Customer.Name SKIP Customer.Address SKIP
     Customer.Address2 SKIP Customer.City SKIP Customer.State SKIP
     Customer.Sales-rep HELP "Use the space bar to scroll the values"
     WITH SIDE-LABELS.

ON " " OF Customer.Sales-rep
   DO:
     FIND NEXT Salesrep NO-LOCK NO-ERROR.
     IF NOT AVAILABLE Salesrep
     THEN FIND FIRST Salesrep NO-LOCK.
     DISPLAY Salesrep.Sales-rep @ Customer.Sales-rep
          WITH FRAME cust-frame.
     RETURN NO-APPLY.
   END.

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

REPEAT WITH FRAME cust-frame:
   PROMPT-FOR Customer.Cust-num.
   FIND Customer USING Cust-num.
   FIND Salesrep OF Customer NO-LOCK.
   UPDATE Name Address Address2 City State Customer.Sales-rep.
END.
```

Note the use of RETURN NO–APPLY in both the ANY–PRINTABLE and spacebar trigger. This is equivalent to omitting the APPLY LASTKEY statement in an EDITING block. Thus, the spacebar trigger brings the next Sales–rep into view, without also inserting a space character in the field.

Note also that in `p-kystk2.p` the ANY–PRINTABLE trigger rejects all keys that enter data characters other than a space. This works together with the spacebar trigger to allow only the spacebar and navigation keys (TAB, etc.) in the field.

Sometimes an EDITING block defines a special key that applies for all active fields:

p-kystk3.p

```
DEFINE FRAME cust-frame
   Customer.Cust-num SKIP Customer.Name SKIP Customer.Address SKIP
   Customer.Address2 SKIP Customer.City Customer.State SKIP
   Customer.Sales-rep WITH SIDE-LABELS.

REPEAT WITH FRAME cust-frame:
   PROMPT-FOR Customer.Cust-num.
   FIND Customer USING Cust-num.
   MESSAGE "Press F6 to see the previous Customer.".
   UPDATE Name Address Address2 City State Sales-rep
       EDITING:
          READKEY.
          IF KEYLABEL(LASTKEY) = "F6"
          THEN DO:
             FIND PREV Customer NO-ERROR.
             IF NOT AVAILABLE Customer
             THEN FIND FIRST Customer.

             DISPLAY Cust-num Name Address City State Sales-rep
                WITH FRAME cust-frame.
          END.
          ELSE APPLY LASTKEY.
       END.
   HIDE MESSAGE.
END.
```

In `p-kystk3.p`, the EDITING block defines a special key, **F6**, that is available in all fields of the UPDATE statement. When the user presses this key, the previous Customer record displays.

The following procedure uses a trigger to achieve the same result:

p-kystk4.p

```
DEFINE FRAME cust-frame
   Customer.Cust-num SKIP Customer.Name SKIP Customer.Address SKIP
   Customer.Address2 SKIP Customer.City Customer.State SKIP
   Customer.Sales-rep WITH SIDE-LABELS.

ON F6 ANYWHERE
  DO:
     IF FRAME-FIELD = "Cust-num"
     THEN RETURN NO-APPLY.

     FIND PREV Customer NO-ERROR.
     IF NOT AVAILABLE Customer
     THEN FIND FIRST Customer.

     DISPLAY Cust-num Name Address City State Sales-rep
        WITH FRAME cust-frame.
  END.

REPEAT WITH FRAME cust-frame:
   PROMPT-FOR Customer.Cust-num.
   FIND Customer USING Cust-num.
   MESSAGE "Press F6 to see the previous Customer.".
   UPDATE Name Address Address2 City State Sales-rep.
   HIDE MESSAGE.
END.
```

If you define a trigger to be active ANYWHERE, then it applies to all widgets. In `p-kystk4.p`, the F6 trigger executes whenever the user presses **F6** while input is enabled. Within the trigger, the FRAME–FIELD function determines whether the t rigger executes from the UPDATE or PROMPT–FOR statement. If it is the PROMPT–FOR statement, then **F6** is ignored; if it is the UPDATE statement, the previous Customer record displays.

NOTE: In a larger program, be careful of the scope of the ANYWHERE trigger. You are usually better off listing the specific widgets to which a trigger applies. For example, you could rewrite the ON statement in `p-kystrk4.p` as follows:

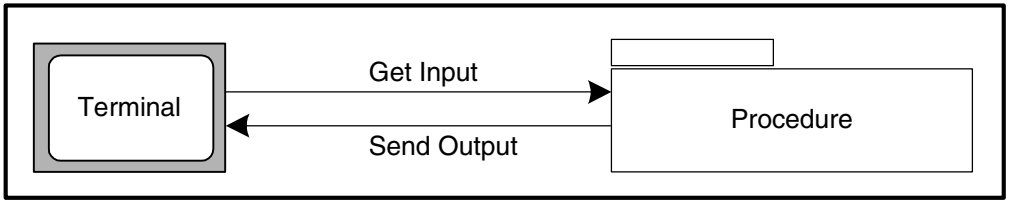
```
ON F6 OF Name, Address, Address2, City, State, Sales-rep
```

If you take this approach, you can remove the code that checks whether FRAME–FIELD is Cust–num within the trigger.

7

Alternate I/O Sources

Most of the procedures you have seen so far have used the terminal as the input source (the user types in data) and as the output destination (data is displayed to the user).



However, you've probably already thought of situations in which you might want to get data from and send data to locations other than the terminal. For example, you might want to send reports to your printer or an operating system file.

This chapter describes input/output and then covers the following topics:

- Changing the output destination and input source
- Defining additional input/output streams and sharing streams
- Importing data from a process and piping data to a processes
- Performing code page (character set) conversions
- Reading the contents of a directory
- Converting non-standard input files

7.1 Understanding Input and Output

When a Progress procedure gets input from the terminal, it uses an *input stream*. Similarly, when the procedure sends output to the terminal, it uses an *output stream*.

Every procedure automatically gets one input stream and one output stream. These are called the *unnamed streams*. By default, Progress assigns both of these unnamed streams to the terminal, as shown in Figure 7–1.

p-io.p

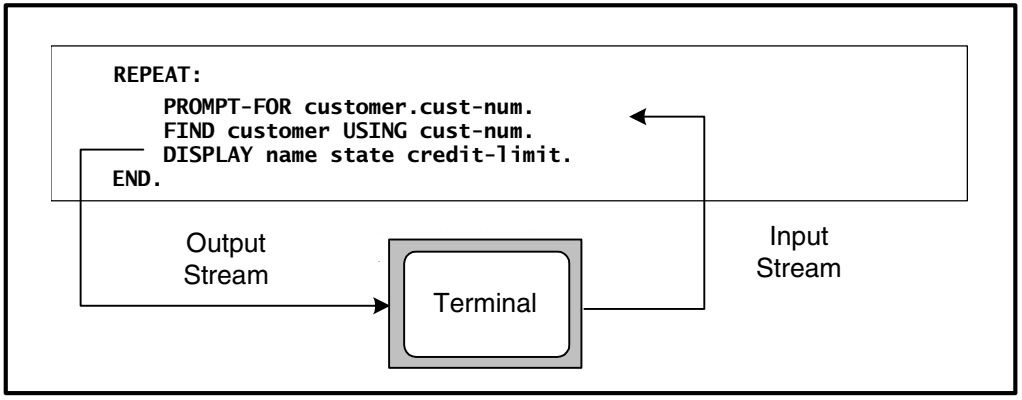


Figure 7–1: The Unnamed Streams

This procedure contains no special syntax about where to get and send data so Progress automatically assigns the input stream and the output stream to the terminal.

7.2 Changing the Output Destination

You use the OUTPUT TO statement to name a new destination for output. All statements that output data (such as DISPLAY or SET) use the new output destination. The possible destinations include:

- The printer
- An operating system file or device
- The terminal (by default)
- The system clipboard (Windows only)

7.2.1 Output to Printers

Figure 7–2 shows the output stream being redirected to a printer using the PRINTER option of the OUTPUT TO statement.

p-iop2.p

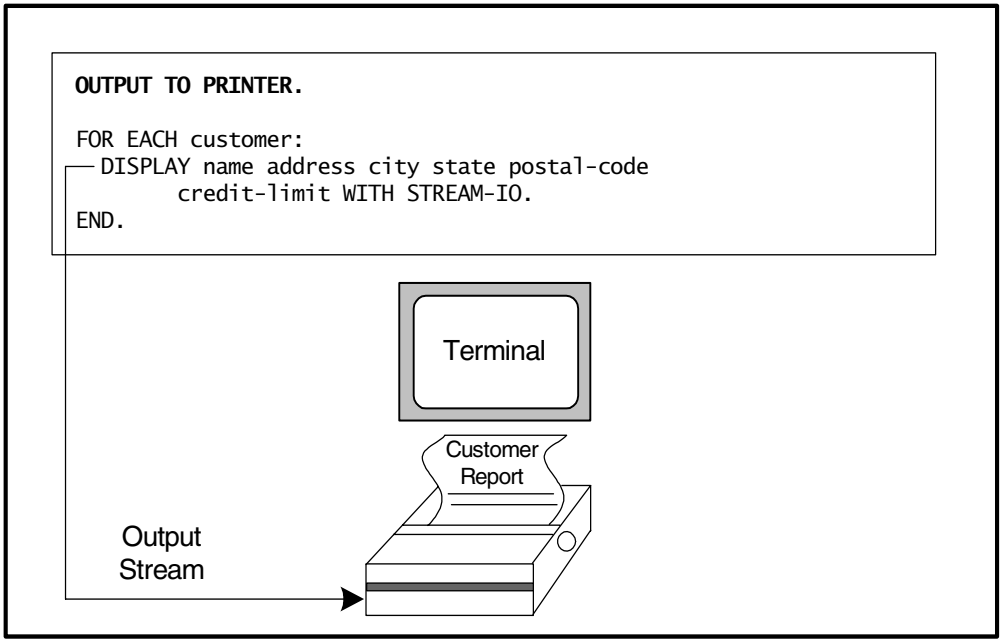


Figure 7–2: Redirecting the Unnamed Output Stream

If you run the procedure, you do not see anything displayed on your terminal because all the output from the DISPLAY statement goes to the printer. The STREAM–IO Frame phrase option formats the output especially for character-based destinations. See the section "Stream I/O vs. Screen I/O" for more information.

7.2.2 Additional Options for Printing on Windows

On Windows, you can also get a list of printers, change the default printer for the session, and provide access to the Print dialog box.

Getting a List of Printers

You can use the GET-PRINTERS option of the SESSION statement to obtain a list of the printers configured for the current system. The following example shows you how to obtain the list of currently configured printers, select a printer, and print to it:

p-wgetls.p

```
/*1*/ DEFINE VARIABLE printer-list AS CHARACTER.
       DEFINE VARIABLE printer-idx AS INTEGER.

/*2*/ printer-list = SESSION:GET-PRINTERS().
/*3*/ printer-idx = LOOKUP("SpecialPrinter", printer-list).

/*4*/ IF printer-idx > 0 THEN DO:
         MESSAGE "output" VIEW-AS ALERT-BOX.
           OUTPUT TO PRINTER "SpecialPrinter".
       END.
/*5*/ ELSE DO:
         OUTPUT TO PRINTER.
       END.

       FOR EACH customer WHERE country = "Finland" :
         DISPLAY NAME country.
       END.

/*6*/ OUTPUT CLOSE.
```

The following list explains the important elements in the example above:

1. Create two variables: one for the output of SESSION:GET–PRINTERS; the other, for the output of LOOKUP.

2. The SESSION: GET–PRINTERS method returns a comma-separated list of printers that are currently configured on the system.

3. The LOOKUP function obtains an integer that gives the position of “SpecialPrinter” in the list. If the printer is not in the list, it returns a 0.

4. The IF statement determines whether the printer “SpecialPrinter” is available and if so, prints to it.

5. If the printer "SpecialPrinter" is not available, the ELSE DO statement prints to the default printer.

6. The OUTPUT CLOSE statement stops sending output to the current destination and redirects output to the destination used prior to OUTPUT TO. See the section "Stream I/O vs. Screen I/O" for more information.

Changing the Default Printer

To change the default printer for the session, you can use the PRINTER–NAME option of the SESSION statement as shown in the following example.

p-wchpr.p

```
/*1*/ DEFINE VARIABLE printername AS CHARACTER.

/*2*/ printername = SESSION:PRINTER-NAME
/*3*/ SESSION:PRINTER-NAME = "\\AB1\hplaser".
/*4*/ OUTPUT TO PRINTER.

      FOR EACH customer:
        DISPLAY name SPACE balance.
      END.

/*5*/ SESSION:PRINTER-NAME = printername.
```

1. Create a variable for the output of SESSION:PRINTER–NAME.

2. The SESSION:PRINTER–NAME attribute returns the name of the default printer.

3. The SESSION:PRINTER–NAME attribute sets another printer, \\AB1\hplaser, as the default printer.

4. The OUTPUT TO PRINTER prints the report on the printer \\AB1\hplaser.

5. The SESSION:PRINTER–NAME attribute restores the original default printer.

Providing Access to the Print Dialog Box

You can use the SYSTEM–DIALOG PRINTER–SETUP statement to provide access to the Windows print dialog box. This allows the application user to set up the printer or even change the printer that receives the output.

7.2.3 Output to Files or Devices

You can direct the report to a standard text file. Replace the PRINTER option of the OUTPUT TO statement in Figure 7–2 as follows:

```
OUTPUT TO rpt-out PAGED.
```

In this OUTPUT TO statement, `rpt-out` is the name of the file where you direct the output. ON UNIX, the filename is case sensitive. That is, UNIX treats `rpt-out` and `RPT-OUT` as two different files. However, to most other operating systems, these are the same file.

The PAGED option indicates that you want a page break in the output every 56 lines. PAGED is automatic for output to a printer. If you do not use the PAGED option, Progress sends the data to the file continuously without any page break control characters.

You can also use the VALUE option to specify filenames stored in variables or fields. For information, see the "Sending Output to Multiple Destinations" section.

Some operating systems (like UNIX) allow you to send output to devices other than printers, terminals, and disk files using a name (like a filename) to reference them. You redirect output to these devices exactly like files.

7.2.4 Output to the Clipboard

In general, you use the CLIPBOARD system handle to write output to the system clipboard (in graphical environments) or Progress clipboard (in character environments). However, on Windows, you can use the OUTPUT TO statement to redirect output to the system clipboard using the "CLIPBOARD" option (quotes required). This allows you to use Progress output statements (such as DISPLAY) to buffer up to 64K of data to the clipboard. You send the buffered data to the clipboard by changing or closing the output destination (using the OUTPUT CLOSE statement). For more information on changing and closing the output destination, see the "Sending Output to Multiple Destinations" section. For more information on using the CLIPBOARD system handle for output (and also input), see the *Progress External Program Interfaces* manual.

7.2.5 Resetting Output to the Terminal

You can reset any output destination to the terminal using the TERMINAL option of the OUTPUT TO statement. For more information, see the "Sending Output to Multiple Destinations" section.

7.2.6 Sending Output to Multiple Destinations

At some points in a procedure, you might want to send output to the terminal, but at other points you might want to send output to a file. Progress does not restrict you to one output destination per procedure.

p-chgot2.p

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

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

OUTPUT TO VALUE(outfile).

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

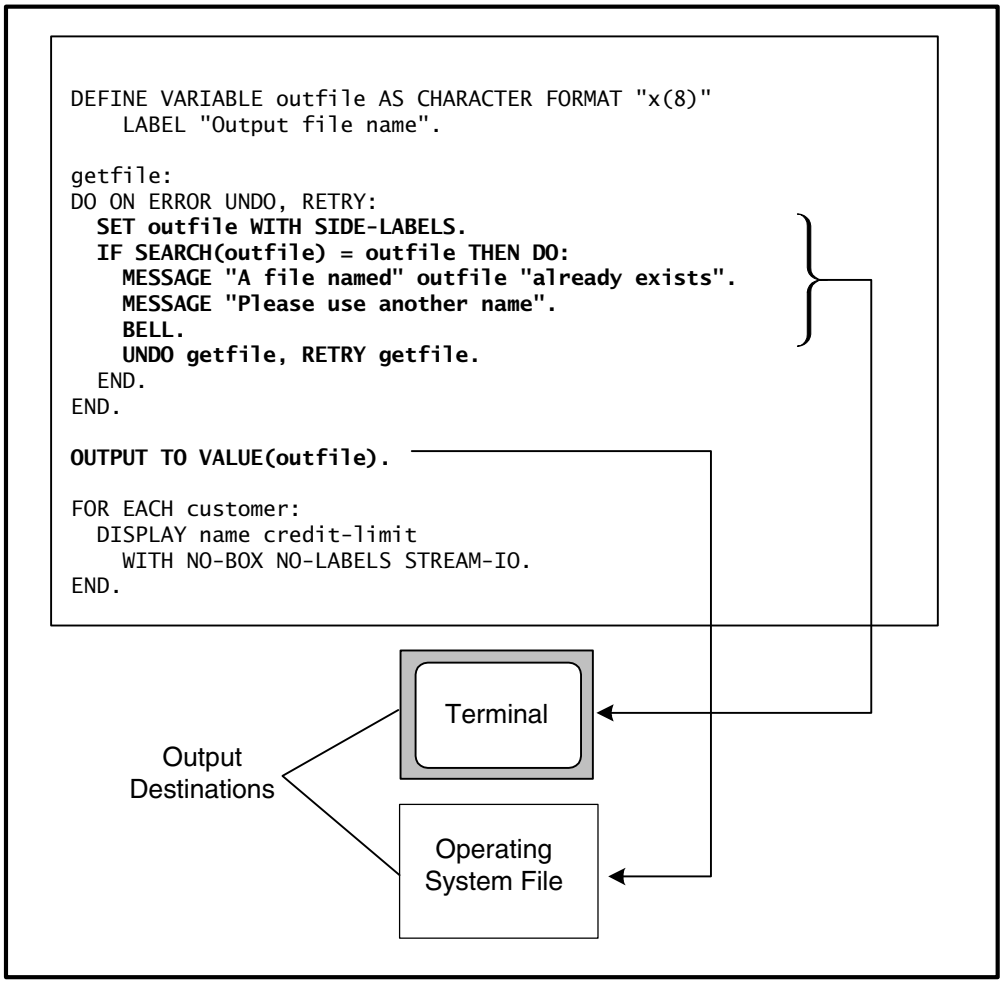


Figure 7–3: Multiple Output Destinations

In the procedure in Figure 7–3, you supply the name of the file where you want to send a customer report and then, if that file does not already exist, send the report to that file.

Here are the specific steps the procedure takes:

1. The SET statement prompts you for the filename.
2. The SEARCH function searches for the file, returning the filename if the file is found.
3. If the file is found, the procedure:
   - Displays a message telling you the file already exists and to use another filename.
   - Rings the terminal bell.
   - Undoes the work done in the DO block and retries the block, giving you the opportunity to supply a different filename.
4. If the file is not found, the procedure uses this statement:

```
OUTPUT TO VALUE(outfile).
```

   This statement redirects the output to the file you specified. You must use the VALUE keyword with the OUTPUT TO statement. The VALUE option tells Progress to use the value of the outfile variable rather than the name “outfile” itself. If instead you say OUTPUT TO outfile, Progress assumes that outfile is the name of the text file to which you want to send output.

You use the OUTPUT CLOSE statement to stop sending output to a destination. Output sent after the OUTPUT CLOSE statement goes to the destination used prior to the OUTPUT TO statement.

For example, the procedure in Figure 7–4 uses the OUTPUT CLOSE statement to reset the output destination from a file to the terminal.

p-out.p

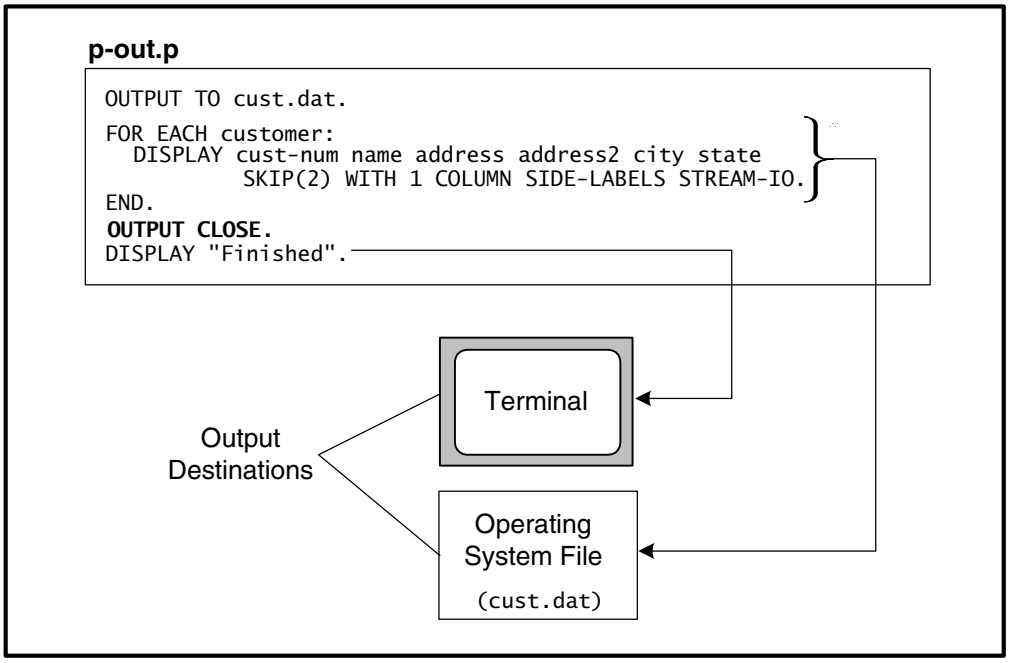


Figure 7–4: The OUTPUT CLOSE Statement

This procedure sends customer information to a file called `cust.dat`. Then the procedure displays the word "Finished" on your terminal screen. The specific steps in the procedure are as follows:

1. The OUTPUT TO statement redirects output so all statements that normally send output to the terminal send output to the `cust.dat` file.

2. The FOR EACH customer and DISPLAY statements produce a report listing each customer's name, address, city, and state. The procedure sends the report to the `cust.dat` file.

3. The OUTPUT CLOSE statement resets the output destination for the procedure from the `cust.dat` file to the terminal.

4. The last DISPLAY statement displays the message "Finished" on the terminal screen.

If the output destination prior to execution of `p-out.p` was the printer (or any destination other than the terminal), you must explicitly set the output destination to the terminal between Steps

3 and 4 using the OUTPUT TO statement with the TERMINAL option. Otherwise the OUTPUT CLOSE statement resets the output destination to the printer (or other prior destination).

7.2.7 Stream I/O vs. Screen I/O

When you compile a procedure, Progress automatically lays out the frames as appropriate for the current screen display. The layout differs across user interfaces because some fields have special decoration. The font also affects the size, and therefore the layout, of fields.

However, when you are displaying a frame to a file or printer rather than the screen, you do not want Progress to design the frame for screen display. You want Progress to lay out the field for a character display using a fixed font and fields without decorations. Otherwise, output written to a printer or operating system file may be unattractive or even partially unreadable. You also want Progress to represent all data fields as text, not as graphical widgets such as editors or sliders.

Progress provides two mechanisms to deal with these issues:

- The STREAM–IO option of the COMPILE statement
- The STREAM–IO and SCREEN–IO options of the Frame phrase

If all output from a procedure is to printers or operating system files, you can use the STREAM–IO option of the COMPILE statement. This forces Progress to lay out all frames using a standard fixed font and to display all data fields as simple text fields. The following widget types are converted to text:

- Editors
- Selection lists
- Sliders
- Radio sets
- Toggle boxes

If you only write a few frames to a printer or file in your procedure, you can use the STREAM–IO frame option to mark those frames. This forces Progress to lay out only those specific frames for output to a printer or file. If you do not specify STREAM–IO in the COMPILE statement, then all other frames are designed for screen display.

If only a few screens in your procedure are displayed to a screen, you use the SCREEN–IO frame option to mark those frames. Then compile with STREAM–IO so that all other frames are laid out for display to a printer or file.

7.2.8 A Printing Solution

The Progress ADE toolset provides a portable solution for printing text files. The solution is a procedure called _osprint.p and it is located in the adecomm directory in the Progress product directory (DLC). You can also access the source code for this procedure in the src/adecomm directory located in the Progress product directory.

The _osprint.p procedure sends a specified text file to the default printer as paged output. Input parameters for the procedure allow you to specify values that configure a print job. On Windows, you can also direct the _osprint.p procedure to display the Print dialog box and print the text in a specified font. Use the following syntax to call the _osprint.p procedure from a Progress procedure:

SYNTAX

```
RUN adecomm/_osprint.p
  ( INPUT parentWindow , INPUT printFile ,
    INPUT fontNumber , INPUT PrintFlags . INPUT pageSize ,
    INPUT pageCount , OUTPUT result
  ).
```

The parameters of the _osprint.p procedure are as follows:

INPUT *parentWindow*

A window handle identifying the parent window for Print dialog box and any print status messages on Windows. The procedure ignores a value specified for this parameter in character interfaces. If you specify the unknown value (?) or an invalid handle on Windows, the procedure uses the CURRENT–WINDOW handle.

INPUT *printFile*

A string value representing the name of a text file to print. You can specify an absolute or relative path for the file. The _osprint.p procedure uses the PROPATH to locate the file.

`INPUT` *`fontNumber`*

An integer value representing an entry in the font table maintained by the FONT–TABLE handle. The `_osprint.p` procedure uses the specified font to print the text file on Windows. The procedure ignores a value specified for this parameter in character interfaces. If you specify the unknown value (?) or an integer value that does not exist in the font table for Windows, the procedure uses the default system font to print the text file.

`INPUT` *`PrintFlags`*

An integer value that determines which printing options are used for a print job on Windows (only). You can use the values in Table 7–1. If you need to use more than one option, add the values of the options together. In all cases, the `_osprint.p` procedure sets the value of the PRINTER–CONTROL–HANDLE attribute of the SESSION handle to zero (0).

Table 7–1: Printing Options for Windows

Printing Option	Value	Selection
Context	0	Default print context
	1	Print dialog box. This allows the user to establish a print context.
Landscape orientation	2	Landscape orientation
Paper Size	4	Letter
	8	Legal
	12	A4
Paper Tray	32	Upper tray
	64	Middle tray
	96	Lower tray
	128	Manual
	160	Auto

INPUT *pageSize*

An integer value representing the number of lines per page. If you specify zero (0) for this parameter, the printer determines the page size. Windows ignores this parameter and calculates the page size based on the Paper Size setting in the Print Setup dialog box and the font specified with the *fontNumber* parameter.

NOTE: The maximum number of character per line is 255.

INPUT *pageCount*

An integer value that determines if _osprint.p prints the entire text file or a range of pages from the text file on Windows. The procedure ignores a value specified for this parameter in character interfaces. If the value of this parameter is not zero (0) on Windows, Progress uses the page range specified for the current print context.

OUTPUT result

A logical value that reports the success or failure of the print job.

To call the _osprint.p procedure from a Progress procedure, you must define a variable for the result output parameter. Here is an example:

_osprint.p

```
/*1*/ DEFINE VARIABLE result AS LOGICAL NO-UNDO.

/*2*/  OUTPUT TO tmp.dat.
       FOR EACH customer:
         DISPLAY Cust-num Name Phone WITH STREAM-IO.
       END.
       OUTPUT CLOSE.

/*3*/   RUN adecomm/_osprint.p (INPUT ?, INPUT "tmp.dat", INPUT ?, INPUT 1,

                      INPUT 0, INPUT 0, OUTPUT result).

/*4*/  OS-DELETE "tmp.p".
```

The following list describes the important elements of the previous example:

1. Create a variable for the OUTPUT parameter of the `_osprint.p` procedure.
2. Generate the temporary text file to be printed. Remember to close the output stream after generating the text file.
3. Run the `_osprint.p` procedure to print the generated text file.
4. Delete the temporary text file.

For more information on the language elements referenced in this section, see the *Progress Language Reference*.

7.3 Changing a Procedure's Input Source

You use the INPUT FROM statement to name a new source for input. The possible sources include:

- An operating system file or device
- The contents of an operating system directory
- The terminal (by default)

All statements (such as UPDATE or SET) that require data use the new input source.

7.3.1 Input From Files

A data file (not a database table) that contains information on new customers might look like the following example:

p-datf1.d

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

This file is in standard Progress format. That is, blanks separate the field values. Field values that contain embedded blanks are surrounded by quotation marks (" "). Later sections in this chapter discuss using alternate formats.

You can write a Progress procedure and tell that procedure to get its input from the `p-datf1.d` file. See Figure 7–5.

p-chgin.p

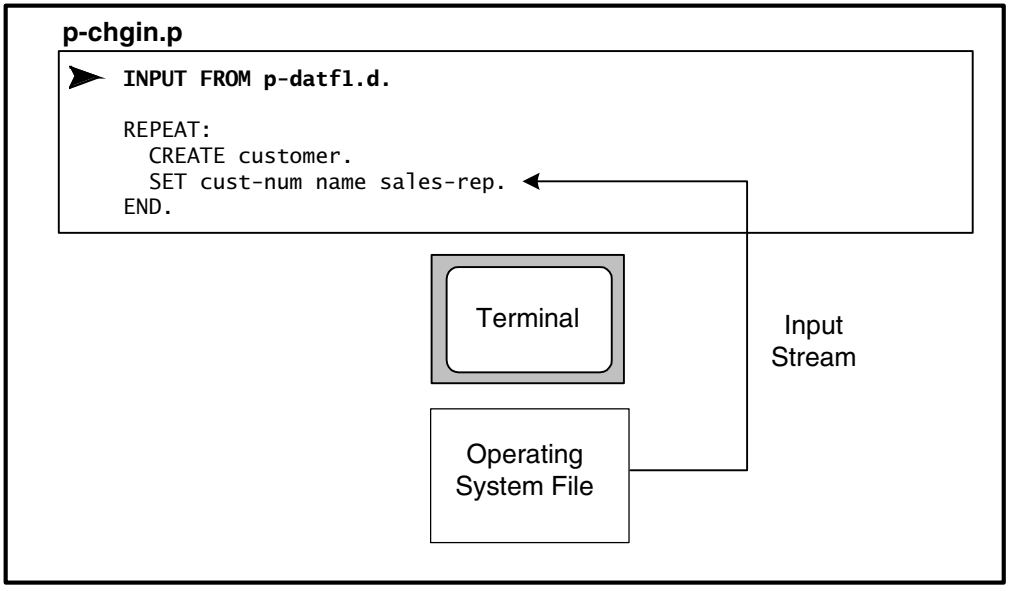


Figure 7–5: Redirecting the Unnamed Input Stream

The SET statement, which normally gets its input from the terminal, gets its input from the `p-datf1.d` file. The cust–num field uses the first data item, 90. The name field uses the next quoted data item, "Wind Chill Hockey," etc. Each time Progress processes a data entry statement, it reads one line from the file.

7.3.2 Input from Devices, Directories, and the Terminal

Some operating systems (like UNIX) let you receive input from devices other than terminals and disk files and let you use a name (like a filename) to reference them. You redirect input from these devices exactly like files. For more information, see the "Input From Files" section.

You can read the contents of a directory using the OS–DIR option of the INPUT FROM statement. For more information on reading the contents of a directory, see the "Reading the Contents of a Directory" section.

You can reset any input source to the terminal using the TERMINAL option of the INPUT FROM statement. For more information, see the "Receiving Input from Multiple Sources" section.

7.3.3 Receiving Input from Multiple Sources

At some points in a procedure you might want to get input from the terminal, but at other points you might want to get input from a file. A single procedure can use multiple input sources.

For example, suppose you want to create records for the customers in the `p-datf1.d` data file. Before creating the records, you probably want to display the customer numbers in the file and ask if the user wants to create customer records for those numbers. To do this, you need input from the terminal. If the user wants to create customer records for the customers in the `p-datf1.d` file, you also need input from the file.

The `p-chgin2.p` procedure uses multiple input sources to perform the work described above. Because `p-chgin2.p` uses the same data file (`p-datf1.d`) you used in the previous section to create customer records, you must delete customers 90, 91, and 92 from your database before you run `p-chgin2.p`. Use the following procedure to delete the customers:

p-io3.p

```
FOR EACH customer WHERE cust-num > 89:
  DELETE customer.
END.
```

Figure 7–6 shows the p-chgin2.p procedure.

p-chgin2.p

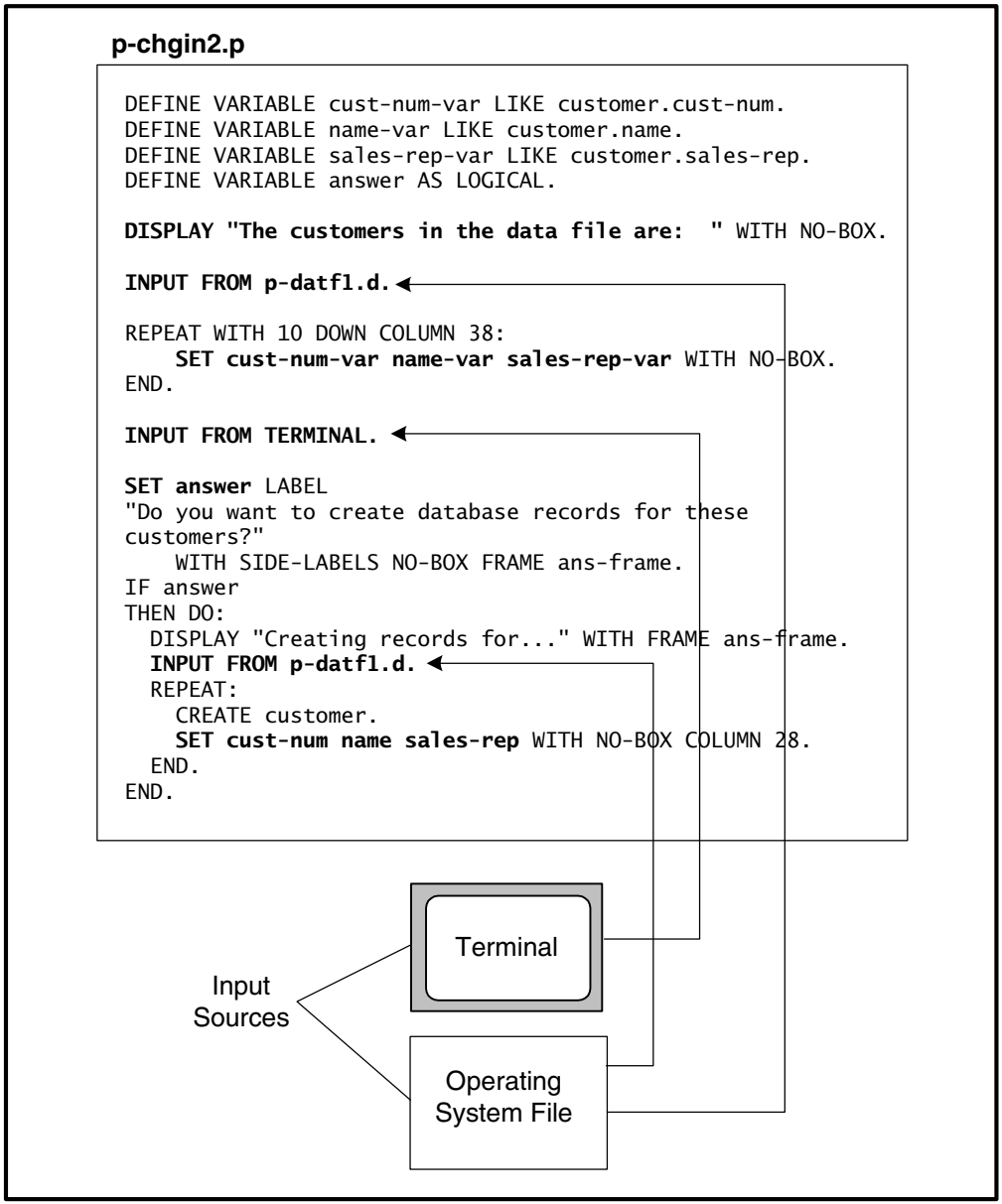


Figure 7–6: Multiple Input Sources

These are the specific steps the procedure follows:

1. The DISPLAY statement displays some text.
2. The first INPUT FROM statement redirects the input source to the `p-datf1.d` file.
3. The SET statement assigns the values in the `p-datf1.d` file to the cust–num–var, name–var, and sales–rep–var variables. As it assigns the values to these variables, you see the values on the terminal screen.
4. The INPUT FROM TERMINAL statement redirects the input source to the terminal. The INPUT CLOSE statement could have been used instead of the INPUT FROM TERMINAL statement. However, since this procedure might have been called from another procedure, it is better to be explicit about the input source you want to use.
5. The SET answer statement prompts you to create database records for the customer data just displayed. If you answer yes, the procedure:
   - Redirects the input source to come from the beginning of the `p-datf1.d` file.
   - Creates a customer record and assigns values to the cust–num., name, and sales–rep fields in that record for each iteration of a REPEAT block.
   - Ends the REPEAT block when the SET statement reaches the end of the input file.

7.3.4 Reading Input with Control Characters

You can use the BINARY option of the INPUT FROM statement to read control characters. Input sources often contain control characters that affect the format or quantity of data that you receive. For example, a text file might contain NUL (“\0”) characters to terminate character strings. Without the BINARY option, Progress ignores all input on a line after the first NUL character. The BINARY option allows you to read all the data in the file, including any NUL and other non-printable control characters without interpretation.

7.4 Defining Additional Input/Output Streams

When you start a procedure, Progress automatically provides that procedure with input and output streams. As described in the previous sections, the default source for the input stream is the terminal and the default destination for the output stream is also the terminal. You saw how to use the INPUT FROM and OUTPUT TO statements to redirect these input and output streams.

You might find that having just one input stream and one output stream is not enough for particular procedures. That is, you might want to get input from more than one source at the same time or send output to more than one destination at the same time.

Suppose you want to produce a report of the items you have in inventory and you want to send the report to a file. You already know how to use the OUTPUT TO statement to redirect the output stream to a file. Suppose that you also want to produce an "exceptions" report at the same time. Any item where the allocated amount is greater than the on-hand amount is an exception. Figure 7–7 illustrates this scenario.

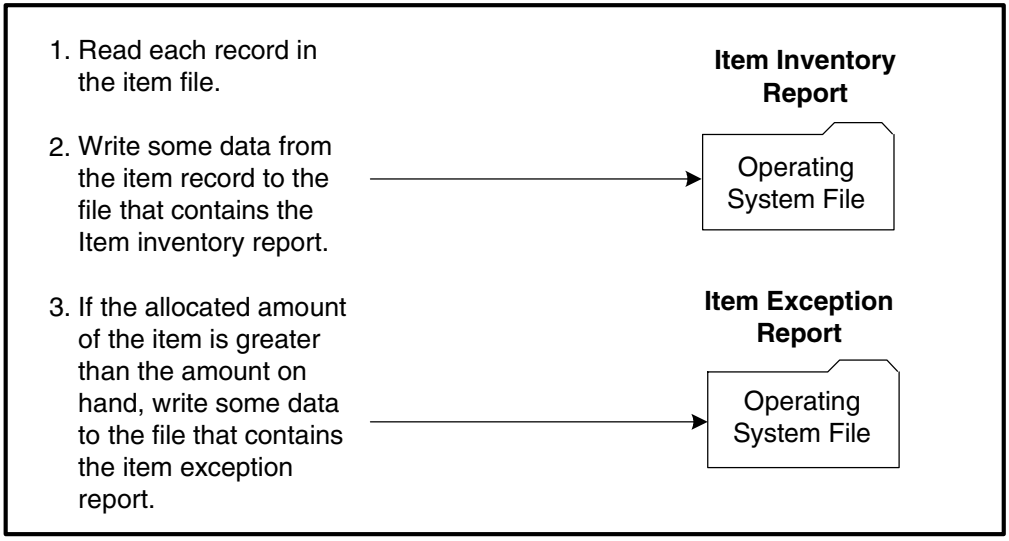


Figure 7–7: Multiple Output Streams Scenario

For items that are exceptions, the procedure needs to send output to a second location. That means you need two different output streams.

You use the DEFINE STREAM statement to define additional streams for a procedure to get input from more than one source simultaneously and send output to more than one destination simultaneously. Streams you name can be operating system files, printers, the terminal, or other non-terminal devices.

The procedure p-dfstr.p uses the two report scenarios shown in Figure 7–7.

p-dfstr.p

```
      DEFINE STREAM rpt.
      DEFINE STREAM exceptions.
      DEFINE VARIABLE fnr AS CHARACTER FORMAT "x(12)".
      DEFINE VARIABLE fne AS CHARACTER FORMAT "x(12)".
      DEFINE VARIABLE excount AS INTEGER LABEL "Total Number of
          exceptions".
      DEFINE VARIABLE exception AS LOGICAL.

/*1*/ SET fnr LABEL "Enter filename for report output" SKIP(1)
          fne LABEL "Enter filename for exception output"
          WITH SIDE-LABELS FRAME fnames.
/*2*/ OUTPUT STREAM rpt TO VALUE(fnr) PAGED.
      OUTPUT STREAM exceptions TO VALUE(fne) PAGED.

/*3*/ DISPLAY STREAM rpt "Item Inventory Report" SKIP(2)
           WITH CENTERED NO-BOX FRAME rpt-frame STREAM-IO.
/*4*/ DISPLAY STREAM exceptions "Item Exception Report" SKIP(2)
           WITH CENTERED NO-BOX FRAME except-frame STREAM-IO.

/*5*/ FOR EACH item:
/*6*/    IF on-hand < alloc THEN DO:
             DISPLAY STREAM exceptions item-num item-name on-hand alloc
                WITH FRAME exitem DOWN STREAM-IO.
             excount = excount + 1.
             exception = TRUE.
END.
/*7*/    DISPLAY STREAM rpt item-num item-name
          WITH NO-LABELS NO-BOX STREAM-IO.
/*8*/    IF exception
         THEN DISPLAY STREAM rpt "See Exception Report".
         exception = FALSE.
     END.

/*9*/ DISPLAY STREAM exceptions SKIP(1) excount
           WITH FRAME exc SIDE-LABELS STREAM-IO.
/*10*/DISPLAY STREAM rpt WITH FRAME exc STREAM-IO.

/*11*/OUTPUT STREAM rpt CLOSE.
      OUTPUT STREAM exceptions CLOSE.
```

The numbers to the left of the procedure correspond to the following step-by-step descriptions:

1. The SET statement prompts you for the filenames you want to use for the Item Inventory Report and for the Item Exception Report. It stores your answers in the fnr and fne variables, respectively.

2. The OUTPUT STREAM statements open two output streams, named rpt and exceptions. These streams were defined at the start of the procedure with the DEFINE STREAM statement.

   The rpt and exceptions streams are directed to the files whose names you supplied: VALUE(fnr) and VALUE(fne). This means that output can now be sent to either or both of those files.

3. The DISPLAY statement displays the text "Item Inventory Report." But instead of displaying that text on the terminal, it displays it to the rpt stream. The file you named for the Item Inventory Report contains the text "Item Inventory Report."

4. This DISPLAY statement also displays text but it uses the exceptions stream. The file you named for the Item Exception Report contains the text "Item Exception Report."

5. The FOR EACH block reads a single item record on each iteration of the block.

6. If the allocated amount of an item is larger than the on-hand amount of that item:

   - The DISPLAY statement displays item data to the exceptions stream. After this DISPLAY statement finishes, the file you named for the Item Exception Report contains item data for a single item.

   - The excount counter variable, defined at the start of the procedure, is incremented by 1. The value of this variable is displayed at the end of the procedure so that you know the total number of exception items in inventory.

   - The exception logical variable, defined at the start of the procedure, is set to TRUE.

7. The DISPLAY statement displays some item data to the rpt stream. After this statement finishes, the file you named for the Item Inventory Report contains item data for a single item.

8. If the item is an exception, determined by the value in the exception logical variable, the DISPLAY statement displays the string "See Exception Report" to the rpt stream. That way you know, when looking at the Item Inventory Report, which items are exceptions.

9. The DISPLAY statement displays the value of the excount variable to the exceptions stream. The value of this variable is the total number of exception items.

10. This DISPLAY statement displays the value of the excount variable to the rpt stream. Although the DISPLAY statement does not explicitly say what is being displayed, it does name the same frame, exc, as is used to display excount in the previous DISPLAY statement. That means that the exc frame already contains the excount value. Thus, all this second DISPLAY statement has to do is name the same frame.

11. The OUTPUT STREAM CLOSE statements close the rpt and exception streams, redirecting all further output to the default output destination.

7.5 Sharing Streams Among Procedures

In some cases, you might want two or more procedures to share the same input or output streams. The following procedures, `p-sstrm.p` and `p-dispho.p`, share the same output stream, phonelist. Notice that phonelist is defined as a shared stream in both procedures:

p-sstrm.p

```
DEFINE NEW SHARED BUFFER xrep FOR salesrep.
DEFINE NEW SHARED STREAM phonelist.

OUTPUT STREAM phonelist TO phonefile.

PAUSE 2 BEFORE-HIDE.

FOR EACH xrep:
  DISPLAY xrep WITH FRAME repname
    TITLE "Creating report for " 2 COLUMNS CENTERED ROW 10.
  DISPLAY STREAM phonelist xrep WITH 2 COLUMNS STREAM-IO.
  RUN p-dispho.p.
END.
```

The `p-sstrm.p` procedure defines a NEW SHARED STREAM called phonelist. The procedure sends the output from the phonelist stream to a file called `phonefile`. The procedure also calls the `p-dispho.p` procedure:

p-dispho.p

```
DEFINE SHARED BUFFER xrep FOR salesrep.
DEFINE SHARED STREAM phonelist.

FOR EACH customer OF xrep BY state:
  DISPLAY STREAM phonelist cust-num name city state phone
          WITH NO-LABELS STREAM-IO.
END.
```

The `p-dispho.p` procedure defines the SHARED STREAM phonelist, and displays the information from that stream on the screen. (It is more efficient to place the FOR EACH and DISPLAY statements in the `p-sstrm.p` procedure. They are in a separate procedure here to illustrate shared streams.)

NOTE: You cannot define or access shared streams in a persistent procedure. If you do, Progress returns a run-time error when you create the persistent procedure. For more information, see the sections on persistent procedures in Chapter 3, "Block Properties."

Sharing streams is much like sharing variables:

- You use a regular DEFINE STREAM statement to define a stream that is available only to the current procedure.
- To define a shared stream, you define the stream as NEW SHARED in the procedure that creates the stream, and as SHARED in all other procedures that use that stream. If you do not explicitly close the stream, Progress closes it automatically at the end of the procedure in which you defined it.
- You define the stream as NEW GLOBAL when you want that stream to remain available even after the procedure that contains the DEFINE NEW GLOBAL SHARED STREAM statement ends.

7.6 Summary of Opening and Closing Streams

Table 7–2 describes how you establish, open, use, and close default streams and streams you name.

Table 7–2: Using Streams

Action	Unnamed Streams	Named Streams
Establish the stream	By default, each procedure gets one unnamed input stream and one unnamed output stream.	You define the stream explicitly by using one of these statements: DEFINE STREAM, DEFINE NEW SHARED STREAM, DEFINE SHARED STREAM, DEFINE NEW GLOBAL SHARED STREAM.
Open the stream	Automatically opened, using the output destination to which the calling procedure's unnamed stream is directed and the input source from which the calling procedure's input is read. You can also explicitly name a destination or source by using OUTPUT TO or INPUT FROM.	You open the stream explicitly by using OUTPUT STREAM *name* TO or INPUT STREAM *name* FROM.
Use the stream	All data-handling statements use the stream by default.	Name the opened stream in the data-handling statement that will use the stream.
Close the stream	Automatically closed at the end of the procedure that opened it. You can also explicitly close it with the OUTPUT CLOSE or INPUT CLOSE statement.	Local streams are automatically closed at the end of the procedure. Shared streams are automatically closed when the procedure that defined the stream as NEW ends. Global streams are closed at the end of the Progress session. You can also explicitly close named streams by using the INPUT CLOSE or OUTPUT CLOSE statement or by opening the stream to a new destination or from a new source.

Initially, a default input stream has as its source the most recent source specified in the calling procedure or, if there is no calling procedure, the terminal. The default output stream has as its destination the most recent destination specified in the calling procedure or, if there is no calling procedure, the terminal. If you are running a procedure in batch or background, you must explicitly indicate a source and/or destination.

When an unnamed stream is closed (either automatically or explicitly), it is automatically redirected to its previous destination (the destination of the procedure it is in). If the stream is not in a procedure, the stream is redirected to or from the terminal.

When you close a named stream, you can no longer use that stream until it is reopened. When you close an input stream associated with a file and then reopen that stream to the same file, input starts from the beginning of the file.

7.7 Processes as Input and Output Streams (NT and UNIX only)

You can import data into Progress from a process or pipe data from Progress to another process using one of the following statements:

- INPUT THROUGH statement to import data into Progress from another process
- OUTPUT THROUGH statement to pipe data from Progress to another process
- INPUT–OUTPUT THROUGH statement to pipe the output of a process into a Progress procedure and to pipe data from Progress back to that same process.

  This allows two-way communication to a program written in C or any other language. You might use this capability to do specialized calculations on data stored in a Progress database or entered during a Progress session.

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

7.8 I/O Redirection for Batch Jobs

You can use the < and > symbols on the Progress command line to redirect I/O for batch jobs. For details, see the entry for the Batch (–b) startup parameter in the *Progress Startup Command and Parameter Reference*.

7.9 Reading the Contents of a Directory

Sometimes, rather than reading the contents of a file, you want to read a list of the files in a directory. You can use the OS–DIR option of the INPUT FROM statement for this purpose.

Each line read from OS–DIR contains three values:

- The simple (base) name of the file.
- The full pathname of the file.
- A string value containing one or more attribute characters. These characters indicate the type of the file and its status. Every file has one of the following attribute characters:
  - **F** — Regular file or FIFO pipe
  - **D** — Directory
  - **S** — Special device
  - **M** — Member of a Progress r-code library
  - **X** — Unknown file type

  In addition, the attribute string for each file might contain one or more of the following attribute characters:
  - **H** — Hidden file
  - **L** — Symbolic link
  - **P** — Pipe file

The tokens are returned in the standard Progress format that can be read by the IMPORT or SET statements.

The following example uses the OS–GETENV function to find the path of the DLC directory. It then uses the OS–DIR option of INPUT FROM to read the contents of the directory:

p-osdir.p

```
DEFINE VARIABLE search-dir AS CHARACTER.
DEFINE VARIABLE file-name AS CHARACTER FORMAT "x(16)" LABEL "File".
DEFINE VARIABLE attr-list AS CHARACTER FORMAT "x(4)" LABEL "Attributes".

search-dir = OS-GETENV("DLC").

INPUT FROM OS-DIR(search-dir).

REPEAT:
   SET file-name ^ attr-list
         WITH WIDTH 70 USE-TEXT TITLE "Contents of " + search-dir.
END.

INPUT CLOSE.
```

In `p-osdir.p`, only the base name of the file and attribute string are read from OS–DIR. The caret (^) is used in the SET statement to skip over the pathname of the file.

For more information on the OS–DIR option, see the INPUT FROM Statement reference entry in the *Progress Language Reference*.

You can find additional information on a single file by using the FILE–INFO system handle. To use the FILE–INFO handle, first assign the pathname of an operating system file to the FILE–INFO:FILE–NAME attribute. You can then read other FILE–INFO attributes. For example, the `p-osfile.p` procedure prompts for the pathname of a file and then uses the FILE–INFO handle to get information on that file:

p-osfile.p

```
DEFINE VARIABLE os-file  AS CHARACTER FORMAT "x(60)" LABEL "File".

REPEAT:
  SET os-file WITH FRAME osfile-info.

  FILE-INFO:FILE-NAME = os-file.

  DISPLAY FILE-INFO:FULL-PATHNAME FORMAT "x(60)" LABEL "Full Path"
    FILE-INFO:PATHNAME FORMAT "x(60)" LABEL "Path"
    FILE-INFO:FILE-TYPE LABEL "Type"
    WITH FRAME osfile-info SIDE-LABELS TITLE "OS File Info".
END.
```

For more information, see the FILE–INFO System Handle reference entry in the *Progress Language Reference*.

7.10 Performing Code-page Conversions

Computer systems store text data using character codes, typically 7 or 8–bit numeric codes that map to specific visual character representations. The series of character codes that make up the character set that a system uses is referred to as a code page.

Progress provides a character set management facility to automatically convert data between the code pages of different data sources and destinations (targets). In general, the supported data sources and targets for code page conversion include memory, streams, and databases. You can specify default code page conversions for a session using conversion tables and startup parameters to specify the code page for each data source and target. For more information on this character set facility, see the *Progress Internationalization Guide*.

The Progress 4GL also allows you to perform I/O that explicitly converts data from one code page to another in order to facilitate I/O between data sources and destinations intended for different computer systems or components. You can specify the name of the code page for a data source and target as parameters to several statements and functions.

To convert between source and target characters or strings in memory, you can specify code page parameters in these functions:

- ASC
- CHR
- CODEPAGE–CONVERT

To convert data input and output, you can specify code page parameters in these statements:

- INPUT FROM (input source to memory target)
- OUTPUT TO (memory source to output target)

To convert piped input and output, you can specify code page parameters in these statements:

- INPUT THROUGH (program source to memory target)
- OUTPUT THROUGH (memory source to program target)
- INPUT–OUTPUT THROUGH (program source to program target)

These statements and functions take available code page name parameters as character expressions (for example, “ibm850”). The code page names you specify must be defined in your Progress conversion map file (`convmap.cp`, by default). Also, the source and target conversion tables must be defined in the conversion map file for each code page to support the specified conversions. For more information on building a conversion map file, see the *Progress Internationalization Guide*.

For example, this procedure writes to two output streams using two different code pages:

p-codpag.p

```
DEFINE VARIABLE xname LIKE customer.name INITIAL ?.

DO WHILE xname <> "":
    ASSIGN xname = "".
    DISPLAY xname LABEL "Starting Name to German List"
        WITH FRAME A SIDE-LABELS.
    SET xname WITH FRAME A.
    IF xname <> "" THEN DO:
        OUTPUT TO german.txt
            CONVERT SOURCE "iso8859-1" TARGET "german-7-bit".

        FOR EACH customer WHERE customer.name >= xname BY name:
DISPLAY customer WITH STREAM-IO.
        END.
        OUTPUT CLOSE.

        OUTPUT TO swedish.txt
            CONVERT SOURCE "iso8859-1" TARGET "swedish-7-bit".

        FOR EACH customer WHERE customer.name < xname BY name:
            DISPLAY customer WITH STREAM-IO.
        END.
        OUTPUT CLOSE.
    END.
END.
```

For this example, assume that the internal code page is “iso8859–1”. If a customer name from the sports database is greater than or equal to a prompted name (xname), then the procedure writes the corresponding customer record to the file, `german.txt`, using the “german–7–bit” code page. Otherwise, it writes the corresponding customer record to the file, `swedish.txt`, using the “swedish–7–bit” code page. These conversions are handled by the conversion tables provided with Progress.

For more information on specifying code page parameters, see the reference entry for each statement or function that supports code page conversion in the *Progress Language Reference*.

7.11 Converting Non-standard Input Files

The input files used in the previous examples contained data that was in a very specific format:

p-datf1.d

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

When using a data file as an input source, Progress, by default, expects that file to conform to the following standards:

- One or more spaces must separate each field value.
- Character fields that contain embedded blanks must be surrounded by quotes (“ ”).
- Any quotes in the data must be represented by two quotes (“ ”).

What if you need to deal with an input file that does not conform to these standards? For example, you might have a file in which each field is on a separate line; or where fields are separated by commas instead of spaces; or where the fields have no special delimiters, but appear at specific column locations.

Progress provides two strategies for dealing with such files:

- Use the QUOTER utility to convert the file to the standard Progress format. You can then use the normal Progress frame-based input statements, such as SET, to read data from the file.
- Use the IMPORT statement to read from the file.

Which method you choose depends on your particular circumstances. For example:

- The frame-based statements, such as SET, validate the format of the incoming data; IMPORT does not. In some cases you want this validation and in others you might not want it.
- If you expect to read the same file repeatedly, preprocessing that file once with QUOTER might be worthwhile. However, if your code reads from different files, or the content of the file is subject to change, you might want to avoid repeated preprocessing by using IMPORT instead.

The following section explains how to use QUOTER. For more information on using the IMPORT statement, see the “Importing and Exporting Data” section.

7.11.1 Using QUOTER to Format Data

The QUOTER utility formats character data in a file to the standard format so it can be used by a Progress procedure. By default, QUOTER does the following:

- Takes the name of a file as an argument
- Reads input from that file
- Places quotes (“) at the beginning and end of each line in the file.
- Replaces any already existing quotes (“) in the data with two quotes (“ ”).

You can use the QUOTER utility directly from the operating system using the operating system statement, as appropriate, to reformat data from a file that is not in the standard Progress input format:

Operating System	Syntax
UNIX Windows	quoter *input-file-name* [> *output-file-name* | -d *character* | -c *startcol-stopcol*] ...

PARAMETERS

input-file-name

Specifies the file you are using.

> *output-file-name*

Specifies the file where you want to send the output.

-d *character*

Identifies a field delimiter. You can specify any punctuation character.

-c *startcol-stopcol*

Specifies the ranges of column numbers to be read as fields. Do not use any spaces in the range list.

Suppose your data file looked like the following example:

p-datf12.d

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

You use QUOTER to put this file into standard format. Use commands shown in Table 7–3 to run QUOTER from the Procedure Editor:

Table 7–3: Quoter Examples

Operating System	Using Quoter—Examples
Windows	`DOS quoter p-dat12.d >p-dat12.q`
UNIX	`UNIX quoter p-datf12.d >p-datf12.q`

In Figure 7–8, p-datfl2.d is the name of the data file you supply to QUOTER while p-datfl2.q is the name of the file in which you want to store QUOTER's output.

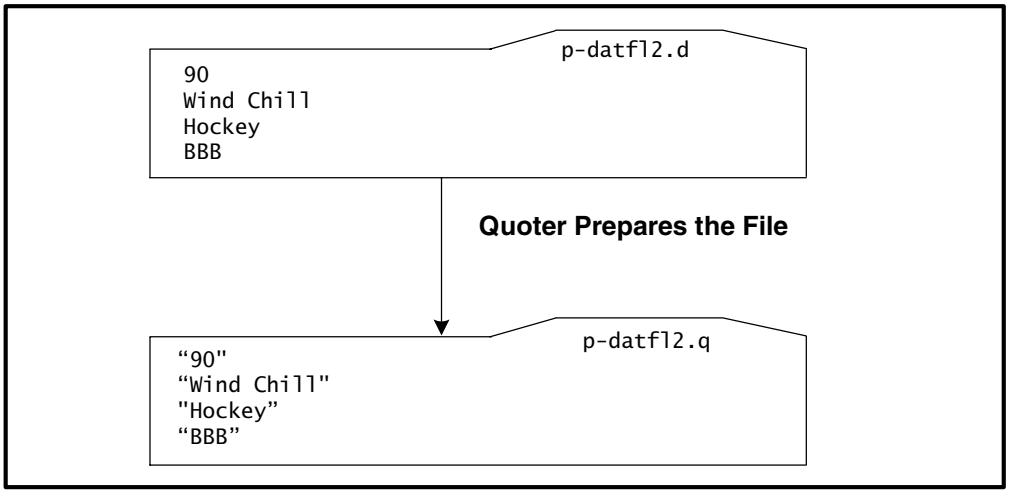


Figure 7–8: How QUOTER Prepares a File

The p-datfl2.q file contains the QUOTER output:

p-datf12.q

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

Now this file is in the appropriate format to be used as input to a Progress procedure.

What if each of the field values in your data file is not on a separate line (unlike p-datfl2.d) and without quotes (unlike p-datf1.d)? That is, your data file looks like p-datfl3.d:

p-datf13.d

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

Suppose you wanted to use this file as the input source to create customer records for customers 90, 91, and 92. You need to make each line of the file into a character string and then assign a substring of this value to each field. The procedure p-chgin3.p does that.

p-chgin3.p

```
       DEFINE VARIABLE data AS CHARACTER FORMAT "x(78)".

/*1*/  OS-COMMAND SILENT quoter p-datfl3.d > p-datfl3.q.

/*2*/  INPUT FROM p-datfl3.q NO-ECHO.

       REPEAT:
/*3*/      CREATE customer.
/*4*/      SET data WITH NO-BOX NO-LABELS NO-ATTR-SPACE WIDTH 80.
           cust-num = INTEGER(SUBSTRING(data,1,2)).
           name = SUBSTRING(data,4,17).
/*5*/      sales-rep = SUBSTRING(data,22,3).
       END.

/*6*/  INPUT CLOSE.
```

The numbers to the left of the procedure correspond to the following step-by-step descriptions:

1. The QUOTER utility takes the data from the p-datfl3.d file and produces data that looks like the following example:

   p-datf13.q

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

2. The INPUT FROM statement redirects the input stream to get input from the p-datfl3.q file.

3. The CREATE statement creates an empty customer record.

4. The SET statement uses the first quoted line in the `p-datfl3.q` file as input and puts that line in the data variable. Once that line of data is in the line variable, the next statements break it up into pieces that get stored in individual customer fields.

5. The SUBSTRING functions take the appropriate pieces of the data in the line variable and store the data in the cust–num, name, and sales–rep fields, as shown in Figure 7–9.

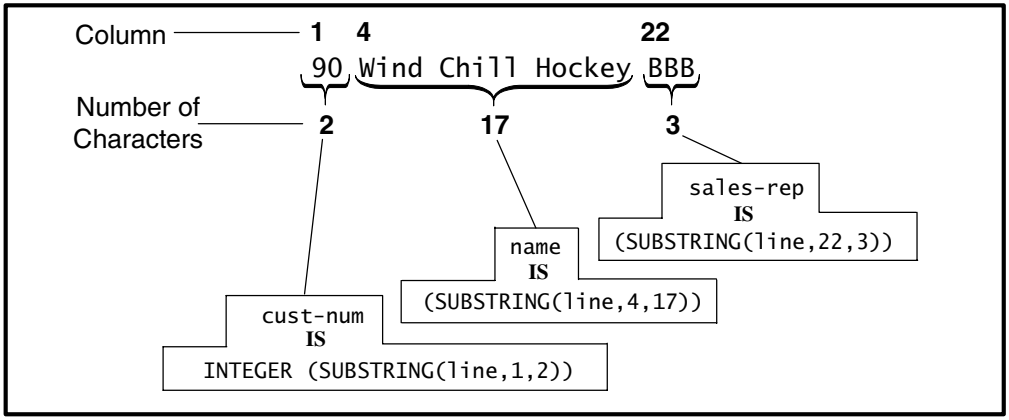


Figure 7–9: Extracting QUOTER Input with SUBSTRING

Because Progress assumes that all the data in the `p-datfl3.q` file is character data, you must use the INTEGER function to convert the cust–num data to an integer value.

6. The INPUT CLOSE statement closes the input stream coming from the `p-datfl3.q` file and redirects the input stream to the terminal.

NOTE: With this method, all trailing blanks are stored in the database. To avoid this problem, use the –c or –d option of QUOTER.

You can use QUOTER to prepare files formatted in other ways as well. For example, suppose the field values in your data file are separated by a specific character, such as a comma (,), as in `p-datfl4.d`:

p-datf14.d

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

You can use a special option, –d, (on UNIX) to tell QUOTER what character separates fields. The procedure `p-chgin4.p` reads the comma-separated data from `p-datfl4.d`:

p-chgin4.p

```
OS-COMMAND SILENT quoter -d , p-datfl4.d >p-datfl4.q.

INPUT FROM p-datfl4.q NO-ECHO.

REPEAT:
    CREATE customer.
    SET cust-num name sales-rep.
END.

INPUT CLOSE.
```

Here, the –d option or the DELIMITER qualifier tells QUOTER that a comma (,) is the delimiter between each field in the data file. The output of QUOTER is shown in `p-datfl4.q`:

p-datf14.q

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

This data file is in the standard blank-delimited Progress format. If your data file does not use a special field delimiter that you can specify with the –d QUOTER option or the /DELIMITER qualifier, but does have each data item in a fixed column position, you can use another special option, –c, on Windows and UNIX.

You use the –c option or /COLUMNS to identify the columns in which fields begin and end. For example, suppose your file looks like `p-datfl5.d`:

p-datf15.d

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

The procedure p-chgin5.p uses this data file to create customer records:

p-chgin5.p

```
OS-COMMAND SILENT quoter -c 1-2,4-20,22-24 p-datfl5.d >p-datfl5.q.

INPUT FROM p-datfl5.q NO-ECHO.

REPEAT:
    CREATE customer.
    SET cust-num name sales-rep.
END.

INPUT CLOSE.
```

Because you used the –c option, this procedure produces a data file without trailing blanks.

You can also use QUOTER interactively to reformat your data. You can access QUOTER interactively through the Administration Tool or, in character interfaces, the Data Dictionary. From the main menu, choose Utilities.

7.11.2 Importing and Exporting Data

Sometimes you send data to a file knowing that it will be used later by a Progress procedure. If so, then you also know that the data file must be in standard format with character fields surrounded by quotes. Therefore, instead of just redirecting the output to the file and using the DISPLAY statement to send output to that file, you might use the EXPORT statement.

NOTE: Do not confuse the EXPORT statement with the EXPORT method of the SESSION handle. The EXPORT statement converts data from one format to another while redirecting it. The EXPORT method of the SESSION handle adds procedure names to the export list (list of procedures a requesting procedure can access) of a Progress AppServer. For more information on Progress AppServers, see *Building Distributed Applications Using the Progress AppServer*.

Using the EXPORT Statement

The EXPORT statement sends data to a specified output destination, formatting it in a way that can be easily used by another Progress procedure. For example, `p-export.p` writes customer information to a file:

p-export.p

```
OUTPUT TO p-datfl6.d.

FOR EACH customer:
EXPORT cust-num name sales-rep;.
END.

OUTPUT CLOSE.
```

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

p-datf16.d

```
1 "Lift Line Skiing" "HXM"
2 "Urpon Frisbee" "DKP"
3 "Hoops Croquet Comp" "HXM"
4 "Go Fishing Ltd" "SLS"
5 "Match Point Tennis" "JAL"
       .
       .
       .
```

Now this file is ready to be used as an input source by another Progress procedure. There is no need to process it through QUOTER.

By default, the EXPORT statement uses the space character as a delimiter between fields. You can use the DELIMITER option of the EXPORT statement to specify a different delimiter.

For example, `p-exprt2.p` writes to a file in which field values are separated by commas:

p-exprt2.p

```
OUTPUT TO p-datfl7.d.

FOR EACH customer:
   EXPORT DELIMITER "," cust-num name sales-rep.
END.

OUTPUT CLOSE.
```

The output from p-exprt2.p is written to p-datfl7.d:

p-datf17.d

```
1,"Lift Line Skiing","HXM"
2,"Urpon Frisbee","DKP"
3,"Hoops Croquet Comp","HXM"
4,"Go Fishing Ltd","SLS"
5,"Match Point Tennis","JAL"
       .
       .
       .
```

You can read this file by using the DELIMITER option of the IMPORT statement. More likely, you would prepare a file like this to be read by another application.

For more information on the EXPORT statement, see the *Progress Language Reference*.

7.11.3 Using the PUT Statement

If you need to prepare a data file in a fixed format, perhaps for use by another system, you can use the PUT statement. The following procedure uses PUT statement to unite to a file:

p-putdat.p

```
OUTPUT TO p-datfl8.d.

FOR EACH customer:
    PUT cust-num AT 1 name AT 10 sales-rep AT 40 SKIP.
END.

OUTPUT CLOSE.
```

The output from p-putdat.p is written to p-datfl8.d:

p-datf18.d

```
     1        Lift Line Skiing         HXM
     2        Urpon Frisbee            DKP
     3        Hoops Croquet Comp       HXM
     4        Go Fishing Ltd           SLS
     5        Match Point Tennis       JAL
            .
            .
            .
```

The PUT statement formats the data into the columns specified with the AT options. Only the data is output: there are no labels and no box. The SKIP option indicates that you want each customer's data to begin on a new line.

For more information on the PUT statement, see the *Progress Language Reference*.

7.11.4 Using the IMPORT Statement

The IMPORT statement is the counterpart of the EXPORT statement. It reads an input file into Progress procedures, one line at a time.

Using IMPORT to Read Standard Data

The following example shows IMPORT reading the file exported by the procedure `p-export.p`:

p-import.p

```
INPUT FROM p-datfl6.d.

REPEAT:
   CREATE customer.
   IMPORT cust-num name sales-rep.
END.

INPUT CLOSE.
```

This relies on the input being space separated. You can also use the DELIMITER option of the IMPORT statement to read a file with a different separator.

For example, `p-imprt2.p` reads the file produced by `p-exprt2.p` in the previous section:

p-imprt2.p

```
INPUT FROM p-datfl7.d.

REPEAT:
   CREATE customer.
   IMPORT DELIMITER "," cust-num name sales-rep.
END.

OUTPUT CLOSE.
```

This example reads one line at a time from `p-datfl7.d` into the character-string variable data. It then breaks the line into discrete values and assigns them to the fields of a customer record.

Using IMPORT to Read Non-standard Data

Although the IMPORT statement is used primarily to read data in the standard format written by the EXPORT statement. However, you can use the UNFORMATTED and DELIMITER options of IMPORT to read data in non-standard formats.

When you use the UNFORMATTED option, the IMPORT statement reads one line from the input file. For example, suppose your input file is formatted as follows:

p-datf12.d

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

The lines containing cust–num and sales–rep values can be read with normal IMPORT statements. However, if you try to read the customer name values with a normal IMPORT statement, only the first word of each name is read—the space character is treated as a delimiter. To prevent this, read the name with the UNFORMATTED option, as in `p-impun1.p`.

p-impun1.p

```
INPUT FROM p-datfl2.d.

REPEAT:
   CREATE customer.
   IMPORT customer.cust-num.
   IMPORT UNFORMATTED customer.name.
   IMPORT customer.sales-rep.
END.

INPUT CLOSE.
```

Now, suppose each line of the file contained a cust–num, name, and sales–rep value, but no special delimiters are used. Instead, the fields are defined by their position within the line:

p-datf13.d

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

In p-datfl3.d, the first three character positions in each line are reserved for the cust-num value, the next 17 positions for the name value, and the last three for the sales-rep value. Space characters may occur between fields, but they may also occur within a field value. To process this file with the IMPORT statement, use the UNFORMATTED option to read one line at a time, as shown in p-impun2.p:

p-impun2.p

```
DEFINE VARIABLE file-line AS CHARACTER.

INPUT FROM p-datfl3.d.

REPEAT:
   CREATE customer.
   IMPORT UNFORMATTED file-line.
   ASSIGN customer.cust-num = INTEGER(SUBSTRING(file-line, 1, 2))
          customer.name = TRIM(SUBSTRING(file-line, 4, 17))
          sales-rep = SUBSTRING(file-line, 22, 3).
END.

INPUT CLOSE.
```

After p-impun2.p reads each line, it uses the SUBSTRING function to break the line into field values. It then assigns these values to the appropriate fields in the customer record.

NOTE: If a line in your input file ends with a tilde (~), Progress interprets that as a continuation character. This means, that line and the following line are treated as a single line. Therefore, the IMPORT statement with the UNFORMATTED option reads both lines into a single variable.

What if fields values are separated by a delimiter other than the space character? For example, in p-datfl4.d, field values are separated by commas:

p-datf14.d

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

You could use the UNFORMATTED option of the IMPORT statement to read this file one line at a time and then use the INDEX function to locate the commas and break the line into field values. Another solution is to use the DELIMITER option of the IMPORT statement as shown in `p-impun3.p`:

p-impun3.p

```
INPUT FROM p-datfl4.d.

REPEAT:
   CREATE customer.
   IMPORT DELIMITER "," cust-num name sales-rep.
END.

INPUT CLOSE.
```

In this example, the DELIMITER option specifies that field values are separated by commas rather than by spaces. Therefore, the IMPORT statement parses each line correctly and assigns each value to the appropriate field.

NOTE: You can only specify a single character as a delimiter. If the value you give with the DELIMITER option is longer than one character, then only the first character is used.

For more information on the IMPORT statement, see the *Progress Language Reference*.

8

The Preprocessor

This chapter discusses the Progress language preprocessor, a powerful tool that enhances your programming flexibility. The Progress preprocessor allows you to write applications that are easy to read, modify, and transport to other operating systems.

The preprocessor is a component of the Progress Compiler. Before the Compiler analyzes your source code and creates r-code, the preprocessor examines your source code and performs text substitutions. The preprocessor also conditionally includes blocks of source code to compile. The preprocessor operates on a *compilation unit*, which is a group of files compiled together to produce one completed program. You can think of the preprocessor as a tool that prepares a final version of your source code just before it is compiled.

You control the preprocessor by placing preprocessor directives throughout your source code. A *preprocessor directive* is a statement that begins with an ampersand (&) and is meaningful only to the preprocessor. These directives are described in this chapter.

8.1 &GLOBAL-DEFINE and &SCOPED-DEFINE Directives

The &GLOBAL-DEFINE and &SCOPED-DEFINE directives allow you to define *preprocessor names,* which are compile-time constants. Their syntax is as follows:

SYNTAX

```
&GLOBAL-DEFINE preprocessor-name definition
```

SYNTAX

```
&SCOPED-DEFINE preprocessor-name definition
```

The *preprocessor–name* is a name that you supply, and *definition* is a string of characters. If *definition* is longer that one line, you can use a tilde (~) to continue onto the next line. You must place these directives at the beginning of a line, preceded only by blank or tab characters. The preprocessor trims all leading and trailing spaces from *definition*. You can abbreviate &GLOBAL-DEFINE and &SCOPED-DEFINE to &GLOB and &SCOP, respectively.

You can use reserved Progress keywords as preprocessor names. However, unless they are defined as part of a preprocessor name, reserved Progress keywords cannot be placed in preprocessor expressions.

8.1.1 Example Using &GLOBAL-DEFINE

After you define a preprocessor name, you can reference it within your source code. Wherever a reference appears, the preprocessor substitutes the string of characters that you defined:

```
&GLOBAL-DEFINE MAX-EXPENSE 5000
```

In this example, MAX-EXPENSE is the preprocessor name. Once it is defined, you can reference MAX-EXPENSE from anywhere within your compilation unit (including internal procedures and include files). A reference to a preprocessor name is specified in the form {&*preprocessor–name*}. (For more information on referencing preprocessor names, see the "Referencing Preprocessor Names" section.) Wherever you reference MAX-EXPENSE in your source code, Progress substitutes the text "5000" at compile time. For example, your source code might contain a line like this:

```
IF tot-amount <= {&MAX-EXPENSE} THEN DO:
```

Before the Compiler analyzes your source code, the preprocessor performs the text substitution. The line of code that the Progress Compiler analyzes and compiles is as follows:

```
IF tot-amount <= 5000 THEN DO:
```

You can see how the preprocessor makes your source code easier to maintain. A preprocessor name like MAX-EXPENSE might be referenced many times in your source code. You only have to change one line of code (the definition of MAX-EXPENSE) to change the value of all of the references. In addition, you can see how preprocessor references make your code easier to read: {&MAX-EXPENSE} is more readable and understandable than 5000. Note that the characters in the name MAX-EXPENSE are all uppercase; this is not required, but it makes your source code more readable. Preprocessor names are case insensitive.

If you want to use a preprocessor name as a text string within an application, you must put quotes around it:

```
DISPLAY "{&MAX-EXPENSE}".
```

8.1.2 Preprocessor Name Scoping

Each preprocessor name that you define has a specific scope. The *scope* of a name is the area within its compilation unit where you can access, or reference, the name. You determine the scope of a preprocessor name by where you define it within the compilation unit and by whether you define it with the &GLOBAL-DEFINE or &SCOPED-DEFINE directive. A name defined with the &GLOBAL-DEFINE directive is *globally* defined; a name defined with the &SCOPED-DEFINE directive is *nonglobally* defined.

The scope of a preprocessor name always begins at the directive that defines it; the name is not accessible before the defining directive.

The scope of a globally defined name extends forward from the point of definition to the end of the compilation unit. The scope of a nonglobally defined name extends forward from the point of definition to the end of the file that contains its definition. It does not extend beyond the end of the file to the rest of the compilation unit. It does, however, extend to any files included within the defining file.

If you define a name in a top-level file of a compilation unit, it makes no difference whether the name is globally defined or nonglobally defined. In either case, the name is accessible from the point of definition to the end of the compilation unit. However, if you define a name within an include file, it does make a difference how the name is defined. Globally defined names are accessible to the end of the compilation unit; nonglobally defined names are only accessible within the defining file. See Figure 8–1.

main.p
include.i

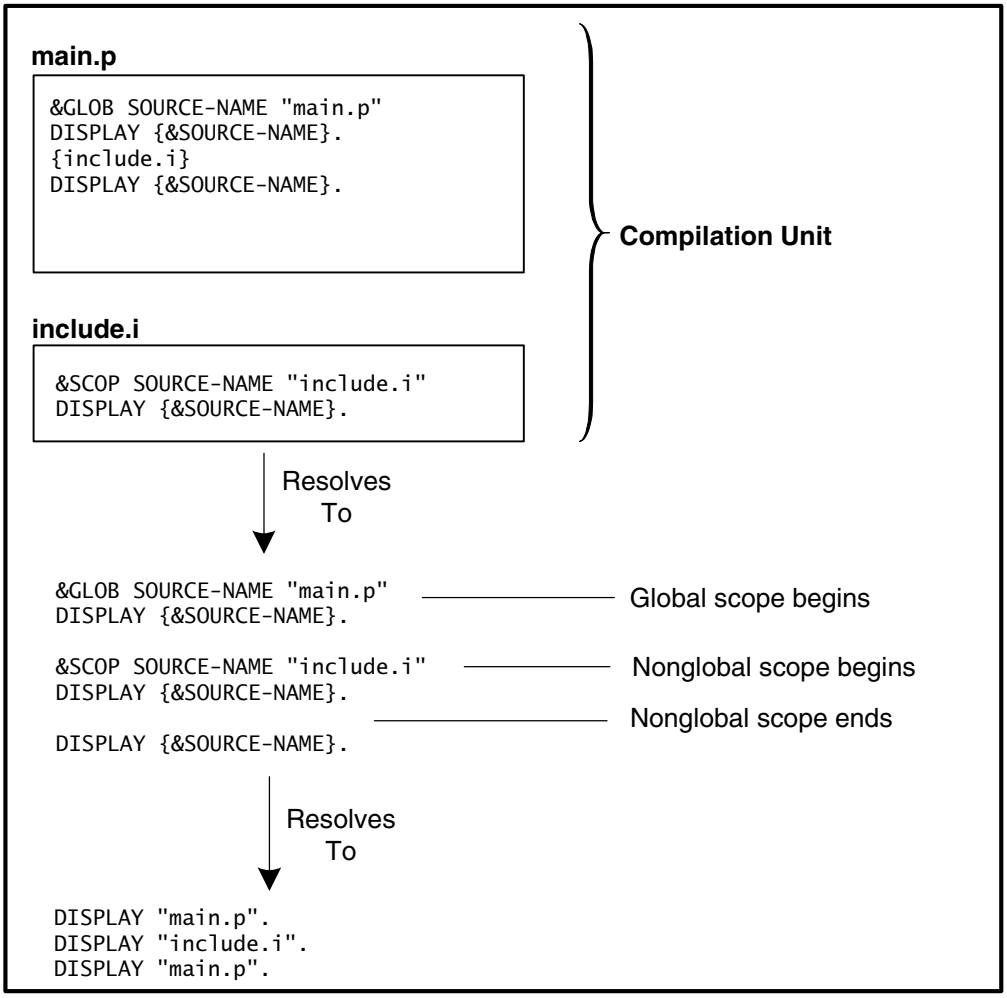


Figure 8–1: Name Scoping

In Figure 8–1, the preprocessor name SOURCE-NAME is defined both globally and nonglobally. In such a situation, the nonglobally defined name takes precedence over the globally defined name within its scope. The nonglobally defined name's scope begins at the first line and ends at the last line of the file `include.i`. Within this scope, all references to SOURCE-NAME access the defined value "include.i".

The innermost name definition always takes precedence within its scope. If you define a name nonglobally and then redefine the name nonglobally within a nested include file, only the innermost name definition is accessible within the nested include file.

You can limit the scope of a name with the &UNDEFINE directive. This directive undefines the name and ends its scope.

8.2 &UNDEFINE Directive

To remove the definition of a preprocessor name, you can undefine the name. This is the syntax for &UNDEFINE:

SYNTAX

```
&UNDEFINE preprocessor-name
```

The *preprocessor–name* is the name you want to undefine.

When you use the &UNDEFINE directive, Progress warns you if the name you want to undefine was not previously defined. The &UNDEFINE directive undefines the most recently defined name within whose scope the &UNDEFINE directive resides. Once you undefine a name, you can redefine it.

The &UNDEFINE directive also undefines named include file arguments. For more information, see the "Arguments to Include Files" section.

8.3 &IF, &THEN, &ELSEIF, &ELSE, and &ENDIF Directives

You use these directives to set logical conditions when compiling blocks of code. This is the syntax for these directives:

SYNTAX

```
&IF expression &THEN
  .
  .
  .
[ &ELSEIF expression &THEN ] ...
  .
  .
  .
[ &ELSE ]
  .
  .
  .
&ENDIF
                                  .
                                  . = block of source code
                                  .
```

The *expression* can contain the following elements: preprocessor name references, the operators listed in Table 8–2, the Progress functions listed in Table 8–3, and the DEFINED( ) preprocessor function. The DEFINED( ) function indicates whether a preprocessor name is defined. For more information on the DEFINED( ) function, see the "DEFINED( ) Preprocessor Function" section.

When the preprocessor evaluates an expression, all arithmetic operations are performed with 32-bit integers. Preprocessor name references used in arithmetic operations must evaluate to integers.

When encountering an &IF directive, the preprocessor evaluates the expression that immediately follows. This expression can continue for more than one line; the &THEN directive indicates the end of the expression. If the expression evaluates to TRUE, then the block of code between it and the next &ELSEIF, &ELSE, or &ENDIF is compiled. If the expression evaluates to FALSE, the block of code is not compiled and the preprocessor proceeds to the next &ELSEIF, &ELSE, or &ENDIF directive. No include files referenced in this uncompiled block of code are included in the final source. You can nest &IF directives.

Table 8–1 indicates how preprocessor expressions are evaluated.

Table 8–1: Preprocessor Expressions

Type of Expression	TRUE	FALSE
LOGICAL	TRUE	FALSE
CHARACTER	Non-empty	Empty
INTEGER	Non-zero	0
DECIMAL	Not supported	Not supported

The expression that follows the &ELSEIF directive is evaluated only if the &IF expression tests FALSE. If the &ELSEIF expression tests TRUE, then the block of code between it and the next &ELSEIF, &ELSE, or &ENDIF directive is compiled. If the &ELSEIF expression tests FALSE, the preprocessor proceeds to the next &ELSEIF, &ELSE, or &ENDIF directive.

The block of code between the &ELSE and &ENDIF directives is compiled only if the &IF expression and the &ELSEIF expressions all test FALSE. If there are no &ELSEIF directives, the block of code is compiled if the &IF expression tests FALSE.

Once any &IF or &ELSEIF expression evaluates to TRUE, no other block of code within the &IF ... &ENDIF block is compiled.

The &ENDIF directive indicates the end of the conditional tests and the end of the final block of code to compile.

Table 8–2: Preprocessor Operators *(1 of 2)*

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
=	Equality
<>	Inequality

Table 8–2: Preprocessor Operators *(2 of 2)*

Operator	Description
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
AND	Logical and
OR	Logical or
NOT	Logical not
BEGINS	Compares the beginning letters of two expressions
MATCHES	Compares two character strings

These operators have the same precedence as the regular Progress 4GL operators.

Table 8–3: Functions Allowed in Preprocessor Expressions

ABSOLUTE	LC	RIGHT–TRIM
ASC	LENGTH	RANDOM
DATE	LIBRARY	REPLACE
DAY	LOG	ROUND
DECIMAL	LOOKUP	SQRT
ENCODE	MAXIMUM	STRING
ENTRY	MEMBER	SUBSTITUTE
ETIME	MINIMUM	SUBSTRING
EXP	MODULO	TIME
FILL	MONTH	TODAY
INDEX	NUM–ENTRIES	TRIM
INTEGER	OPSYS	TRUNCATE
KEYWORD	PROPATH	WEEKDAY
KEYWORDALL	PROVERSION	YEAR
LEFT–TRIM	R–INDEX	–

You can use any of these expressions within a preprocessor expression; they are evaluated at compile time.

8.4 DEFINED( ) Preprocessor Function

The DEFINED( ) preprocessor function takes a preprocessor name or include file parameter as an argument. The preprocessor name argument is not quoted, does not have an ampersand in front of it, and is not in the reference form, {&*preprocessor–name*}. For example, if you had defined the preprocessor name MAX–EXPENSE, the argument would appear as follows:

```
DEFINED(MAX-EXPENSE)
```

The DEFINED( ) function returns a value of 1 if the argument was a name defined with the &GLOBAL–DEFINE directive; a value of 2 if the argument was passed as an include file parameter; and a value of 3 if the argument was a name defined with the &SCOPED–DEFINE directive. If the argument was not defined and was not an include file parameter, this function returns a value of 0. The value returned refers to the definition that is current at the point of the call.

8.5 The &MESSAGE Directive

The &MESSAGE directive allows you to display a message at compile time. This is the syntax for &MESSAGE:

SYNTAX

```
&MESSAGE text-string
```

The *text–string* is a string of characters, preprocessor name references, named include file arguments, or any combination of these that results in a character string to display. The argument *text–string* does not have to be quoted.

8.6 Referencing Preprocessor Names

Use this syntax to reference a preprocessor name:

SYNTAX

```
{ &preprocess-name }
```

The *preprocessor–name* is a name defined in a &GLOBAL–DEFINE or &SCOPED–DEFINE directive, or is a built-in preprocessor name. (For information on built-in preprocessor names, see the "Using Built-in Preprocessor Names" section.) Note that this syntax is identical to the syntax used for referencing argument names in an include file. For more information on how the preprocessor handles interactions between preprocessor names and include file argument names, see the "Arguments to Include Files" section.

A reference to a preprocessor name can occur anywhere within your Progress source code:

```
&GLOBAL-DEFINE FIELD-LIST cust-name cust-num addr city
                  .
                  .
                  .
DISPLAY {&FIELD-LIST}.
```

When you reference a preprocessor name in your source code,Progress replaces the reference with the value that you defined in the &GLOBAL–DEFINE or &SCOPED–DEFINE directive. In this example, Progress replaces {&FIELD–LIST} with the following code:

```
cust-name cust-num addr city
```

8.7 Expanding Preprocessor Names

When Progress encounters a reference to a preprocessor name, Progress replaces the reference with the value of the preprocessor name's definition. This is called expanding the reference:

```
&GLOBAL-DEFINE TABLE customer
&GLOBAL-DEFINE FIELD cust-name
&GLOBAL-DEFINE WHERE-CLAUSE WHERE {&FIELD} <> "Al's Antiques"

FIND NEXT {&TABLE} {&WHERE-CLAUSE}.
```

The first reference in this example is {&FIELD}. The curly brace ({) and ampersand (&) direct Progress to expand the reference. By checking the definition of the preprocessor name FIELD, Progress determines that its value is cust–name. Progress therefore replaces the reference with cust–name. As a result, WHERE–CLAUSE is defined as follows:

```
WHERE cust-name <> "Al's Antiques"
```

The second reference is {&TABLE}. Progress replaces this reference with customer; customer is the definition of the preprocessor name TABLE.

The third reference is {&WHERE–CLAUSE}. Progress replaces WHERE–CLAUSE with its definition, which has already been expanded.

After Progress has made all of these evaluations, the preprocessing phase of compilation is complete. This is the FIND NEXT statement that Progress compiles:

```
FIND NEXT customer WHERE cust-name <> "Al's Antiques".
```

In the previous example, the reference {&FIELD} was evaluated first. Progress also allows you to defer the evaluation of a reference. By placing a tilde (~) in front of a preprocessor name reference, you indicate that you do not want to expand the reference when it is first encountered. (On UNIX, you can also use the backslash character (\ ) to defer evaluation.):

```
&GLOBAL-DEFINE FORM-LIST cust-num ~
FORMAT >>>9 ~{&OTHER-FIELDS} WITH FRAME X
&GLOBAL-DEFINE OTHER-FIELDS cust-name SKIP city state
FIND FIRST customer.
DISPLAY {&FORM-LIST}.
&SCOPED-DEFINE OTHER-FIELDS phone.
DISPLAY {&FORM-LIST}.
```

Progress does not expand the reference {&OTHER–FIELDS} in the definition of FORM–LIST, because {&OTHER–FIELDS} has a tilde in front of it. In this instance, the curly brace has no significance to Progress because of the tilde.

This is how Progress defines FORM–LIST.

```
cust-num FORMAT >>>9 {&OTHER-FIELDS} WITH FRAME X
```

Instead of trying to expand the reference {&OTHER–FIELDS}, Progress simply includes it in the definition of FORM–LIST.

When the preprocessor reaches the first DISPLAY statement, Progress expands the {&FORM–LIST} reference. Since the definition of FORM–LIST includes the {&OTHER–FIELDS} reference, Progress recursively expands {&OTHER–FIELDS}. As a result, Progress replaces {&FORM–LIST} with this.

```
cust-num FORMAT >>>9 cust-name SKIP city state WITH FRAME X
```

The &SCOPED–DEFINE directive changes the value of OTHER–FIELDS. As a result, the second {&FORM–LIST} reference expands like this:

```
cust-num FORMAT >>>9 phone WITH FRAME X.
```

When you defer the evaluation of a reference, make sure that the reference will have a value at the point of expansion. The following example does not do this:

p-proc.p

```
&GLOB file customer
{p-incl.i}

FOR EACH {&file}:
  DISPLAY {&fields}.
END.
```

p-incl.i

```
&GLOB fields ~{&field_list}

&SCOP field_list name max-credit
```

In this example, the mistaken intention was to defer the evaluation of {&field_list} until {&fields} is expanded. However, since field_list is nonglobally defined, its value does not extend beyond the end of `p-incl.i`. As a result, there is no value assigned to field_list at the point of expansion of {&fields}.

8.8 Using Built-in Preprocessor Names

Progress automatically provides the following preprocessor names:

- BATCH–MODE
- OPSYS
- FILE–NAME
- LINE–NUMBER
- SEQUENCE
- WINDOW–SYSTEM

You can reference these built-in preprocessor names anywhere within your source code, including preprocessor expressions. They are read-only; you cannot redefine them with &GLOBAL–DEFINE or &SCOPED–DEFINE directives. However, you can pass them as include file arguments.

8.8.1 Built-in Preprocessor Name References

Table 8–4 lists the name reference for each built-in preprocessor name, and describes the value it provides.

Table 8–4: Built-in Preprocessor Name References *(1 of 2)*

The Reference . . .	Expands to an Unquoted String . . .
{&BATCH–MODE}	Equal to “yes” if the Batch (–b) startup parameter was used to start the client session. Otherwise, it expands to “no”.
{&FILE–NAME}	That contains the pathname of the file being compiled.[1] If you want only the name of the file as specified in the { }Include File Reference, the RUN statement, or the COMPILE statement, use the argument reference {0}.
{&LINE–NUMBER}	That contains the current line number in the file being compiled. If you place this reference in an include file, the line number is calculated from the beginning of the include file.

Table 8–4: Built-in Preprocessor Name References *(2 of 2)*

The Reference . . .	Expands to an Unquoted String . . .
{&OPSYS}	That contains the name of the operating system on which the file is being compiled. The OPSYS name can have the same values as the OPSYS function. The possible values are "UNIX" and "WIN32".[2]
{&SEQUENCE}	Representing a unique integer value that is sequentially generated each time the SEQUENCE preprocessor name is referenced. When a compilation begins, the value of {&SEQUENCE} is 0; each time {&SEQUENCE} is referenced, the value increases by 1. To store the value of a reference to SEQUENCE, you must define another preprocessor name as {&SEQUENCE} at the point in your code you want the value retained.
{&WINDOW–SYSTEM}	That contains the name of the windowing system in which the file is being compiled. The possible values include "MS–WIN95","MS–WINDOWS", and "TTY".[3]

1 When running the source code of a procedure file loaded into the Procedure Editor or the AppBuilder, {&FILE–NAME} expands to a temporary filename, not the name of the file under which the source code might be saved.

2 Progress supports an override option for the &OPSYS preprocessor name that enables applications that need to return the value of MS-DOS for all Microsoft operating systems to do so. For example, if you do not want the value WIN32 to be returned when either Windows 95 or Windows NT operating systems are recognized, you can override this return value by defining the Opsys key in Startup section of the current environment, which can be in the registry or in an initialization file. If the Opsys key is located, the OPSYS preprocessor name returns the value associated with the Opsys key on all platforms.

3 Progress supports an override option for the &WINDOW–SYSTEM preprocessor name that provides backward compatibility. This option enables applications that need the WINDOW–SYSTEM preprocessor name to return the value of MS–WINDOWS for all Microsoft operating systems to do so. To establish this override value, define the WindowSystem key in Startup section of the current environment, which can be in the registry or in an initialization file. If the WindowSystem key is located, the WINDOW–SYSTEM preprocessor name returns the value associated with the WindowSystem key on all platforms.

8.8.2 Quoting Built-in Names and Saving {&SEQUENCE} Values

Note that all built-in name values are returned as unquoted strings. You must quote or not quote the name reference according to the 4GL or preprocessor context of the reference.

Thus, a first reference to {&SEQUENCE}:

```
DEFINE VARIABLE nexti AS INTEGER.
nexti = {&SEQUENCE}.
```

expands to:

```
DEFINE VARIABLE nexti AS INTEGER.
nexti = 0.
```

A second reference to {&SEQUENCE}:

```
DEFINE VARIABLE nextc AS CHARACTER.
nextc = "{&SEQUENCE}".
```

expands to:

```
DEFINE VARIABLE nextc AS CHARACTER.
nextc = "1".
```

Third through sixth references to {&SEQUENCE}, with a value-saving definition:

```
&GLOBAL-DEFINE OCCURENCE {&SEQUENCE} + 1
DEFINE VARIABLE triple AS INTEGER EXTENT 3
    INITIAL [{&SEQUENCE}, {&SEQUENCE}, {&SEQUENCE}].
MESSAGE "Definition of triple complete as variable number" {&OCCURENCE}
    VIEW-AS ALERT-BOX.
```

expands to:

```
DEFINE VARIABLE triple AS INTEGER EXTENT 3
    INITIAL [3, 4, 5].
MESSAGE "Definition of triple complete as variable number" 2 + 1
    VIEW-AS ALERT-BOX.
```

8.9 Nesting Preprocessor References

During compilation, Progress recursively expands nested references, moving from the innermost reference to the outermost reference. To see how Progress expands nested references, look at the following procedure:

p-main.p

```
&GLOBAL-DEFINE X Y
&GLOBAL-DEFINE Y Z
&GLOBAL-DEFINE Z "Hello"

DISPLAY {&{&{&{p-inclde.i}}}}.
```

Progress expands the nested reference {&{&{&{p–inclde.i}}}} by starting with the include file `p-inclde.i`. For this example, `p-inclde.i` is as follows:

p-inclde.i

```
X
```

Progress replaces {p–inclde.i} with the contents of `p-inclde.i`. After replacing {p-inclde.i} with X, Progress then has to expand the following nested reference:

```
DISPLAY {&{&{&X}}}.
```

Progress expands this reference to the following code:

```
DISPLAY {&{&Y}}.
```

This expands to the following code:

```
DISPLAY {&Z}.
```

Finally, this expands to the following code:

```
DISPLAY "Hello".
```

This last DISPLAY statement is the one that Progress finally compiles into your r-code.

8.10 Arguments to Include Files

If a named argument to an include file and a preprocessor name are the same, the named include file argument overrides any previous preprocessor names. However, inside the include file any preprocessor names override the include file arguments.

You can use the &UNDEFINE directive to undefine a named include file argument. However, you cannot undefine an include file argument of the form {1}, {2}, etc.

8.11 Sample Application

The following application illustrates the use of preprocessor names. It allows you to update all or some of the fields in a given database table. By changing a few preprocessor name definitions, you can compile the application to modify different tables.

In addition, you can supply different FORM statements to display your data:

p-editrc.p

```
/* A procedure to illustrate the use of preprocessor names                */
/* This procedure includes p-editrc.i, an include file that uses          */
/* preprocessor names to allow you to easily access and modify any        */
/* file in the demo database.  Before you compile this procedure,         */
/* you must set the following preprocessor definitions:                   */
/*                                                                        */
/* NOTE: This file does NOT set a value to the form_file                  */
/*       preprocessor name.                                               */
/*                                                                        */
/* PREPROCESSOR              DESCRIPTION                                  */
/*     NAME                                                               */
/*                                                                        */
/* file_name   : The database file to modify.                             */
/*               This name must have a value.                             */
/* file_fields : The list of fields in the database file to modify.       */
/*               This name must have a value.                             */
/* form_file   : The name of an external form definition include (.f)     */
/*               file. This name is optional.                             */
/* frame_name  : The name of the frame to be used in accessing the        */
/*               database file.                                           */
/* frame_title : The title to be added to the frame upon display.  If a   */
/*               form include file is used, TITLE should not be part of   */
/*               the form statement.                                      */
/* use-buts    : A comma-separated list of actions you may perform        */
/*               against the database in the browse utility.  Available   */
/*               actions are Next, Prev, Update, Create, and Delete.      */
/* banner      : An example of deferred evaluation, this name references  */
/*               the ifile_ver preprocessor definition, which is set      */
/*               in the include file p-editrc.i.                          */

&GLOBAL-DEFINE file_name customer
&GLOBAL-DEFINE file_fields customer.cust-num customer.name~
               customer.address customer.address2~
               customer.city  customer.st customer.postal-code~
               customer.contact customer.phone
&GLOBAL-DEFINE form_file p-cust.f
&GLOBAL-DEFINE frame_name cust_fr
&GLOBAL-DEFINE frame_title Customer
&GLOBAL-DEFINE Banner File Modification Utility, Version  ~{&ifile_ver}
&GLOBAL-DEFINE use-buts Next,Prev,Update,Create,Delete
{p-editrc.i}
```

The p-editrc.p procedure defines form_file to be a reference to p-cust.f. It also includes p-editrc.i, which references form_file.

p-editrc.i *(1 of 6)*

```
/* An include file to illustrate the use of preprocessor names.         */
/* PREPROCESSOR             DESCRIPTION                                 */
/*     NAME                                                             */
/*                                                                      */
/* ifile_ver : The version of this file. The name is used with the      */
/*             preprocessor name banner defined in the main             */
/*             procedure through deferred evaluation.                   */
/* display   : Combines the file field list and the frame name to make  */
/*             a more manageable statement when displaying and updating */
/*             database records.                                        */
/* but-list  : A list of allowable actions that can be performed        */
/*             during execution of the browse utility.                  */
/* but-nums  : The number of user-specified actions allowed.            */
/*             Generated through the use of {&SEQUENCE}.                */
/*             Displayed during compile time using &MESSAGE.            */

DEFINE BUFFER {&file_name} FOR {&file_name}.
DEFINE VARIABLE do_return AS LOGICAL INITIAL FALSE.

/* Do some initial setup work */
PAUSE 0 BEFORE-HIDE.

&IF "{&frame_name}" = "" &THEN  /* Make up a default frame name */
  &MESSAGE A frame name was not supplied, using default name
  &SCOPED-DEFINE frame_name Default_Frame
&ENDIF

&IF "{&form_file}" <> "" &THEN   /* Define a frame for the user */
  {{&form_file}}   /* This includes the form
                      definition file if one is given */
&ELSE

  &MESSAGE A Form file was not supplied, using default form
  FORM {&file_fields}
  WITH FRAME {&frame_name} ROW 7 CENTERED.
&ENDIF

&SCOPED-DEFINE ifile_ver 1.1a
&SCOPED-DEFINE display {&file_fields} WITH FRAME {&frame_name}30
```

p-editrc.i *(2 of 6)*

```
/* Define buttons to use in browse as indicated by developer through  */
/* the preprocessor name {&use-buts}  */

DEFINE BUTTON next_but LABEL "Next"
  TRIGGERS:
  ON CHOOSE RUN getrec (INPUT "Next")./* On choose find next record */
  END TRIGGERS.

&IF LOOKUP("Next", "{&use-buts}") <> 0 &THEN  /* Will enable button
                                                 for NEXT */
  &SCOPED-DEFINE but-list next_but
  &SCOPED-DEFINE but-nums {&SEQUENCE}
&ENDIF

DEFINE BUTTON prev_but LABEL "Prev"
  TRIGGERS:
  ON CHOOSE RUN getrec (INPUT "Prev").        /* On choose find
                                                 prev record   */
  END TRIGGERS.

&IF LOOKUP("Prev", "{&use-buts}") <> 0 &THEN  /* Will enable button
                                                 for PREV */
  &SCOPED-DEFINE but-list {&but-list} prev_but
  &SCOPED-DEFINE but-nums {&SEQUENCE}
&ENDIF

DEFINE BUTTON up_but LABEL "Update"       /* On choose update record */
  TRIGGERS:
  ON CHOOSE UPDATE {&display}.
  END TRIGGERS.

&IF LOOKUP("Update", "{&use-buts}") <> 0 &THEN  /* Will enable button
                                                   for UPDATE */
  &SCOPED-DEFINE but-list {&but-list} up_but
  &SCOPED-DEFINE but-nums {&SEQUENCE}\
&ENDIF
```

p-editrc.i *(3 of 6)*

```
DEFINE BUTTON cr_but LABEL "Create"   /* On choose create and update */
  TRIGGERS:
  ON CHOOSE
  DO:
    DEFINE BUFFER localbuf FOR {&file_name}.
    /* First save current location in the file */
    FIND FIRST localbuf WHERE ROWID(localbuf) = ROWID({&file_name}).
    /* Make the new record */
    CREATE {&file_name}.
    UPDATE {&display}.
    /* Restore the location in the file */
    FIND FIRST {&file_name} WHERE ROWID({&file_name}) = ROWID(localbuf).
  END.

  ON END-ERROR RETURN NO-APPLY.
/* If end-error, restores current location */
  END TRIGGERS.

&IF LOOKUP("Create", "{&use-buts}") <> 0 &THEN
/* Enable button for CREATE */
  &SCOPED-DEFINE but-list {&but-list} cr_but
  &SCOPED-DEFINE but-nums {&SEQUENCE}
&ENDIF

DEFINE BUTTON del_but LABEL "Delete"     /* On choose, delete record */
  TRIGGERS:
  ON CHOOSE RUN delrec.
  END TRIGGERS.

&IF LOOKUP("Delete", "{&use-buts}") <> 0 &THEN
/* Enable button for DELETE */
  &SCOPED-DEFINE but-list {&but-list} del_but
  &SCOPED-DEFINE but-nums {&SEQUENCE}
&ENDIF

DEFINE BUTTON ret_but LABEL "Return"
  TRIGGERS:
  ON CHOOSE do_return = TRUE.
  END TRIGGERS.

DEFINE BUTTON quit_but LABEL "Quit"
  TRIGGERS.
  ON CHOOSE QUIT.
  END TRIGGERS.

&SCOPED-DEFINE but-list {&but-list} ret_but quit_but
&MESSAGE {&but-nums} Out of 5 user definable buttons are active
```

p-editrc.i *(4 of 6)*

```
FORM  next_but Prev_but up_but cr_but Del_but ret_but quit_but
  WITH FRAME but_fr ROW 4 CENTERED
  TITLE "Work With {&file_name} Records" .

DISPLAY "{&Banner}" WITH FRAME ban_fr CENTERED .
VIEW FRAME but_fr.
ENABLE {&but-list} WITH FRAME but_fr.

REPEAT:  /* Cycle through records, or add one. */
  WAIT-FOR CHOOSE OF next_but,Prev_but,up_but,cr_but,Del_but,
                          ret_but, quit_but IN FRAME but_fr.
  IF do_return THEN
    RETURN.
  DISPLAY {&display}.
END.
```

p-editrc.i *(5 of 6)*

```
/*************************************************************************/
/* Procedure to find records in the database file being modified.     */
/* It takes one parameter, action, which indicates whether you move   */
/* forward or backward in the file.                                      */
/*************************************************************************/

PROCEDURE getrec.
  DEFINE INPUT PARAMETER action AS CHARACTER.
  DEFINE BUFFER localbuf FOR {&file_name}.

  IF action = "Next" OR action = "" THEN
  DO:
    FIND LAST localbuf.
    IF ROWID(localbuf) = ROWID({&file_name}) THEN
    DO:
      BELL.
      MESSAGE "You are already at the end of this file.".
      RETURN ERROR.
    END.
    ELSE
      FIND NEXT {&file_name} .
  END.
  ELSE
  DO:
    FIND FIRST localbuf.
    IF ROWID(localbuf) = ROWID({&file_name}) THEN
    DO:
      BELL.
      MESSAGE "You are already at the start of this file.".
      RETURN ERROR.
    END.
    ELSE
    DO:
      FIND PREV {&file_name} .
    END.
  END.
END PROCEDURE.   /* getrec */
```

p-editrc.i *(6 of 6)*

```
/*************************************************************************/
/* Procedure to delete a record from the file.  Cannot delete the last  */
/* record from the file if it is the only record in the file. When the  */
/* record is deleted, the next record is found. If this is not available*/
/* the previous record is found.                                         */
/*************************************************************************/

PROCEDURE delrec.
  DEFINE BUFFER localbuf FOR {&file_name}.

  /* Before delete of record, find another record to display after */
  FIND localbuf WHERE ROWID(localbuf) = ROWID({&file_name}).
  /* find the next record */
  FIND NEXT {&file_name} NO-ERROR.
  IF NOT AVAILABLE {&file_name} THEN
  DO:
    /* reset location in file */
    FIND {&file_name} WHERE ROWID({&file_name}) = ROWID(localbuf).
    /* find previous record */
    FIND PREV {&file_name} NO-ERROR.
  END.
  IF NOT AVAILABLE {&file_name} THEN
  DO:
    BELL.
    MESSAGE "You may not delete the last record in the file.".
    RETURN ERROR.
  END.
  ELSE
    DELETE localbuf.
END PROCEDURE.  /* delrec */
```

p-cust.f

```
/* Form definition for updating the customer file.                  */
/* Please note that there is no TITLE in the frame phrase.          */

FORM
  customer.cust-num  AT 5
  customer.name      AT 5
  customer.address   AT 5
  customer.address2  AT 5
  customer.city AT 5 customer.st  AT 25 customer.postal-code AT 37
  SKIP (1)
  customer.contact AT 5
  customer.phone AT 5
  WITH FRAME cust_fr SIDE-LABELS ROW 9 CENTERED.
```

9

Database Access

This chapter describes how to access records in a Progress database. It includes information on the following topics:

- Connecting and disconnecting databases
- Logical database names and aliases
- Data-handling statements
- Adding and deleting records
- Fetching records and field lists
- Joining tables
- Word indexes
- Sequences and how to access them
- Database trigger considerations
- Using the RAW data type
- Multi-database programming techniques
- Creating and using a schema cache file
- Using query, buffer, and buffer-field objects

9.1 Database Connections

A Progress application can access one or more Progress or non-Progress databases simultaneously. The databases can be located on different operating systems using different networking protocols. To access non-Progress databases, you must use the appropriate DataServer, such as ORACLE, or C-ISAM. You must connect to a database before you can access it. There are four ways to connect to a database:

- As an argument when starting Progress
- With the CONNECT statement (in the Progress Procedure Editor or in a Progress procedure)
- With the Progress Data Dictionary
- Using the auto-connect feature

Note that a multi-user application can simultaneously connect to a database for only as many times as specified in the Number of Users (–n) startup parameter. For information on exceeding this user count, see the "Connection Failures and Disruptions" section. For more information on database connection management, see the *Progress Startup Command and Parameter Reference*, the *Progress Installation and Configuration Guide Version 9 for Windows* or *Progress Installation and Configuration Guide Version 9 for UNIX* and the *Progress Client Deployment Guide*.

9.1.1 Connection Parameters

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

For multiple database programming, the most important connection parameter is the Physical Database Name (–db) startup parameter, which allows you to specify multiple databases.

The Number of Databases (–h) startup parameter lets you set the number of connected databases allowed during a Progress session, up to a maximum of 240 (the default is 5).

You can group connection parameters in a text file called a *parameter file*. The Parameter File (–pf) connection parameter invokes the parameter file when you connect to a database. Parameter files ordinarily have .pf file extensions. For more information on parameter files, see the *Progress Startup Command and Parameter Reference*.

9.1.2 The CONNECT Statement

The CONNECT statement allows you to connect to a database from a Progress procedure or directly from the Progress Procedure Editor. The CONNECT statement has the following syntax:

SYNTAX

```
CONNECT {   physical-name       [ parameters ]
        | -db physical-name     [ parameters ]
        | -pf parameter-file    [ parameters ]
      }
      [
        {   -db physical-name   [ parameters ]
          | -pf parameter-file  [ parameters ]
        }
      ] ... [ NO-ERROR ]
```

physical-name

An argument that represents the actual name of the database on a disk. The first *physical–name* argument you specify in a CONNECT statement does not require the Database Name (–db) parameter. All subsequent *physical–names* must be preceded by the –db parameter.

parameter-file

The name of a parameter file that contains database connection information. See the *Progress Startup Command and Parameter Reference* for more information on parameter files.

parameters

One or more database connection parameters. Each connection parameter applies to the most recently specified database (–db). For the parameters you can specify, see the information on client connection parameters in the *Progress Startup Command and Parameter Reference*.

NO-ERROR

This argument suppresses the error condition, but still displays the error message when an attempt to CONNECT to a database fails.

Although you can connect to several databases within one CONNECT statement, it is a good idea to connect only one database per CONNECT statement, because a connection failure for one database causes a termination of the current CONNECT statement. However, databases already connected when the statement terminates stay connected. In cases like this, it is a good idea to use the CONNECTED functions to see which databases were connected and which were not.

Here is an example of using a parameter file with the CONNECT statement:

```
CONNECT -pf parm3.pf.
```

In this example, the CONNECT statement uses the `parm3.pf` file to connect to the appldb1 database.

parm3.pf

```
-db appldb1
```

A single procedure cannot connect to and then access a database. The following code fragment does not run:

```
/* NOTE: this code does NOT work */

CONNECT sports.
FOR EACH sports.customer:
  DISPLAY customer.
END.
```

By splitting this example into a procedure and a subprocedure, you can connect to the database in the main procedure, and then access it in the subprocedure:

topproc.p

```
CONNECT sports.
RUN p-subproc.p.
```

subproc.p

```
FOR EACH sports.customer:
  DISPLAY customer.
END.
```

For more information on the CONNECT statement, see the *Progress Language Reference*.

9.1.3 Auto-connect

The auto-connect feature uses information stored in one database to connect to a second database at run time. The database that contains the connect information is the primary application database; it must be connected before Progress can execute the auto-connect.

Progress executes the auto-connect when a precompiled procedure references data from the second database. It executes immediately prior to running the precompiled procedure. It does not work with procedures run from the Editor or otherwise compiled on-the-fly.

If you use a CONNECT statement while you are connected to the primary database, Progress merges the information in the CONNECT statement with the information in the primary database's auto-connect list. If there is a conflict, the information in the CONNECT statement takes precedence. For more information, see the CONNECT Statement reference entry in the *Progress Language Reference*.

For information on how to set up an auto-connect list in the primary database, see the *Progress Client Deployment Guide*.

9.1.4 Multi-database Connection Considerations

When you develop a multi-database application, keep in mind the following information on database connections:

- Connect all databases accessed by the application at compile time. Once compiled, the application can run no matter where you store the databases or how you connect to them.
- Establish unique logical names for each of the databases you connect. Use these names to reference the databases within your application. These names are stored in the r-code of the application at compilation time. For more information on logical database names, see the "Logical Database Names" section.

9.1.5 Run-time Connection Considerations

As mentioned in the "Database Connections" section, there are various ways to connect at run time. For multi-database applications, you must decide which technique to use.

When connecting in the run-time environment, you must use the same logical database names that you used during compilation. If you need to use a different logical database name for some reason, you may add an alias to the database to allow r-code files with other names to run.

9.1.6 Connection Failures and Disruptions

Database connections may fail for a number of reasons:

- The database server for multi-user database access is not available—perhaps the network is down or it has not been started or the network parameters are incorrect.
- The logical name of the application database already exists for the current session.
- The syntax for a connection parameter is wrong.
- The connection exceeds the maximum number of users per database (–n).

Table 9–1 lists the default failure behavior for each of the run-time connection techniques

.

Table 9–1: Connection Failure Behavior

Connection Technique	Default Connection Failure Behavior
At Progress startup	The Progress session does not run.
CONNECT statement	The procedure executes up to the CONNECT statement where the connection failure occurs. The connection failure raises the error condition for the procedure. Progress error processing does not undo database connections or disconnections. Any database connected in the procedure before the failed database connection remains connected. See Chapter 5, "Condition Handling and Messages," for more information on error handling.
Auto-connect	The procedures that contain a reference to the auto-connect database do not run, and a stop or break condition results in the calling procedure.

Progress displays any connection error messages at the point of failure.

Before running a procedure or subprocedure, Progress checks the procedure for database references. For each reference, Progress searches through all of the connected databases, trying to find the corresponding database. If the database is not found, Progress again searches all of the connected databases for an auto-connect list with an entry for that database. If an entry is found, Progress connects the database. If no entry is found, Progress does not run the procedure and raises a stop or break condition in the calling procedure.

Server, network, or machine failures can disrupt an application, even after a successful connection. If a failure occurs while a subprocedure is accessing a database, Progress raises a stop or break condition in the calling procedure.

The following sections present a number of recommendations to help you manage database connection failures and disruptions in an application.

Using CONNECT with NO–ERROR

If you designate a startup procedure with the Startup Procedure (–p) parameter, always use the NO–ERROR option with the CONNECT statement. If a CONNECT statement fails, the NO–ERROR option suppresses the resulting error condition, allowing the procedure to continue executing. Although the NO–ERROR option bypasses ordinary error processing, Progress still displays an error message. For more information on using the NO–ERROR option, see Chapter 5, "Condition Handling and Messages."

After a connection failure, if a subprocedure tries to access the unconnected database, Progress raises a stop or break condition in the calling procedure. Therefore, before you execute any subprocedures that access a database, you test whether the database is connected. You can do this with the CONNECTED function.

Using the CONNECTED Function

The CONNECTED function tests whether a database is connected. This function helps you to route program control around portions of an application that might be affected by database connection failures and disruptions. This example tests a database connection with the CONNECTED function.

```
IF NOT CONNECTED(logical-name)
  THEN DO:
    MESSAGE "CONNECTING logical-name".
    CONNECT logical-name... NO-ERROR.
  END.
IF CONNECTED(logical-name)
THEN RUN procedure.
ELSE MESSAGE "DATABASE NOT AVAILABLE".
```

If the database is not connected, the above code attempts to connect the database. This example only runs the procedure if the database is connected. Auto-connect precludes the use of the CONNECTED function to test for database connections. For more information on the CONNECTED function, see the *Progress Language Reference*.

9.1.7 Progress Database Connection Functions

Table 9–2 lists the Progress functions that allow you to test database connections, get information on connected databases, and get information on the types of databases that you can access.

Table 9–2: Progress Database Functions

Progress Function	Description
CONNECTED	Tests whether a given database is connected.
DATASERVERS	Returns a character string containing a list of database types supported by the installed Progress product. For example, “Progress,ORACLE.”
DBTYPE	Returns the database type of a currently connected database. For example “Progress,” “ORACLE,” etc.
DBRESTRICTIONS	Returns a character string that describes the Progress features that are not supported for a particular database. For example, if the database is an ORACLE database, the return string is: “LAST,PREV,RECID,SETUSERID”.
DBVERSION	Returns a “7” if a connected database is a Version 7 database and an “8” if it is a Version 8 database. For non-Progress databases, you see the appropriate version number of your database.
FRAME–DB	Returns a character string that contains the logical name of the database for the field in which the cursor was last positioned for input.
NUM–DBS	Returns the number of connected databases.
LDBNAME	Returns the logical name of a currently connected database.
PDBNAME	Returns the physical name of a currently connected database.
SDBNAME	Returns the logical name of a schema holder for a database.

Some of these functions take arguments; for more information on these functions, see the *Progress Language Reference*.

Use these functions to perform various tasks related to connection, such as determining connection status. The following procedure displays a status report for all connected databases:

p-infor.p

```
DEFINE VARIABLE x AS INTEGER FORMAT "99".
DO x = 1 TO NUM-DBS WITH DOWN:
  DISPLAY PDBNAME(x) LABEL "Physical Database"
    LDBNAME(x) LABEL "Logical Name"
    DBTYPE(x) LABEL "Database Type"
    DBRESTRICTIONS(x) LABEL "Restrictions"
    SDBNAME(LDBNAME(x)) LABEL "Schema Holder DB".
END.
```

9.1.8 Conserving Connections vs. Minimizing Overhead

As more and more users connect to a database, the number of available connections decreases. You can conserve the number of connections by connecting temporarily, and then disconnecting when the application is done accessing the database. This frees up the connection for another user. If the number of available connections is scarce, an application should release connections wherever possible.

However, each connect and disconnect generates *connection overhead*, which is the sum of operations necessary for an application to connect and disconnect a database. The time used for connection overhead depends on the nature of the database connections. A connection to a database over a network generally takes longer than a connection to a local database. While the application connects to a database, the end user waits.

When there are plenty of available connections, you might want to reduce the number of connects and disconnects to minimize overhead. The best way to do this is to connect all application databases only once at application startup. With this technique, all connection overhead occurs once at application startup and does not occur again throughout the life of the application.

9.2 Disconnecting Databases

By default, Progress disconnects all databases at the end of a session. You can explicitly disconnect a database with the DISCONNECT statement, which has the following syntax:

SYNTAX

```
DISCONNECT logical-database-name
```

The *logical–database–name* represents the logical name of a connected database. It can be an unquoted string, a quoted string, or a character expression.

A DISCONNECT statement does not execute until all active procedures that reference the database end or stop.

mainprc1.p

```
CONNECT -db mydb -1.
RUN p-subproc1.p.
```

subproc1.p

```
RUN subproc2.p.
FOR EACH mydb.customer:
  UPDATE name address city state postal-code.
END.
```

subproc2.p

```
DISCONNECT mydb.
```

In these procedures, the mydb database is not disconnected until the end of the `p-subproc1.p` procedure.

9.3 Logical Database Names

When you connect to a database, Progress automatically assigns that database a default logical name for the current Progress session. The default logical name consists of the physical database name without the .db file extension. For example, if you connect to a database with the physical name `mydb1.db`, the default logical database name is mydb1. Progress uses the logical database name mydb1 to resolve database references, and stores it in the compiled r-code of any procedures that you compile that reference the `mydb1.db` database.

The Logical Database Name (–ld) connection parameter allows you to specify a logical database name other than the default.

The example below establishes the logical name firstdb for the physical database `mydb1.db` during the current Progress session:

```
pro mydb1 -ld firstdb
```

When you develop and compile an application to run on the `mydb1.db` database, it is the logical name, not the physical name, that Progress stores in the r-code. You must use the logical name firstdb in your procedures (.p) to reference the `mydb1.db` database.

Logical database names allow you to change physical databases without recompiling an application. To run a compiled application on a new physical database without recompiling, the new database must have identical structure and time stamp or Cyclic Redundancy Check (CRC) values for the tables accessed by the application and must be connected with the same logical name (or alias) used to compile the application:

```
pro mydb2 -ld firstdb
```

The example above establishes the logical name firstdb for a new physical database `mydb2.db`.

NOTE: Progress does not allow you to run the Progress Data Administration tool or character Data Dictionary against a database connected with the logical name DICTDB.

A database connection fails if the logical database name of the database that you connect to has the same logical name as an already connected database of the same database type (Progress, ORACLE, etc.). If you try to do this, Progress assumes that database is already connected and ignores the request.

For information about the characters allowed in the logical name, see the *Progress Startup Command and Parameter Reference*.

9.4 Database Aliases

An *alias* is a synonym for a logical database name. You use an alias as a database reference in Progress procedures in place of a logical database name.

You establish a logical database name when you connect a Progress session to a physical database. You create and assign an alias to a logical database name of an already connected database using the CREATE ALIAS statement. By reassigning an alias to different logical database names, you can run a compiled procedure on other connected databases that have identical structure and time stamps or CRC values for the tables referenced by the procedure. A logical database name can have more than one alias, but each alias refers to only one logical database name at a time.

The Progress Data Dictionary offers an example of alias usage. The Progress Data Dictionary is a general-purpose Progress application that works on any database that has DICTDB as an alias. The first database connected during a Progress session automatically receives the alias DICTDB. During a Progress session, you can reassign the DICTDB alias to another connected database with the Select Working Databases option on the Database menu in the Progress Data Dictionary.

NOTE: Because of the need to reassign of the DICTDB alias, Progress does not allow you to run the Progress Data Dictionary against a database connected with the logical name DICTDB.

Use the CREATE ALIAS statement during a Progress session to assign or reassign an alias to a connected database. You can use this statement from the Progress editor or from an application procedure. This is the syntax for the CREATE ALIAS statement:

SYNTAX

```
CREATE ALIAS alias FOR DATABASE logical-name [ NO-ERROR ]
```

The *alias* argument can be an unquoted string, a quoted string, or an expression. The *logical–name* argument represents the existing logical name of a connected database. It can be an unquoted string, a quoted string, or an expression. You cannot create an alias that is the same as the logical database name of a connected database. The named database must be connected unless you use the NO–ERROR option.

When you create an alias, Progress logs the alias assignment in a table that remains in memory for the current session. If you use the DISCONNECT statement to disconnect a database from within an application, all existing aliases assigned to the logical database name remain in the alias table until the end of the Progress session. Later, if you connect to a database with the same logical database name during the same Progress session, you can use the same aliases to reference that logical database name. If you create an alias that already exists in the session alias table, Progress replaces the existing alias with the new alias. This allows you to reassign existing aliases to new logical database names.

The DELETE ALIAS statement allows you to delete an alias from the alias table of the current Progress session:

SYNTAX

```
DELETE ALIAS alias
```

The *alias* argument represents an alias that exists in the current alias session table.

9.4.1 Creating Aliases in Applications

You cannot assign and reference an alias in the same procedure. You must assign an alias to a logical database name prior to compiling and running procedures that use that alias. For example, `alias1.p` below fails to compile when it reaches the FOR EACH statement, because you cannot assign and reference the alias myalias in a single procedure:

alias1.p

```
/* Note that this procedure does not work */
CREATE ALIAS myalias FOR DATABASE sports.
FOR EACH myalias.customer:
  DISPLAY customer.
END.
```

To solve this problem, split `alias1.p` into two procedures, as in the following examples:

alias2.p

```
CREATE ALIAS myalias FOR DATABASE sports.
RUN dispcust.p.
```

dispcust.p

```
FOR EACH myalias.customer:      /* myalias.customer */
  DISPLAY customer.             /* myalias.customer */
END.
```

9.4.2 Compiling Procedures with Aliases

As a general rule, you should not compile procedures while using aliases. This potentially leads to confusion about what database name ends up in the r-code. For example, the Progress dictionary programs which contain DICTDB as a qualifier must be compiled in a session where the database concerned has the logical name DICTDB. That way, when the dictionary r-code is run, it can run for any database name as long as the database uses the DICTDB alias. In summary, logical names are useful for compiling, aliases are useful at run time.

However, you can still compile procedures using aliases. Suppose you have three databases called eastdb, centraldb, and westdb, all of which contain customer tables with identical structure and time stamps or CRC values. Your application requires a general report procedure that can run against any of these customer tables. To begin developing your general report procedure, start Progress and connect to the eastdb, centraldb, or westdb database using the logical name myalias. Develop and compile the customer report procedure using myalias to prefix table and field references as shown in the previous procedure `dispcust.p`. All unqualified table and field references in the report procedure `dispcust.p` resolve to the myalias logical name at compilation time. When you finish compiling your procedure, disconnect from the database represented by the myalias logical database name.

You must assign an alias to the logical database name of a connected database prior to running any procedure that uses that alias as a database reference. Therefore, you need to develop a procedure that uses the CREATE ALIAS statement to assign the myalias alias to a logical database name and run the report procedure (`dispcust.p`) as a subprocedure. See the `alias2.p` procedure above.

Although it is not recommended because of the possible confusion it can cause, you may need to compile a procedure with an alias. To do this, you must know how Progress places database references in the r-code of the procedure at compilation time. Remember, you must assign the alias to a logical database name prior to compiling the procedure. In general, use only the alias as a database prefix for all table and field references in the general-purpose procedures, and always fully qualify every database reference within such a procedure. Use the following syntax:

SYNTAX

```
alias.table-name alias.table-name.field-name
```

If you use this syntax for every table or field reference in your procedure, only the alias will be represented as the database reference in the procedure's r-code after compilation. Note that this is the only exception to the rule that you should never compile using aliases.

Unqualified table and field references within procedures may cause both the alias and the logical database name for a particular physical database to be represented in the r-code for the procedure at compilation time.

9.4.3 Using Shared Record Buffers with Aliases

Be careful when using shared buffers with aliases. If you reference a shared buffer after changing the alias that initially was used in defining it, a run-time error results:

main2.p

```
CREATE ALIAS myalias FOR DATABASE sports1.
RUN makebuf.p.
```

makebuf.p

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

disp.p

```
DEFINE SHARED BUFFER mybuf FOR myalias.customer.
FOR EACH mybuf:
  DISPLAY mybuf.
END.
```

In this example, procedure `main2.p` calls `makebuf.p`, which in turn calls `disp.p`. The alias myalias is created in `main.p`, with reference to database sports1. In `makebuf.p`, the shared buffer mybuf is defined for the table myalias.customer. Then, in the next line, myalias is changed, so that it now refers to database sports2. When an attempt is made to reference shared buffer mybuf in procedure `disp.p`, a run-time error occurs, with the message "disp.p unable to find shared buffer for mybuf."

9.5 Data-handling Statements

Statements that move data from one location to another are called *data-handling statements*.

Progress stores data in various locations—a database, a record buffer, a screen buffer, etc. A database stores application data on disk. A record buffer stores data temporarily while a procedure accesses the data, and stores the values of variables used in the procedure. A screen buffer stores data being displayed on the screen or being sent to another output destination; it also stores data that is being entered from the terminal or other input source.

Figure 9–1 shows how the data-handling statements move data.

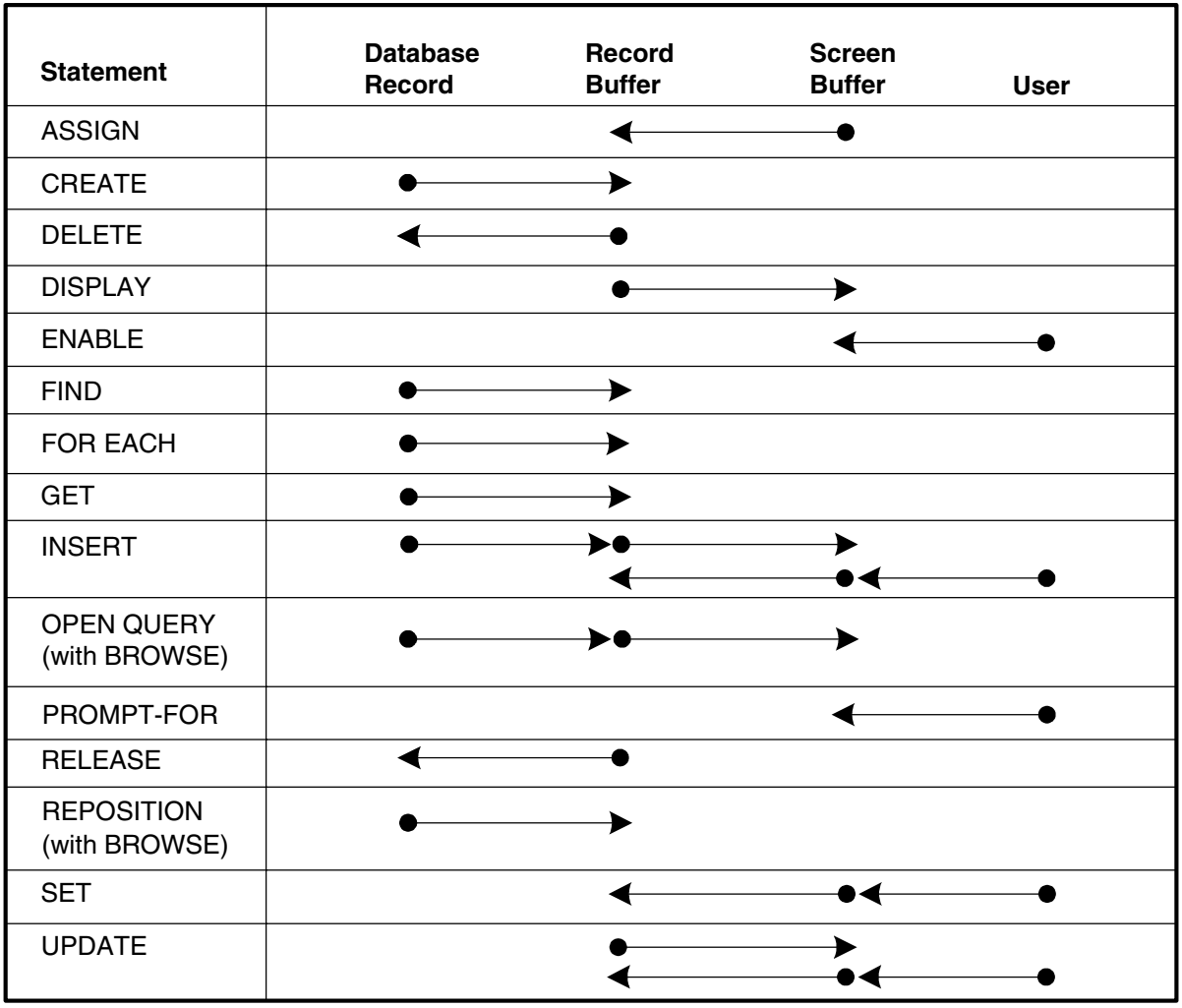


Figure 9–1: Data Movement

To use Figure 9–1, read each statement from top to bottom. The beginning of the first arrow (the dot) indicates the source of the data. The arrowhead of the last arrow indicates where the data is finally stored. For example, the DISPLAY statement gets data from a record buffer and moves it into a screen buffer. Once in the screen buffer, the data is displayed on screen. Also, note that INSERT creates a new database record, moves it to the record buffer, and then to the screen buffer where the user enters data, which is then moved back to the record buffer.

Some statements are made up of other statements. For example, the UPDATE statement is made up of the DISPLAY, PROMPT–FOR, and ASSIGN statements, and performs the same steps as these other statements. First, it copies data from the record buffer to the screen buffer (DISPLAY). Second, it allows data to be entered into the screen buffer (PROMPT–FOR). Third, it copies the data back to the record buffer (ASSIGN).

You can use statements as building blocks, using only as many as you need to do specific tasks. For example, the INSERT statement is very powerful and combines several processing steps into one statement. But in some situations, it is less flexible than using the CREATE, DISPLAY, and SET statements individually (or the CREATE and UPDATE statements).

Figure 9–2 shows the primary data-handling statements and shows which statements are composed of other statements.

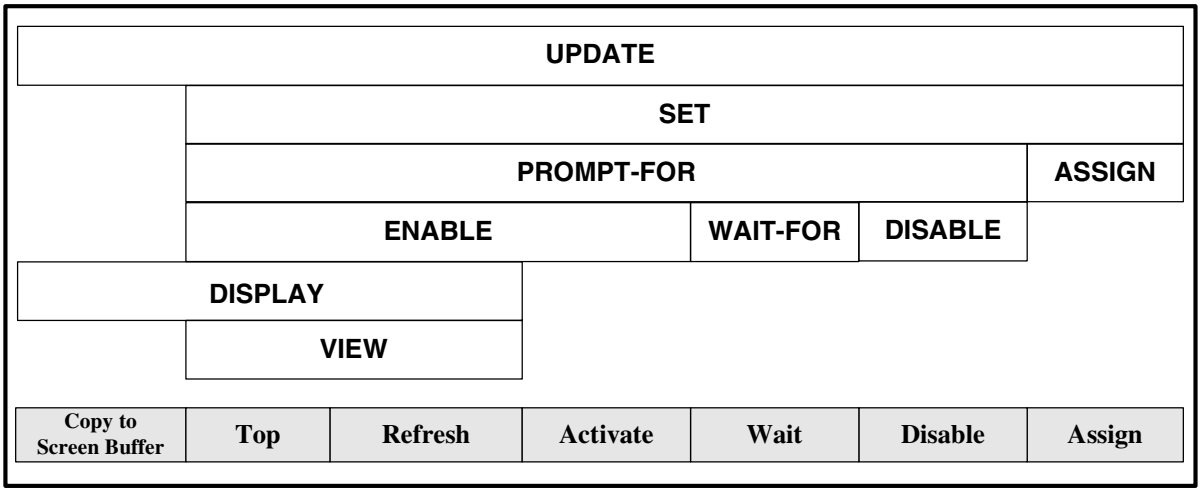


Figure 9–2: The Primary Data-handling Statements

Note that the UPDATE, SET, and ASSIGN statements do not actually write records to the database. However, at the end of a transaction (or at the end of the record scope or after an explicit RELEASE), Progress writes all modified records to the database.

If you modify a record using INSERT, UPDATE, or SET, Progress assigns the change to the record buffer (and, hence, eventually to the database). However, if you use the ENABLE statement, you must explicitly assign any changes with the ASSIGN statement.

For more information on the statements in Figure 9–2, see the *Progress Language Reference*.

9.6 Adding and Deleting Records

To add records to a database, you can use either the CREATE or INSERT statements. The CREATE statement places a newly created record in the database, but does not display the record or request user input. All fields in the record are set to the initial values specified in the Data Dictionary. The CREATE statement causes any CREATE trigger associated with the table to execute. This trigger may set fields in the record to new values. The INSERT statement not only creates the record, but also displays the record and requests input.

The INSERT statement is composed of four other statements—the CREATE, DISPLAY, PROMPT–FOR and ASSIGN statements. Three of these statements (DISPLAY, PROMPT–FOR, and ASSIGN) comprise the UPDATE statement. You can emulate an INSERT statement by using the four statements, or by using the CREATE and UPDATE statements. This is more flexible than using the INSERT statement alone.

For example, the INSERT statement always displays fields in the order they are defined in the schema. The UPDATE statement lets you specify the order they are displayed. Note that if you use an INSERT statement, the CREATE trigger is executed before the record is displayed.

To delete records from a database, use the DELETE statement. The DELETE statement causes any DELETE trigger associated with the table to execute. For more information on database triggers, see Chapter 11, “Database Triggers.”

9.7 Defining a Set of Records to Fetch

To fetch records, you must first define the set of records that you want Progress to fetch. For example, the following statement defines the set of all customer records:

```
FOR EACH Customer:
```

Progress allows you to define a set of records in a variety of ways. You define the set of records in a Record phrase. For more information on the syntax of Record phrases, see the *Progress Language Reference*. You can specify a Record phrase for the following Progress statements:

- FIND
- FOR [EACH]
- OPEN QUERY
- DO PRESELECT
- REPEAT PRESELECT

The examples given below illustrate some of the flexibility that you have when defining a set of records. The Record phrases are highlighted. You can build complex Record phrases using the AND and OR operands.

This example defines a set of one record (customer 11):

```
FIND Customer WHERE cust-num = 11.
```

This example uses the word-indexed field Comments to define a set of records (all customer records containing the word “ship”):

```
FOR EACH Customer WHERE Comments CONTAINS "ship":
```

This example creates the subset of all Order records with the Customer number = 11:

```
OPEN QUERY ordqry FOR EACH Order WHERE Cust-Order = 11.
```

This example defines a set of customer records (those between 25 and 50) to be preselected:

```
REPEAT PRESELECT EACH Customer WHERE cust-num > 25 AND cust-num < 50:
```

9.8 Fetching Records

After you define a set of records, Progress must fetch the records. How Progress fetches records depends in part on which statements you use to fetch the records. Table 9–3 summarizes the differences in record fetching between the FIND, FOR EACH, OPEN QUERY, and PRESELECT statements.

Table 9–3: Record Fetching

Statement to Define Set of Records	Statement to Fetch the Records	Structure Used to Locate Records
FIND	FIND	Index
FOR EACH	FOR EACH	Results List or Index
OPEN QUERY	GET	Results List or Index
DO PRESELECT or REPEAT PRESELECT	FIND	Results List

9.8.1 Order of Fetched Records

The FOR EACH statement, OPEN QUERY statement, and PRESELECT option may use multiple indexes to satisfy a query. When multiple indexes are used, the order of returned records is not guaranteed. You can enforce an order by using the BY option.

The following example returns the selected customer records in ascending credit–limit order and within credit limit in name order:

```
FOR EACH customer BY Credit-limit BY Name:
```

9.8.2 Sample Record Fetches

The following examples show some of the many ways you can access records. These examples are not meant to be exhaustive, but merely to show some of the flexibility provided by Progress.

- Specify the FIRST, LAST, NEXT, or PREV options to step through all records in a particular sequence:

```
FIND FIRST Customer.
```

- Specify boolean expressions to describe the record or records to be fetched:

```
FOR EACH Customer WHERE Credit-Limit > 5000 AND Balance < 12000:
```

- Specify a constant value of the primary index for the record. This works only if the primary index is single-component. Also, this technique is not supported for the OPEN QUERY statement:

```
FIND Item 12.
```

- Specify one or more field values that are currently in the screen buffer or record buffer:

```
PROMPT-FOR Customer.Cust-num WITH FRAME abc.
FIND Customer USING FRAME abc Customer.Cust-num.
DISPLAY Customer.Name.
```

- Specify a previously found related record from another table (The two tables must share a field with the same name. That field must have a UNIQUE index in at least one of the tables.):

```
FIND FIRST Customer.
FOR EACH Order OF Customer: /* This uses the cust-num field */
```

- Specify a CONTAINS clause on a word-indexed field:

```
FOR EACH Customer WHERE Comments CONTAINS "ship":
```

You cannot use a CONTAINS clause with the FIND statement. You can use CONTAINS only with the OPEN QUERY and FOR EACH statements.

9.8.3 ROWID and RECID Data Types

Progress provides two structure types to support record fetches. One structure type, the index, you define in the schema of your database. The other structure type, a *results list*, is temporary; Progress builds it at run time. The results list associated with a DO, REPEAT, or OPEN QUERY statement with the PRESELECT option is sometimes called a *preselect list*.

In addition, there are two data types, ROWID and RECID, that allow you to retrieve a pointer to a fetched record. You can use this pointer to:

- Position to and retrieve a record from a results list.
- Refetch a record and modify its lock status.
- Store as a future record reference.

In addition to the examples in this section, you can see ROWID at work in the chapters that describe the following Progress features:

- **Browse interactions** — Chapter 10, "Using the Browse Widget."
- **Transactions and updates** — Chapter 12, "Transactions."
- **General lock management** — Chapter 13, "Locks."

ROWID versus RECID

ROWID is supported by all DataServers. Earlier Progress versions provide RECID as the only way to fetch a record pointer (supported in this version for backward compatibility). RECID is limited to a 4-byte record address supported by only a few DataServers and standard Progress. ROWID provides a variable byte string that can represent a record address for any type of DataServer. For DataServers that use the 4-byte address supported by RECID (including Progress), ROWID also uses a four-4 value. Thus, there is no loss in performance using the more portable ROWID instead of RECID.

Returning Record ROWID Values

Progress provides a function named after the ROWID data type to return ROWID values. Given a buffer name, the ROWID function returns the ROWID of the current record in the buffer. This example fetches the first customer record, and if it has a balance, refetches it to lock it for update:

```
DEFINE VARIABLE custrid AS ROWID.

FIND FIRST customer NO-LOCK.
custrid = ROWID(customer).

IF balance > 0 THEN DO:
    FIND customer WHERE ROWID(customer) = custrid EXCLUSIVE-LOCK.
    UPDATE customer.
END.
```

Storing and Retrieving ROWID and RECID Values

As shown in the previous example, you can store ROWID values in ROWID variables. You can also store them in work table fields. Thus, the following are valid ROWID storage definitions:

```
DEFINE VARIABLE wkrid AS ROWID EXTENT 20.
DEFINE WORK-TABLE wtrid FIELD wkrid AS ROWID.
```

You cannot store ROWID values in database or temporary tables, but you can store their hexadecimal string representations using the STRING function. You can then retrieve the string as a ROWID value using the TO–ROWID function:

```
DEFINE TEMP-TABLE ttrid
    FIELD ridfld AS CHARACTER.

FOR EACH customer FIELDS (balance) WHERE balance = 0 NO-LOCK:
    CREATE ttrid.
    ASSIGN ttrid.ridfld = STRING(ROWID(customer)).
END.

DO TRANSACTION:
    FOR EACH ttrid:
        FIND customer WHERE ROWID(customer) = TO-ROWID(ttrid.ridfld).
        DELETE customer.
    END.
END.
```

You can store RECID values directly in a database or temporary table.

Additional 4GL Support

Several additional statements use ROWID and RECID values directly. For example, the REPOSITION statement sets the query position to a record given by its ROWID or RECID. For more information, see the "Results Lists" section.

Also, because RECID is not supported by all DataServers, Progress provides the DBRESTRICTIONS function to indicate whether a particular DataServer supports it.

Converting from RECID to ROWID

When changing an application to use ROWID that currently uses RECID, you can complete the change with only a keyword substitution if your application:

- Does not reference RECID values as integers
- Does not store RECID values in database or temporary tables

Otherwise, after you change all "RECID" references to "ROWID", you must rewrite your integer references to use character strings. If you use database or temporary tables, you must also convert the relevant fields to CHARACTER fields, and use the STRING and TO–ROWID functions to store and retrieve ROWID values. However, note that some DataServers build a string for a single ROWID that can reach up to several hundred bytes (including a complete WHERE clause).

All DataServer tables support ROWID references except those, such as views, that do not have unique row identifiers. DataServers from earlier Progress versions also support ROWID references. Versions 7.3A and later use an internal RECID that transparently converts to a ROWID in the client.

Writing DataServer-portable Applications

The least portable feature of ROWID references is the scope of a ROWID reference and when it changes for each DataServer. To maximize portability, follow these rules:

- Always assign values to unique keys before returning the ROWID value of a record. Some DataServers use a unique key to generate the ROWID value.
- If you UNDO a DELETE of a record for which you have stored the ROWID value, return the ROWID value again after the UNDO. It might be different after the record is recreated.
- If you update a unique key value for a record, return its ROWID again to replace any prior value.
- Never expect a record to have the same ROWID value between sessions.

Note that each DataServer uses a different method to generate a ROWID for a table, and sometimes for different tables in the same database. Therefore, never expect ROWID values to be identical, or even compatible, between otherwise duplicate tables from different DataServers.

For more information on ROWID value construction and scope for your DataServer, see your Progress DataServer guide.

9.8.4 Results Lists

A results list is a list of ROWIDs that satisfy a query. The results list allows you to quickly access the records in the record set you define, and allows Progress to make the records available one at a time, as needed.

When more than one row satisfies a query, Progress doesn't lock all of the records and hold them until you release them. Instead, when a specific record is needed, Progress uses the record's ID to go directly to the record, locking only that record. By going directly to the record, Progress also ensures that you have the latest copy of the record.

When you open a query, Progress does not normally build the entire results list. Instead it initializes the results list and adds to it as needed. The NUM–RESULTS function returns the number of records currently in the results list. This is not necessarily the total number of records that satisfy the query.

Whether and when Progress builds the results list depends on the type of the query. As shown in Table 9–3, the DO or REPEAT PRESELECT statements always use a results list, while the FOE EACH and OPEN QUERY statements sometimes use a results list.

Queries have the following characteristics:

- **Scrolling versus non-scrolling** — A query is scrolling if you specify SCROLLING in the DEFINE QUERY statement or if you define a browse for the query. You can use the REPOSITION statement to change your current position within the results list. For a non-scrolling query, you can only move sequentially through the rows by using the FIRST, LAST, NEXT, and PREV options of the GET statement. Scrolling queries must use a results list (often initially empty); non-scrolling queries might not.

- **Index-sorted** — If the order of the rows in the query can be determined by using a single index, the query is *index-sorted*. For an index-sorted query, Progress can use the index to order records without a results list. However, if the query requires additional sorting it must also be presorted.

- **Presorted** — If the query requires any sorting without an index or with multiple indexes, it must be *presorted*. For a presorted query, Progress must read all the records, sort them, and build the complete results list before any records are fetched.

- **Preselected versus non-preselected** — You can force Progress to build a preselected results list by specifying PRESELECT on the OPEN QUERY, DO, or REPEAT statement.

Table 9–4 summarizes how the results list is built for each type of query.

Table 9–4: Results Lists for Specific Query Types

Query Type	Results List
Non-scrolling, index-sorted, no preselection	None.
Scrolling, no sorting, no preselection	Empty list[1] established when query is opened. Records are added to the results list as needed.
Presorted or preselected	Complete list built when query is opened.

[1] If a browse is defined for the query, the results list initially contains one row.

There are two cases where Progress has to build the entire results list when you first open the query:

- When you have explicitly used the PRESELECT option. In this case, Progress performs a preselection phase in which it reads each record of the query. You can specify the lock type to use during the preselection phase. (For an OPEN QUERY, the lock type you specify in the OPEN QUERY statement is used for the preselection phase.) These locks are released immediately unless the preselection occurs within a transaction.

- When you have not used the PRESELECT option, but have specified sort criteria that cannot be performed using an index. In this case, Progress performs a presort phase in which it reads each record of the query with NO–LOCK and builds the results list.

For example, the following statement explicitly uses the PRESELECT option. This forces Progress to build the entire results list immediately:

```
OPEN QUERY cust-query PRESELECT EACH customer WHERE credit-limit > 1500.
```

If you had used FOR instead of PRESELECT, Progress would not have had to build the entire results list because it uses the primary index to fetch the records. It could use this index to find the first or last record for the query; it only needs to search forward or backward through the index until it finds a record that satisfies the WHERE clause.

You can use the PRESELECT option of the OPEN QUERY statement when you need to know immediately how many records satisfy the query or you can use it to immediately lock all the records that satisfy the query.

Progress also builds a complete results list when you open a query with a sort condition that cannot be resolved using a single index. Suppose you open a query on the customer table as follows:

```
OPEN QUERY cust-query FOR EACH customer BY city.
```

Because there is no index for the city field, Progress must retrieve all the records that satisfy the query (in this case, all the customer records), perform the sort, and build the entire results list before any records can be fetched. Until it performs this sort, Progress cannot determine the first or last record for the query. If an index were defined on the city field, Progress could use that index to fetch the records in sorted order (forwards or backwards) and would not need to build the results list in advance.

If the sort conditions for a query can be resolved using a single index, you can use the GET statement with the FIRST, LAST, NEXT, and PREV options on that query. For example, the following query is sorted using the primary index:

```
OPEN QUERY custqry FOR EACH customer.
GET FIRST custqry.
DISPLAY cust-num name.    /* Display first record */
PAUSE.
GET NEXT custqry.         /* Display second record */
DISPLAY cust-num name.
PAUSE.
GET LAST custqry.         /* Display last record */
DISPLAY cust-num name.
PAUSE.
GET PREV custqry.
DISPLAY cust-num name.   /* Display second-to-last record */
```

Because the sorting is done with a single index, you can move freely forwards and backwards within the query.

NOTE: If you want to use the REPOSITION statement on a query, you must make the query scrolling by specifying the SCROLLING option in a DEFINE QUERY statement.

Navigating a Results List

As shown in Table 9–3, results lists are associated with the OPEN QUERY and GET statements. However, Progress only guarantees a results list if you first define the query with the SCROLLING option:

```
DEFINE QUERY custqry FOR customer SCROLLING.
```

This option indicates to Progress that you want to use the results list for multi-directional navigation.

You can use the REPOSITION statement to specify how many places forward or backward you want to move, so that you can skip over a given number records. It also allows you to move to a specific ROWID.

The REPOSITION statement changes your location in the results list but does not actually fetch the record (unless the query is associated with a browse widget). To actually fetch records in a results list, you use the GET statement. The following example illustrates how the REPOSITION statement works:

```
DEFINE QUERY q FOR customer SCROLLING.
DEFINE VARIABLE rid AS ROWID.       /* to save the ROWID of cust 4 */
OPEN QUERY q FOR EACH cust.
GET NEXT q.                 /* gets cust no. 1 */
GET NEXT q.                 /* gets cust no. 2 */
                            /* query is positioned ON cust 2 */
GET PREV q.                 /* gets cust no. 1 */
REPOSITION q FORWARD 0.     /* query is positioned BETWEEN cust 1 and 2 */
GET NEXT q.                 /* gets cust no. 2 */
                            /* query is positioned ON cust 2 */
REPOSITION q FORWARD 1.     /* query is positioned BETWEEN cust 3 and 4 */
GET NEXT q.                 /* gets cust no. 4 */
rid = ROWID(cust).          /* query is positioned ON cust 4 */
REPOSITION q BACKWARD 2.    /* query is positioned BETWEEN cust 2 and 3 */
GET PREV q.                 /* gets cust no. 2 */
REPOSITION q TO ROWID(rid). /* query is positioned BETWEEN cust 3 and 4 */ GET
NEXT q.                     /* gets cust no. 4 */
```

After a record is fetched (with a GET statement), the results list position is on the ROWID, so that GET NEXT gets the next record, and GET PREV gets the previous record. After a REPOSITION, the position is always **between** two records. Thus, REPOSITION FORWARD 0 repositions the results list immediately after the current record. GET NEXT fetches the next record; GET PREV fetches the previous record. REPOSITION FORWARD 1 repositions the results list between the next record and the record after it.

To find the total number of rows in a results list, you can use the NUM–RESULTS function. To find the current position within a results list, you can use the CURRENT–RESULT–ROW function.

As Table 9–3 shows, Progress also creates results lists for FOR EACH statements and for DO and REPEAT statements with the PRESELECT phrase. However, you cannot navigate freely through a results list created for the FOR EACH statement. If the results list was created for the FOR EACH statement, then Progress automatically steps through the results list in sorted order:

```
FOR EACH customer BY name:
   DISPLAY name city state.
END.
```

Within a PRESELECT block, you can use the FIND statement to move backwards and forwards through the results list:

```
/* This code fragment displays all customers in descending order */

DO PRESELECT EACH customer:
  FIND LAST customer.    /* last position in list */
  DISPLAY cust-num name WITH FRAME a DOWN.
  REPEAT:
    FIND PREV customer.  /* move backward through list */
    DOWN WITH FRAME a.
    DISPLAY cust-num name WITH FRAME a.
  END.
END.
```

NOTE: A powerful way of navigating a record set is with the browse widget. For more information, see Chapter 10, "Using the Browse Widget."

9.8.5 FIND Repositioning

After executing FOR EACH statements or FIND statements, Progress might reposition subsequent FIND statements to the last record fetched (except for FIND statements occurring in PRESELECT blocks). For repositioning to occur, the same record buffer must be used. Also, repositioning after FOR EACH statements can differ between Version 8.0B and Versions 8.0A and earlier, depending on the options you use.

NOTE: Repositioning does not occur for a subsequent FIND if the FIND specifies a unique key (that is, the FIND does not use the NEXT or PREV options).

Repositioning After FIND Fetches

Progress uses index cursors to keep track of what record you last fetched. This is important if you use the FIND statement to fetch a record. For example, depending on what was the last record fetched, the following statement returns a different record:

```
FIND NEXT customer.
```

If you had last fetched the first customer record, this statement would fetch the second customer record. If you had just fetched the fourth customer record, this statement would fetch the fifth customer record.

A table can have multiple indexes, and the cursor position in each index dictates what the next record is in that index. For example, the following code fragment fetches customers 1 and 21:

```
FIND FIRST customer.
DISPLAY cust-num name country  postal-code.
PAUSE.
FIND NEXT customer USE-INDEX country-post.
DISPLAY cust-num name country postal-code.
```

In the country–post index, the **next** record after customer 1 is customer 21. Progress uses the index cursor to establish the correct context.

Sometimes cursor repositioning is tricky. For example, the following code fragment returns customer 6 and customer 7 (you might expect customer 6 and customer 2):

```
FIND FIRST customer WHERE cust-num > 5.
DISPLAY cust-num name.
PAUSE.
FIND NEXT customer WHERE cust-num > 1.
DISPLAY cust-num name.
```

The first FIND statement causes Progress to reposition the cust–num index cursor to point to customer 6. The second FIND statement begins the search from that location, not from the beginning of the cust–num index.

9.9 Fetching Field Lists

When fetching records with a FOR EACH statement or query, Progress typically retrieves all the fields of a record, whether or not your application needs them. This can have a costly impact on performance, especially when browsing records over a network.

Progress automatically optimizes preselected and presorted fetches from remote Progress databases using field lists. A *field list* is a subset of the fields that define a record and includes those fields that the client actually requires from the database server. For preselected and presorted fetches, Progress can deduce this field list at compile time from the code. You can also specify field lists explicitly for many types of Progress record fetches, including:

- Queries
- FOR statements
- DO PRESELECT statements
- REPEAT PRESELECT statements
- SQL SELECT statements

This section explains how to use field lists in the Progress 4GL. For information on specifying field lists in SQL SELECT statements, see the *Progress SQL-89 Guide and Reference*.

9.9.1 Field List Benefits

Field lists can provide significant performance benefits when:

- **Browsing over a network** — With the reduction in network traffic, tests show that field lists can increase the performance of fetches from both remote Progress and DataServer databases by factors of from 2 to 10, depending on the record size and number of fields in the list.
- **Fetching from local DataServers** — Some local DataServers can yield significant improvements in fetch performance when only a portion of the record is read.
- **Fetching from remote DataServers that send multiple rows at a time** — DataServers that package multiple rows per network message show noticeable performance gains using field list fetches.

In general, these benefits mean that a multi-user query application can handle more network clients when fetching field lists than when fetching whole records. For more information on the availability and benefits of field list fetches with DataServers, see the Progress DataServer guides.

9.9.2 Specifying Field Lists in the 4GL

You can specify a field list in two different 4GL contexts:

- Following each buffer name specified in a DEFINE QUERY statement
- Following the buffer name specified for fetching in each Record phrase of a FOR, DO PRESELECT, or REPEAT PRESELECT statement

This is the syntax for specifying a field list in all cases:

SYNTAX

```
record-bufname
  [ FIELDS [ ( [ field ... ] ) ]
  | EXCEPT [ ( [ field ... ] ) ]
]
```

The *record–bufname* reference specifies the table buffer you are using for the fetch, and a *field* reference specifies a field in the table. If *field* is an array reference, the whole array is fetched. The FIELDS form lists the fields included in the fetch, and the EXCEPT form lists the fields excluded from the fetch. FIELDS without *field* references fetches enough information to return the ROWID of a record, and EXCEPT without *field* references or *record–bufname* alone fetches a complete record.

Queries versus Record Selection Blocks

For a query, you must specify the field lists in the DEFINE QUERY statement, not the Record phrase of the OPEN QUERY statement. Thus, the following two procedures, `p-fldls1.p` and `p-fldls2.p`, are functionally equivalent:

p-fldls1.p

```
DEFINE QUERY custq FOR customer FIELDS (name cust-num balance).
OPEN QUERY custq PRESELECT EACH customer.
REPEAT:
    GET NEXT custq.
    IF AVAILABLE(customer)
        THEN DISPLAY name cust-num balance.
        ELSE LEAVE.
END.
```

p-rldls2.p

```
REPEAT PRESELECT EACH customer FIELDS (name cust-num balance):
    FIND NEXT customer.
    DISPLAY name cust-num balance.
END.
```

Shared Queries

If you specify field lists in a NEW SHARED query, the matching SHARED query definitions in external procedures only have to include the FIELDS or EXCEPT keyword as a minimum field list reference. The complete field list is optional in the SHARED query definitions, but required in the NEW SHARED query definition.

However, Progress raises the ERROR condition when you run a procedure with a SHARED query if you:

- Specify a field list in the SHARED query to match a NEW SHARED query that has no field list.
- Do not specify a field list reference in the SHARED query to match a NEW SHARED query that has a field list.

9.9.3 Avoiding Implied Field List Entries

Under certain conditions, Progress adds fields to a specified field list when they are required by the client to complete the record selection. The most common case is when you specify a join condition with the OF option and you do not include the join field in the field list.

```
FOR EACH customer FIELDS (name),
    EACH invoice FIELDS (invoice-num amount) OF customer:
    DISPLAY customer.name customer.cust-num
            invoice.invoice-num invoice.amount.
```

In this case, Progress adds customer.cust–num to the list because the client requires it to complete the join between the customer and invoice tables.

However, never rely on implied entries in a field list to provide the fields you need. If you reference an unfetched field, Progress raises the ERROR condition at run time. Future versions of Progress might change the criteria used to distribute record selection between the client and server. Thus in the previous example, if the server does not return customer.cust–num to the client to complete the join, the DISPLAY statement executes with ERROR. This might happen, for example, if the DataServer you use or a future version of Progress actually completes the specified join on the server.

Therefore, **always** specify all the fields you plan to reference in your field lists. There is no extra cost for specifying a field that Progress can also add implicitly.

9.9.4 Updating and Deleting with Field Lists

After fetching a field list, if your procedure updates the fetched record, Progress always rereads the complete record before completing the update. In fact, if you fetch a field list with EXCLUSIVE–LOCK, Progress reads the complete record anyway. This is to ensure proper operation of updates and the before-image (BI) file. (For information on BI files, see the *Progress Client Deployment Guide*.)

Also, if you delete a record after fetching a field list for it, Progress rereads the complete record for the following cases:

- If you delete from a Progress database, Progress always rereads the complete record.
- If you delete from a DataServer database, Progress rereads the complete record if the delete occurs in a subtransaction, in order to create the local before-image (LBI) note. (For information on LBI files, see the *Progress Client Deployment Guide*.)

Thus, if you fetch with NO–LOCK or SHARE–LOCK, avoid using field lists if you expect to perform a high ratio of updates or deletes to fetches. For example, this is an inefficient construction with a Progress database:

```
FOR EACH customer FIELDS (name balance):
    DISPLAY "Deleting customer" name "with balance:" balance.
    DELETE customer.
END.
```

This procedure rereads the complete record for each field list that it fetches, and thus fetches twice per record. Without the field list, the same procedure fetches only once per record.

Updating and Deleting with Query Field Lists

For queries, especially those attached to a browse widget, there is little concern about updates and deletes, because the complete results list for a query is built before any updates or deletes take place. In this case, updates and deletes are selective over the entire query. Therefore, field lists can greatly enhance query performance, no matter how many updates or deletes a browse user completes. For more information on browse widgets, see Chapter 10, “Using the Browse Widget.”

9.9.5 Cursor Repositioning and Field Lists

FOR and relative FIND statements reposition all open index cursors for the same buffer. (For more information, see the “FIND Repositioning” section.) However, in order to reposition a buffer’s index cursors, Progress must have all the index fields available in the buffer. If you fetch a field list that excludes some of these fields, Progress marks the relevant indexes as being incorrectly positioned.

This does not matter for the query or fetch loop that uses the field list, because Progress might never reference the relevant indexes. However, if you later execute a FIND NEXT or FIND PREV using one of the badly positioned indexes, Progress raises the ERROR condition and the FIND fails.

To avoid this error, always specify the fields in your field lists that participate in the indexes you reference.

9.9.6 Field List Handling in Degenerate Cases

When you specify field lists, Progress is very flexible where it cannot make use of them. Progress either returns complete records automatically or allows you to bypass field list processing completely for the following cases:

- **Progress server versions** — Earlier database servers interpret field list requests as requests for complete records. Progress sends queries in such a way that earlier servers do not need to know that the field list requests are included.

- **DataServers that do not support SHARE–LOCK** — If you execute a SHARE–LOCK fetch from a DataServer that does not support SHARE–LOCK, Progress ignores the field list and retrieves the complete record. In some circumstances, Progress must replace the fetched record with a new version. Because the SHARE–LOCK is meaningless, the new version can be different from the previous one, and Progress requires the complete record to ensure that the user receives the correct data.

  NOTE: You can avoid SHARE–LOCK fetches with the help of the CURRENT–CHANGED function. For more information, see Chapter 13, "Locks."

- **Multiple queries returning the same record** — If you specify two queries that return two different field lists for the same record, Progress cannot always consolidate the two lists. In that case, Progress must reread the complete record. Such occurrences are rare, but they can impose a minor performance penalty with little impact on overall performance.

- **Deployment problems** — While programmers must ensure that their field lists are complete, run-time errors can still occur during application deployment. This is especially likely when a new database (schema) trigger is defined that references an unfetched field. To work around this type of problem, Progress provides the Field List Disable (–fldisable) client session parameter. This is a run-time parameter that causes Progress to ignore field lists in the r-code and fetch complete records. This might degrade performance, but allows the application to run until a fix can be made.

Thus, while using field lists can lead to difficulties, Progress provides a way around most of them. And when used optimally, the performance gains can make field lists well worth the extra attention they might require.

9.10 Joining Tables

When you read from multiple tables using a single statement, such as a FOR EACH or OPEN QUERY statement, Progress returns the results as a join of the tables. A *join* is a binary operation that selects and combines the records from multiple tables so that each result in the results list contains a single record from each table. That is, a single join operation combines the records of one table with those of another table or combines the records of one table with the results of a previous join. Figure 9–3 shows how you can join three tables.

```
FOR EACH Table1, EACH Table2 WHERE C11 = C21, EACH Table3 WHERE C22 = C31:
    DISPLAY C11 C12 C21 C22 C31 C32 WITH TITLE "Join123".
```

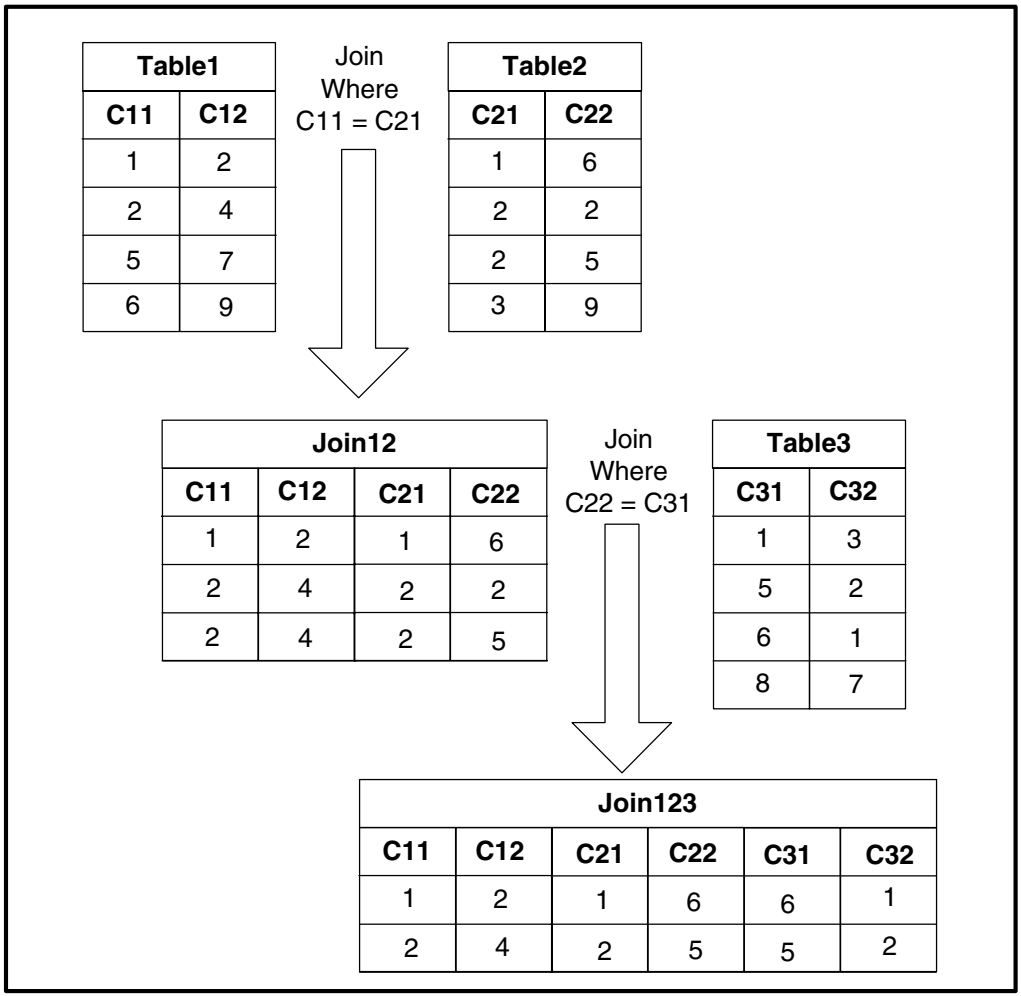


Figure 9–3: Inner Joins

A table or prior join can be either on the left- or right-hand side of a join operation. Thus, the results of joining the three tables in Figure 9–3 depends on two join operations—one join between Table1 (left-hand side) and Table2 (right-hand side) and one join between the first join (left-hand side) and Table3 (right-hand side). The relations C11 = C21 and C22 = C31 represent *join conditions*, conditions that determine how one table is related to the other (that is, which records selected from one table join with the records in the other table). How the records from joined tables are combined depends on the order of the tables in the join, the type of join operation used, and the selection criteria applied to each table.

9.10.1 Specifying Joins in the 4GL

Progress supports two types of joins in the 4GL:

- **Inner join** — Supported in all statements capable of reading multiple tables, including the FOR, DO, REPEAT, and OPEN QUERY statements. An *inner join* returns the records selected for the table (or join) on the left side combined with the related records selected from the table on the right. For any records not selected from the right-hand table, the join returns no records from either the left or right sides of the join. Thus, only related records that are selected from both sides are returned for an inner join. Table 9–3 shows an example of inner joins.

- **Left outer join** — Supported only in the OPEN QUERY statement. A *left outer join* returns the records selected for an inner join. In addition, for each set of records selected from the table (or join) on the left side, a left outer join returns unknown values (?) from the table on the right where there is no record selected or otherwise related to the records on the left. That is, records from the left-hand table (or join) are preserved for all unmatched records in the right-hand table. Figure 9–4 shows an example of left outer joins using the same tables as in Figure 9–3.

NOTE: For versions earlier than V8, Progress supports only the inner join.

Specifying the Type of Join

The Record phrase that specifies the right-hand table of a join also indicates the type of join operation. A Record phrase specifies an inner join by default. To specify a left outer join, you include the [LEFT] OUTER–JOIN option anywhere in the Record phrase. Where you specify a list of multiple Record phrases in a record-reading statement, the join logic allows you to specify only one set of contiguous inner joins at the beginning (left side) of the list and one set of contiguous left outer joins at the end (right side) of the list. Each right-hand Record phrase of a left outer join must contain the OUTER–JOIN option up to and including the last left outer join in the list. For more information, see the "Mixing Inner and Left Outer Joins" section.

Relating and Selecting Tables

You can specify join conditions (table relations) using the OF option or WHERE option of the Record phrase that specifies the join. The OF option specifies an implicit join condition based on one or more common field names in the specified tables. The common field names must participate in a unique index for at least one of the tables. The WHERE option can specify an explicit join based on any field relations you choose, and you can use this option further to specify selection criteria for each table in the join. For an inner join, if you do not use either option, Progress returns a join of all records in the specified tables. For a left outer join, you must relate tables and select records using the OF option, the WHERE option, or both options.

NOTE: Work tables and temporary tables can also participate in joins. However, work tables do not have indexes. So, if you specify join conditions using the OF option with a work table, the other table in the join must be a database or temporary table with a unique index for the fields in common. For more information on work tables and temporary tables, see Chapter 15, "Work Tables and Temporary Tables."

The following code fragment generates the left outer joins shown in Figure 9–4. Note that the Record phrase for the right-hand table of each join specifies the OUTER–JOIN option. As Figure 9–4 shows, the primary benefit of a left outer join is that it returns every record on the left-hand side, whether or not related data exists on the right.

```
DEFINE QUERY q1 FOR Table1, Table2, Table3.
OPEN QUERY q1 FOR EACH Table1, EACH Table2 OUTER-JOIN WHERE C11 = C21,
    EACH Table3 OUTER-JOIN WHERE C22 = C31.
GET FIRST q1.
DO WHILE AVAILABLE(Table1):
    DISPLAY C11 C12 C21 C22 C31 C32 WITH TITLE "Join123".
    GET NEXT q1.
END.
```

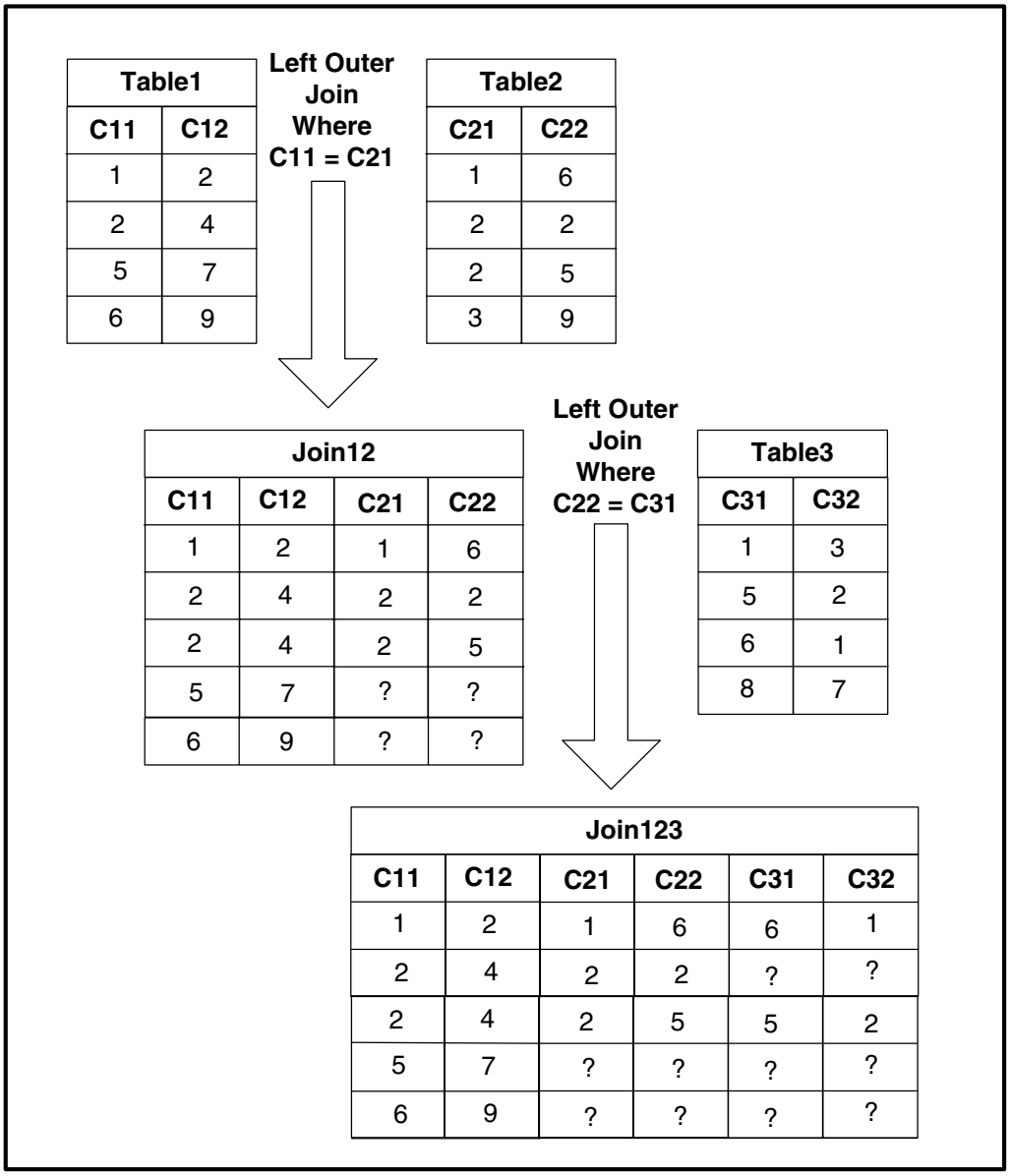


Figure 9–4: Left Outer Joins

Why Use Joins Instead of Nested Reads?

Using joins provides an opportunity for Progress to optimize the retrieval of records from multiple related tables using the selection criteria you specify. When you perform a nested read, for example using nested FOR EACH statements for different tables, you are actually implementing a join in a 4GL procedure. However, by specifying one or more contiguous joins in a single FOR EACH statement or in the PRESELECT phrase of single DO or REPEAT statement, you minimize the complexity of your 4GL code and leave the complexity of joining tables to the Progress interpreter.

For a single 4GL query (OPEN QUERY statement), there is no other way to retrieve data from multiple tables except by using joins. With both inner and left outer join capability, you can use the OPEN QUERY statement to implement most queries that are possible using nested FOR EACH, DO, or REPEAT statements. As such, query joins provide the greatest opportunity for optimized multi-table record retrieval in the 4GL. Also, because browse widgets read their data from queries, you must use query joins to display multiple related tables in a browse. (For more information on browsing records, see Chapter 10, “Using the Browse Widget.”)

However, use nested FOR EACH, DO, and REPEAT blocks wherever you require much finer control over how you access and manipulate records from multiple tables.

9.10.2 Using Inner Joins

The inner join is the default join type in a multi-table read or query. Use this type of join where you are only concerned with the data on the left side of the join for which there is related data on the right. For example, you might want to see only those customers whose purchase of a single item comes close to their credit limit.

The query in `p-join1.p` performs three inner joins on the sports database to return this data. These three items correspond to the `/*1*/`, `/*2*/`, and `/*3*/` that label the joins in the `p-join1.p` example:

1. To each Customer, join all related Order records.

2. To each Order in the previous join, join each related Order–Line record whose total purchase is greater than two-thirds the customer’s credit limit.

3. To each Order–Line in the previous join, join the related Item record to get the item name.

p-join1.p

```
DEFINE QUERY q1 FOR Customer, Order, Order-Line, Item.
DEFINE BROWSE b1 QUERY q1
    DISPLAY Customer.Name Customer.Credit-Limit Order.Order-num
            Item.Item-Name
WITH 10 DOWN.

OPEN QUERY q1 PRESELECT EACH Customer,
    EACH Order OF Customer,
    EACH Order-Line OF Order
        WHERE (Order-Line.Price * Order-Line.Qty) >
              (.667 * Customer.Credit-Limit),
    EACH Item OF Order-Line.

ENABLE b1 WITH SIZE 68 BY 10.
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

When executed, you get the output shown in Figure 9–5. Thus, the relation of Order to Customer and the selection criteria on Order–Line reduces the total number of query rows from thousands of possible rows to four.

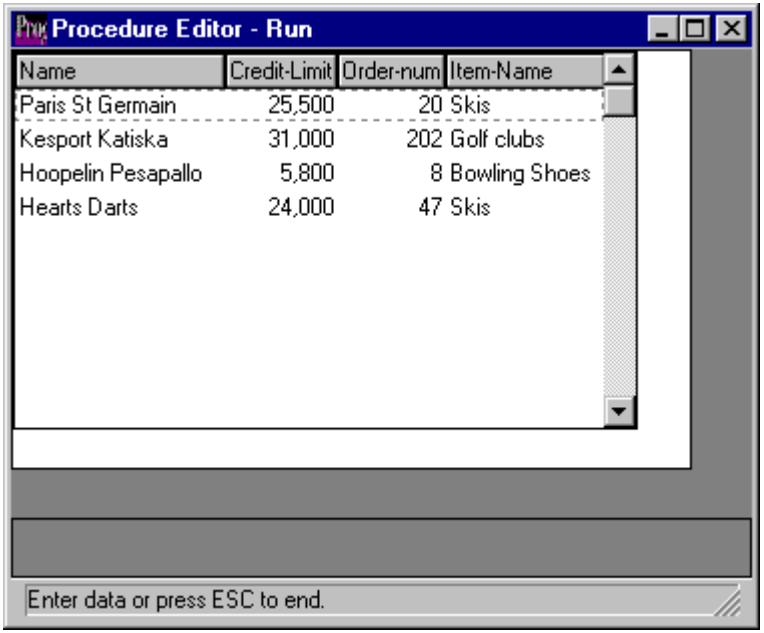


Figure 9–5: Inner Join Example

9.10.3 Using Left Outer Joins

A left outer join is useful where you want to see all the data on the left side, whether or not there is related data on the right. For example, you might want to see the proportion of customers who are ordering close to their credit limit as against those who are not.

The query in `p-join2.p` is identical to the one in `p-join1.p` (see the "Using Inner Joins" section) except that all the joins are left outer joins instead of inner joins. Thus, you see all customers, whether or not they order close to their credit limit. These three items correspond to the `/*1*/`,`/*2*/`, and `/*3*/` that label the joins in the `p-join2.p` example:

1. To each Customer, join all related Order records, or join a null Order record if the customer has no orders.

2. To each Order in the previous join, join each related Order–Line record whose total purchase is greater than two–thirds the customer's credit limit, or join a null Order–Line record if all item purchases are less than or equal to two–thirds the customer's credit limit. Also, join a null Order–Line record if the order is null (the customer has no orders).

3. To each Order–Line in the previous join, join the related Item record to get the item name, or join a null Item record if the the Order–Line is null (no Order or selected purchase for that customer).

p-join2.p

```
DEFINE QUERY q1 FOR Customer, Order, Order-Line, Item.
DEFINE BROWSE b1 QUERY q1
    DISPLAY Customer.Name Customer.Credit-Limit Order.Order-num
            Item.Item-Name
WITH 10 DOWN.

OPEN QUERY q1 PRESELECT EACH Customer,
    EACH Order OUTER-JOIN OF Customer,
    EACH Order-Line OUTER-JOIN OF Order
        WHERE (Order-Line.Price * Order-Line.Qty) >
              (.667 * Customer.Credit-Limit),
    EACH Item OUTER-JOIN OF Order-Line.

ENABLE b1 WITH SIZE 68 BY 10.
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

When executed, you get the output shown in Figure 9–6. In this example, you see the same golf club order as in Figure 9–5 along with many other orders that do not meet the selection criteria and some customers who have no orders at all.

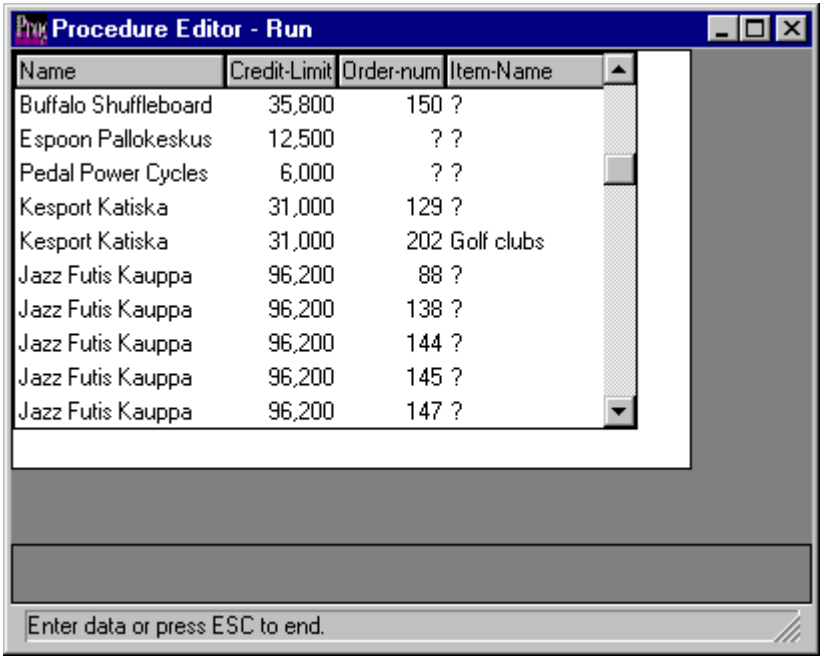


Figure 9–6: Left Outer Join Example

9.10.4 Implementing Other Outer Joins

In addition to left outer joins, there are right and full outer joins. A *right outer join* reverses the join order for the same tables joined with a left outer join. In the 4GL, you can implement a right outer join by doing a left outer join with the tables in reverse order, but leaving the order of displayed fields the same as for the left outer join. Thus, unknown values from the right side appear on the left side of each displayed row, as if the tables were joined from right to left.

A *full outer join* combines the results of a left and right outer join into a single join. This is rarely used, but you can implement a full outer join by building one temporary table from the results of both a left and right outer join. For more information on temporary tables, see Chapter 15, "Work Tables and Temporary Tables."

9.10.5 Mixing Inner and Left Outer Joins

You might want to mix inner and left outer joins in order to filter and reduce the amount of data you need on the left side of your left outer joins. When mixing these two types of join, keep in mind that the last inner join in a query forces the results of all prior joins in the query to be inner joins. This is because any rows that contain unknown values from a prior left outer join are eliminated by the following inner join. The effect is that the work of the prior left outer joins is wasted, and the same result is achieved as with contiguous inner joins, but much less efficiently. Therefore, in any query, keep your inner joins contiguous on the left with any left outer joins contiguous on the right.

9.11 The CONTAINS Operator, Word Indexes, and Word-break Tables

You can use the CONTAINS operator of the WHERE option of the record phrase to query a Progress database. For example, the following query displays each item record of the sports database whose cat–description field contains the string “hockey”:

```
FOR EACH item WHERE cat-description CONTAINS "hockey":
  DISPLAY item.
END.
```

Using the CONTAINS operator involves word indexes and word-break tables. This section explains why and tells you how to use word indexes and word-break tables. Specifically, the section covers:

- Understanding the CONTAINS operator, word indexes, and word-break tables
- Using word indexes
- Using word-break tables
- Using the CONTAINS operator
- Word indexing external documents
- Word indexing non-Progress databases

NOTE: Word indexes and word-break tables have international implications. To understand these implications, first read this section, then see the *Progress Internationalization Guide*.

9.11.1 Understanding the CONTAINS Operator, Word Indexes, and Word-break Tables

To process queries containing the CONTAINS operator, Progress uses word indexes. Similarly, to create, maintain, and use word indexes, Progress uses word-break tables. Let us examine this relationship chain one link at a time.

The CONTAINS Operator and Word Indexes

For Progress to process a query that uses the CONTAINS operator, the field mentioned in the WHERE option must participate in a word index. For example, to process the following query:

```
FOR EACH item WHERE cat-description CONTAINS "hockey":
  DISPLAY item.
END.
```

Progress looks for the word indexes associated with the item record and its cat–description field. If no such word indexes are found, Progress reports a run-time error. Otherwise, Progress uses the word indexes to retrieve the item records whose cat–description field contains the string "hockey."

Word Indexes and Word-break Tables

In order for Progress to use word indexes, it must first build and maintain them, which it does as you add, delete, and modify records that have fields that participate in them. Consider the sports database's item table, whose cat–description field participates in a word index. Every time an item record is added, Progress examines the contents of the cat–description field, breaks it down into individual words, and, for each individual word, creates or modifies a word index. To break down the contents of a field into individual words, Progress must know which characters act as word delimiters. To get this information, Progress consults the database's word-break table, which lists characters and describes the word-delimiting properties of each.

9.11.2 Creating and Viewing Word Indexes

To define a new word index or to view the word indexes already defined on a database table, use the Progress Data Dictionary utility. For example, you can use this utility to view the word indexes of the sports database, including the word index associated with the item table and its cat–description field. For more information on creating and viewing word indexes using the Progress Data Dictionary utility, see the *Progress Basic Development Tools* manual or the Progress Data Dictionary utility online help.

Word Indexing Very Large Fields

NOTE: If a word index is defined on a CHARACTER field that is extremely large, you might have to increase the value of the Stash Area (–stsh) startup parameter. Doing so increases the amount of space Progress uses to temporarily storage of modified indexes. For more information, see the reference entry for the Stash Area (–stsh) startup parameter in the *Progress Startup Command and Parameter Reference*.

Word Indexing Double-byte and Triple-byte Code Pages

You can use word indexing with double-byte and triple-byte code pages. For more information, see the *Progress Internationalization Guide*.

9.11.3 Using Word-break Tables

You can create word-break tables that specify word separators using a rich set of criteria. To specify and work with word-break tables involves:

- Specifying word delimiter attributes
- Understanding the syntax of word-break tables
- Compiling word-break tables
- Associating compiled word-break tables with databases
- Rebuilding word indexes
- Providing access to compiled word-break tables

Specifying Word Delimiter Attributes

As mentioned previously, to break down the contents of a word-indexed field into individual words, Progress needs to know which characters delimit words and which do not. The distinction can be subtle and sometimes depends on context. For example, consider the function of the dot in the character strings in Table 9–5.

Table 9–5: Is the Dot a Word Delimiter?

Character String	Function of the Dot	Is the Dot a Word Delimiter?
“Balance is $25,125.95”	Decimal separator	No
“Shipment not received.Call customs broker”	Period at end of sentence.	Yes

In the first character string, the dot functions as a decimal point and does not divide one word from another. Thus, you can query on the word “$25,125.95.” In the second character string, by contrast, the dot functions as a period, dividing the word “received” from the word “call.”

To help define word delimiters systematically while allowing for contextual variation, Progress provides eight word delimiter attributes, which you can use in word-break tables. The eight word delimiter attributes appear in Table 9–6.

Table 9–6: Word Delimiter Attributes *(1 of 2)*

Word Delimiter Attribute	Description	Default
LETTER	Always part of a word.	Assigned to all characters that the current attribute table defines as letters. In English, these are the uppercase characters A–Z and the lowercase characters a–z.
DIGIT	Always part of a word.	Assigned to the characters 0–9.

Table 9–6: Word Delimiter Attributes *(2 of 2)*

Word Delimiter Attribute	Description	Default
USE_IT	Always part of a word.	Assigned to the following characters: • Dollar sign ($) • Percent sign (%) • Number sign (#) • At symbol (@) • Underline (_)
BEFORE_LETTER	Part of a word only if followed by a character with the LETTER attribute. Else, treated as a word delimiter.	–
BEFORE_DIGIT	Treated as part of a word only if followed by a character with the DIGIT attribute.	Assigned to the following characters: • Period (.) • Comma (,) • Hyphen (–) For example, “12.34” is one word, but “ab.cd” is two words.
BEFORE_LET_DIG	Treated as part of a word only if followed by a character with the LETTER or DIGIT attribute.	-
IGNORE	Ignored.	Assigned to the apostrophe (’). For example, “John’s” is equivalent to “Johns.”
TERMINATOR	Word delimiter.	Assigned to all other characters.

Understanding the Syntax of Word-break Tables

Word delimiter attributes form the heart of word break tables, and you specify them using the following syntax:

SYNTAX

```
[ #define symbolic-name symbol-value ] ...

[ Version = 9
   Codepage = codepage-name
   wordrules-name = wordrules-name
   type = table-type
]

word_attr =
{
  { char-literal | hex-value | decimal-value } , word-delimiter-attribute
      [ , { char-literal | hex-value | decimal-value }
          , word-delimiter-attribute ] ...
};
```

symbolic-name

The name of a symbol.

For example: DOLLAR-SIGN

symbol-value

The value of the symbol.

For example: '$'

NOTE: Although some versions of Progress let you compile word-break tables that omit all items within the second pair of square brackets, Progress Software Corporation (PSC) recommends that you always include these items. If the source-code version of a compiled word-break table lacks these items, and the associated database is not so large as to make this unfeasible, PSC recommends that you add these items to the table, recompile the table, reassociate the table with the database, and rebuild the indexes.

codepage-name

The name, not surrounded by quotes, of the code page the word-break table is associated with. The maximum length is 20 characters.

For example: UTF–8

wordrules-name

The name, not surrounded by quotes, of the compiled word-break table. The maximum length is 20 characters.

For example: utf8sample

table-type

The number 2.

NOTE: Some versions of Progress allow a table type of 1. Although this is still supported, Progress Software Corporation (PSC) recommends, if feasible, that you change the table type to 2, recompile the word-break table, reassociate it with the database, and rebuild the indexes.

char-literal

A character within single quotes or a *symbolic–name,* which represents a character in the code page.

For example: '#'

hex-literal

A hexadecimal value or a *symbolic–name,* which represents a character in the code page.

For example:0xAC

decimal-literal

A decimal value or a *symbolic–name*, which represents a character in the code page.

For example: 39

word-delimiter-attribute

In what context the character is a word delimiter. You can use one of the following:

- LETTER
- DIGIT
- USE_IT
- BEFORE_LETTER
- BEFORE_DIGIT
- BEFORE_LET_DIG
- IGNORE
- TERMINATOR

Examples of Word-break Tables

The following is an example of a word-break table for Unicode:

```
/* a word-break table for Unicode */

#define DOLLAR-SIGN '$'

Version = 9
Codepage = utf-8
wordrules-name = utf8sample
type = 2

word_attr =
{
  '.',          BEFORE_DIGIT,
  ',',          BEFORE-DIGIT,
  0x2D,         BEFORE_DIGIT,
  39,           IGNORE,
  DOLLAR-SIGN, USE-IT,
  '%',          USE-IT,
  '#',          USE_IT,
  '@',          USE-IT,
  '_',          USE_IT,
};
```

As the preceding example illustrates, word-break tables can contain comments delimited as follows:

```
/* this is a comment */
```

For more examples, see the word-break tables that Progress provides in source-code form. They reside in the DLC/prolang/convmap directory and have the file extension .wbt.

NOTE: Progress supplies a word-break table for each code page it supports.

Compiling Word-break Tables

After you create or modify a word-break table, you must compile it with the PROUTIL utility. The syntax is as follows:

Operating System	Syntax
UNIX Windows	proutil -C wbreak-compiler *src-file rule-num*

src-file

The name of the word-break table file to be compiled.

rule-num

A number between 1 and 255 inclusive that identifies this word-break table within your Progress installation.

The PROUTIL utility names the compiled version of the word-break table proword.*rule–num*. For example, if *rule–num* is 34, PROUTIL names the compiled version proword.34.

Associating Compiled Word-break Tables with Databases

After you compile a word-break table, you must associate the compiled version with a database using the PROUTIL utility. The syntax is as follows:

Operating System	Syntax
UNIX Windows	`proutil` *`database`* `-C word-rules` *`rule-num`*

database

The name of the database.

rule-num

The value of *rule–num* you specified when you compiled the word-break table.

To associate the database with the default word-break rules, set *rule–num* to zero.

NOTE: Setting *rule–num* to zero associates the database with the default word-break rules for the current code page. For more information on code pages, see the *Progress Internationalization Guide*.

Rebuilding Word Indexes

For word indexing to work as expected, the word-break table Progress uses to write the word indexes (to add, modify, or delete a record that contains a word index) and the word-break table Progress uses to read word indexes (to process a query that contains the CONTAINS operator) must be identical. To ensure this, when you associate the compiled version of a word-break table with a database, Progress writes cyclical redundancy check (CRC) values from the compiled word-break table into the database. When you connect to the database, Progress compares the CRC values in the database to the CRC value in the compiled version of the word-break table. If they do not match, Progress displays an error message and terminates the connection attempt.

If a connection attempt fails and you want to avoid rebuilding the indexes, you can try associating the database with the default word-break rules.

NOTE: This might invalidate the word indexes and require you to rebuild them anyway.

To rebuild the indexes, you can use the PROUTIL utility with the IDXBUILD or IDXFIX qualifier.

The syntax of PROUTIL with the IDXBUILD qualifier is:

Operating System	Syntax
UNIX Windows	`proutil` *db-name* `-C idxbuild [ all ] [ -T` *dir-name* `] [ -TB` *blocksize* `] [ -TM` *n* `] [ -B` *n* `]`

The syntax of PROUTIL with the IDXFIX qualifier is:

Operating System	Syntax
UNIX Windows	`proutil` *db-name* `-C idxfix`

For more information on the PROUTIL utility, see the *Progress Database Administration Guide and Reference*.

Providing Access to the Compiled Word-break Table

To allow database servers and shared-memory clients to access the compiled version of the word-break table, it must reside either in the Progress installation directory or in the location pointed to by the environment variable `PROWD`*rule–num*. For example, if the compiled word-break table has the name `proword.34` and resides in the `DLC/mydir/mysubdir` directory, set the environment variable `PROWD34` to `DLC/mydir/mysubdir/proword.34`.

NOTE: Although the name of the compiled version of the word-break table has a dot, the name of the corresponding environment variable does not.

9.11.4 Writing Queries Using the CONTAINS Operator

Once you associate a compiled word-break table with a database that has word indexes, if necessary, populate the database and rebuild the word indexes. You can then write queries that use the CONTAINS operator.

Syntax of the CONTAINS Operator of the WHERE Option

The CONTAINS operator has the following syntax in the WHERE option of the record phrase:

SYNTAX

```
WHERE field CONTAINS string-expression
```

field

A field or array of type CHARACTER that participates in a word index.

string-expression

An expression of type CHARACTER that represents possible contents of *field*.

The syntax of *string–expression* is as follows:

SYNTAX

```
"word [ [ & | | | ! | ^ ] word ] ..."
```

word

The word to search for.

The ampersand (&) represents logical AND, while the vertical bar (|), the exclamation point (!), and the caret (^) represent logical OR. AND limits your search to records that contain all words you specify, while OR enlarges your search to include any word you specify. You can combine ANDs and ORs within *string–expression* . You can also group items with parentheses to create complex search conditions.

You can use a wild card on the end of a string. For example, the string “sales*” represents “sales,” “saleswoman,” “salesman,” “salesperson,” and similar strings.

You can also define a character variable and assign a value to that variable.

Examples of the CONTAINS Operator

Now that you know the syntax of the CONTAINS operator, you can write queries that use it.

The following query, which displays all Item records whose Cat–description field contains the word "hockey," demonstrates the CONTAINS operator in its simplest form:

```
FOR EACH item
  WHERE cat-description CONTAINS "hockey":
  DISPLAY item.
END.
```

The following is the equivalent query in SQL-89, which also allows CONTAINS:

```
SELECT * FROM item
  WHERE cat-description CONTAINS "hockey".
```

A CONTAINS string can contain multiple words connected by the AND operator (AND or &) and the OR operator (OR, |, or ^), optionally grouped by parentheses. For example:

```
...CONTAINS "free | gratis | (no & charge)"...
```

NOTE: The AND operator takes precedence over the OR operator. To override this default, use parentheses. Using parentheses can also make the text of a query clearer.

A CONTAINS string containing multiple contiguous words, such as:

```
...CONTAINS "credit hold"...
```

is equivalent to a CONTAINS string containing multiple words connected by AND, such as:

```
...CONTAINS "credit AND hold"...
```

If a CONTAINS string contains multiple words, the order of the words is not significant. To retrieve records in a specific order, use the CONTAINS operator with the MATCHES operator. The following WHERE clause retrieves records whose comments field contains the words "credit" and "hold" in that order, perhaps with other words in between:

```
...WHERE comments CONTAINS "credit hold"
    AND comments MATCHES "*credit*hold*"...
```

Word indexes are case insensitive unless a field participating in the word index is case sensitive. The following two WHERE clauses are equivalent:

```
...WHERE comments CONTAINS "CREDIT HOLD"...
```

```
...WHERE comments CONTAINS "credit hold"...
```

You can combine CONTAINS with other search criteria, as in the following WHERE clause, which searches for records whose city field is Boston and whose comments field contains the word “credit” and either the word “hold” or “watch”:

```
...WHERE city = "Boston"
    AND comments CONTAINS "credit (hold ^ watch)"...
```

The following example demonstrates the use of a variable with the CONTAINS operator within the WHERE clause:

```
DEFINE VARIABLE search_wrd AS CHARACTER.

ASSIGN search_wrd = "The".

FOR EACH customer WHERE name CONTAINS search_wrd:
    DISPLAY cust-num name.
END.
```

9.11.5 Word Indexing External Documents

To create a word index on an existing document, import the text into a Progress database, then index the text by line or by paragraph.

Indexing by Line

To index a set of documents by line, you might create a table called line with three fields: document_name, line_number, and line_text. Define the primary index on document_name and a word index on line_text. Next, write a text-loading Progress program that reads each document and creates a line record for each line in the document. To decrease the amount of storage required by the line table and to normalize the database, you might replace its document_name field with a document_number field, and create a document table to associate a document_name with each document_number.

When base documents change, you must update the line index. You can store a document ID as part of the record for each line. When a document changes, you can delete all lines with that document ID and reload the document.

The following program queries the line table using the word index:

```
DEFINE VARIABLE words AS CHAR FORMAT "x(60)"
  LABEL "To find document lines, enter search words".

REPEAT:
  UPDATE words.
  FOR EACH line WHERE line_text CONTAINS words:
    DISPLAY line.
  END.
END.
```

The example prompts for a string of words, then displays each line that matches the search criteria.

Indexing by Paragraph

Instead of indexing by line, you can index by paragraph. The technique resembles line indexing, but the text from a paragraph can be much longer. You can use paragraph indexes the same way you use line indexes. You can also index by chapter, by page, and by other units of text. The only difference is how your text-loading program parses the document into character fields. Otherwise, your word search code, as in the line table example, can be identical.

9.11.6 Word Indexing Non-Progress Databases

With the exception of the Progress/400 DataServer, Progress DataServers do not support word indexing. To create a word index on text from a non-Progress database accessed by a DataServer that does not support word indexing, you must write a Progress routine to read records from the non-Progress database and copy the text into a table in a Progress database.

You might define a Progress table with the fields nonprog_primary_key and nonprog_text. Define a word index on nonprog_text, then load text and keys from the non-Progress database into the table. Then, use the word index to find the primary keys for records that contain specific words.

For more information on the Progress/400 DataServer wording indexing, see the *Progress/400 Product Guide*.

9.12 Sequences

Sequences are database objects that provide incremental values to an application. They can generate sequential values within any range of a Progress integer (–2,147,483,648 to 2,147,483,647) and with your choice of increment (positive or negative).

You can also define sequences to cycle or terminate when a specified limit is reached. A *cycling sequence* restarts after it reaches the specified limit, providing non-unique values for the life of the sequence. A *terminating sequence* stops incrementing after it reaches the specified limit, providing unique values for the life of the sequence as long as you do not explicitly reset it.

Using a sequence, for example, you can generate a unique series of key values automatically. Progress allows you to guarantee that the key value is unique (for terminating sequences) and created in a specified order. You can also use a sequence to generate an audit trail that creates sequential audit records in the same order they are written to the database.

You can also perform the same kinds of operations by using a *control table* instead of sequences. A control table is a database table that contains a single record with one or more fields. Each field holds a piece of control information such as the name of the application, the company that owns the application, and the last-used value for each unique integer ID in the database. For example, the control table might have fields that hold the last customer number used and the last order number used. Each time a new customer record is created, the record is read from the control table and the last customer number is incremented. This number is then used in the new customer record.

The following sections describe how to:

- Choose between sequences and control tables for an application.
- Define sequences in a database.
- Access and increment sequences from the 4GL.

9.12.1 Choosing Between Sequences and Control Tables

Sequences are objects, like fields, that you can create and maintain in any Progress database. Unlike fields, sequences reside in a separate database block independent of application tables. Figure 9–7 shows the basic relationship between database fields and sequences.

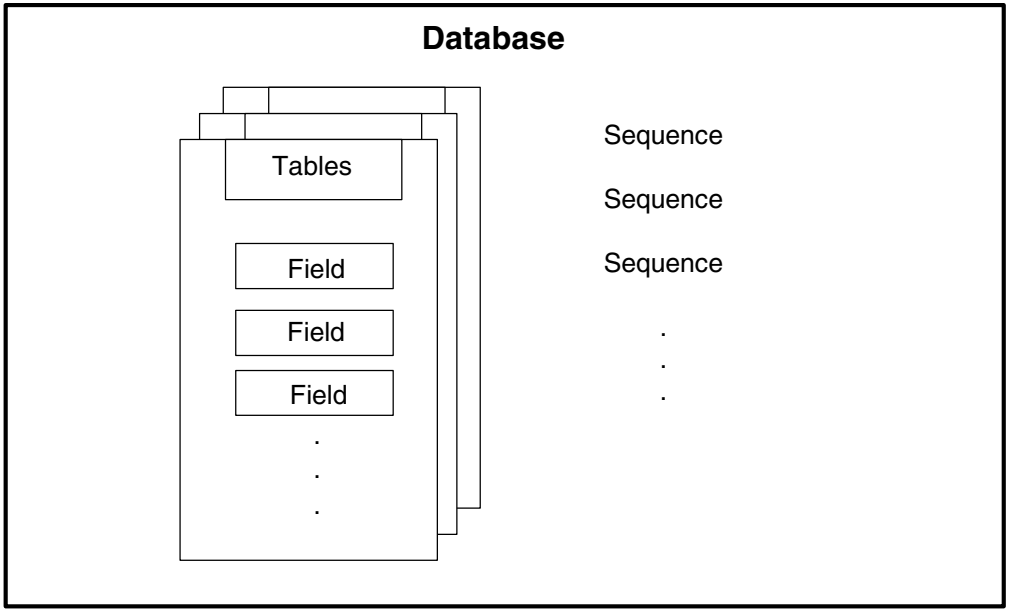


Figure 9–7: Database Fields and Sequences Comparison

Each sequence has a name and a value, similar to a field. But unlike a field, a sequence also has different schema attributes that determine how the sequence value can change. These include:

- Whether the sequence increments or decrements and by what interval
- Whether the sequence cycles or terminates
- The upper and lower boundaries for sequence values

Progress provides a dedicated set of 4GL functions and statements to access and increment sequences strictly according to the defined attributes.

Sequences vs. Control Tables

Before Progress Version 7, the only way Progress could generate sequential values was to maintain integer fields in a table record created specifically for this purpose. This is often done in a control table, separate from all other application tables.

For example, you might have a control table named syscontrol that contains the field last–cus–num. This field holds the value of the Cust–num field for the last Customer record. The following code fragment generates a new Customer record with a unique customer number:

```
. . .
DEFINE VARIABLE next-cust-num   LIKE Customer.Cust-num.

DO FOR syscontrol:
   DO TRANSACTION:
      FIND FIRST syscontrol EXCLUSIVE-LOCK.
      next-cust-num = syscontrol.last-cus-num + 1.
      syscontrol.last-cus-num = next-cust-num.
   END. /* transaction */
   RELEASE syscontrol.
END.

DO TRANSACTION:
   CREATE Customer.
   Customer.Cust-num = next-cust-num.
   DISPLAY Customer.
   UPDATE Customer EXCEPT Cust-num.
END. /* transaction */
. . .
```

Note that access to the syscontrol table must be made within a small transaction to avoid lock contention problems.

Sequences provide a built-in means for generating incremental values, but they are not suitable for all applications. In certain situations (described in the following sections), you might need to use a control table, instead.

Table 9–7 compares sequences and control tables.

Table 9–7: Comparison of Sequences and Control Tables

Capability	Sequences	Control Tables
Access speed	Fast	Slow
Transaction independent	Yes	No
Guaranteed order	Yes	No
Auto initializing	Yes	No
Auto cycling	Yes	No
Bounds checking	Yes	No
Database limit	100	Field limit

Performance vs. Capabilities

In general, sequences provide a much faster and more automated mechanism to generate sequential values than control tables. Sequences are faster because their values are all stored together in a single, table-independent, database block, and they do not participate in transactions. As a result, the principal limitations of sequences for certain applications include:

- Because of transaction independence, sequences do not roll back when transactions are undone, but maintain their last value for all UNDOs. However, fields defined in control tables **do** roll back to their pretransaction values, as long as all updates to control tables respect transaction boundaries.
- You cannot define more than 100 sequence objects at a time in a database; whereas the number of fields in a control table are limited only by the table size.

Transaction Independence

Transaction independence guarantees that each subsequent sequence value is incremented (positively or negatively) beyond its previous value, but does not guarantee the extent of the increment. In other words, with sequences you can have incremental gaps in the sequential values actually used in your application. These gaps result when sequences are incremented during a transaction that is subsequently undone, leaving the sequences set to their latest values. If the transaction restarts, it uses these latest values, not those generated for the previously undone transaction. Figure 9–8 shows how two overlapping transactions with rollback can create records and commit consecutive sequence values with gaps.

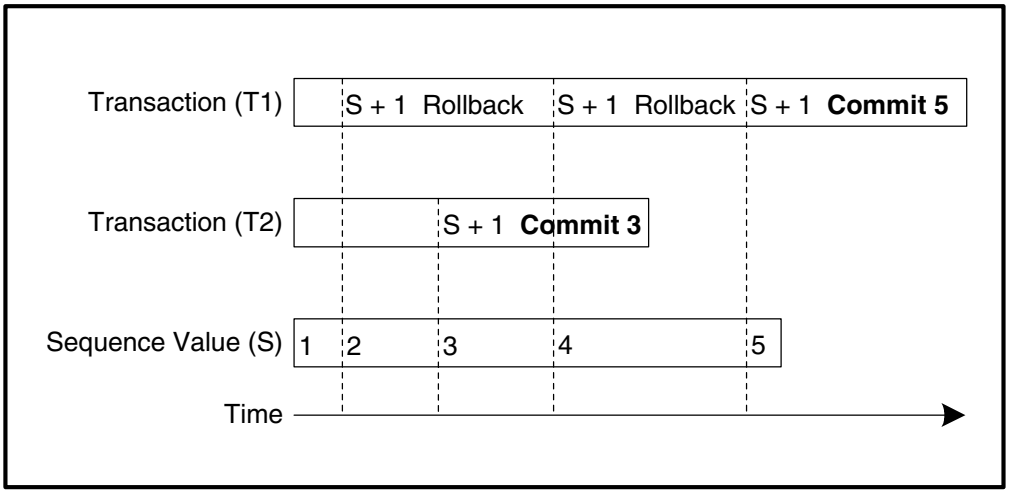


Figure 9–8: Sequences and Overlapping Transactions

Both transactions start out with the sequence set to 1, which has already been used in the database. Transaction T1 increments the sequence first, assigns, and then rolls back leaving the sequence value at 2. Then transaction T2 increments, assigns, and commits the sequence value 3 to a database field. Transaction T1 then increments, assigns, and rolls back a second time, leaving the sequence value at 4. Finally, transaction T1 increments, assigns, and commits the sequence value 5 to the same database field in a different record. Thus, the sequence values 2 and 4 are skipped and never used in the database. All subsequent transactions increment and commit sequence values greater than 5.

NOTE: Transaction independence is provided for standard sequence increment/decrement operations, but not for operations that set the sequence value directly. Setting the value directly is a special operation intended only for maintenance purposes. For more information, see the "Using the CURRENT–VALUE Statement" section.

Control tables, however, obey the same transaction rules as any other table, and can ensure that their field values are reset for undone transactions. If it is essential to generate sequence values without incremental gaps, you must use a control table rather than a sequence.

Storage Limits

If you need to maintain more than 100 sequence objects at a time, you must either add another database to define more sequences or use a control table to generate incremental values in your application. The number of incremental fields you can maintain in a control table is limited only by the number of integer fields you can store in the table.

9.12.2 Creating and Maintaining Sequences

You can create and maintain sequence definitions in a database using Progress's Data Dictionary. Progress stores all sequence values together in a single database block, and stores sequence names and their remaining attributes in a metaschema table named _Sequence.

For information on defining, editing, or deleting sequences in your database, see the *Progress Basic Development Tools* (Character only) manual and, in graphical interfaces, the on-line help for the Progress Data Dictionary.

For information on using sequences in the 4GL, see the "Accessing and Incrementing Sequences" section in this chapter.

Providing Sequence Security

You can restrict sequence access by assigning user privileges to the _Sequence metaschema table. The Can–Read privilege grants users permission to read the current value of a sequence with the CURRENT–VALUE function or to increment and read the next value of the sequence with the NEXT–VALUE function. The Can–Write privilege grants users permission to change the current value of a sequence with the CURRENT–VALUE statement. Any user privileges assigned to the _Sequence metaschema table apply to all sequences defined in the database.

For more information on providing security to a database table, see the *Progress Database Administration Guide and Reference*.

Dumping and Loading Sequences

You can dump and load sequence definitions and values using the Database Administration tool. Select the Database Administration item from the Tools menu of the ADE Desktop. For more information on dumping and loading sequences, see the *Progress Database Administration Guide and Reference*.

9.12.3 Accessing and Incrementing Sequences

Figure 9–8 lists Progress statements and functions you can use to access and increment sequence values from the 4GL.

Table 9–8: Sequence Statements and Functions

Statement or Function	Description
CURRENT–VALUE function	Returns the current value of a sequence.
NEXT–VALUE function	Increments and returns the incremented value for a sequence.
CURRENT–VALUE statement	Sets a new current value for a sequence.

Whenever a CURRENT–VALUE statement or NEXT–VALUE function changes the value of a sequence, the new value persists in the database where the sequence is defined until it is changed again, or the sequence is deleted from the database.

The sequence statements and functions have the following syntax:

SYNTAX

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

SYNTAX

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

`sequence`

An identifier that specifies the name of a sequence defined in the Data Dictionary. Note that a sequence can have the same name as a database field, but they are distinct entities.

`logical-dbname`

An identifier that specifies the logical name of the database in which the sequence is defined. The database must be connected. You can omit this parameter only if you have a single database connected. Otherwise, you must specify the database in which the sequence is defined.

`expression`

An expression that evaluates to an integer value. The CURRENT–VALUE statement assigns the value of *expression* to the specified sequence. The value of *expression* must be within the range defined for the specified sequence.

In general, use the CURRENT–VALUE and NEXT–VALUE functions for mission-critical applications that depend on access to reliable and orderly sequence values. Use the CURRENT–VALUE statement only for database maintenance and initialization, primarily during the development cycle.

Using the CURRENT–VALUE Function

Use the CURRENT–VALUE function in any integer expression to retrieve the current value of a sequence without incrementing it. The current value of a sequence can be any one of the following:

- The Initial value specified in the Dictionary
- The last value set with either the CURRENT–VALUE statement or the NEXT–VALUE function

The following example gets the the most recent customer number in the default database (maintained by the cust–num sequence), and displays each order record for that customer:

```
DEFINE VARIABLE current-cust AS INTEGER.
               .
               .
               .
current-cust = CURRENT-VALUE (cust-num).

FOR EACH order WHERE cust-num = current-cust:
    DISPLAY order.
               .
               .
               .
```

Using the NEXT–VALUE Function

Use the NEXT–VALUE function to increment a sequence by its defined positive or negative increment value. If the sequence cycles and NEXT–VALUE increments it beyond its Upper or Lower limit, the function sets and returns the defined Initial value for the sequence. If the sequence terminates and NEXT–VALUE tries to increment it beyond its Upper or Lower limit, the function returns the unknown value (?) and leaves the sequence value unchanged.

The following example creates a new customer record with the next available customer number generated by the cust-num sequence:

```
DEFINE VARIABLE input-cust-num AS INTEGER.

input-cust-num = NEXT-VALUE (cust-num, sports).
FIND sports.customer WHERE cust-num = input-cust-num NO-ERROR.

DO WHILE (AVAILABLE customer):
  input-cust-num = NEXT-VALUE (cust-num, sports).
  FIND sports.customer WHERE cust-num = input-cust-num
    NO-ERROR.
END.

CREATE sports.customer.
customer.cust-num = input-cust-num.
UPDATE sports.customer EXCEPT cust-num.
                  .
                  .
                  .
```

Because this example does not check the cust-num sequence for termination, it implies that cust-num is a cycling sequence. Because it does check for and ignore existing records containing the generated cust-num value, the example can reuse previously deleted (or otherwise skipped) customer numbers after the sequence cycles.

Using the CURRENT–VALUE Statement

Use the CURRENT–VALUE statement to explicitly set a sequence to a new value. You can assign the defined initial value of a sequence, its upper or lower limit, or any integer value in between. Trying to set a value outside these bounds causes an error. Note that you cannot assign the unknown value (?) to a sequence. Unlike the NEXT–VALUE function, the CURRENT–VALUE statement always sets a new sequence value in a transaction, starting one, if necessary.

CAUTION: Avoid using this statement in mission-critical applications. Use of this statement, especially in a multi-user context, can compromise the referential integrity of your database. Any database application designed to rely on the orderly values provided by sequences cannot reliably reset existing database fields according to potentially unexpected sequence values.

The purpose of the CURRENT–VALUE statement is to maintain a database off-line, under program control. Note that the Data Dictionary uses this statement when you create a new sequence to set the initial value. Possible uses include:

- Restarting a terminating sequence that has terminated.
- Otherwise, resetting a sequence value for a specific test condition.

Keep in mind that all such uses must respect and might require changes to the _Sequence table. This table contains one record for each sequence defined in the database. Creating a record in this table automatically allocates and assigns the data storage for a new sequence. Deleting a record in this table automatically releases the data storage for the specified sequence.

Progress provides fault detection that prevents a sequence from being created with inconsistent attribute values in the _Sequence table (such as, the minimum value greater than the maximum value). It also prevents the assignment of a new current value that is inconsistent with the the corresponding attribute values in the _Sequence table (such as, a current value outside the minimum and maximum value boundary).

The following code fragment sets a new current value for sequence used for testing.

```
DEFINE VARIABLE SeqValue AS INTEGER.

DO TRANSACTION ON ERROR UNDO, RETRY:
    SET SeqValue LABEL "TEST-Stepper Start" WITH SIDE-LABELS.
    CURRENT-VALUE(TEST-Stepper) = SeqValue.
END. /* TRANSACTION */
                    .
                    .
                    .
```

Progress also uses this statement in the Data Administration tool to load sequence definitions and values from dump-formatted text files. You can accomplish the same task implemented in the code fragment above using this tool. For more information, see the *Progress Database Administration Guide and Reference*.

Compiling Procedures That Reference Sequences

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

Resetting a Terminating Sequence

When a terminating sequence stops at its upper or lower limit, you can reset it by assigning a valid sequence value to it with the CURRENT–VALUE statement. You can also reset a stopped terminating sequence by changing it to a cycling sequence. The first use of the NEXT–VALUE function for the new cycling sequence resets the sequence to its initial value.

Referencing Sequences Within a WHERE Clause

You cannot invoke sequence functions from within a WHERE clause. This generates a compiler error, because sequence expressions do not participate in the index resolution and optimization phase of the Compiler.

If you want to use a sequence value within a WHERE clause, set a variable or field to the sequence value, and reference the variable or field in your WHERE clause. See the sections that describe each sequence function for examples.

9.13 Database Trigger Considerations

You can store a database's trigger procedures in an operating system directory or in a Progress r-code library. There are advantages and disadvantages with each technique of storage. Typically, when deploying an application, it is advantageous to store your trigger procedures in an r-code library. When developing the application, it is usually better to store the procedures in a directory.

The Trigger Location (–trig) startup parameter allows you to specify the directory or r-code library containing your trigger procedures.

9.13.1 Storing Triggers in an Operating System Directory

In a development environment, it is usually a good idea to store trigger procedures in a directory. In a directory, you can store compiled procedures (.r files) and source procedures (.p files). You can modify the source code of the procedures as necessary.

Also, you can run uncompiled procedures against any database that has the appropriate schema. In contrast, if a procedure is precompiled, the r-code stores the logical name of the database you compiled the procedure against. The precompiled procedure can only run against a database with the same logical name.

9.13.2 Storing Triggers in an R-code Library

When deploying your application, it is usually a good idea to place trigger procedures in an r-code library. This is usually faster than running the procedures from an operating system directory, even if the procedures are precompiled. However, procedures run from a library **must** be precompiled. Moreover, because they are precompiled, they are necessarily associated with a logical database name. You can only run them against a database connected with the same logical name.

9.14 Using the RAW Data Type

Progress supports the RAW data type, which lets you manipulate raw data without converting it in any way.

You can use the RAW data type to:

- Define Progress variables in a 4GL procedure
- Define fields in a Progress database
- Retrieve and manipulate raw data from non-Progress databases

You can use the RAW data type to import non–Progress data that has no parallel Progress data type. By using the RAW data type statements and functions, Progress allows you to bring data from any field into your procedure, manipulate it, and write it back to the non-Progress database. The functions and statements let you define RAW data type variables, write data into a raw variable, find the integer value of a byte, change the length of a raw variable, and perform logical operations.

The following procedure demonstrates how to retrieve raw values from the database, how to put bytes into variables, and how to write raw values back to the database:

```
/* You must run this procedure against a non-Progress sports database. */

DEFINE VAR r1 AS RAW.
DEFINE VAR i AS INT.

FIND FIRST cust.
r1 = RAW(name).
PUTBYTE(r1,1) = 115.
RAW(name) = r1.
DISPLAY name.
```

This procedure first creates the variable r1 and defines it as a RAW data type. Next, it finds the first customer in the database (“Lift Line Skiing”), and with the RAW function, takes the raw value of the field name, and writes it into the variable r1. The PUT–BYTE statement then puts the character code value of “s” (115) into the first byte of r1. The RAW statement takes the raw value of r1 and writes it back to the database. Finally, the procedure displays the customer name. Thus, “Lift Line Skiing” has become “sift Line Skiing.”.

The next procedure shows how you can pull bytes from a field:

```
/* You must run this procedure against a non-Progress sports database. */

DEFINE VAR i AS INT.
DEFINE VAR a AS INT.

FIND cust WHERE cust-num = 27.
i = 1.
REPEAT:
  a = GETBYTE(RAW(name),i).
  DISPLAY a.
  IF a = ? THEN LEAVE.
  i = i + 1.
END.
```

This procedure finds the customer with the customer number 27, and then finds the character code value of each letter in the customer name. To do this, it retrieves the bytes from the name one at a time, then places them into the variable a. The GET–BYTE function returns the unknown value (?) if the byte number you try to retrieve is greater than the length of the expression from which you are retrieving it. The next procedure demonstrates how you find the length of a raw value and how to change length of a raw expression:

```
/* You must run this procedure against a non-Progress sports database. */

DEFINE VAR r3 AS RAW.

FIND FIRST cust.
r3 = RAW(name).
DISPLAY LENGTH(r3) name WITH DOWN. /* length before change */
DOWN.

LENGTH(r3) = 2.
DISPLAY LENGTH(r3) name. /* length after change */
```

This procedure simply finds the number of bytes in the name of the first customer in the database then truncates the number of bytes to two. The procedure first displays 16 because the customer name Lift Line Skiing contains 16 bytes. (If you are using a C-ISAM dataserver the procedure displays a larger value because C-ISAM uses fixed-length strings.) It then displays 2 because you truncated the raw value to two bytes.

NOTE: When you use the STRING function to retrieve a raw value as a string, you must supply a format for the string as the second function argument.

9.15 Multi-database Programming Techniques

Once you've settled on a design and run-time connection technique for your multi-database application, you can begin programming. The following sections list a variety of programming techniques and issues to consider as you develop your multi-database application with Progress.

9.15.1 Referencing Tables and Fields

Unique table and field names do not require fully qualified references within application procedures. In a single database Progress application, references to non-unique field names require a table prefix to avoid ambiguous field references. Use the following syntax to reference a non-unique field in a procedure:

SYNTAX

```
table-name.field-name
```

The *table–name* is the name of the database table that contains the field *field–name*.

The ability to connect to several databases from a single Progress session introduces the possibility of non-unique table names and increases the possibility of non-unique field names. References to non-unique table names within multi-database Progress applications require database prefixes to avoid ambiguous table references. Ambiguous table and field references cause compilation failures. Use the following syntax to include a database prefix in a table or field reference:

SYNTAX

```
database-name.table-name
database-name.table-name.field-name
```

The *database–name* is the logical name (or an alias for a logical name) that represents the database that contains the table *table–name*.

For example, suppose you are connected to two databases db1 and db2, both of which contain a table called customer. The following procedure will not compile due to an ambiguous table reference:

```
FOR EACH customer: /* In db1 or db2 ? */
  DISPLAY name.
END.
```

The procedure below uses fully qualified table references to display customer names from both connected databases (db1 and db2):

```
FOR EACH db1.customer:
  DISPLAY name.
END.
FOR EACH db2.customer:
  DISPLAY name.
END.
```

Notice that the two references to the name field do not need to be qualified because they appear within a FOR EACH block that already contain fully qualified table references.

NOTE: When Progress encounters a statement such as “DISPLAY X.Y”, it first attempts to process “X.Y” as a *databasename.table–name*. If that fails, Progress then attempts to process “X.Y” as *table–name.field–name.*

9.15.2 Positioning Database References in an Application

If possible, isolate database references within your application to help minimize the effects of a database connection failure on your application. This allows you to use the CONNECTED function effectively to test for a particular database connection, prior to passing program control to a subprocedure that accesses that database.

p-mainproc.p
db1proc.p
db2proc.p

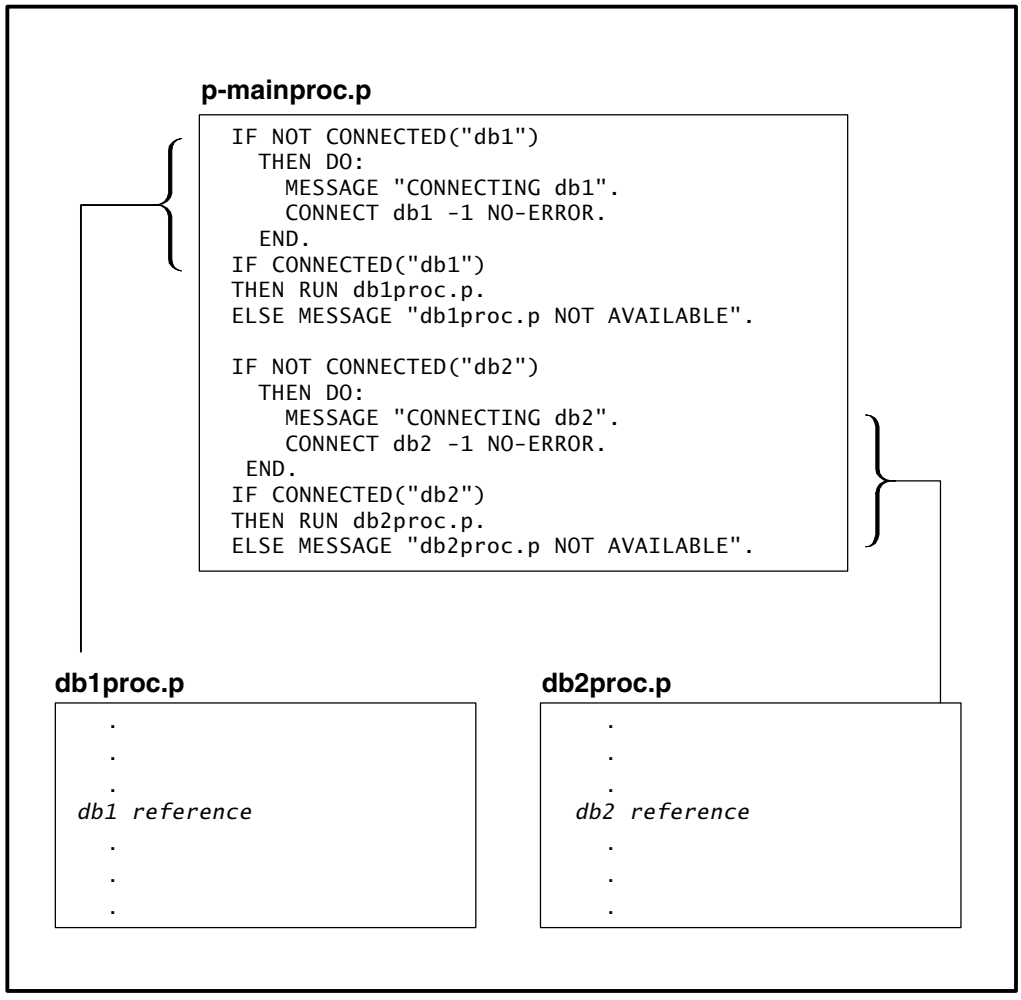


Figure 9–9: Positioning Database References

Figure 9–9 shows a main procedure that connects two databases and runs subprocedures, depending on whether or not a database is connected. If the CONNECT statement in `mainproc.p` fails to connect db2, only `db2proc.p` subprocedure is effected by the database connection failure. This technique is useful for applications that run subprocedures from a menu.

Do not reference a database in your application startup procedure, except possibly to connect it. Remember, you cannot connect to a database and reference a database in the same procedure. If you reference a database on an auto-connect list within the startup procedure of your application and the auto-connect fails for that database, the startup procedure does not run.

9.15.3 Using the LIKE option

The Progress LIKE option in a DEFINE VARIABLE statement, DEFINE WORKFILE statement, DEFINE WORK–TABLE, or Format phrase requires that a database is connected. Use the LIKE option with caution in procedures that do not otherwise access a database.

9.16 Creating Schema Cache Files

When Progress starts a client application, it loads the schema for each database that the application requires into memory. Ordinarily, Progress reads the schema from the database, which can be on a remote server. If the network is slow, or the server, itself, is overloaded, this can increase startup time significantly.

You can shorten client startup time by building and saving a schema cache file on a disk local to the client. A *schema cache file* is a machine-portable binary file containing a complete schema or subschema for a single Progress database. When you start a client application with local schema cache files, Progress reads the required schema for each database almost instantaneously from the local disk rather than waiting for network or server traffic.

9.16.1 Building a Schema Cache File

To build a schema cache file, you use the SAVE CACHE statement together with a connected database. You must first decide whether you need the entire schema cache or only the schema cache for selected tables of a database. If your application accesses all the tables in the database, you need the complete cache. Otherwise, you can build a schema cache file for only the tables that are accessed by your application.

In general, you build a schema cache file off-line, after making a schema change in the database. You can do this in the Procedure Editor directly, or you can write a small maintenance procedure to generate the file.

This is the syntax of the SAVE CACHE statement:

SYNTAX

```
SAVE CACHE { CURRENT | COMPLETE }
  database-name TO pathname
```

The *database–name* can be the literal logical name of any Progress database or the VALUE(*expression*) option, where *expression* is a character expression that evaluates to the database name.

NOTE: For a DataServer, Progress saves the schema cache for the entire schema holder database. You cannot save the schema cache for a non-Progress database separately. For more information on schema cache files for DataServers, see your Progress DataServer guide.

The *pathname* can be the literal pathname of an operating system file or the VALUE(*expression*) option, where *expression* is a character expression that evaluates to the pathname.

Saving the Entire Schema Cache

To save the entire schema cache.

1 ♦ Connect to the database.

2 ♦ Execute the SAVE CACHE statement using the COMPLETE option.

For an example procedure that saves the entire schema cache for one or more databases, see the SAVE CACHE Statement reference entry in the *Progress Language Reference*.

Saving a Partial Schema Cache for Selected Tables

To save a partial schema cache for selected tables of a database.

1 ♦ Connect to the database.

2 ♦ Read each table you want to include in the schema cache with a FIND statement.

3 ♦ Execute the SAVE CACHE statement using the CURRENT option.

4 ♦ If you want to save a different partial schema cache for the same database, disconnect the database and repeat Steps 1 through 3.

9.16.2 Example Schema Cache File Creation

The following event-driven application allows you to create partial schema cache files for any combination of tables in the sports database. The main procedure, `p-schcs1.p`, presents all the available sports database tables in a multiple selection list and a field to enter a name for the schema cache file. When you choose the Save to File button, it connects to the sports database, and calls `p-schcs2.p`, which reads the selected tables and saves the resulting schema cache file in the current working directory. The main procedure then disconnects the sports database to allow new table selections for a different schema cache file.

p-schcs1.p

```
DEFINE VARIABLE filen AS CHARACTER FORMAT "x(8)"
LABEL "Schema Cache Name".
DEFINE VARIABLE icnt AS INTEGER.

DEFINE VARIABLE db-table AS CHARACTER LABEL "Select Tables"
       VIEW-AS SELECTION-LIST
       MULTIPLE NO-DRAG SIZE 32 BY 7
       LIST-ITEMS "customer", "invoice", "item", "local-default",
                  "order", "order-line", "ref-call", "salesrep",
                  "state".

DEFINE BUTTON bsave LABEL "Save to File".
DEFINE BUTTON bcancel LABEL "Cancel".

DEFINE FRAME SchemaFrame
    SPACE(1)
    db-table
    VALIDATE(db-table <> "" AND db-table <> ?, "You must select a table.")
    filen
    VALIDATE(filen <> "" AND filen <> ?, "You must enter filename.")
    SKIP(1)
    SPACE(20) bsave bcancel
WITH TITLE "Save Schema Cache File" SIDE-LABELS SIZE 80 by 11.

ON CHOOSE OF bcancel IN FRAME SchemaFrame QUIT.

ON CHOOSE OF bsave IN FRAME SchemaFrame DO:
    ASSIGN filen db-table.
    IF NOT filen:VALIDATE() THEN RETURN NO-APPLY.
    IF NOT db-table:VALIDATE() THEN RETURN NO-APPLY.
    DO WHILE NOT CONNECTED("sports"):
        BELL.
        PAUSE MESSAGE "When ready to connect the sports database, press
<RETURN>".
        CONNECT sports -1 NO-ERROR.
        IF NOT CONNECTED("sports") THEN
            DO icnt = 1 to ERROR-STATUS:NUM-MESSAGES:
                MESSAGE ERROR-STATUS:GET-MESSAGE(icnt).
            END.
        ELSE
            MESSAGE "Sports database connected.".
    END.
    RUN p-schcs2.p (INPUT db-table, INPUT filen).
    DISCONNECT sports NO-ERROR.
END.

ENABLE ALL WITH FRAME SchemaFrame.
WAIT-FOR CHOOSE OF bcancel IN FRAME SchemaFrame.
```

p-schcs2.p

```
DEFINE INPUT PARAMETER db-table AS CHARACTER.
DEFINE INPUT PARAMETER filen AS CHARACTER.

DEFINE VARIABLE itab AS INTEGER.

Table-Add: DO itab = 1 to NUM-ENTRIES(db-table):
    CASE ENTRY(itab, db-table):
        WHEN "customer"         THEN FIND FIRST customer NO-ERROR.
        WHEN "invoice"          THEN FIND FIRST invoice NO-ERROR.
        WHEN "item"             THEN FIND FIRST item NO-ERROR.
        WHEN "local-default"    THEN FIND FIRST local-default NO-ERROR.
        WHEN "order"            THEN FIND FIRST order NO-ERROR.
        WHEN "order-line"       THEN FIND FIRST order-line NO-ERROR.
        WHEN "ref-call"         THEN FIND FIRST ref-call NO-ERROR.
        WHEN "salesrep"         THEN FIND FIRST salesrep NO-ERROR.
        WHEN "state"            THEN FIND FIRST state NO-ERROR.
        OTHERWISE               LEAVE Table-Add.
    END CASE.
END.

SAVE CACHE CURRENT sports to VALUE(filen + ".csh") NO-ERROR.
IF NOT ERROR-STATUS:ERROR THEN
    MESSAGE "Saved partial schema cache for the sports database"
            "in" filen + ".csh.".
ELSE DO:
    BELL.
    DO itab = 1 TO ERROR-STATUS:NUM-MESSAGES:
        MESSAGE ERROR-STATUS:GET-MESSAGE(itab) VIEW-AS ALERT-BOX.
    END.
END.
```

9.16.3 Using a Schema Cache File

You specify the schema cache file for each database of an application using the Schema Cache File (–cache) startup parameter. If the schema cache file for any database is invalid, Progress displays a message, ignores the file with a warning, and reads the cache from the database.

```
The time stamp in the database does not match the time stamp in the
    cache file: sports.csh (840)
```

The user can notify the database administrator or application developer to provide the correct schema cache file. For more information on using schema cache files, see the *Progress Client Deployment Guide*, and for a description of –cache, see the *Progress Startup Command and Parameter Reference*.

10

Using the Browse Widget

This chapter describes how to design and interact with static browse widgets and the queries they represent. For information on the dynamic browse and the dynamic query, see the "Creating and Using Dynamic Queries" section of Chapter 20, "Using Dynamic Widgets."

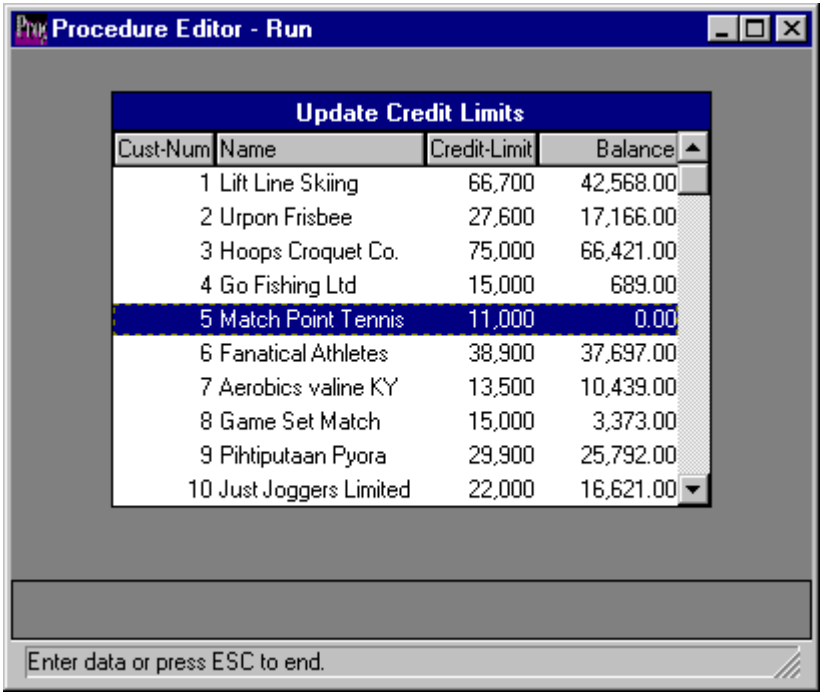
This chapter covers browse design and programming topics divided into six general areas:

- Understanding browse widgets
- Defining queries for browse widgets
- Defining browse widgets
- Programming with browse widgets
- Resizable browse widgets
- Using browse widgets in character interfaces

NOTE: The first five sections describe the essential features of static browse programming and browse appearance and functionality on Windows. Occasional notes call out differences in character interfaces. The last section focuses on the unique features of browse widgets in character interfaces.

10.1 Understanding Browse Widgets

A browse widget is a representation of a query. A query is the list of database records from one or more tables that match the supplied selection criteria. A browse is an object that displays data from the query in rows and columns. A row represents the data of a single record. A column represents all the values of a particular field. A row and column intersection, called a cell, represents the value of the field (column name) in that particular record. A user can scroll up and down the rows, and left and right through the columns, as shown below:



The browse widget comes with a broad range of functionality and design options that can accommodate many different browse uses. For example, the browse can be a read-only index to database records from which users launch dialog boxes and windows that let them work with the selected record. The browse can also be a large spread-sheet-like application with in-line editing, color and font support, calculated fields, and user input validation.

NOTE: Progress does not support fonts in character interfaces.

Whether a single browse widget is the center of your application or just one component, the browse assumes an important role in event-driven database applications. To use browse widgets effectively, you need to understand default behavior and functionality, as well as how to hook browse widgets into the rest of your code. This chapter presents the fundamental design and programming topics related to browse widgets. For a description of all browse functionality, including attributes and methods not covered here, see the DEFINE BROWSE Statement reference entry and the Browse Widget reference entry, both in the *Progress Language Reference*.

10.2 Defining a Query for a Browse

By default, a query uses SHARE–LOCKs and it is not scrollable. *Scrollable* is the ability to move backward as well as forward in the query. The optimum query for a browse should use NO–LOCKs and be scrollable. When a browse is associated with a query (by way of the DEFINE BROWSE statement), Progress automatically changes the query to use the NO–LOCK and SCROLLABLE options.

Locking is an important topic with browse widgets. Since the browse widget is a tool for displaying data from multiple records, you do not want to tie up these records with SHARE–LOCKs. Even when you are working with updatable browse widgets, you will find that NO–LOCK is the best choice for the associated query. Functionality covered later in the chapter describes how to successfully update browse records that start in the NO–LOCK state.

Once you define the query, define the browse, and open the query, the browse and the query become virtually identical. The currently selected row and the results list cursor are in sync and remain so. When the user manipulates the browse, the user is also performing the same manipulation on the cursor of the results list. Many programmatic actions performed on the results list or the browse automatically update the other, although this is not universally true. (You could say that learning all the subtleties of the browse involves learning what occurs by default, what you have to manage, and what behaviors you can override.)

As a rule, a query associated with a browse should be used exclusively by that browse. While you can use GET statements to manipulate the database cursor, the results list, and the associated buffers, you run the risk of putting the browse out of sync with the query. If you do mix browse widgets and GET statements with the same query, you will have to use the REPOSITION statement to manually keep the browse in sync with the query.

10.2.1 Using Field Lists

Using field lists with browse widgets can be an important way to speed up your application and minimize resource overhead. For example, if your application works with three fields from a database table with 15 fields, your application reads five times more data than it needs. Instead of reading the whole record in the query, you can use the field list syntax to list the subset of fields needed by your application. The subset of fields consists of those required by the browse and those required by the other parts of your application that get data from the buffers associated with the query.

The code fragment below shows an example of a field list in an OPEN QUERY statement:

```
OPEN QUERY my-query FOR EACH customer FIELDS (cust-num name credit-limit).
```

For more information on field lists, see Chapter 9, "Database Access" and the Record phrase entry in the *Progress Language Reference*.

10.2.2 Planning for the Size of the Data Set

The number of records in your results list can have a serious impact on the performance of your browse. You may want to optimize your query definition to take advantage of features that work well with small and large data sets.

When a query is opened, it does not necessarily build the entire results list immediately. In Chapter 9, "Database Access," there is a detailed discussion about how results lists are initialized and built. For small data sets, it may be more advantageous to force the query to build the complete results list with a PRESELECT option. This technique has the added benefit of giving the scrollbar access to all the records in the query from the time the browse is initialized. While the results list is being built, the scrollbar represents the range of records currently in the results list.

If you have a query to a large data set, using the REPOSITION TO ROWID statement may significantly affect performance. The INDEXED–REPOSITION option of the OPEN QUERY statement attempts to optimize REPOSITION statements that use ROWID. To accomplish this, the query essentially starts over. It begins building the results list from the new position, on each REPOSITION statement. The results list will only be built as much as necessary to satisfy the REPOSITION statement. This option can dramatically speed up browse repositions. The option has side effects, however. Because of the new results list, your view of the query can change. For example, new records entered into the browse will appear in their correct sorted order when the results list is initialized.

Using the INDEX–REPOSITIONED option on a query associated with a multiple-select browse is legal, but may have too many drawbacks to be useful. When the results list is restarted, all selected rows in the browse are deselected. Also, the vertical scrollbar thumb is disabled.

Since PRESELECT forces a complete results list to be created and maintained, you cannot use the INDEXED–REPOSITION option with a PRESELECT query.

10.3 Defining a Browse Widget

This section describes some of the major functionality and style choices that you have when you define a browse widget, and the issues involved with those choices.

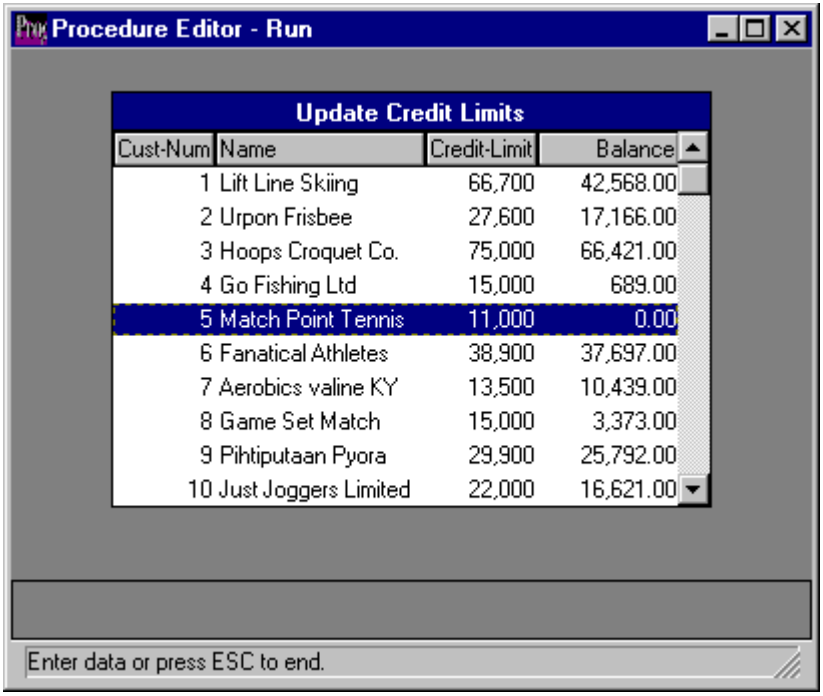
10.3.1 Defining a Read-only Browse

You can browse database records by defining a browse widget for the query and opening the query. This is known as a read-only browse. Once the user finds and selects a record, your application uses the selected record, which the associated query puts in the associated buffer or buffers. The following code example illustrates a simple read-only browse for the customer table:

p-br01.p

```
DEFINE QUERY q1 FOR customer SCROLLING.
DEFINE BROWSE b1 QUERY q1 DISPLAY cust-num name credit-limit balance
    WITH 10 DOWN TITLE "Update Credit Limits".

DEFINE FRAME f1
    b1
        WITH SIDE-LABELS ROW 2 CENTERED NO-BOX.
OPEN QUERY q1 FOR EACH customer NO-LOCK.
ENABLE ALL WITH FRAME f1.
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

When you run this example, the following browse appears:



The DOWN option of the DEFINE BROWSE statement tells Progress how many rows to display. If the TITLE option is not used, then the title bar above the column labels does not appear.

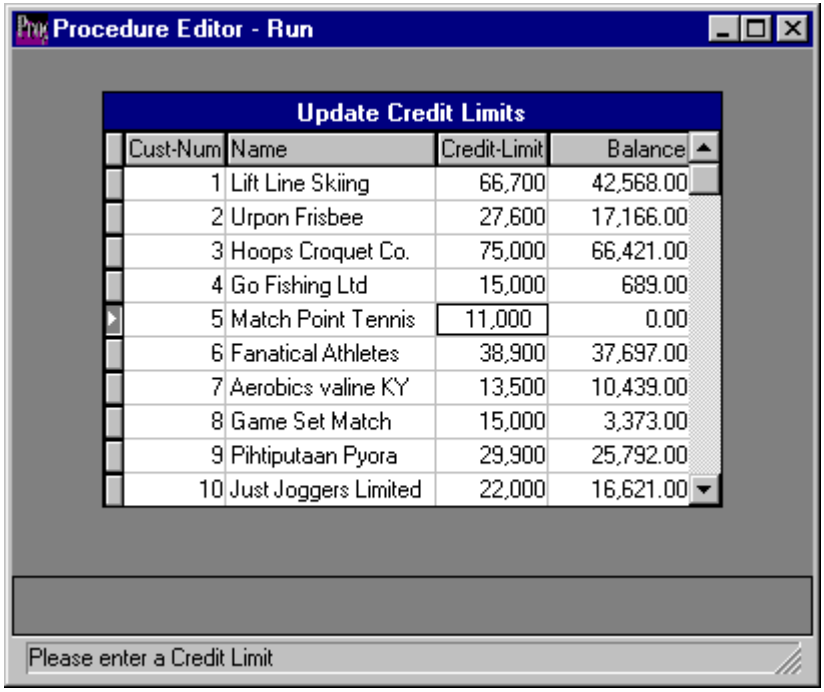
10.3.2 Defining an Updatable Browse

An updatable browse supports in-line editing of data. The code example below shows the syntax for enabling a single field in the browse for input:

p-br02.p

```
DEFINE QUERY q1 FOR customer SCROLLING.
DEFINE BROWSE b1 QUERY q1 DISPLAY cust-num name credit-limit balance
    ENABLE credit-limit WITH 10 DOWN SEPARATORS
    TITLE "Update Credit Limits".

DEFINE FRAME f1
    b1
        WITH SIDE-LABELS ROW 2 CENTERED NO-BOX.
OPEN QUERY q1 FOR EACH customer NO-LOCK.
ENABLE ALL WITH FRAME f1.
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

When you run this example, the following browse appears:



Notice that the updatable browse has row markers on the left. In a read-only browse, clicking anywhere in a row selects the row. In an updatable browse, clicking in a row is interpreted as a request to edit that field, if the field is enabled. Otherwise, the row is selected. The normal way to select rows in an updatable browse is to select row markers. (Note that row markers can also be turned off programmatically with the NO–ROW–MARKERS option of the DEFINE BROWSE statement.)

NOTE: In character interfaces, a key sequence (**ESC–R**) toggles between two modes, to select rows or edit cells. By default, the selected row is the highlighted row that you change with **CURSOR–DOWN** and **CURSOR–UP**. Character row markers appear as an asterisk (*).

An instance of a field in a row (the intersection of a row and column) is called a browse cell. An updatable browse cell behaves in much the same manner as a fill-in field. Tabbing or clicking moves the user from one updatable cell to the next.

NOTE: In character interfaces, the EDITOR–TAB and BACK–TAB key functions (usually **CTRL–G** and **CTRL–U**) tab from cell to cell.

One very important fact to remember is that a read-only browse cannot be enabled, or "turned on," by the programmer after it has been defined. If you expect to need any of the capabilities of the updatable browser, enable the appropriate fields in the definition and then use the READ–ONLY attribute to temporarily turn them off.

The SEPARATORS option provides dividers between columns and rows. Using separators with an updatable browse makes the browse look more like a spreadsheet, which might suggest that it has inline editing capability to the user.

NOTE: The difference in behavior between read-only and updatable multiple-select browse widgets represents a commitment to backward compatibility on one hand and a commitment to supporting native standards on the other.

10.3.3 Defining a Single- or Multiple-select Browse

A browse can support single and multiple selections. By default, a browse is single select. The MULTIPLE option sets up a multiple-select browse. Multiple selection allows you to select any combination of rows from the browse, which you can then access using the appropriate browse attributes and methods.

In a multiple-select read-only browse, a user clicks each row to select it. Selecting a second row does not deselect the previous row. In a multiple-select updatable browse, there are two ways to select multiple records. First, a user can click on one row marker, hold down the mouse button and drag to select multiple contiguous rows. Second, a user can hold down the **CTRL** key and click on any row marker to toggle its selection state. Rows selected in this manner do not have to be contiguous. Clicking a row marker without holding down **CTRL** selects that row and deselects all other selected rows. Note also, that the **CTRL** click feature only operates on row markers, not on disabled columns in updatable browse widgets.

NOTE: For a multiple-select browse in character interfaces, selection of each row is toggled by the space bar. The close angle bracket (>) indicates each selected row. The same technique applies to both read-only and updatable multiple-select browse widgets.

For a multiple-select browse, the concepts of focus and selection separate. Selection indicates that the user has selected a record. There can be many selected records. The last selected record is in the one in the record buffer. Focus indicates that the record has input focus. There can be only one focused row, and it might or might not be a selected row. In a single-select browse, selection and focus are one and the same.

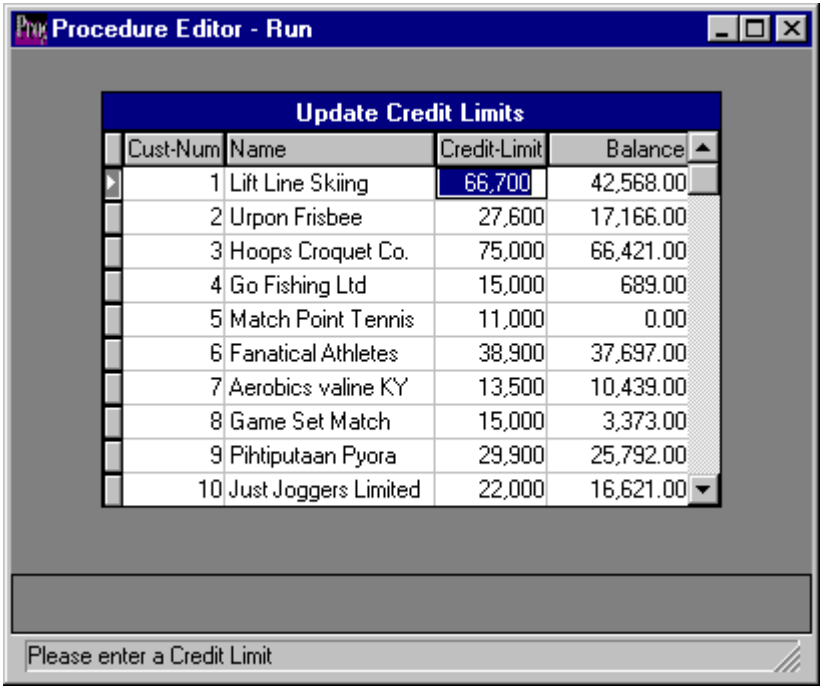
NOTE: When a browse is initialized, the buffer contains the first record in the query, since no record has been selected by the user. To keep the browse in sync with the query, the buffer contains the first record in the browse viewport while there is no selected record.

Note that an updatable browse can only have one selected row when it is in edit mode. So, if the user selects five rows and then an updatable cell, the four rows that do not contain the active cell are deselected. The code example below shows a simple multiple-select browse:

p-br03.p

```
DEFINE QUERY q1 FOR customer SCROLLING.
DEFINE BROWSE b1 QUERY q1 DISPLAY cust-num name credit-limit balance
    ENABLE credit-limit WITH 10 DOWN SEPARATORS MULTIPLE
    TITLE "Update Credit Limits".

DEFINE FRAME f1
    b1
        WITH SIDE-LABELS ROW 2 CENTERED NO-BOX.
OPEN QUERY q1 FOR EACH customer NO-LOCK.
ENABLE ALL WITH FRAME f1.
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

When you run this example, the following browse appears:



Browse Selection and Query Interaction

A browse can support either a single selection or multiple selections. The last selected row of the browse always becomes the current row of the query. In a multiple-select browse, if you deselect the current row, the query is repositioned to the currently selected row that was last selected. If no rows are selected, the query is repositioned to the first record in the viewport.

Table 10–1 summarizes how actions on the browse affect the associated query.

Table 10–1: Browse and Query Interaction

Browse Action	Effect on Query
Vertical keyboard navigation	Moves results list cursor for single-select browse widgets.
Select row	Put record(s) for that row into the record buffer(s).
Deselect current row	If other records are selected in the browse, put record(s) for the most recently selected of those rows into the record buffer(s). Otherwise, reposition to the first row in the viewport.
Deselect non-current row	None.

Using the GET statement to navigate within the results list of the query has no effect on the browse. However, the REPOSITION statement does update the current position of the browse. If you use GET statements for a query on which a browse is defined, you should use the REPOSITION statement to keep the browse synchronized with the query. Also, when you open or reopen the query with the OPEN QUERY statement, the browse is positioned to the first record.

10.3.4 Using Calculated Fields

A browse widget can show a calculated result of browse fields as a separate column. This is accomplished by adding the expression to the DEFINE BROWSE statement. All valid browse format phrase options are legal extensions to the expression (for example, a LABEL option). When the browse is opened, the calculated field is displayed for each row.

For updatable browse widgets, however, you need to refresh the calculation if one of the fields in the expression changes. The browse will display the appropriate calculations only when the query refreshes the data. It is more appropriate to programmatically refresh the calculated data when the user finishes editing the affected row. To accomplish this, use a base field as a placeholder for the expression. You then can reference the calculation and refresh the result as needed.

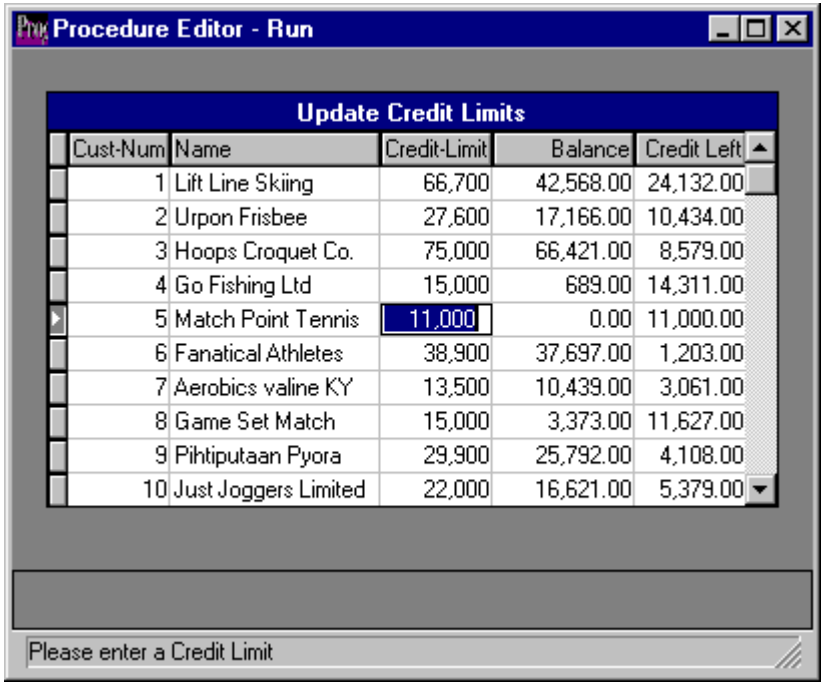
The code example below sets up a calculated field using a base field in the browse widget. The LEAVE trigger on the browse column forces a refresh of the calculated data as soon as the user leaves the browse cell:

p-br04.p

```
DEFINE VARIABLE credit-left AS DECIMAL LABEL "Credit Left".
DEFINE QUERY q1 FOR customer SCROLLING.DEFINE BROWSE b1 QUERY q1 DISPLAY
cust-num name credit-limit balance
    (credit-limit - balance) @ credit-left
    ENABLE credit-limit WITH 10 DOWN SEPARATORS
    TITLE "Update Credit Limits".

DEFINE FRAME f1
    b1
        WITH SIDE-LABELS ROW 2 CENTERED NO-BOX.

ON LEAVE OF credit-limit IN BROWSE b1 DO:
    DISPLAY (INTEGER(customer.credit-limit:SCREEN-VALUE) - balance)
            @ credit-left WITH BROWSE b1.
END.

OPEN QUERY q1 FOR EACH customer NO-LOCK.
ENABLE ALL WITH FRAME f1.
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

NOTE: With calculated fields in the DISPLAY phrase, you cannot use the ENABLE ALL option, unless you use the EXCEPT option to exclude the calculated field, since calculated fields cannot be edited.

When you run this example, the following browse appears:



10.3.5 Sizing a Browse and Browse Columns

Normally, the browse calculates its size based on the column width of each field and the number of display rows requested with the DOWN option. If the resultant browse is too big for the frame, Progress displays an error message at compile time.

Use the DOWN option to specify the number of rows to display and the WIDTH option to specify the width of the browse. If the total column width is wider than the specified browse width, the browse displays a horizontal scrollbar to access the columns that don't display initially, and you avoid the compilation error.

NOTE: In character interfaces, there is no horizontal scrollbar for a wide browse, but you can scroll the browse one column at a time, using **CURSOR–LEFT** and **CURSOR–RIGHT**.

The horizontal scrollbar can work in two different ways. By default, the browse scrolls whole columns in and out of the viewport. To change this behavior to pixel scrolling, specify the NO–COLUMN–SCROLLING option of the DEFINE BROWSE statement or set the COLUMN–SCROLLING attribute to NO.

You can also use the SIZE phrase to set an absolute outer size of the browse in pixel or character units and Progress will determine how many rows and columns can be displayed. (Note that the User Interface Builder uses the SIZE phrase.)

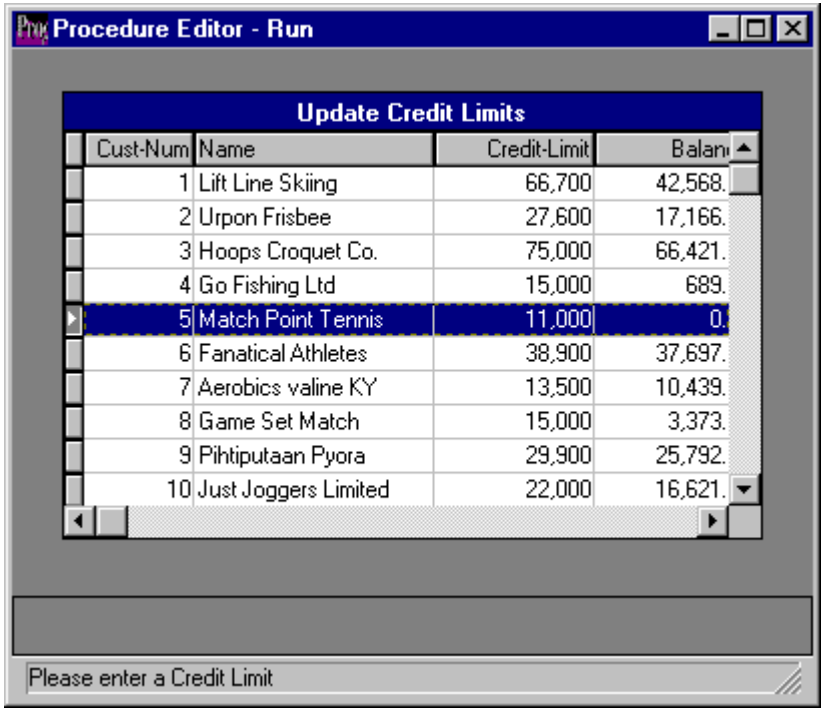
The browse also supports a column WIDTH option. Use this option to set the width of individual columns. When a column WIDTH option sets the width of an updatable column smaller than the size specified by the FORMAT string, the browse cell will scroll to accommodate extra input up to the size specified by FORMAT.

The example below uses the WIDTH option on each column to set the desired column width and the WIDTH option with the DOWN option to size the browse as a whole:

p-br05.p

```
DEFINE VARIABLE credit-left AS DECIMAL LABEL "Credit Left".
DEFINE QUERY q1 FOR customer SCROLLING.
DEFINE BROWSE b1 QUERY q1 DISPLAY cust-num WIDTH 10 name WIDTH 23
    credit-limit WIDTH 15 balance WIDTH 15
    (credit-limit - balance) @ credit-left WIDTH 15
    ENABLE credit-limit WITH 10 DOWN WIDTH 70 SEPARATORS
    TITLE "Update Credit Limits".

DEFINE FRAME f1
    b1
        WITH SIDE-LABELS ROW 2 CENTERED NO-BOX.

OPEN QUERY q1 FOR EACH customer NO-LOCK.
ENABLE ALL WITH FRAME f1.
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

When you run this example, the following browse appears:



10.3.6 Searching Columns (Windows Only)

When a read-only browse has focus, typing any printable key (an alphanumeric character), forces Progress to reposition the browse to the first row where the first character of the first browse column matches the typed character. Thus, the read-only browse allows searching, but based only on the data in the first column. If the browse finds no match, it wraps to the top of the column and continues searching.

In an updatable browse, the user can select the column that will be the basis of the character search. First, the user clicks a column label. The column label depresses. Typing any printable key searches for the first row in that column that matches and Progress focuses that row. Column searches also wrap to the top if necessary.

NOTE: A column search on a browse associated with a query with the INDEXED–REPOSITION option does not wrap to the top of the column if it cannot find a record to satisfy the search. This behavior is a side effect of the reposition optimization. To work around this, you can apply HOME to the query before starting the search.

There are two ways to extend this basic functionality. First, you can configure an updatable browse to look like a read-only browse. This technique gives you selectable columns and searching on those columns. Second, you can use the START–SEARCH and END–SEARCH events to trap the beginning and end of search mode. You can use this functionality to write your own search routines. For example, you can perform incremental searches using multiple keys.

Implementing Column-Searching on a Read-Only Browse

The column-searching feature is exclusive to the updatable browse. The read-only browse can do one-character deep searches on the initial column of the browse. You can work around this by defining your browse with one field enabled for input and the NO–ROW–MARKERS option specified. After the definition, disable the enabled column by using the READ–ONLY attribute. This provides an updatable browse that looks like a read-only browse. However, the updatable browse retains the ability for the user to select individual columns and perform searches on them.

Implementing Incremental Searching

The following code example depicts a multi-character incremental search on the customer name field. You can modify the basic algorithm to provide this type of search on any column:

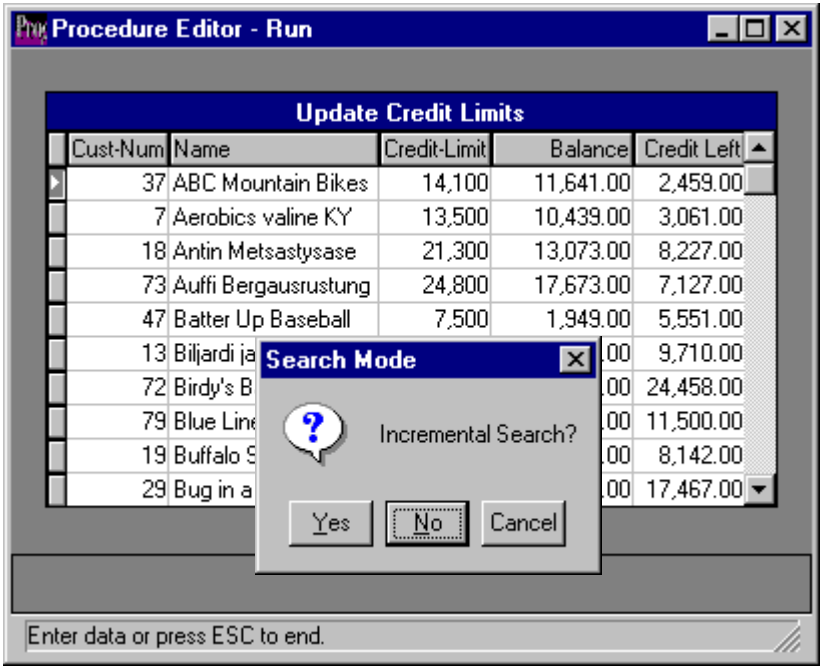
p-br06.p *(1 of 2)*

```
DEFINE QUERY q1 FOR customer SCROLLING.
DEFINE VARIABLE credit-left AS DECIMAL LABEL "Credit Left".
DEFINE BROWSE b1 QUERY q1 DISPLAY cust-num name credit-limit balance
    (credit-limit - balance) @ credit-left ENABLE credit-limit
    WITH 10 DOWN SEPARATORS TITLE "Update Credit Limits".
DEFINE VARIABLE method-return AS LOGICAL.
DEFINE VARIABLE wh AS WIDGET-HANDLE.
DEFINE VARIABLE lblflag AS LOGICAL.
DEFINE FRAME f1
    b1
WITH SIDE-LABELS ROW 2 CENTERED NO-BOX. ON END-SEARCH OF b1 IN FRAME f1
DO:
    IF lblflag THEN
        wh:LABEL = "Name".
    lblflag = FALSE.
END.

ON START-SEARCH OF b1 IN FRAME f1
DO:
    wh = b1:CURRENT-COLUMN.
    IF wh:LABEL = "name" THEN DO:
       lblflag = TRUE.
       wh:label = "SEARCHING...".
       MESSAGE "Incremental Search?" VIEW-AS ALERT-BOX QUESTION
          BUTTONS YES-NO-CANCEL TITLE "Search Mode"
         UPDATE answ AS LOGICAL.
       CASE answ:
          WHEN TRUE THEN DO:
              /* SETTING REPOSITIONED ROW */
              method-return = b1:SET-REPOSITIONED-ROW(3,"CONDITIONAL").
              RUN inc-src.
              RETURN.
          END.
          WHEN FALSE THEN DO:
              MESSAGE "Search mode defaults to the first character."
              VIEW-AS ALERT-BOX INFORMATION BUTTONS OK.
              RETURN NO-APPLY.
          END.
          OTHERWISE APPLY "END-SEARCH" TO SELF.
      END CASE.
    END.
END.
```

p-br06.p *(2 of 2)*

```
OPEN QUERY q1 FOR EACH customer NO-LOCK BY customer.name.
ENABLE ALL WITH FRAME f1.
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.PROCEDURE inc-src: DEFINE VARIABLE
curr-record AS RECID.
DEFINE VARIABLE initial-str AS CHARACTER.APPLY "START-SEARCH" TO BROWSE b1.ON
ANY-PRINTABLE OF b1 IN FRAME f1
DO:
   ASSIGN curr-record = RECID(customer)
          initial-str = initial-str + LAST-EVENT:LABEL.

   match-string:
   DO ON ERROR UNDO match-string, LEAVE match-string:
       IF RETRY THEN
           initial-str = LAST-EVENT:LABEL.

       FIND NEXT customer WHERE customer.name BEGINS initial-str
            USE-INDEX name NO-LOCK NO-ERROR

IF AVAILABLE customer THEN
            curr-record = RECID(customer).
        ELSE IF RETRY THEN
             DO:
                initial-str = "".
                BELL.
             END.
             ELSE
                UNDO match-string, RETRY match-string.
    END.

    REPOSITION q1 TO RECID curr-record.
    END.    ON VALUE-CHANGED OF b1 IN FRAME f1
    DO:
        initial-str = "".
    END.

    WAIT-FOR "END-SEARCH" OF BROWSE b1.
    APPLY "ENTRY" TO name IN BROWSE b1.
END PROCEDURE.
```

The screen below shows what happens when you choose the Name column:



In brief, the START–SEARCH trigger traps the user action and presents the user with choices. This is good for demonstration purposes, but in practice, you may want your custom search behavior to override the default behavior.

If the user chooses Yes at the Incremental Search alert box, then the trigger enables search mode. The search mode is instantiated and maintained entirely by the internal procedure, inc–src. This procedure executes a trigger when any printable key is typed. While the code can build up a valid search string, it keeps going. So, if the user types B L U E, the procedure finds Blue Line Hockey. As soon as the procedure encounters a character that creates a string it can't match, it assumes it is starting a new search. So if the User types B L U E A, the procedure focuses Blue Line Hockey, but discards the string and moves to ABC Mountain Bikes when it encounters the A.

Another way to implement multiple character searches is to provide the user with a Search button. The Search button would simply apply START–SEARCH to the correct column, and the START–SEARCH trigger and internal procedure would take over.

10.4 Programming with a Browse Widget

The following sections discuss important programming techniques involving browse widgets.

10.4.1 Browse Events

This section covers:

- Basic events
- Row events
- Column events

Basic Events

The basic browse events include:

- VALUE–CHANGED event occurs each time the user selects or deselects a row.
- HOME event repositions the browse to the beginning query list.
- END event repositions the browse to the end of the query list.
- DEFAULT–ACTION event occurs when you press `RETURN` or `ENTER` or when you double-click a row. DEFAULT–ACTION also has the side effect of selecting the row that is currently highlighted.
- The SCROLL–NOTIFY event occurs when the user adjusts the scrollbar.

Typically, you use the VALUE–CHANGED and DEFAULT–ACTION events to link your browse to other parts of your application, in the code example below, both events are used to handle displaying data that is not ideally suited for display in the browse. In this case, a supplemental editor widget is used to display the lengthy text strings found in the customer comments field.

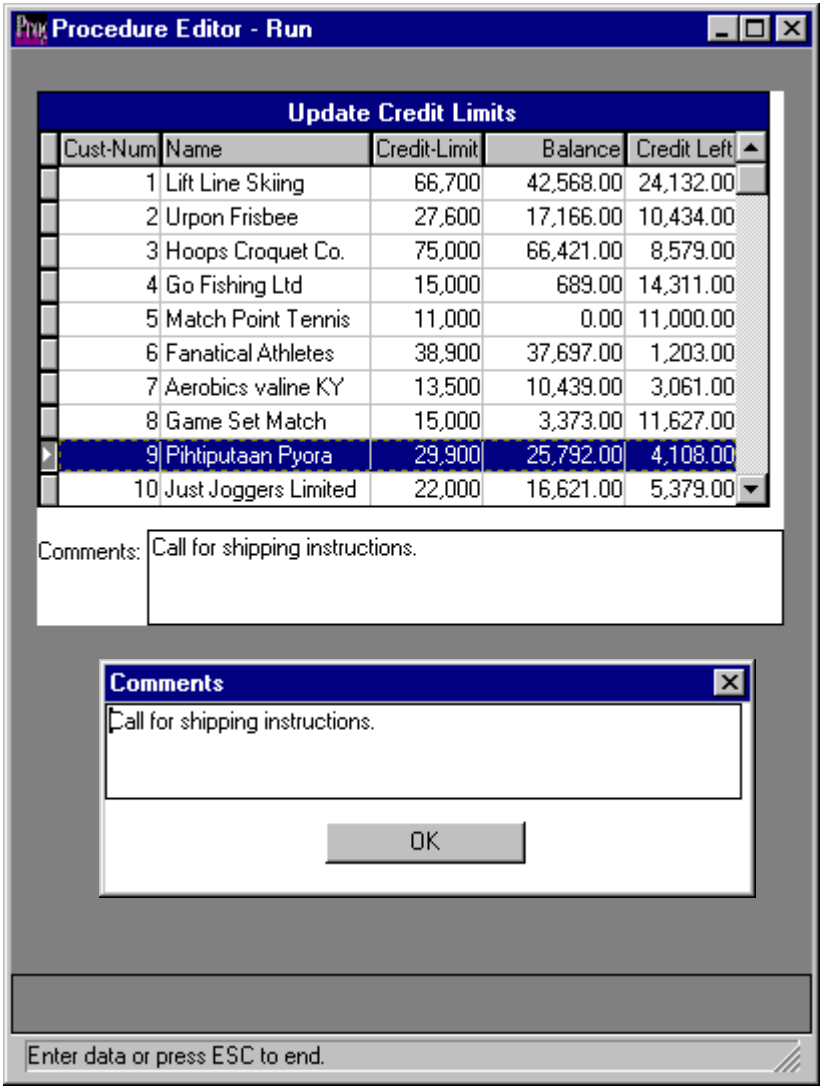
The trigger on VALUE–CHANGED refreshes the data in the comments editor as you navigate through the browse. This type of trigger is often used to keep related data in the same frame or window in sync with a browse.

The trigger on DEFAULT–ACTION displays the comments editor in a dialog box that pops up when you double-click a browse row. One typical use of DEFAULT–ACTION triggers is to display a supplementary view of data related to the record, as the following code sample demonstrates:

p-br07.p

```
DEFINE VARIABLE credit-left AS DECIMAL LABEL "Credit Left".
DEFINE QUERY q1 FOR customer SCROLLING.
DEFINE BROWSE b1 QUERY q1 DISPLAY cust-num name credit-limit balance
    (credit-limit - balance) @ credit-left
    ENABLE credit-limit WITH 10 DOWN SEPARATORS
    TITLE "Update Credit Limits".

DEFINE BUTTON b-ok LABEL "OK" SIZE 20 BY 1.

DEFINE FRAME f1
    b1 skip(.5)
    customer.comments VIEW-AS EDITOR INNER-LINES 3
        INNER-CHARS 62
        WITH SIDE-LABELS ROW 2 CENTERED NO-BOX.

DEFINE FRAME f2
    customer.comments NO-LABEL VIEW-AS EDITOR INNER-LINES 3
        INNER-CHARS 62 SKIP(.5)
    b-ok to 42 SKIP(.5)
        WITH SIDE-LABELS ROW 2 CENTERED
        VIEW-AS DIALOG-BOX TITLE "Comments".

ON VALUE-CHANGED OF b1
DO:
    DISPLAY customer.comments WITH FRAME f1.
END.
ON DEFAULT-ACTION OF b1
DO:
    ASSIGN customer.comments:READ-ONLY IN FRAME f2 = TRUE.
    DISPLAY customer.comments WITH FRAME f2.
    ENABLE ALL WITH frame f2.
    WAIT-FOR CHOOSE OF b-ok.
    HIDE FRAME f2.
END.

ASSIGN customer.comments:READ-ONLY IN FRAME f1 = TRUE.
OPEN QUERY q1 FOR EACH customer NO-LOCK.
ENABLE ALL WITH FRAME f1.
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

The following screen shows both ways that the comments field is handled in this example:



One use for the HOME and END events might be to force the results list to be built in a particular direction when working with INDEXED–REPOSITION queries.

The SCROLL–NOTIFY event is important if you plan to overlay other widgets on browse cells. SCROLL–NOTIFY gives you the ability to move the overlay widget to keep pace with the user's scrolling. See the "Overlaying Widgets on Browse Cells" section later in this chapter for more information on this technique.

Row Events

The browse supports three row-specific events:

- ROW–ENTRY occurs when a user enters edit mode on a selected row.
- ROW–LEAVE occurs when the user leaves edit mode.
- ROW–DISPLAY occurs when a row becomes visible in the browse viewport.

You can find examples of how ROW–ENTRY and ROW–LEAVE work in the "Updating Browse Rows" and "Creating Browse Rows" sections later in the chapter.

The ROW–DISPLAY event lets you change color and font attributes of a row or individual cells or to reference field values in the row. On Windows, the ROW–DISPLAY event lets you change the format of a browse-cell (by changing the value of its FORMAT attribute). The ROW–DISPLAY event cannot be used for any other purpose.

For example, you might want color overdue accounts in red. This kind of cell manipulation is only valid while the cell is in the viewport. For this reason, you need to use ROW–DISPLAY to check each new row as it scrolls into the viewport. The code fragment below shows a sample trigger:

```
ON ROW-DISPLAY OF b1
DO:
    IF balance > credit-limit THEN
        ASSIGN balance:BGCOLOR IN BROWSE browse1 = white-color
               balance:FGCOLOR IN BROWSE browse1 = red-color.
END.
```

NOTE:

- If you want to use this technique to change the colors of an extent field, you must do so by directly referencing each member.
- In character interfaces, the DCOLOR attribute specifies the color of an individual cell.

Column Events

The LEAVE and ENTRY events reference the entire browse widget. For example:

```
ON ENTRY OF b1 DO:
```

You can also write triggers for column names. You can use these triggers to change an attribute of the entire column (change a column background color), or to modify a single cell in that column. The affected cell is the named column in the focused row. For example:

```
ON LEAVE OF customer.credit-limit IN BROWSE b1 DO:
    SELF:FONT = bold-font.
END.
```

There are also two events that allow you to interact with search mode. START–SEARCH occurs when the user chooses a column label. END–SEARCH occurs when the user enters edit mode on a cell or selects a row. See the "Searching Columns (Windows Only)" section earlier in this chapter for more information on these events. Note that both of these events can be applied to columns to force the start and end of search mode.

10.4.2 Refreshing Browse Rows

If you are using a read-only browse and a record is updated through a dialog box or window, you need a way to refresh the currently focused browse row with the new data. Use the following statement at the end of the update code to refresh the browse:

SYNTAX

```
DISPLAY column-name . . . WITH BROWSE browse-widget.
```

See the "Using Calculated Fields" section earlier in this chapter for information on refreshing calculated fields.

10.4.3 Repositioning Focus

The browse and the query must remain in sync. At times, you might have to do some behind-the-scenes manipulation of the results list. To resync, you can use the REPOSITION statement. The REPOSITION statement:

- Moves the database cursor to the specified position
- Adjusts the browse viewport to display the new row

To avoid display flashing when doing programmatic repositions, you can set the REFRESHABLE attribute to FALSE, do the REPOSITION, and then set REFRESHABLE to TRUE.

The SET–REPOSITIONED–ROW( ) method gives you control over the position in the viewport where the browse will display the repositioned row.

The code fragment below demonstrates these techniques:

```
method-return = browse1:SET-REPOSITIONED-ROW(3,"CONDITIONAL").
    /* Displays the record at row 3 with focus, unless the record
       is already in the viewport. In this case, the record receives
       focus at its current row. */
   .
   .
   .
ON CHOOSE OF b-find-cust
DO:
    FIND FIRST customer WHERE /* your selection criteria */.

    ASSIGN temp-rowid = ROWID(customer)
           browse1:REFRESHABLE = NO.

    REPOSITION query1 TO ROWID temp-rowid.

    ASSIGN browse1:REFRESHABLE = YES.
END.
```

Note that SET–REPOSITIONED–ROW( ) is normally set once for the session. You can also use the GET–REPOSITIONED–ROW( ) method to find out the current target row for repositions.

10.4.4 Updating Browse Rows

By default, Progress handles the process of updating data in an updatable browse. Assuming that the browse starts with the record in NO–LOCK state, Progress follows these steps:

- On any user action that modifies data in an editable browse cell, Progress re-gets the record with a SHARE–LOCK. (No other user can change the data.) If the data has changed, Progress warns the user and redisplays the row with the new data.
- When the user leaves the row and has made changes to the row, Progress starts a transaction (or subtransaction) and gets the record from the database with EXCLUSIVE–LOCK NO–WAIT. If no changes were made, Progress does not start a transaction.
- If the GET with EXCLUSIVE–LOCK is successful, Progress updates the record, disconnects it (removes the lock), ends the transaction, and downgrades the lock to its original status.
- If the get with EXCLUSIVE–LOCK fails, Progress backs out the transaction, displays an error message, keeps the focus on the edited row, and retains the edited data.

The 4GL programmer also has the option to disable this default behavior and programmatically commit the changes by way of a trigger on the ROW–LEAVE event. First, you must supply the NO–ASSIGN option in the DEFINE BROWSE statement.

The following code example shows how to use a ROW–LEAVE trigger:

p-br08.p

```
DEFINE QUERY q1 FOR customer SCROLLING.
DEFINE BROWSE b1 QUERY q1 DISPLAY cust-num name credit-limit
    ENABLE credit-limit WITH 10 DOWN SEPARATORS NO-ASSIGN
    TITLE "Update Credit Limits".

DEFINE FRAME f1
    b1
        WITH SIDE-LABELS ROW 2 CENTERED NO-BOX.

ON ROW-LEAVE OF BROWSE b1 DO:
    IF INTEGER(customer.credit-limit:SCREEN-VALUE IN BROWSE b1) > 100000
    THEN DO:
        MESSAGE "Credit limits over $100,000 need a manager's approval."
            VIEW-AS ALERT-BOX ERROR BUTTONS OK
            TITLE "Invalid Credit Limit".
       DISPLAY credit-limit WITH BROWSE b1.
       RETURN NO-APPLY.
    END.

    DO TRANSACTION:
        GET CURRENT q1 EXCLUSIVE-LOCK NO-WAIT.
        IF CURRENT-CHANGED(customer) THEN DO:
             MESSAGE "The record changed while you were working."
                 VIEW-AS ALERT-BOX ERROR BUTTONS OK TITLE "New Data".
             DISPLAY credit-limit WITH BROWSE b1.
             RETURN NO-APPLY.
        END.
        ELSE ASSIGN INPUT BROWSE b1 credit-limit.
    END. /* TRANSACTION */
    GET CURRENT q1 NO-LOCK.
END.
OPEN QUERY q1 FOR EACH customer NO-LOCK.
ENABLE ALL WITH FRAME f1.
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

The ROW–LEAVE trigger checks for a special condition where the new data should not be committed to the database. If it encounters such data, the trigger is canceled without any writes taking place. If the data is valid, the GET CURRENT statement is used to re-get the current record with an EXCLUSIVE-LOCK NO–WAIT. It also uses the CURRENT–CHANGED function to ensure that another user has not changed the record since the query first retrieved it. Note that the lock must be downgraded manually at the end of the trigger with another GET CURRENT statement.

10.4.5 Creating Browse Rows

In an updatable browser, you might want to allow the user to add new records to the database. Programmatically, this requires three separate steps:

1. Create a blank line in the browse viewport with the INSERT–ROW() method and populate it with new data.

   NOTE: You can use the INSERT–ROW() browse method in an empty browser. It places a new row at the top of the viewport.

2. Use the CREATE statement and ASSIGN statement to update the database.

3. Add a reference to the results list with the CREATE–RESULT–LIST–ENTRY( ) method. (This step is only necessary if you do not plan to reopen the query after the update. However, this method makes reopening the query unnecessary for most applications.)

All three steps are required to create the record and keep the database, query, and browse in sync. Also, there are several possible side effects to allowing the user to add a record through an updatable browse. They include placing new records out of order and adding records that do not match the query. To eliminate these side effects, you can reopen the query after each new record.

The code example below shows one possible algorithm for creating new records with an updatable browse:

p-br09.p

```
DEFINE QUERY q1 FOR item SCROLLING.
DEFINE BROWSE b1 QUERY q1 DISPLAY item-num item-name price
    ENABLE item-name price WITH 10 DOWN SEPARATORS
    TITLE "Update Inventory".

DEFINE VARIABLE method-status AS LOGICAL.
DEFINE VARIABLE temp-rowid AS ROWID.
DEFINE BUTTON b-new LABEL "Add New Record" SIZE 45 BY 1.5.

DEFINE FRAME f1
    b1 skip
    b-new
        WITH SIDE-LABELS ROW 2 CENTERED NO-BOX.

ON CHOOSE OF b-new DO:
    method-status = b1:INSERT-ROW("AFTER").
END.

ON ROW-LEAVE OF BROWSE b1 DO:
    IF b1:NEW-ROW IN FRAME f1 THEN DO:
        DO ON ERROR UNDO, RETURN NO-APPLY:
            CREATE item.
            ASSIGN item-num = NEXT-VALUE(next-item-num)
                   INPUT BROWSE b1 item-name price.
            method-status = b1:CREATE-RESULT-LIST-ENTRY().
        END.
    END.
END.
OPEN QUERY q1 FOR EACH item NO-LOCK.
ENABLE ALL WITH FRAME f1.
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

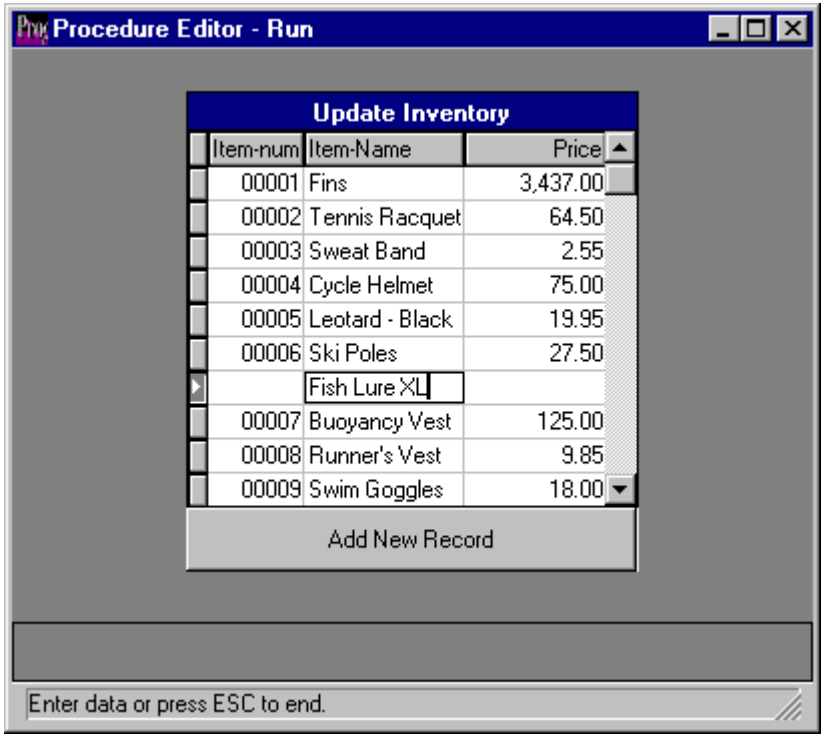
In this example, the user requests a new record by choosing a button. The button triggers the INSERT–ROW( ) browse method. This method takes two possible strings: BEFORE and AFTER, indicating where to place the new record—before or after the current selected row. The method places a new blank browse row in the browse viewport. The method has no effect on the database or the results list of the query.

Since you know that you have allowed the user to add new rows, you have to check for them. A ROW–LEAVE trigger provides the perfect vehicle for checking for this special case. Because the code does not contain the NO–ASSIGN option of the DEFINE BROWSE statement, the normal default row assignment occurs for updated existing records.

The NEW–ROW attribute tells you whether the current row is a new one. When you encounter one, you have to create a new blank database record and populate it with all the necessary data, which may come exclusively from the browse or from a mixture of sources. In this example, the item-num is generated by an existing database sequence.

At this point, the browse has correct data, the database has correct data, but the query results list does not. The CREATE–RESULT–LIST–ENTRY() method takes care of this task.

The following screen shows a new record being entered into the browse:



10.4.6 Deleting Browse Rows

Deleting a record by way of a browse is a two-step process. When the user indicates a deletion, you should re-get the records EXCLUSIVE–LOCK NO-WAIT and then use the DELETE statement to remove the records from the database. Next, you would use the DELETE–SELECTED–ROWS( ) (**plural**) method to delete one or many selected records from both the browse widget and the query results list. DELETE–SELECTED–ROWS( ) is a newer and more efficient browse method and supercedes the old technique of using the DELETE–SELECTED–ROW( ) (**singular**) and DELETE–CURRENT–ROW( ) methods.

The code example below demonstrates an algorithm for deleting many rows from both the database and the browse. The technique also works for deleting single rows and records:

p-br11.p

```
DEFINE VARIABLE method-return AS LOGICAL.
DEFINE BUTTON b-delete LABEL "Delete All Selected Rows".
DEFINE QUERY q1 FOR customer SCROLLING.
DEFINE BROWSE b1 QUERY q1 DISPLAY cust-num name ENABLE name
    WITH 10 DOWN SEPARATORS MULTIPLE.
DEFINE FRAME f1
    b1 SKIP(.5)
    b-delete
       WITH NO-BOX SIDE-LABELS ROW 2 CENTERED.
ON CHOOSE OF b-delete IN FRAME f1
DO:
    DEFINE VARIABLE i AS INTEGER.
    DO i = b1:NUM-SELECTED-ROWS TO 1 by -1:
        method-return = b1:FETCH-SELECTED-ROW(i).
        GET CURRENT q1 EXCLUSIVE-LOCK.
        DELETE customer.
    END.
    method-return = b1:DELETE-SELECTED-ROWS().
END.
OPEN QUERY q1 FOR EACH customer.
ENABLE b1 b-delete WITH FRAME f1.
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW
```

A button indicates that the user wants to delete the selected rows. Since the record begins with NO–LOCK, you must first re-get the record with EXCLUSIVE–LOCK NO–WAIT. After using the DELETE statement, to remove the record from the database, the DELETE–SELECTED–ROWS( ) method updates the query and browse.

10.4.7 Locking Columns

You can use the NUM–LOCKED–COLUMNS attribute to prevent one or more browse columns from scrolling out of the browse viewport when the horizontal scrollbar is used. A non-horizontal-scrolling column is referred to as a *locked column*.

Locked columns are always the left-most columns in the browse. In other words, if you set NUM–LOCKED–COLUMNS to 2, the first two columns listed in the DEFINE BROWSE statement will be locked. In the example below, the customer number and name never move out of the browse viewport, no matter which of the remaining fields the user accesses with the horizontal scrollbar:

p-br12.p

```
DEFINE QUERY q1 FOR customer SCROLLING.
DEFINE BROWSE b1 QUERY q1 DISPLAY cust-num name address address2 city
    state postal-code country contact phone sales-rep credit-limit balance
    ENABLE ALL WITH 10 DOWN SEPARATORS
    TITLE "Update Customer".

DEFINE FRAME f1
    b1
        WITH SIDE-LABELS AT ROW 2 COLUMN 10 NO-BOX.

ASSIGN b1:NUM-LOCKED-COLUMNS = 2.
OPEN QUERY q1 FOR EACH customer NO-LOCK.
ENABLE ALL WITH FRAME f1.
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

10.4.8 Moving Columns

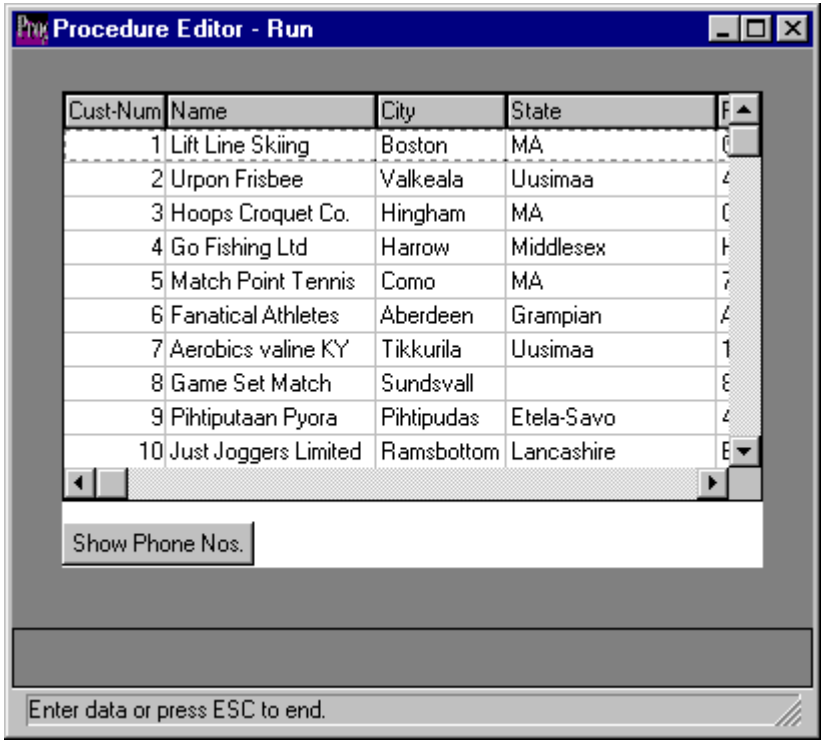
You can use the MOVE–COLUMN( ) method to rearrange the columns within a browse. For example, rather than forcing the user to scroll horizontally to see additional columns, you might allow them to reorder the columns.

The following example sets up a button that brings the phone column up to the third column position in the browse from the sixth position:

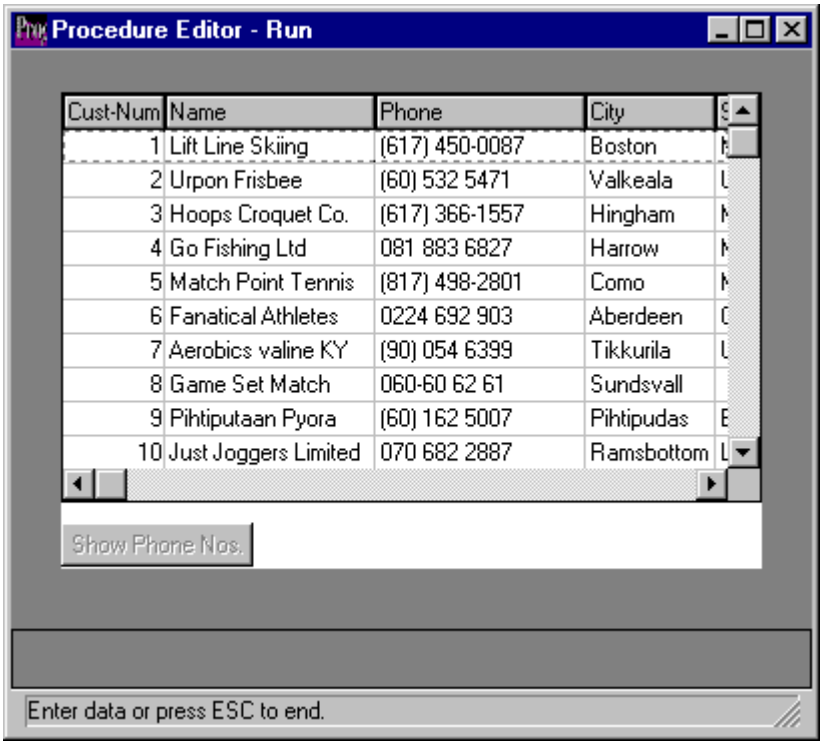
p-br13.p

```
DEFINE QUERY q1 FOR customer.
DEFINE BROWSE b1 QUERY q1 DISPLAY cust-num name city
    state postal-code phone WITH 10 DOWN WIDTH 70 SEPARATORS.

DEFINE BUTTON b-phone LABEL "Shown Phone Nos.".
DEFINE VARIABLE method-return AS LOGICAL.

DEFINE FRAME f1
   b1 SKIP(.5)
   b-phone
        WITH SIDE-LABELS ROW 2 CENTERED NO-BOX.
ON CHOOSE OF b-phone
DO:
    method-return = b1:MOVE-COLUMN(6, 3).  /* Move the phone column up. */
    DISABLE b-phone WITH FRAME f1.
END.
OPEN QUERY q1 FOR EACH customer.
ENABLE b1 b-phone WITH FRAME f1.
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

When you run this procedure, the following browse appears:



If you choose the button, the phone column moves to the third column, bumping the others back by one, as the following screen shows:



10.4.9 Overlaying Widgets on Browse Cells

One popular way to extend the functionality of an updatable browse is to overlay an updatable browse cell with another widget when the user enters the cell to edit it. For example, on ENTRY of a cell, you could display a combo box or a toggle box. The user selects an entry from the combo box, or checks the toggle box, and you use those values to update the SCREEN–VALUE attribute of the browse cell. The values get committed or undone along with the rest of the row.

NOTE: This technique is especially effective in graphical interfaces.

There are two problems to overcome. The first is in calculating the geometry to let you precisely position the overlay widget. You do this by adding the X and Y attributes of the browse cell to the X and Y attributes of the browse and assigning the result to the X and Y attributes of the overlay widget.

Next, you may need to move the overlay widget if the user accesses the scrollbar. A trigger on the SCROLL–NOTIFY event notifies you when the user scrolls, and you can update the X and Y positioning accordingly.

The code example below overlays a toggle box on the backorder field of the order–line table of the sports database.

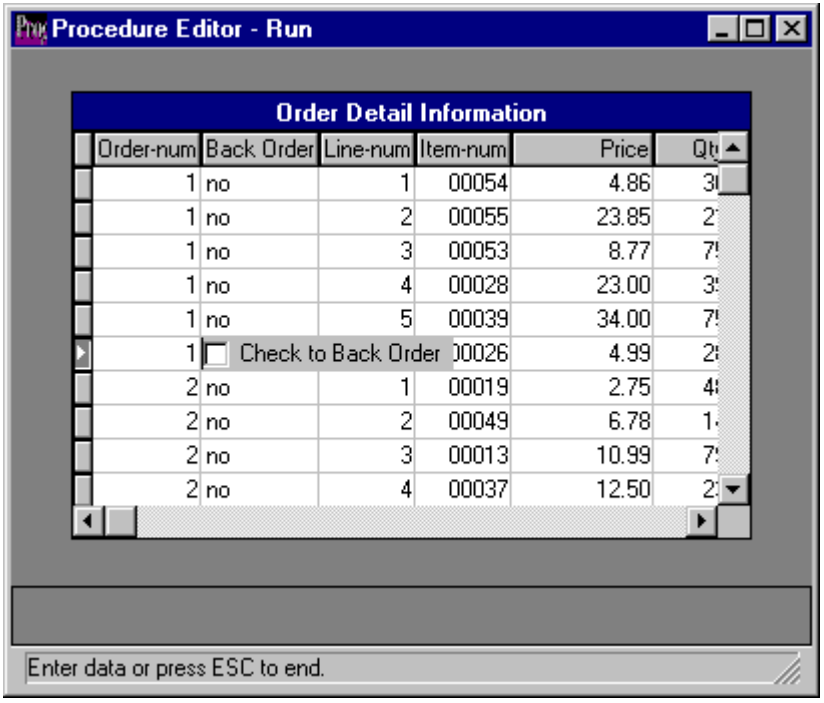
p-br14.p *(1 of 2)*

```
DEFINE VARIABLE wh AS WIDGET-HANDLE.
DEFINE VARIABLE eprice AS DECIMAL.
DEFINE VARIABLE initial-color AS LOGICAL.
DEFINE VARIABLE method-return AS LOGICAL.
DEFINE VARIABLE toggle1 AS LOGICAL LABEL "Check to Back Order"
    VIEW-AS TOGGLE-BOX.DEFINE QUERY q1 FOR order-line SCROLLING.
DEFINE BROWSE b1 QUERY q1 DISPLAY order-num backorder line-num item-num
      price qty (price * qty) @ eprice COLUMN-LABEL "Extended Price"
      ENABLE backorder line-num item-num price qty
      WITH 10 DOWN WIDTH 68 SEPARATORS TITLE "Order Detail
Information".DEFINE FRAME f1
    b1
    toggle1 AT ROW 2 COLUMN 2
        WITH THREE-D NO-BOX SIDE-LABELS ROW 2 CENTERED.
ON SCROLL-NOTIFY OF b1 IN FRAME f1
DO:
  RUN TOGGLE-PLACEMENT.
END.ON ENTRY OF backorder IN BROWSE b1
DO:
  RUN toggle-placement.
END.ON LEAVE OF backorder IN BROWSE b1
DO:
  toggle1:VISIBLE IN FRAME f1 = FALSE.
END.ON VALUE-CHANGED OF toggle1 IN FRAME f1
DO:
  toggle1:VISIBLE IN FRAME f1 = FALSE.
  IF toggle1:CHECKED THEN BACKORDER:SCREEN-VALUE IN BROWSE b1 = "YES".
  ELSE BACKORDER:SCREEN-VALUE IN BROWSE b1  = "NO".  APPLY "ENTRY" TO
BACKORDER IN BROWSE b1.
END.OPEN QUERY q1 FOR EACH order-line.
ASSIGN toggle1:HIDDEN IN FRAME f1 = TRUE
       toggle1:SENSITIVE IN FRAME f1 = TRUE.
ENABLE b1 WITH FRAME f1.
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
/* Continued on the next page. */
```

p-br14.p *(2 of 2)*

```
PROCEDURE toggle-placement:

ASSIGN WH = backorder:HANDLE IN BROWSE b1.
IF wh:X < 0 THEN toggle1:VISIBLE IN FRAME f1 = FALSE.
ELSE toggle1:X IN FRAME f1 = wh:X + b1:X.
IF wh:Y < 0 THEN toggle1:VISIBLE IN FRAME f1 = FALSE.
ELSE toggle1:Y IN FRAME f1 = wh:Y + b1:Y.

IF backorder:SCREEN-VALUE = "" THEN toggle1:CHECKED = FALSE.
ELSE toggle1:SCREEN-VALUE = backorder:SCREEN-VALUE.

IF wh:X >= 0 and wh:Y >= 0 THEN toggle1:VISIBLE IN FRAME f1 = TRUE.

END PROCEDURE.
```

This code also enables the THREE–D attribute on the parent frame, which provides a better contrast between the cell and the overlay widget, as the following screen shows:



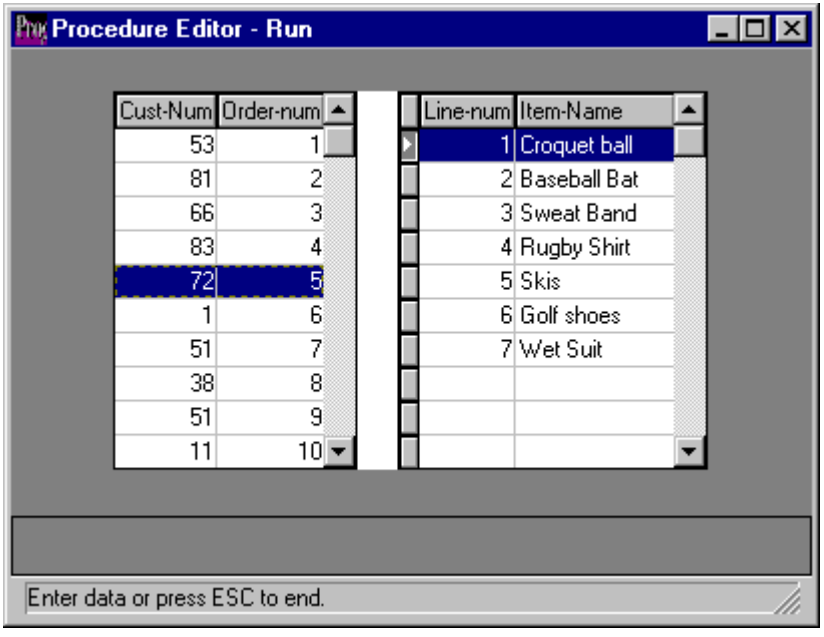
10.4.10 Using Multiple Browse Widgets for Related Tables

This next example shows a multi-table browse (or join) of the order–line and item tables:

p-br15.p

```
DEFINE QUERY orders FOR order.
DEFINE BROWSE ord QUERY orders DISPLAY order.cust-num order.order-num
    WITH 10 DOWN SEPARATORS.
DEFINE QUERY items FOR order-line, item.
DEFINE BROWSE itm QUERY items
    DISPLAY order-line.line-num item.item-name
    ENABLE item-name
    WITH 10 DOWN SEPARATORS.
DEFINE FRAME f1
  ord space(4) itm
  WITH NO-BOX SIDE-LABELS ROW 2 CENTERED.ON "VALUE-CHANGED" OF BROWSE ord
DO:
    ENABLE itm WITH FRAME f1.
    OPEN QUERY items FOR EACH order-line OF order,
        EACH item OF order-line.
END.
OPEN QUERY orders FOR EACH order.
ENABLE ord WITH FRAME f1.
APPLY "VALUE-CHANGED" TO BROWSE ord.WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

The following is the output of this example:



Each time you select a row in the ord browse, Progress displays the associated order line numbers and item descriptions in the itm browse. The itm browse joins the order–line and item tables.

The APPLY statement causes the order lines for the first order to display before you select a record.

A one-to-one (order–line item) join is best displayed with a single browse widget, because there is a single display line for both tables. In a one-to-many join (order and order–line), it is best to use two separate browse widgets.

10.4.11 Browse Style Options

This section offers a few ideas for changing the look of your browse.

Using Stacked Labels

To use more than one vertical line for your column labels, use the stacked label syntax. Here is an example:

```
DEFINE BROWSE b1 QUERY q1 DISPLAY cust-num COLUMN-LABEL "Customer!Number"
    name COLUMN-LABEL "Customer!Name" WITH 10 DOWN.
```

You must use the COLUMN–LABEL option instead of the LABEL option. The exclamation point character indicates the line breaks.

Justifying Labels

Column labels in the browse are left justified by default. You can use the C, L, and R options (Center, Left, Right) of the LABEL option to modify the justification of column labels:

```
DEFINE BROWSE b1 QUERY q1 DISPLAY cust-num LABEL "Cust. No.":C
    name WITH 10 DOWN.
```

Note the colon (:) syntax. To use this option, the justification option has to be attached to the end of the label option. So, even if you want to use the default labels, you need to re-enter them here in order to append the justification option.

Using Color to Distinguish Updatable Columns

You can make the updatable columns in your browse a different color. For example, you can make the read-only columns grey with black text and the updatable columns blue with yellow text. The code fragment that follows assumes you have defined variables to hold the standard color values:

```
ASSIGN cust-num:COLUMN-BGCOLOR IN BROWSE browse1 = gray-color
       cust-num:COLUMN-FGCOLOR IN BROWSE browse1 = black-color
       name:COLUMN-BGCOLOR IN BROWSE browse1 = blue-color
       name:COLUMN-FGCOLOR IN BROWSE browse1 = yellow-color.
```

NOTE: In character interfaces, the COLUMN–DCOLOR attribute specifies the column color.

Using Color and Font to Distinguish Cells

On top of your basic color scheme, you may want individual cells that have key values to display in a different color or font. For example, you might want to color overdue accounts in red. This kind of cell manipulation is only valid while the cell is in the viewport. For this reason, you need to use the special event ROW–DISPLAY to check each new row as it is scrolled into the viewport. See the "Browse Events" section earlier in this chapter for examples and implementation notes.

Establishing ToolTip Information

The DEFINE BROWSE statement supports the TOOLTIP option. You can elect to specify ToolTips, a brief text message string that automatically displays when the mouse pointer pauses over a browse widget for which a ToolTip value is defined. A ToolTip value can be set for a variety of field-level widgets. However, they are most commonly defined for button widgets. For more information on ToolTips and other specific Windows interface design options, see Chapter 25, "Interface Design."

Using a Disabled Updatable Browse as a Read-only Browse

In one sense, the default read-only browse is the precursor to the Version 8 updatable browse. It might be a good idea to compare the behavior and functionality of a read-only browse with a disabled updatable browse and decide which is the best standard for you. Using the default read-only browse guarantees compatibility with Version 7 applications. Using an updatable browse that has had its enabled columns turned off by way of the READ–ONLY attribute provides these benefits:

- Column searching (Windows only)
- Selection behaviors closer to native Windows standards (Windows only)
- Row markers
- Ability to enable the browse programmatically

10.5 Resizable Browse Widgets

In graphical interfaces, the user and the programmer can:

- Resize the browse
- Move the browse
- Resize a browse-column of a browse
- Move a browse-column of a browse
- Change the row height of a browse

NOTE: For a complete description of the 4GL elements described in this section, see the *Progress Language Reference*.

10.5.1 Resizing the Browse

To let the user resize a browse, the programmer sets the browse's RESIZABLE and SELECTABLE attributes to TRUE, as the following code fragment demonstrates:

```
mybrowse:RESIZABLE = TRUE.
mybrowse:SELECTABLE = TRUE.
```

To resize a browse through direct manipulation, the user clicks on the browse to display the resize handles, then drags a resize handle as desired.

To resize a browse programmatically, the programmer sets the browse's WIDTH–CHARS, WIDTH–PIXELS, HEIGHT–CHARS, HEIGHT–PIXELS, and DOWN attributes as desired.

The following code fragment programmatically resizes a browse to 50 characters wide by 40 characters high:

```
ASSIGN mybrowse:WIDTH-CHARS = 50
       mybrowse:HEIGHT-CHARS = 40.
```

10.5.2 Moving the Browse

To let the user move a browse, the programmer sets the browse's MOVABLE attribute to TRUE, as the following code fragment demonstrates:

```
mybrowse:MOVABLE = TRUE.
```

To move a browse through direct manipulation, the user drags the browse as desired.

To move a browse programmatically, the programmer sets the browse's X and Y attributes as desired.

The following code fragment programmatically moves a browse to the point (50,50) (in pixels) relative to the parent widget or display:

```
ASSIGN mybrowse:X = 50
       mybrowse:Y = 50.
```

10.5.3 Resizing the Browse-column

To let the user resize a browse-column, the programmer uses one of the following techniques:

- To let the user resize any browse-column of a browse, the programmer sets the browse's COLUMN–RESIZABLE attribute to TRUE, as the following code fragment demonstrates:

```
mybrowse:COLUMN-RESIZABLE = TRUE.
```

- To let the user resize a single browse-column of a browse, the programmer sets the browse-column's RESIZABLE attribute to TRUE, as the following code fragment demonstrates:

```
mybrowsecol:RESIZABLE = TRUE.
```

To resize a browse-column through direct manipulation, the user drags a column separator horizontally as desired.

To resize a browse-column programmatically, the programmer sets the browse-column's WIDTH–CHARS and WIDTH–PIXELS attributes as desired.

The following code fragment programmatically changes the width of the "Name" browse-column (representing the Name field of the Customer table of the Sports database) to 20 pixels:

```
customer.name:WIDTH-PIXELS IN BROWSE mybrowse = 20.
```

10.5.4 Moving the Browse-column

To let the user move the browse-columns of a browse, the programmer uses one of the following techniques:

- To let the user move any browse-column of a browse, the programmer sets the browse's COLUMN–MOVABLE attribute to TRUE, as the following code fragment demonstrates:

```
mybrowse:COLUMN-MOVABLE = TRUE.
```

- To let the user move a single browse-column of a browse, the programmer sets the browse-column's MOVABLE attribute to TRUE, as the following code fragment demonstrates:

```
mybrowsecol:MOVABLE = TRUE.
```

To move a browse-column through direct manipulation, the user drags a column label horizontally. (If the user drags a column label to either edge of the viewport, Progress scrolls the viewport.)

NOTE: To move the second browse-column to the left, the user should move the leftmost browse-column to the right instead, thereby swapping slots.

To move a browse-column programmatically, the programmer uses the MOVE–COLUMN method of the browse.

The following code fragment programmatically moves the first browse-column of a browse to the third position:

```
mybrowse:MOVE-COLUMN(1,3).
```

10.5.5 Changing the Row Height

To let the user change the row height of a browse, the programmer sets the browse's ROW–RESIZABLE attribute to TRUE, as the following code fragment demonstrates:

```
mybrowse:ROW-RESIZABLE = TRUE.
```

To change the row height of a browse, the user drags a row separator vertically as desired.

To change the row height programmatically, the programmer sets the browse's ROW–HEIGHT–CHARS and ROW–HEIGHT–PIXELS attributes as desired.

The following code fragment programmatically changes the row height to 20 pixels:

```
mybrowse:row-height-pixels = 20.
```

10.5.6 Additional Attributes

When the programmer sets up a resizable browse, whether for user or programmatic manipulation, she can set additional attributes.

User Manipulation

When setting up a browse for user manipulation, the programmer can set the following attributes:

- SEPARATORS attribute of the browse

  The SEPARATORS attribute indicates if the browse displays separators between each row and each column.

- SEPARATOR–FGCOLOR attribute of the browse

  The SEPARATOR–FGCOLOR attribute sets the color of the row and column separators, if they appear.

Programmatic Manipulation

When setting up a browse for the programmatic manipulation, the programmer can set the following attributes:

- SEPARATORS attribute of the browse

  See above.

- SEPARATOR–FGCOLOR attribute of the browse

  See above.

- EXPANDABLE attribute of the browse

  The EXPANDABLE attribute indicates whether Progress extends the right-most browse-column to the right edge of the browse, covering any white space that might appear.

- AUTO–RESIZE attribute of the browse-column

  The AUTO–RESIZE attribute indicates whether Progress automatically resizes the browse-column if a change occurs in its font or in its label's font or text.

10.5.7 User Manipulation Events

When the user moves or resizes a browse or one of its components, Progress fires one or more of the following events:

- START–MOVE event
- END–MOVE event
- START–RESIZE event
- END–RESIZE event
- START–ROW–RESIZE event
- END–ROW–RESIZE event
- SELECTION event
- DESELECTION event

These events do not fire when the browse is manipulated programmatically.

The programmer does not have to monitor these events; they appear here only for the programmer's reference. For example, the programmer might want to write trigger code for one or more of these events.

For more information on these events, see the "Events Reference" chapter of the *Progress Language Reference*.

10.5.8 Code Example

The following program displays a browse that the user can resize and move:

NOTE: The program works in graphical interfaces only.

p-rszbrw.p

```
/* p-rszbrw.p */
/* demonstrates resizable browse */
/* p-br01.p made resizable, etc. */

DEFINE QUERY q1 FOR customer SCROLLING.

DEFINE BROWSE b1 QUERY q1 DISPLAY cust-num name credit-limit balance
    WITH 10 DOWN WIDTH 50 TITLE "Resizable Browse".

DEFINE FRAME f1
    b1 AT COL 15 ROW 3
    WITH SIZE 80 BY 15 SIDE-LABELS ROW 2 CENTERED NO-BOX.

ASSIGN b1:MOVABLE                  = TRUE
       b1:RESIZABLE                = TRUE
       b1:SELECTABLE               = TRUE
       b1:ROW-RESIZABLE            = TRUE
       b1:COLUMN-MOVABLE           = TRUE
       b1:COLUMN-RESIZABLE         = TRUE
       b1:SEPARATORS               = TRUE
       b1:EXPANDABLE               = TRUE.

OPEN QUERY q1 FOR EACH customer NO-LOCK.
ENABLE ALL WITH FRAME f1.
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

When you run the program and click anywhere except on a separator, column label, or scroll button, the resize handles appear, as Figure 10–1 illustrates:

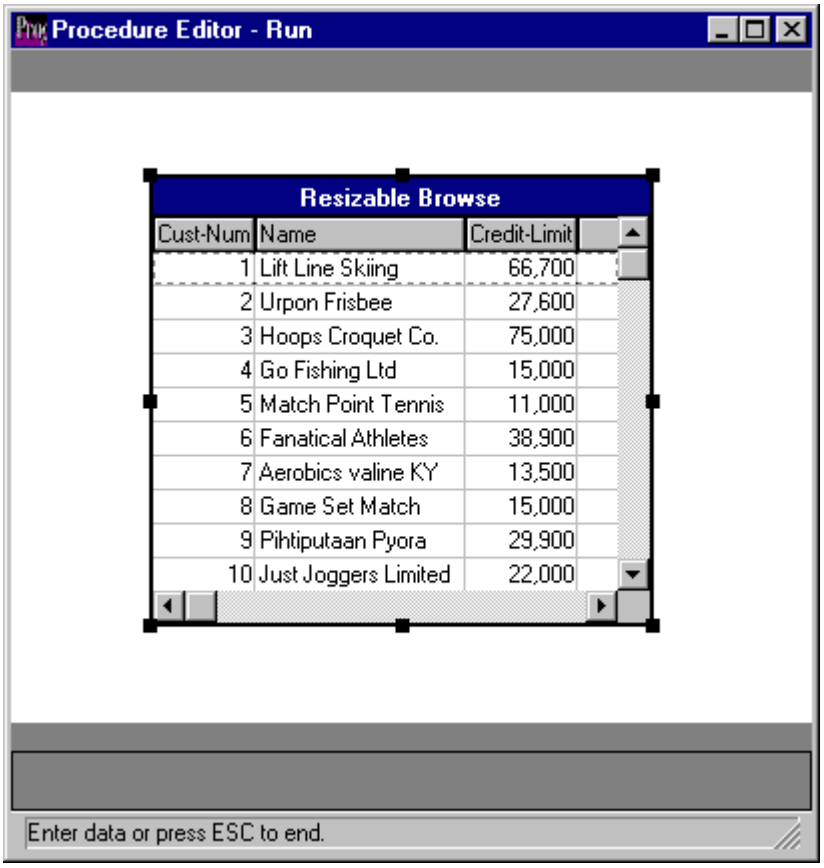


Figure 10–1: Resizable Browse

10.6 Using Browse Widgets in Character Interfaces

Browse widgets in character interfaces operate in two distinct modes:

- Row mode
- Edit mode

In row mode, the user can select and change focus among rows, as well as navigate (tab) to widgets outside the browse. In edit mode, the user can edit and tab between enabled cells within the browse. A user can only enter edit mode in an updatable browse, but can operate in row mode in either a read-only or an updatable browse.

10.6.1 Character Browse Modes

Figure 10–2 and Figure 10–3 (displayed using procedure `p-br03.p`) show a character browse in row mode and edit mode, respectively.

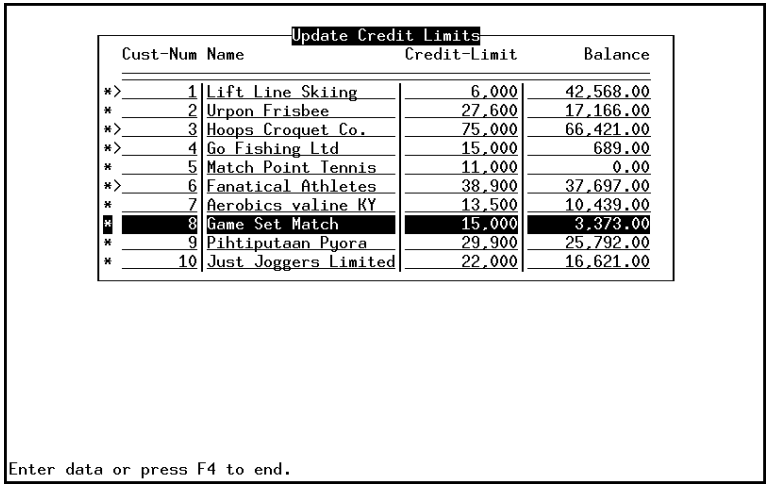


Figure 10–2: Character Browse in Row Mode

The asterisks (*) are row markers that indicate editable rows in an updatable browse. Row markers do not appear

- In a read-only browse.
- In an editable browse defined with the NO–ROW–MARKERS option, which makes the browse read-only.

In row mode, the highlight indicates the row that has focus and the close angle brackets (>) are tick marks that indicate selected rows. In a single selection browse, the tick mark follows the highlight because focus and selection are the same. In a multiple selection browse, focus is independent of selection, and tick marks indicate the selected rows.

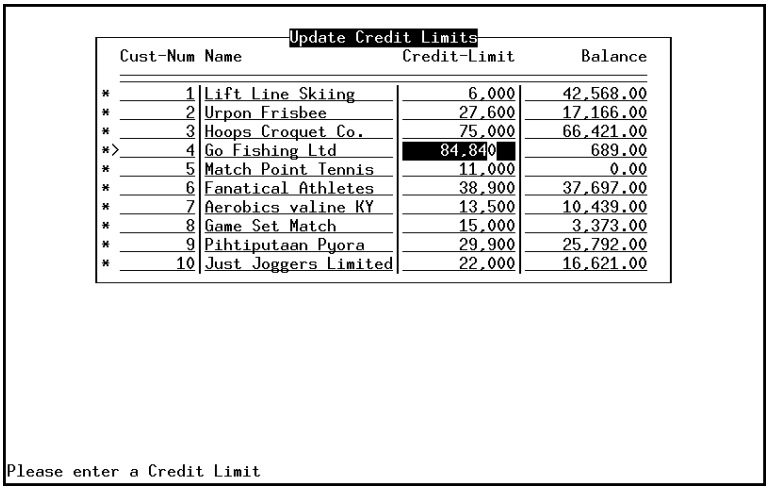


Figure 10–3: Character Browse in Edit Mode

In edit mode, all rows become deselected except the edited row. The highlight indicates the edited cell within the edited row. As the user moves from row to row during editing, the tick mark and highlight both move to indicate the row and cell being edited.

10.6.2 Control Keys

To control operation of a character browse, Progress provides a set of dedicated control keys. These control keys apply differently in row mode or edit mode. However, in both modes, any control key that changes (or attempts to change) the selected row fires the VALUE–CHANGED event.

These control keys correspond to Progress key functions that you can redefine. On Windows, you can redefine them in the registry or in an initialization file. On UNIX, you can redefine them in the PROTERMCAP file. For more information on redefining control key functions, see the chapter on user interface environments in the *Progress Client Deployment Guide*.

Table 10–2 describes the key functions, default key labels, and operation of the row mode control keys.

Table 10–2: Row Mode Control Keys *(1 of 2)*

Key Function	Default Key Label	Description
(Space Bar)[1]	" "	Fires the VALUE–CHANGED event. In multiple selection mode, this also selects and deselects the row that has focus, alternately displaying and erasing the tick mark.
CURSOR–LEFT	**CURSOR–LEFT**	Scrolls the browser widget horizontally one column to the left.
CURSOR–RIGHT	**CURSOR–RIGHT**	Scrolls the browser widget horizontally one column to the right.
CURSOR–DOWN	**CURSOR–DOWN**	Moves focus down one row.[1]
CURSOR–UP	**CURSOR–UP**	Moves focus up one row.[2]
END	**ESC–.**	Moves focus to the last row in the browse widget.[2]
HOME	**ESC–,**	Moves focus to the first row in the browse widget.[2]
PAGE–DOWN	**ESC–CURSOR–DOWN**	Pages down one full page of data.[2]
PAGE–UP	**ESC–CURSOR–UP**	Pages up one full page of data.[2]
REPLACE	**ESC–R**	Changes the browser to edit mode, places focus in the first enabled cell of the currently focused row, and fires the VALUE–CHANGED event. In multiple selection mode, this also selects the focused row and deselects any other selected rows. If you specify NO–ROW–MARKERS in the browse definition or set the ROW–MARKERS attribute to FALSE, REPLACE has no effect.

Table 10–2: Row Mode Control Keys *(2 of 2)*

Key Function	Default Key Label	Description
RETURN	**RETURN**	Fires the DEFAULT–ACTION event.
TAB	**TAB**	Leaves the browse widget and sets input focus to the next sibling widget of the browse in the tab order.

[1] Note that Progress has no key function identifier for the space bar. In code, you reference a character string containing a single space (" ").

[2] In single selection mode, this also fires the VALUE–CHANGED event, because selection follows focus.

Table 10–3 describes the key functions, default key labels, and operation of the edit mode control keys.

Table 10–3: Edit Mode Control Keys *(1 of 2)*

Key Function	Default Key Label	Description
(Space Bar)[1]	" "	Enters a space or zeroes numeric data in the focused cell.
CURSOR–LEFT	**CURSOR–LEFT**	Moves the cursor one character to the left in the cell. (Does not leave the cell.)
CURSOR–RIGHT	**CURSOR–RIGHT**	Moves the cursor one character to the right in the cell. (Does not leave the cell.)
CURSOR–DOWN	**CURSOR–DOWN**	Moves focus down to the next cell in the column and fires the VALUE–CHANGED event.
CURSOR–UP	**CURSOR–UP**	Moves focus up to the previous cell in the column and fires the VALUE–CHANGED event.
BACK–TAB	**CTRL–U**	Moves focus to the previous enabled cell in the browse (right to left; bottom to top).[1] When focus is on the first cell, **BACK–TAB** does not function.

Table 10–3: Edit Mode Control Keys *(2 of 2)*

Key Function	Default Key Label	Description
EDITOR–TAB	**CTRL–G**	Moves focus to the next enabled cell in the browse (left to right; top to bottom).[2] When focus is on the last cell the **EDITOR–TAB** does not function.
END	**ESC–.**	Moves focus to the last cell in the current column and fires the VALUE–CHANGED event.
HOME	**ESC–,**	Moves focus to the first cell in the current column and fires the VALUE–CHANGED event.
PAGE–DOWN	**ESC–CURSOR–DOWN**	Pages down one full page of data and fires the VALUE–CHANGED event.
PAGE–UP	**ESC–CURSOR–UP**	Pages up one full page of data and fires the VALUE–CHANGED event.
REPLACE	**ESC–R**	Changes the browser to row mode, with selection set to the currently focused row.
RETURN	**RETURN**	Moves focus to the next cell in the current column and fires the VALUE–CHANGED event.
TAB	**TAB**	Leaves the browse widget and sets input focus to the next sibling widget of the browse in the tab order.

[1] Note that Progress has no key function identifier for the space bar. In code, you reference a character string containing a single space (" ").

[2] If this action changes or attempts to change the selected row, it also fires the VALUE–CHANGED event.

10.6.3 Functional Differences from the Windows Graphical Browse

The character browse shares most of the same functional capabilities of the Windows graphical browse, including all methods and most attributes and events. For information on using these features, see the Chapter 10, "Programming with a Browse Widget" section.

However, there are differences from the Windows graphical browse in:

- Font management
- Color management
- Row and cell navigation

Font Management

Because there is no font management within character interfaces, all font attributes are inactive for the character browse.

Color Management

For color terminals, the character browse supports the following attributes to manage its color:

LABEL–DCOLOR

Specifies the color of a column label (like LABEL–FGCOLOR for the Windows browse).

COLUMN–DCOLOR

Specifies the color of a single column (like COLUMN–FGCOLOR for the Windows browse).

DCOLOR

Specifies the color of an individual cell (like FGCOLOR for the Windows browse). You can only specify the color of an individual cell as it appears in the view port. For more information on specifying individual cell color, see the "Browse Events" section.

COLUMN–PFCOLOR

Specifies the color of the enabled (updatable) column with input focus (unsupported for the Windows browse).

PFCOLOR

Specifies the color of the updatable cell with input focus (handled by default processing for the Windows browse).

By default, the COLUMN–PFCOLOR and PFCOLOR values are both set from the INPUT color value. On Windows, you can specify this value in the registry or in an initialization file. On UNIX, you can specify this value in the PROTERMCAP file.

Row and Cell Navigation

Unlike the Windows graphical browse, the character browse does not support column searching (positioning to a row or cell by typing a character contained in that row or cell).

Unlike the Windows graphical browse, the character browse uses a different set of key functions to tab between updatable cells than between the browse and other sibling widgets. Tabbing between cells occurs only in the edit mode of the character browse using the **EDIT–TAB** and **BACK–TAB** key functions.

Tabbing forward from the character browse to a sibling widget occurs in either row mode or edit mode using the **TAB** key function. However, tabbing backward from the character browse to a sibling widget occurs **only** in row mode using the **BACK–TAB** key function.

For more information on character browse key functions, see the "Control Keys" section.

11

Database Triggers

A *database trigger* is a block of 4GL code that executes whenever a specific *database event* occurs. A database event is an action performed against a database. For example, when you write a record to a database, a WRITE event occurs.

Because database triggers execute whenever a database event occurs, they are useful for tasks such as referential integrity. For example, if you delete a customer record from a database, you may also want to delete all of the customer's order records. The event (deletion of a customer record) initiates the task (deletion of all associated order records). A database trigger is ideal for this type of processing, because the same task must always be performed when the specific event occurs. Other suitable tasks are maintaining database security or writing database audit trails.

This chapter describes database events and database triggers. It also explains how to implement trigger-based replication.

11.1 Database Events

Database triggers associate a table or field with a database event. When the event occurs, the trigger executes.

Progress does not provide events for all database actions. For example, although you can dump database definitions from a database, you cannot write a trigger for a DUMP event, because Progress does not provide a DUMP event.

However, Progress does provides replication-related triggers in addition to standard triggers for certain events. Replication-related triggers help you implement database replication.

The database events that Progress supports, along with the standard triggers and replication-related triggers are:

- **CREATE** — When Progress executes a CREATE or INSERT statement for a database table, Progress creates the record, fires all applicable CREATE triggers, and then fires all applicable REPLICATION–CREATE triggers.

- **DELETE** — When Progress executes a DELETE statement for a database table, Progress fires all applicable DELETE triggers, fires all applicable REPLICATION–DELETE triggers, validates the delete, and then performs the delete.

- **FIND** — When Progress reads a record in a database table using a FIND or GET statement or a FOR EACH loop, Progress fires all applicable FIND triggers. FIND triggers fire only for records that completely satisfy the full search condition, such as a WHERE clause specifies. FIND triggers do not fire in response to the CAN–FIND function.

  If a FIND trigger fails, Progress behaves as though the record had not met the search criteria. If the FIND is within a FOR EACH block, Progress simply proceeds to the next record.

  If your application uses the BREAK option of the PRESELECT phrase (which forces Progress to retrieve two records at a time, so it can find the break), Progress executes the FIND trigger twice during the first FIND, which is actually two FINDs in succession. Thereafter, Progress looks one record ahead of the record currently in the record buffer, and executes the FIND trigger before it places the next record in the buffer.

- **WRITE** — When Progress changes the contents of a record and validates it for a database table, Progress first fires all applicable WRITE triggers and then all applicable REPLICATION–WRITE triggers. Progress automatically validates a record when releasing it. You can also use the VALIDATE statement to explicitly validate a record. In either case, WRITE triggers execute before the validation occurs (so WRITE triggers can correct values and do more sophisticated validation). Progress might execute the WRITE triggers for a single record more than once before it writes the record to the database (if it validates the record more than once and you modify the record between validations).

- **ASSIGN** — When Progress updates a field in a database record, Progress fires all applicable ASSIGN triggers. Unlike the other database events, this trigger monitors a specific field rather than a table. ASSIGN triggers execute when the contents of the associated field are modified. The trigger procedure executes at the end of a statement that assigns a new value to the field and after any necessary re-indexing. If the statement contains several field assignments (for example, UPDATE name city st), Progress fires each applicable ASSIGN trigger at the end of the statement. If any trigger fails, Progress undoes the statement (unless the code specifies NO–UNDO).

For more information on replication-related triggers and database replication, see the *Progress Database Administration Guide and Reference*.

11.2 Schema and Session Database Triggers

Progress supports two types of database triggers: *schema* and *session*. A schema trigger is a .p procedure that you add, through the Data Dictionary, to the schema of a database. A session trigger is a section of code that you add to a larger, enclosing procedure.

11.2.1 Differences Between Schema and Session Triggers

Although their syntax is slightly different, schema and session triggers provide similar functionality. The important difference between them is that schema triggers are independent procedures; whereas session triggers are contained within a larger procedure. Because of this difference, schema triggers always execute when a specified event occurs, regardless of what application initiates the event. Session triggers are defined as part of an application and are only in effect for that application.

Since session triggers are executed from within an enclosing procedure, they have access to the frames, widgets, and variables defined in the enclosing procedure. Since schema triggers are compiled separately from the procedure that initiates their execution, they do not have access to the procedure's frames, widgets, and variables.

Use schema triggers for processing that you **always** want to perform for a specific event. For example, when an order record is deleted, you may always want to delete the corresponding order-line records. Use session triggers to perform additional or independent processing when the event occurs.

Both types of triggers can return ERRORS that cause the associated event to fail. For more information on the ERROR option of the RETURN statement, see the *Progress Language Reference*.

11.2.2 Trigger Interaction

You can define a schema and a session trigger for the same table/event or field/event pair. How the triggers interact depends on how you define them. Ordinarily, both triggers execute. When both execute, the session trigger executes first, except for a FIND event. A schema trigger defined for a FIND executes before a session trigger for the same table. This ensures that any the schema FIND trigger you define for security reasons applies to the record first. For a WRITE, DELETE, CREATE, or ASSIGN event, the schema trigger can override the session trigger. For a FIND event, the schema trigger can preempt the session trigger.

However, the session trigger executes in place of the schema trigger if you do the following:

- define the schema trigger as overridable (OVERRIDE=YES)
- specify the OVERRIDE option in the session trigger

11.2.3 Schema Triggers

You create schema triggers through the Table or Field Properties dialog box in the Data Dictionary. When you use the Data Dictionary to define a schema trigger for a table or field, the trigger is automatically added to the table or field's data definitions. Progress allows you to define the trigger while you are creating or modifying a table or field. This trigger definition is stored in a trigger procedure. A *trigger procedure* is a Progress procedure file that begins with a *trigger header* statement. When defining the trigger, you have the option to specify CRC checking. Thus Progress can verify that the trigger procedure is valid before running it.

When application procedures are compiled against a database, schema triggers are not compiled into the application r-code, unlike delete or field validation in the Data Dictionary.

For information on using the Data Dictionary to create and delete triggers, see the *Progress Basic Database Tools* manual (character only), and for graphical interfaces, the on-line help for the Data Dictionary. For more information on CRCs and schema triggers, see Appendix A, "R-code Features and Functions."

CREATE, DELETE, and FIND Headers

This is the syntax for the trigger header for a CREATE, DELETE, FIND, REPLICATION–CREATE, or REPLICATION–DELETE trigger:

SYNTAX

```
TRIGGER PROCEDURE FOR
  {   FIND
    | CREATE
    | DELETE
    | REPLICATION-CREATE
    | REPLICATION-DELETE
   } OF table
```

References to the record can be made through an automatic buffer defined as part of the trigger header. Its name is the same as the table name.

WRITE Header

This is the syntax for the trigger header for a WRITE or REPLICATION–WRITE event:

SYNTAX

```
TRIGGER PROCEDURE FOR { WRITE | REPLICATION-WRITE } OF table
  [ NEW [ BUFFER ] buffer-name1 ]
  [ OLD [ BUFFER ] buffer-name2 ]
```

When executing a WRITE trigger, or a REPLICATION–WRITE trigger, Progress makes two record buffers available to the trigger procedure. The NEW buffer contains the modified record that is being validated. The OLD buffer contains the most recent version of the record before the latest set of changes were made (this is the template record if it is a newly created record, the record from the DB if it is not been validated, or the most recently validated record if it has been validated). You can modify the contents of the NEW buffer, but the OLD buffer is read-only. You can compare the contents of these two buffers and perform actions based on the results.

If you do not specify a NEW buffer, you can use the table name to make any comparisons to the OLD buffer. If you specify neither OLD or NEW buffers, you may reference the new record via the automatically created buffer named for the table itself.

You can determine whether the record being written is newly created by using the Progress NEW function. This function returns TRUE if Progress has not written the record to the

database; otherwise, it returns FALSE. However, the NEW function continues to return TRUE even if Progress has validated the record but has not written it to the database.

ASSIGN Header

This is the syntax for the trigger header for an ASSIGN event:

SYNTAX

```
TRIGGER PROCEDURE FOR ASSIGN
  {    OF table.field
    | NEW [ VALUE ] parameter1 { AS data-type | LIKE field
  }
      [    COLUMN-LABEL label
         | FORMAT string
         | INITIAL constant
         | LABEL string
         | NO-UNDO
      ]
  [ OLD [ VALUE ] parameter2 { AS data-type | LIKE field }
  ]
      [   COLUMN-LABEL label
         | FORMAT string
         | INITIAL constant
         | LABEL string
         | NO-UNDO
  ]
```

You can obtain the newly assigned value from the record buffer if you specify the OF option (this gives you access to the rest of the record also; whereas NEW and OLD give access to the fields only), or you can obtain the value from *parameter1* if you specify the NEW option. If you want to access the value stored in the database, use the OLD option. The value of the field before the assignment is stored in *parameter2*. You can use the NEW and OLD options to compare the value being written to the existing value. Progress treats the old value as an input run time parameter; you can modify it, but this modification has no effect on the triggering procedure.

11.2.4 CREATE Schema Trigger Example

The following example CREATE schema trigger procedure uses a sequence called Next–Cust–Num to generate a unique number for creating a customer record. For more information on sequences, see Chapter 9, "Database Access."

crcust.p

```
TRIGGER PROCEDURE FOR Create OF Customer.

/* Automatically Increment Customer Number using Next-Cust-Num Sequence */

ASSIGN Customer.Cust-Num =
   NEXT-VALUE(Next-Cust-Num).
```

11.2.5 DELETE Schema Trigger Example

This is an example of a DELETE schema trigger procedure:

p-dltcst.p

```
TRIGGER PROCEDURE FOR DELETE OF customer.
  FOR EACH invoice OF customer:
    IF invoice.amount > invoice.total-paid + invoice.adjustment THEN DO:
    MESSAGE "Outstanding unpaid invoice exists. Cannot delete.".
    RETURN ERROR.
    END.
END.
FOR EACH order OF customer:
  DELETE order.
END.
```

This DELETE trigger executes whenever you try to delete a customer record. The procedure finds outstanding invoices of the customer and checks to see if they are paid. If not, the trigger returns an error and the deletion fails. If so, the trigger deletes all of the customer's orders.

The delete trigger executes before Progress evaluates validation criteria, so you can perform any actions that would allow the delete to pass the validation.

All DELETE triggers execute before Progress actually deletes the record and can therefore refer to fields in the deleted record.

Even though Progress has not yet deleted the record when the DELETE trigger executes, any FIND request in the same application for that record fails. Since the record is in the process of being deleted, Progress treats it as though it is not available.

11.2.6 ASSIGN Schema Trigger Example

ASSIGN triggers execute when you assign a value to a database field (all other database triggers apply to tables). This is an example of an ASSIGN schema trigger procedure:

p-ascust.p

```
TRIGGER PROCEDURE FOR ASSIGN OF customer.cust-num
    OLD VALUE oldcust.

FOR EACH order WHERE order.cust-num = oldcust:
  order.cust-num = customer.cust-num.
END.
```

This ASSIGN trigger changes the value of the order.cust–num field to equal the newly assigned value of customer.cust–num.

NOTE: If you enable a WRITE trigger on a table and an ASSIGN trigger on a field within that table, the ASSIGN trigger always executes before the WRITE trigger. However, if the WRITE trigger procedure assigns a value to the trigger field, this re-invokes the ASSIGN trigger.

11.2.7 REPLICATION–CREATE Schema Trigger Example

REPLICATION–CREATE triggers execute just after Progress fires any applicable CREATE triggers. Here is an example of one:

```
TRIGGER PROCEDURE FOR REPLICATION-CREATE OF Customer.
CREATE Replication.
ASSIGN
      Replication.Entry-Id = NEXT-VALUE(Replication-entry)
      Replication.Table-Name = "Customer"
      Replication.Task-Id = DBTASKID(LDBNAME(BUFFER Replication))
      Replication.Repl-Event = "Create".
RAW-TRANSFER Customer TO Replication.Record.
```

This REPLICATION–CREATE trigger captures all the information on a newly-created Customer record you need in order to replicate the create. For more information, see the section "Implementing Trigger-based Replication."

11.2.8 REPLICATION–DELETE Schema Trigger Example

REPLICATION–DELETE triggers execute just after Progress fires any applicable DELETE triggers. This is an example of one:

```
TRIGGER PROCEDURE FOR REPLICATION-DELETE OF Customer.
CREATE Replication.
ASSIGN
      Replication.Entry-Id = NEXT-VALUE(Replication-entry)
      Replication.Table-Name = "Customer"
      Replication.Task-Id = DBTASKID(LDBNAME(BUFFER Replication))
      Replication.Repl-Event = "Delete"
      Replication.Key-Val = STRING(Customer.Cust-num).
```

This REPLICATION–DELETE trigger captures all the information on a newly-deleted Customer record you need in order to replicate the delete. For more information, see the section "Implementing Trigger-based Replication."

11.2.9 REPLICATION–WRITE Schema Trigger Example

REPLICATION–WRITE triggers execute just after Progress fires any applicable DELETE triggers. Here is an example of one:

```
/* custwrr.p */
TRIGGER PROCEDURE FOR REPLICATION-WRITE OF Customer OLD BUFFER oldbuf.
CREATE Replication.
ASSIGN
      Replication.Entry-Id = NEXT-VALUE(Replication-entry)
      Replication.Table-Name = "Customer"
      Replication.Task-Id = DBTASKID(LDBNAME(BUFFER Replication))
      Replication.Repl-Event = "Write"
      Replication.Key-Val = STRING(IF NEW(Customer) THEN Customer.Cust-num
            ELSE oldbuf.Cust-num).
RAW-TRANSFER Customer TO Replication.Record.
```

This REPLICATION–WRITE trigger captures all the information on a newly-modified Customer record you need in order to replicate the modify. For more information, see the section "Implementing Trigger-based Replication."

11.3 Implementing Trigger-based Replication

You implement trigger-based replication in Progress by using the 4GL, which provides the language constructs used to perform data replication. The Data Dictionary Tool also provides support. It allows defining of schema level replication triggers, as well as storing the replication procedure name in the Table Properties Window.

Progress also provides log–based replication. For information on log–based replication, see the *Progress Database Administration Guide and Reference*.

11.3.1 4GL Support

Table 11–1 summarizes the Progress core elements supporting replication. For complete information about statements and functions, see the appropriate entry in the *Progress Language Reference*.

Table 11–1: Replication 4GL Elements

Element Name	Type	Description
RAW	data type	Contains the database record to be replicated.
RAW–TRANSFER	statement	Copies a database record buffer to a field whose data type is RAW, or copies a RAW field to a buffer.
REPLICATION–CREATE	trigger	Executes when a record is created.
REPLICATION–DELETE	trigger	Executes when a record is deleted.
REPLICATION–WRITE	trigger	Executes when a record is written to the database.
DISABLE TRIGGERS FOR LOAD	statement	Disables triggers before loading records from one database to another.
ALLOW–REPLICATION	phrase	Option for the DISABLE TRIGGERS FOR LOAD statement. It allows replication to occur.
DBTASKID	function	Takes a logical database name (or alias) and returns a unique identifier for the database identifying a transaction.

11.3.2 Data Dictionary Support

The Data Dictionary Tool provides support for replication development. The three replication triggers are included in the standard TRIGGER area of the Table Maintenance screen. In addition, a *Replication* field is included in the Table Properties window. This field specifies the replication procedure for a table. There is no validation or automatic functionality for this field. It is only a storage area that can be accessed through the Data Dictionary. For more information about the Data Dictionary, see *Progress Basic Development Tools*.

11.3.3 Trigger-based Replication Procedure

This section uses a simple one-way replication scheme to illustrate how to implement replication using Progress 4GL. In the description, the source indicates the database from which records are being transferred. The target is the database to which records are being transferred. Depending on the replication scheme, each database (or site) can fill the roles of source and target. Replication occurs record by record, and the source and target database tables must have identical field layouts.

Implementation of replication occurs in two stages:

1. Logging changes to tables in a separate log table at the source database.

2. Setting up a process to inspect the log periodically and replicate the data operations on the appropriate target databases.

Creating the Replication Changes Log

Create a database table to log changes for replication. Follow these steps to create the replication changes log:

1 ♦ Using the Data Dictionary Tool, create a table with the fields shown in Table 11–2. (For the remainder of this procedure, this table is called *Replog*.)

Table 11–2: Replication Changes Log Table Schema

Field	Datatype	Description
TransactionID	integer	Represents a unique transaction number. This number is obtained through the DBTASKID function.
TableName	char	Identifies the table to be changed.
KeyValue	char	Represents a unique key value that identifies the corresponding record in the target database. Applies to deleted or modified records only.
Event	char	Indicates that the record was created, deleted, or modified.
DataRecord	RAW	Contains the created or modified record.
LogDate	date	Standard date format.
LogTime	char	String representing time.

2 ♦ Select a table to be replicated.

3 ♦ Using the Data Dictionary Tool, register database triggers REPLICATION–CREATE, REPLICATION–DELETE, and REPLICATION–WRITE for the selected table.

These triggers will execute following the execution of any corresponding CREATE, DELETE, or WRITE trigger.

4 ♦ In each of the trigger procedures, do the following to create a record to register the data change:

- Identify the event and table (*TableName* and *Event* fields).
- Use the DBTASKID function to get the transaction number (*TransactionID*).
- Use the RAW–TRANSFER statement to move the record to the *DataRecord* field.

The following code example shows a generic trigger include file with trigger code:

```
/* Trigger Include File 'reptrig.i' for replication*/

CREATE RepLog

  ASSIGN
   RepLog.TransactionID = DBTASKID(LDBDNAME(BUFFER {&RepLog}))
   RepLog.TableName = '{&Table}'
   RepLog.KeyValue = '{&Key}'
   RepLog.Event = '{&Event}'
   RepLog.LogDate = TODAY
   RepLog.LogTime = TIME.

  RAW-TRANSFER BUFFER {&Table} TO FIELD RepLog.DataRecord.
```

```
TRIGGER PROCEDURE FOR REPLICATION-WRITE OF Customer OLD Buffer oldcust.
{reptrig.i
&Table = Customer
&Key = STRING (oldcust.cust-num)
&Event = WRITE}
```

NOTE: More examples with sample source code can be found in `DLC/src/prodict/rplctn`.

Replicating the Data

A batch Progress client on the server machine monitors the replication changes log regularly and carries out replication. The batch Progress client transfers the data to the target database in a number of ways. In the simplest scenario, the batch client connects to the target database, reads the log, and executes a RAW–TRANSFER statement to transfer the data from the RAW *DataRecord* field to the target table on the target database.

If the connection to the target database is not feasible, one alternative is to dump pertinent log records to a text file and ship the file to the server machine. Then a monitoring batch client re-creates the log and transfers the data to the target database.

RAW-TRANSFER Statement

The RAW-TRANSFER statement allows you to transfer a record from a RAW field to a buffer, or from a buffer to a RAW field. You can also transfer directly from buffer to buffer. In 4GL replication, you use RAW–TRANSFER to move whole records to the RepLog and to move records from the log to the target database.

The following code example shows how you can use RAW-TRANSFER:

```
DEFINE VAR rvar AS RAW NO-UNDO.

FOR EACH sourcedb.customer:

  RAW-TRANSFER BUFFER sourcedb.customer TO FIELD rvar.

  RAW-TRANSFER FIELD rvar TO BUFFER targetdb.customer.

  RELEASE targetdb.customer.

END.
```

In this example, the first RAW–TRANSFER statement copies a record buffer in the source database to a variable of type RAW. The second RAW–TRANSFER statement copies the RAW field to the corresponding record buffer in the target database.

This replication could be written more efficiently as a buffer-to-buffer transfer, as shown in the following example:

```
FOR EACH sourcedb.customer:

  RAW-TRANSFER sourcedb.customer TO targetdb.customer.

  RELEASE targetdb.customer.

END.
```

For more information, see the entry for the RAW–TRANSFER statement in the *Progress Language Reference*.

DISABLE TRIGGERS FOR LOAD Statement

The DISABLE TRIGGERS FOR LOAD statement allows you to disable database triggers before replicating data. The following are reasons why you might want to include this statement in your code before executing the RAW–TRANSFER statement:

- Replication in Progress is based on the original scope of the transaction. If the database trigger code in the source database changes data in other tables, those other tables can also participate in replication. Using DISABLE TRIGGERS FOR LOAD allows the replication process itself to keep the database records consistent.

- In a two-way replication scheme (peer-to-peer model) DISABLE TRIGGERS FOR LOAD prevents a target database from creating a record of a change generated by the replication itself. This would result in changes being replicated back and forth between databases.

ALLOW REPLICATION Phrase

ALLOW-REPLICATION is an optional qualifier on the DISABLE TRIGGERS FOR LOAD statement. If ALLOW-REPLICATION is indicated, only CREATE, DELETE, ASSIGN, and WRITE triggers will be disabled. REPLICATION-CREATE, REPLICATION-DELETE, and REPLICATION-WRITE will execute when necessary.

This qualifier is useful, for example, in a cascading replication, where the target database is replicated to other targets in a cascading fashion. In this scenario, you might want to disable standard database triggers but allow replication triggers to fire in the target database table. This allows you to capture changes in the target replication log table.

Figure 11–1 shows an example of a cascading scheme. In this example, external data is received at one database, then replicated in a cascading series to other databases.

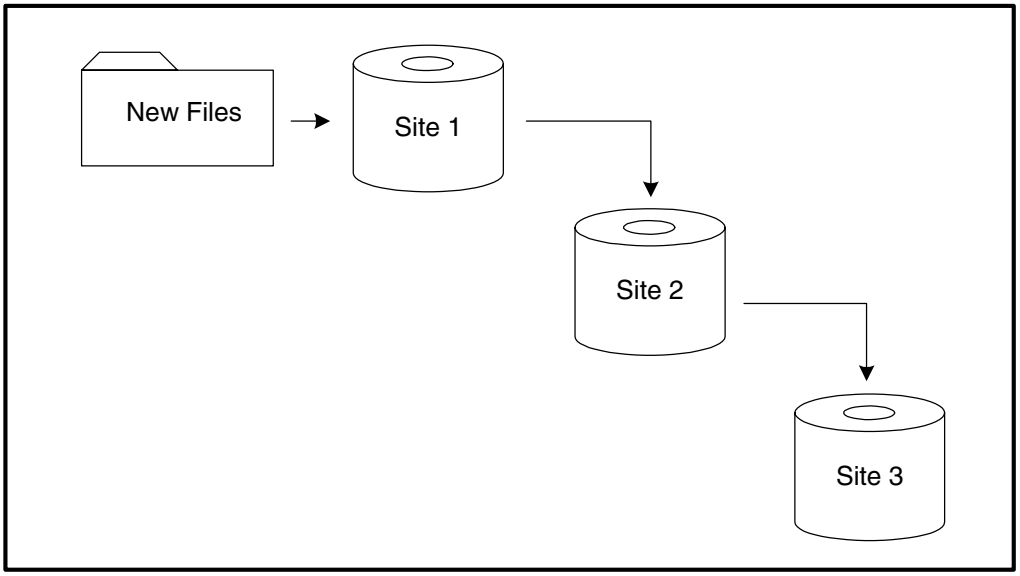


Figure 11–1: Cascading Replication

For more information, see the entry for the DISABLE TRIGGERS FOR LOAD statement in the *Progress Language Reference*.

11.4 Detecting and Capturing Data Collision

Data collision results under the following conditions:

1. User 1 updates a record in database A.
2. User 2 updates the same record in database B.
3. The record is replicated from database A to database B, or from B to A.

Data collision is a crucial issue you must address in replication, particularly in an asynchronous implementation with peer-to-peer user updating.

To maintain data integrity, data collisions must be addressed transaction by transaction. Addressing data collision involves detection and resolution. To address data collisions you might apply the following strategy:

1. Detect collisions by comparing the records in the replication change logs for the respective sites. When there is a collision, isolate all records associated with the transactions in a collision log.
2. Analyze records in the collision log. If necessary, determine the differences between records down to the field level.
3. Resolve the collisions and propagate the results.

11.4.1 Create a Collision Log

The collision log is an important concept in data collision resolution. It is crucial that you isolate the physical records involved in collisions from the original records, and from the replication change log. Users are able to change records in the database again before the collisions are resolved, thereby changing data needed for the resolution. Isolating the data in the physical records at the point they collided maintains the integrity of your collision resolution process.

To create the collision log, write a procedure that (1) compares the replication change logs between respective sites to determine changes to identical records, and (2) places these records in a separate table. This log must contain a sequence number identifying all records involved in a collision.

NOTE: This sequence number is important. There might be records contained in the transaction that have not collided, but still need to be included in the log. This is because replication occurs by transaction. All of the records must be replicated together, or not at all.

Table 11–3 shows a sample format for the collision log.

Table 11–3: Collision Log Table Schema

Field	Datatype	Description
CollisionID	integer	A sequence number that identifies all records involved in a collision. This transaction relates the local and remote records that collided.
TransactionID	integer	Identical to the transaction id (see Table 11–2).
KeyValue	char	Identical to the value shown in Table 11–2.
TableName	char	Identical to the value shown in Table 11–2.
Event	char	Identical to the value shown in Table 11–2.
DataRecord	RAW	RAW data field shown in Table 11–2.
CollisionDate	date	Transaction date shown in Table 11–2.
CollisionTime	char	Transaction time shown in Table 11–2.
Collided	logical	Logical value that flags whether the record actually collided with another record. This value is created when the records are created in this table.

11.4.2 Analyze the Log Records

After you isolate collided records in the collision log, you can use Progress 4GL to analyze records down to the field level. Follow these steps:

1 ♦ Find the records in the collision log that have actually collided (*Collision* field = Y).

2 ♦ Use the RAW–TRANSFER statement to move the RAW data fields to expanded buffer records in a *temp* table.

3 ♦ Use the BUFFER–COMPARE statement to compare the fields within the records.

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

Resolve Collisions and Propagate Results

After you determine the differences between records that have collided, you can resolve the collisions in any number of ways. For example:

- After determining field differences using BUFFER–COMPARE, you can use Progress 4GL to select the record to replicate based on a hierarchy of key fields, for example, time or user.
- If certain collisions need to be addressed manually by an administrator, you could write the conflicts to another table for manual resolution or reporting purposes.

The way you resolve collisions depends upon your business requirements. The Progress 4GL allows you to develop procedures that support all of your requirements.

11.4.3 Replication Process Configuration

After you determine your replication scheme, you need to develop a replication process that is both reliable and flexible enough to address your business requirements. Figure 11–2 shows an example of a replication process configuration, based on the concepts described previously in this section.

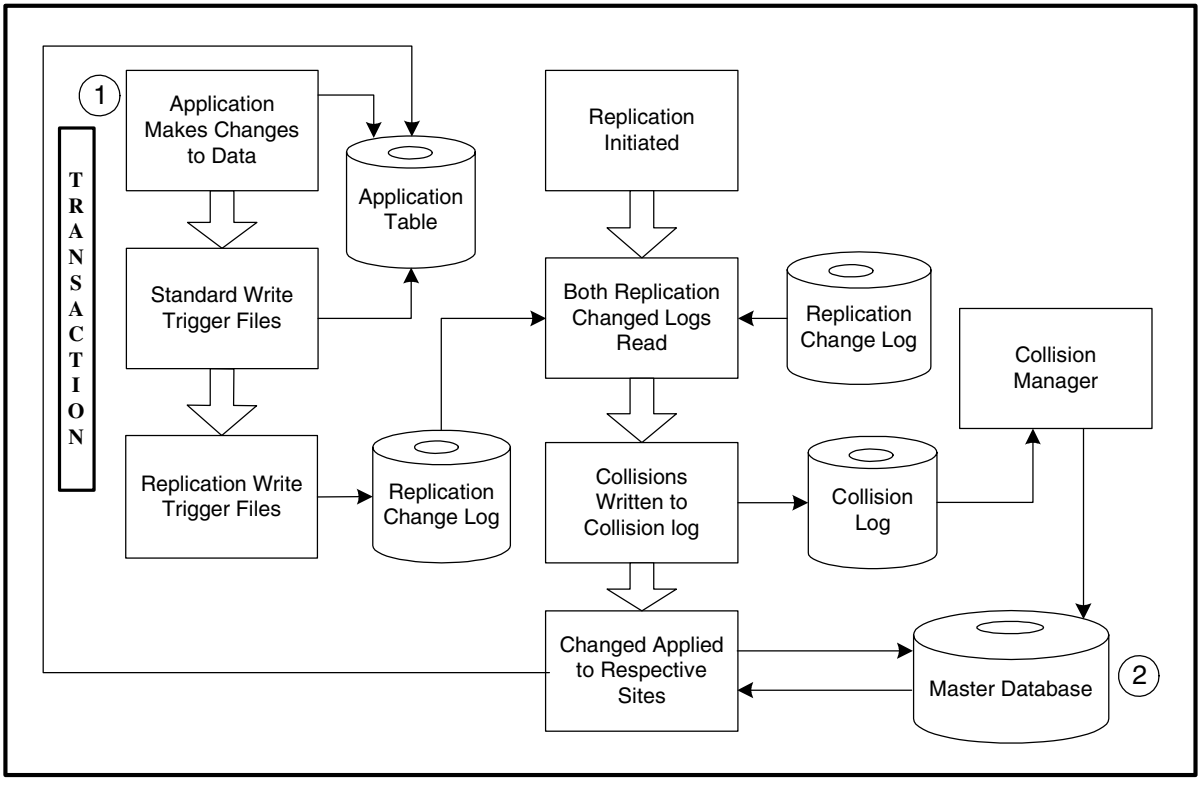


Figure 11–2: Example Replication Configuration

Figure 11–2 represents a replication process between a site database on the left (1) where transactions are shown originating, and the master database on the right (2) where the transactions are replicated and propagated to all sites. In an actual scheme, more than two sites might share the functions illustrated.

The large arrows between the boxed items represent the process flow beginning with the data transaction on the left, and continuing through the replication process on the right. The smaller arrows represent the flow of data.

Collision Manager

The *collision manager* (on the right side in Figure 11–2) represents a central area where all data collisions in the database network would be resolved. The collision manager could be located on a separate machine. The results of data collision resolution could be transferred to the master database, then propagated to all other sites.

11.4.4 Session Triggers

You define a database session trigger within a Progress procedure. The trigger persists while the procedure that defined it and any subprocedures are running. A session trigger takes effect when Progress encounters it at run time. It remains in effect until the defining procedure terminates. However, if a subprocedure contains another trigger for the same table/event or field/event pair, the trigger in the subprocedure overrides the original trigger. After the subprocedure returns, the original trigger goes back into effect.

NOTE: There are no session triggers related to database replication. In other words, Progress does not support:

```
ON REPLICATION-WRITE of Customer DO:
  /* some action */
  END.

/* Progress does not support this */
```

A session trigger has access to the defining procedure's context, regardless of which procedure initiates the event that causes it to execute. The session trigger can refer to the defining procedure's variables and other local objects.

The following is the syntax for session database triggers.

SYNTAX

```
ON event OF object [ reference-clause ]
  [ OVERRIDE ] { trigger-block | REVERT }
```

event

The event parameter is a CREATE, WRITE, DELETE, FIND, or ASSIGN event.

object

The object for table-related events (CREATE, DELETE, FIND, or WRITE) has this syntax:

SYNTAX

```
OF table
```

The object-reference for a field-related event (ASSIGN) has this syntax:

SYNTAX

```
OF table.field
```

reference-clause

The *reference-clause* for a table related event (WRITE triggers only) has this syntax:

SYNTAX

```
[ NEW [ BUFFER ] buffer1 ] [ OLD [ BUFFER ] buffer2 ]
```

Use *reference-clause* for the WRITE trigger if you want to refer to the values in the updated record before and after the record changed. The *reference-clause* uses the same syntax and rules as the schema database WRITE trigger.

The *reference clause* for a field-related event (ASSIGN) has this syntax:

SYNTAX

```
OLD [ VALUE ] parameter
  [   COLUMN-LABEL label
    | FORMAT string
    | INITIAL constant
    | LABEL string
    | NO-UNDO
  ]
```

Session ASSIGN triggers always have access to the table's buffer; the *reference–clause* allows access to the old value of the trigger field.

OVERRIDE

The OVERRIDE option causes the session trigger to execute *instead* of the schema trigger, if the schema trigger is defined with OVERRIDE=YES. If you attempt to OVERRIDE a schema trigger defined as OVERRIDE=NO, Progress returns a runtime error.

trigger-block

A single 4GL statement or a DO block. The trigger block executes whenever the specified event occurs. The context of the block of 4GL code is similar to that of internal procedures. In fact, this code executes when the event occurs, almost as if it were an internal procedure. (See Chapter 3, "Block Properties," for information on internal procedures.)

REVERT

The REVERT option cancels the trigger for the given event/table or event/field pair and reactivates any previously defined trigger for that event/object pair defined in a parent procedure. If no previously defined trigger exists, the trigger for the event/object pair is effectively disabled.

11.4.5 FIND Session Trigger Example

The following is an example of a FIND session trigger for the customer table:

```
ON FIND OF customer DO:
  FIND salesrep OF customer.
  IF sales-rep NE userid THEN
  RETURN ERROR.
END.
```

This trigger checks the user ID of the current user and compares it to the customer's sales representative. If they are different, then the customer record is not displayed.

If you plan to use a FIND trigger for security reasons, be aware that the user can still check for the existence of records with the CAN–FIND function. This function does not fire the FIND trigger.

11.4.6 WRITE Session Trigger Example

Whenever the contents of a record are changed and the record is validated, Progress executes the WRITE trigger. After the WRITE trigger executes, Progress writes the modified record to the record buffer. Note that if the trigger is executed by a VALIDATE statement, the record is not necessarily written to the record buffer.

During trigger execution, Progress makes two record buffers available to the trigger procedure. The NEW buffer contains the modified record that is being validated. The OLD buffer contains the most recent version of the record before the latest changes were made (this will be the template record if it is a newly created record, the record from the DB if it has not been validated, of the most recently validated record if it has been validated). You can modify the contents of the NEW buffer, but the OLD buffer is read-only. You can compare the contents of these two buffers and perform actions based on the results.

The following is an example of a WRITE session trigger:

p-sestrg.p

```
ON WRITE OF customer OLD BUFFER ocust DO:
    IF customer.sales-rep <> ocust.sales-rep
    THEN DO:
       FIND salesrep OF customer NO-LOCK.
       customer.comments = customer.comments + " Salesrep changed to " +
                           salesrep.rep-name + " on " + STRING(TODAY).
    END.
END.
```

This trigger tests whether you modified the Sales–rep field. If so, it adds a comment to the customer record.

The trigger executes before the record is actually written, which means that the trigger can make modifications to the record before it is released. These modifications do **not** cause the trigger to execute again.

NOTE: If you create a record and then write it to the database without updating it (the initial field values remain the same), the WRITE trigger does not execute; only the CREATE trigger executes. If you update one or more fields before you write the record, both the CREATE and WRITE triggers execute. On Embedded SQL (ESQL), an INSERT statement always executes both the CREATE and WRITE triggers.

11.5 General Considerations

When you write triggers, there are several general considerations that you should keep in mind.

11.5.1 Metaschema Tables

Progress does not allow any database triggers on events for metaschema tables and fields (tables or fields named with an initial underscore). You can only intercept database events for an application database object.

11.5.2 User-interaction Code

Progress allows you to include any type of 4GL statement within a database trigger block, including those that involve user interaction. However, it is usually not a good idea to include any statements that call for input from the user. For example, if the user runs your procedure in batch mode, a trigger with a prompt causes the procedure to stop, waiting for user input. Also, if an application is written in ESQL, it can not receive user input.

11.5.3 FIND NEXT and FIND PREV

You cannot delete or advance any record in a buffer passed to a database trigger procedure (as with a FIND NEXT or FIND PREV statement) within the trigger procedure.

11.5.4 Triggers Execute Other Triggers

An action within one trigger procedure may execute another trigger procedure. For example, if a trigger procedure assigns a value to a field and you defined an ASSIGN trigger for that field, that ASSIGN trigger executes. You must carefully design your triggers to avoid conflicts. For example, trigger A could change data, which could cause trigger B to execute. Trigger B could change the same data, a change you did not anticipate or want.

11.5.5 Triggers Can Start Transactions

By their nature, CREATE, DELETE, WRITE, and ASSIGN triggers execute from within a transaction. (The triggering event itself must occur within a transaction.) This means that while you can start a subtransaction within a trigger procedure for these events, you cannot start a new transaction. However, a FIND may or may not be executed within a transaction. Therefore, you should not assume that you can start a subtransaction within a FIND trigger; it might turn out to be a complete transaction.

11.5.6 Default Error Phrase

For all blocks in a database trigger, the default ON ERROR phrase is ON ERROR UNDO, RETURN ERROR; this is in contrast to the usual Progress default: ON ERROR UNDO, RETRY. As with any default, you can override this with an explicit ON ERROR phrase.

11.5.7 RETURN ERROR

If you use the NO–ERROR qualifier on a statement that executes a trigger, and the trigger returns an error (with RETURN ERROR), the ERROR–STATUS system handle is set to TRUE. You can check the value of this system handle after the statement executes:

```
DELETE customer NO-ERROR.
IF ERROR-STATUS:ERROR THEN DO:

/* Add logic to perform when an error occurs */
```

11.5.8 Where Triggers Execute

Database triggers (including replication-related triggers) execute in the application logic, which consists of Progress AppServers and local 4GL procedures. For more information on Progress AppServers, see *Building Distributed Applications Using the Progress AppServer*.

11.5.9 Storing Trigger Procedures in Libraries

You can store precompiled schema trigger procedures in a procedure library. When you connect to the database, you can use the Triggers (–trig) startup parameter to specify the procedure library that contains them. If you do not use this option, Progress searches the PROPATH for the trigger procedure when the associated event occurs. Note that you can also use the –trig parameter to specify the operating system directory holding your trigger procedures. For more information on the –trig startup parameter, see the *Progress Startup Command and Parameter Reference*.

11.5.10 ESQL Access to Database Objects

If you access a Progress database via ESQL, database schema triggers will execute appropriately: the DELETE statement executes the DELETE trigger; the CREATE statement executes the CREATE trigger, etc. Note that ESQL applications executing 4GL database schema triggers cannot compile .p code on the fly—all database triggers must be precompiled.

If a database trigger is executed by an ESQL application, any user interface operations within that trigger (such as UPDATE, PROMPT–FOR, etc.) will cause the ESQL request to fail.

11.5.11 Disabling Triggers for Dump/Load

When you dump and load database records, you may want to disable the schema triggers of the involved databases. If you do not disable them, when you dump records from a database, Progress executes any of the database's associated FIND schema triggers. Also, when you load records into a database, Progress executes any of the database's associated CREATE, WRITE, and ASSIGN schema triggers.

Executing schema triggers can interfere with the dump/load operation and can increase the time needed for the operation. For more information on how to disable schema triggers, see the *Progress Database Administration Guide and Reference*.

NOTE: If your site implements database replication, and the replication scheme involves cascading replication, you might have to use ALLOW–REPLICATION option of the DISABLE TRIGGERS FOR LOAD statement. This option tells Progress not to disable replication-related triggers while it loads data from the replication log to the database.

11.5.12 SQL Considerations

Because of inherent differences between the Progress 4GL and SQL, triggers may not execute in exactly the same way. For more information on how SQL and database triggers interact, see the *Progress SQL-89 Guide and Reference*.

12

Transactions

A transaction is a unit of work that is either completed as a unit or undone as a unit. Proper transaction processing is critical to maintaining the integrity of your databases.

This chapter covers the following topics:

- Transaction basics
- Using transactions

For information on Progress blocks, see Chapter 3, “Block Properties.” For information on UNDO processing, see Chapter 5, “Condition Handling and Messages.”

12.1 Introduction

Imagine this situation: you are entering new customer records into your database. You entered 98 records and are working on entering the 99th customer record and your machine goes down. Are the first 98 records you entered lost? No, Progress does just what you want in this situation:

- Keeps the first 98 records in the database
- Discards the partial 99th record

This is just one simple scenario. Suppose the procedure was updating multiple tables. You want to make sure that Progress saves any completed changes and discards partial changes **in all tables**.

System failures are just one kind of error. There are other kinds of errors that can occur while a procedure is running. Regardless of the kind of error you are dealing with, data integrity is all important. *Data integrity* means that Progress stores completed data and does not store incomplete data in the database. Progress uses transactions to automatically handle this processing.

12.2 Transactions Defined

A *transaction* is a set of changes to the database, which the system either completes or discards, leaving no modification to the database. The terms *physical transaction* and *commit unit* refer to the same concept as the Progress transaction. For example, in the above scenario where you are adding customer records, each customer record you add is a transaction.

p-trans.p

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

Each iteration of the REPEAT block is a transaction. The transaction is undone (or backed out) if:

- The system goes down (or crashes).
- The user presses **STOP** (**CTRL–BREAK** on Windows; usually **CTRL–C** on UNIX).

In either of these cases, Progress undoes all work it performed since the start of the transaction, as shown in Figure 12–1.

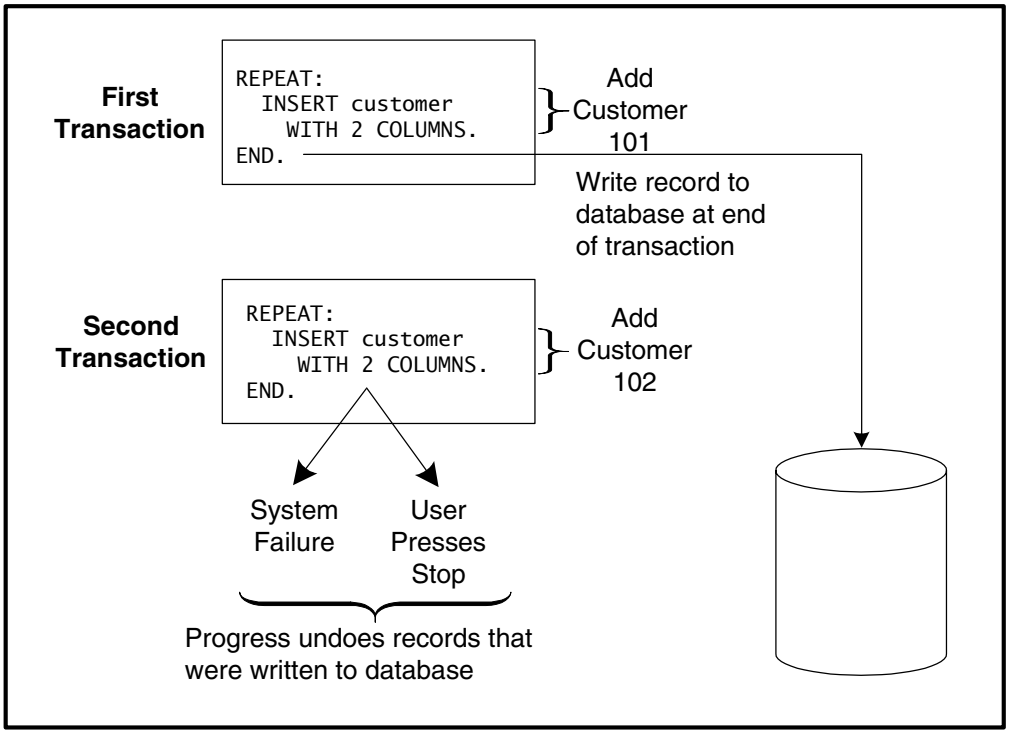


Figure 12–1: Transaction Undo Processing

So far, you have seen how a transaction can be useful in a situation that involves only a single table. Transactions take on additional importance when you make database changes in multiple tables or records.

12.3 All-or-nothing Processing

Suppose a customer calls to change an order for four sweat bands to nine sweat bands. This means you must make two changes to your database:

- First, you must look at the customer's order and change the quantity field in the appropriate order–line record.
- Second, you must change the value of the allocated field in the record for that item in the item table.

What if you changed the quantity field in the order–line record and are in the midst of changing the allocated field in the item record, when the machine goes down? You want to restore the records to their original state. That is, you want to be sure that Progress changes both records or neither.

Figure 12–2 shows this scenario for a database similar to sports (same tables, different data).

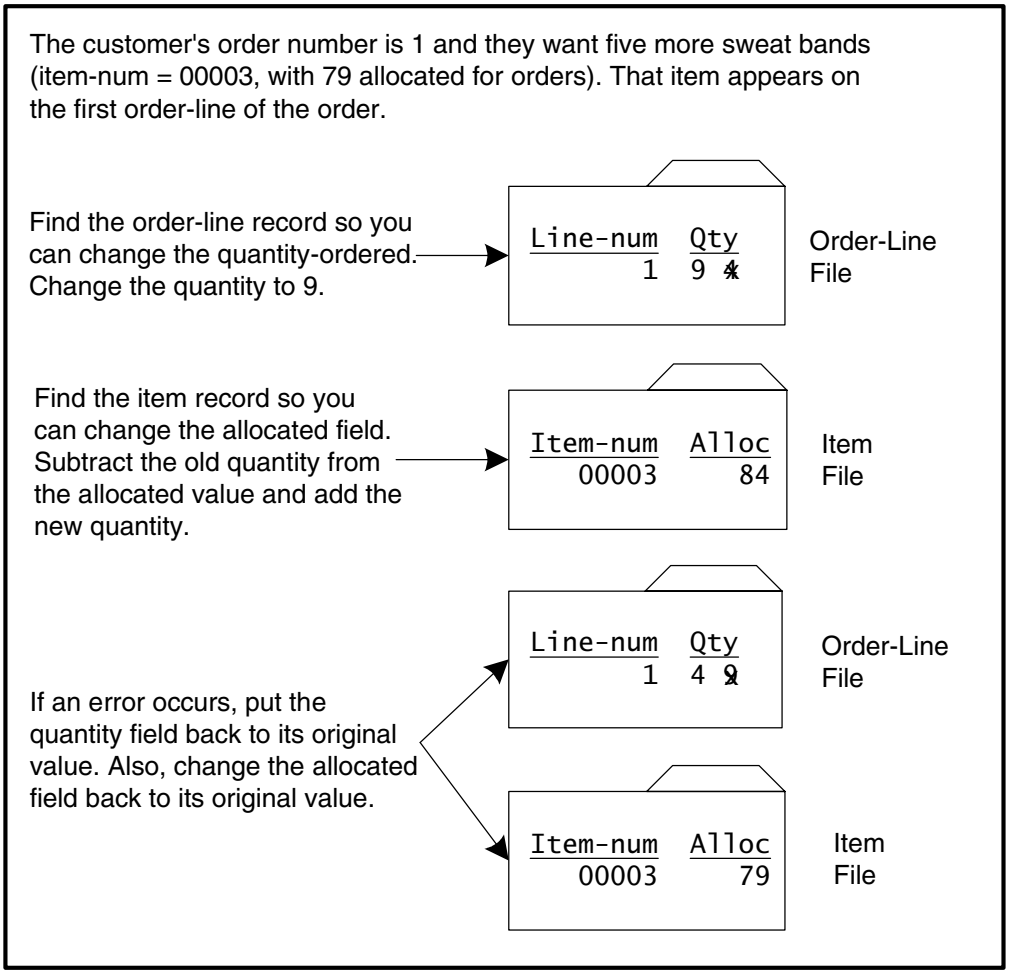


Figure 12–2: Transaction Involving Two Tables

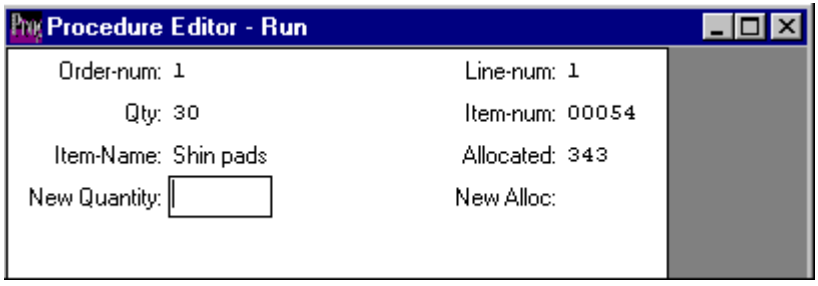
The following procedure, p-txn.p, executes the scenario shown in Figure 12–2, using the sports database.

p-txn1.p

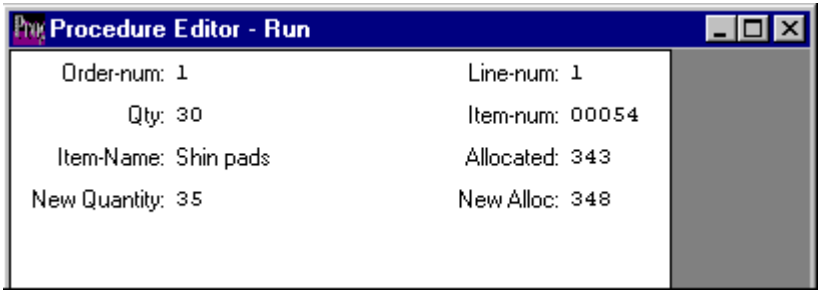
```
DEFINE VARIABLE newqty LIKE qty LABEL "New Quantity".
REPEAT WITH 2 COLUMNS:
    PROMPT-FOR order-line.order-num line-num.
    FIND order-line USING order-num AND line-num.
    FIND item OF order-line.
    DISPLAY order-line.qty item.item-num
            item.item-name item.allocated.
    SET newqty.
    item.allocated = item.allocated - order-line.qty + newqty.
    order-line.qty = newqty.
    DISPLAY item.allocated @ new-alloc
            LIKE item.allocated LABEL "New Alloc" SKIP(1).
    PAUSE.
END.
```

Progress starts a transaction at the beginning of each iteration of the REPEAT block and ends that transaction at the end of each iteration of the REPEAT block.

Run p-txn.p. Type **1** as the order number and **1** as the line number and press **GO**:

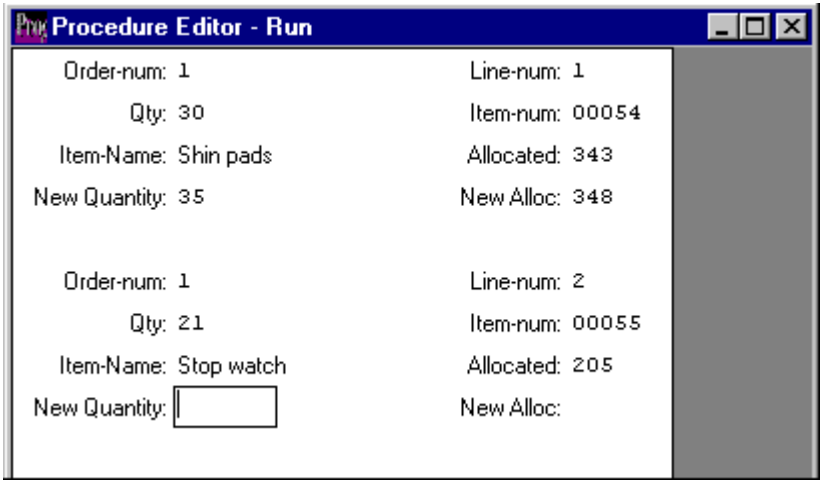


Now type **35** as the New Quantity and press **RETURN**. The procedure calculates and displays the new allocated value:

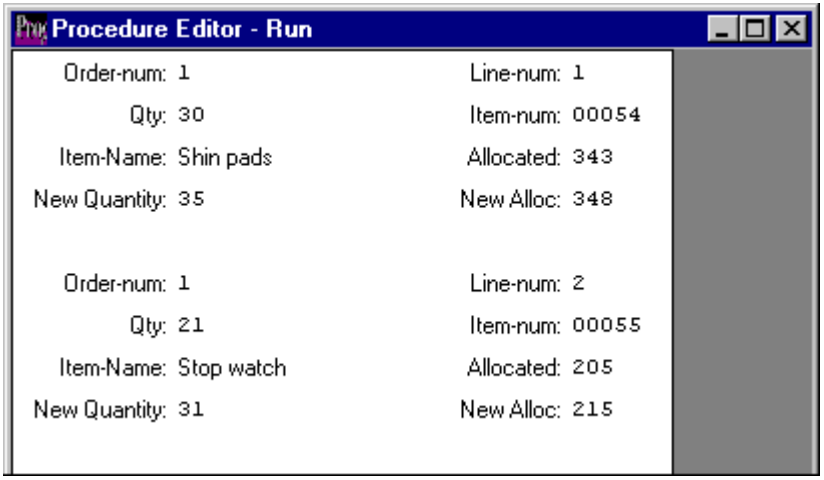


The quantity has been changed to 35 and the amount allocated changed to 187. At this point control has reached the PAUSE statement at the end of the REPEAT block:

Press **SPACEBAR**. The procedure prompts you for another order number and line number. Type **1** for the order number and **2** for the line number:



Change the number of ordered stop watches to **31**:



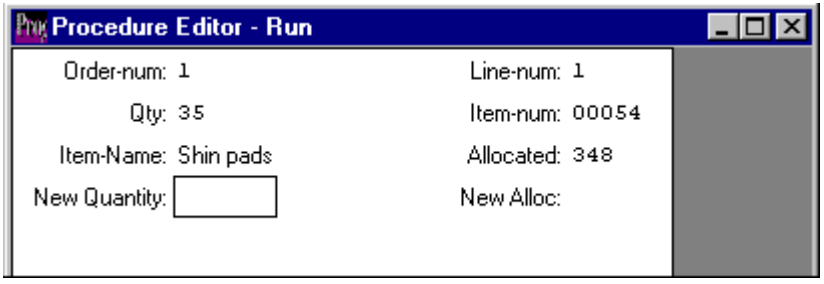
You can see that the quantity has been changed to 31 and the amount allocated has been increased to 215 (you increased the number ordered by 10). You have once again reached the PAUSE statement just before the end of the REPEAT block.

What if, for some reason, your machine goes down or you decide to press **STOP** at this point in the procedure? Progress undoes any database changes made during the current iteration of the REPEAT block. Go ahead and press **STOP**.

Now you are back in the Procedure Editor. Run the `p-txn.p` procedure again.

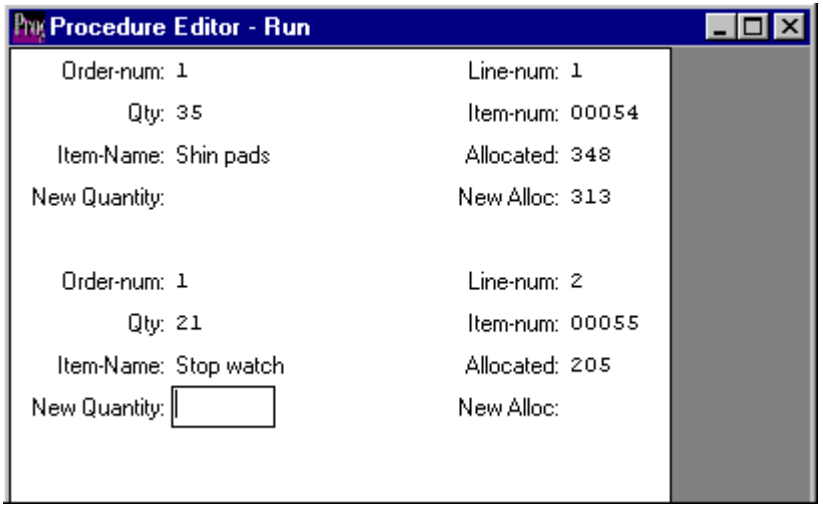
Remember that you had changed the quantity to 35 on line 1 of order 1.

Type **1** as the order number and **1** as the line number. The following screen appears:



You can see that the first change you made is safely stored in the database. Both the order–line table and the item table reflect the new values of 35 and 348. Press **GO** to continue on until you reach the PAUSE statement just before the end of the REPEAT block. Then press **SPACEBAR** to continue on to the next iteration.

Type **1** for the order number and **2** for the line number.



Notice that for line 2 of order number 1, both the quantity field in the order-line record and the allocated field in the item record have been returned to their original values of 21 and 0, respectively.

Remember that each iteration of the REPEAT block started a new transaction. Once the first iteration (the shin pads) completed successfully, Progress ended the transaction and committed the changed data to the database.

Progress started another transaction on the next iteration of the REPEAT block. When you pressed **STOP** during that iteration, Progress backed out the transaction, undoing all database changes made since the start of the transaction.

12.4 Understanding Where Transactions Begin and End

How does Progress know where to start the transaction and how much work to undo or back out? The following *transaction blocks* start a transaction if one is not already active:

- Any block that uses the TRANSACTION keyword on the block statement (DO, FOR EACH, or REPEAT).
- A procedure block, trigger block, and each iteration of a DO ON ERROR, FOR EACH, or REPEAT block that directly updates the database or directly reads records with EXCLUSIVE–LOCK. You use EXCLUSIVE–LOCK to read records in multi-user applications. See Chapter 13, “Locks” for more information on locking.

Directly updating the database means that the block contains at least one statement that can change the database. CREATE, DELETE, and UPDATE are examples of such statements.

If a block contains FIND or FOR EACH statements that specify EXCLUSIVE–LOCK, and at least one of the FIND or FOR EACH statements is not embedded within inner transaction blocks, then the block is directly reading records with EXCLUSIVE–LOCK.

Note that DO blocks do not automatically have the transaction property. Also, if the procedure or transaction you are looking at is run by another procedure, you must check the calling procedure to determine whether it starts a transaction before the RUN statement.

Once a transaction is started, all database changes are part of that transaction, until it ends. Each user of the database can have just one active transaction at a time:

p-txn2.p

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

This procedure has two blocks: the procedure block and the REPEAT block. The procedure block has no statements directly in it that are not contained within the REPEAT block. The REPEAT block contains an INSERT statement that lets you add order records to the database. Because the REPEAT block is the outermost block that contains direct updates to the database, it is the transaction block.

At the start of an iteration of the REPEAT block, Progress starts a transaction. If any errors occur before the END statement, Progress backs out any work done during that transaction.

Note that data-handling statements that cause Progress to automatically start a transaction for a regular table will not cause Progress to automatically start a transaction for a work table or temporary table.

Consider another example:

p-txn3.p

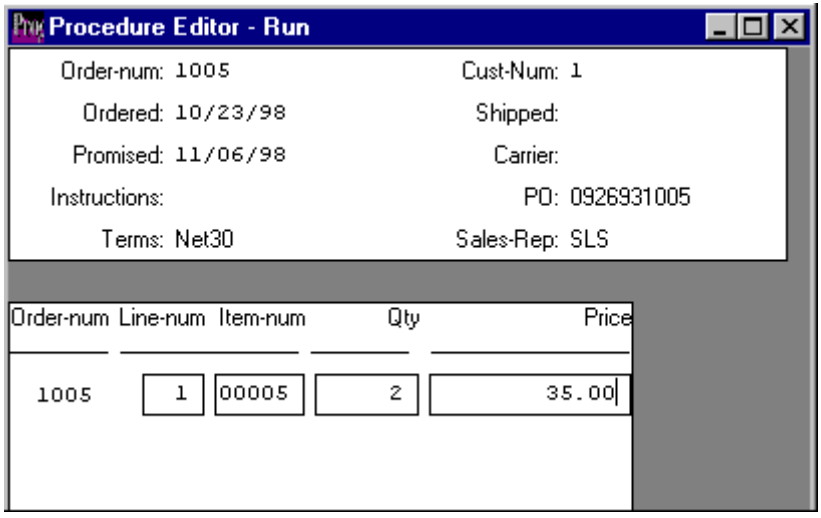
```
REPEAT:
    INSERT order WITH 2 COLUMNS.
    FIND customer OF order.
    REPEAT:
       CREATE order-line.
       order-line.order-num = order.order-num.
       DISPLAY order-line.order-num.
       UPDATE line-num order-line.item-num qty price.
    END.
END.

FOR EACH salesrep:
    DISPLAY sales-rep rep-name.
    UPDATE region.
END.
```

This procedure has four blocks:

- **Procedure block** — There are no statements in this block, so Progress does not start a transaction at the start of the procedure.
- **Outer REPEAT block** — The outermost block that directly updates the database (INSERT order WITH 2 COLUMNS). Therefore, it is a transaction block. On each iteration of this block, Progress starts a transaction. If an error occurs before the end of the block, all work done in that iteration is undone.
- **Inner REPEAT block** — Directly updates the database but it is not the outermost block to do so. Therefore, it is not a transaction block. It is, however, a subtransaction block. Subtransactions are discussed later in this chapter.

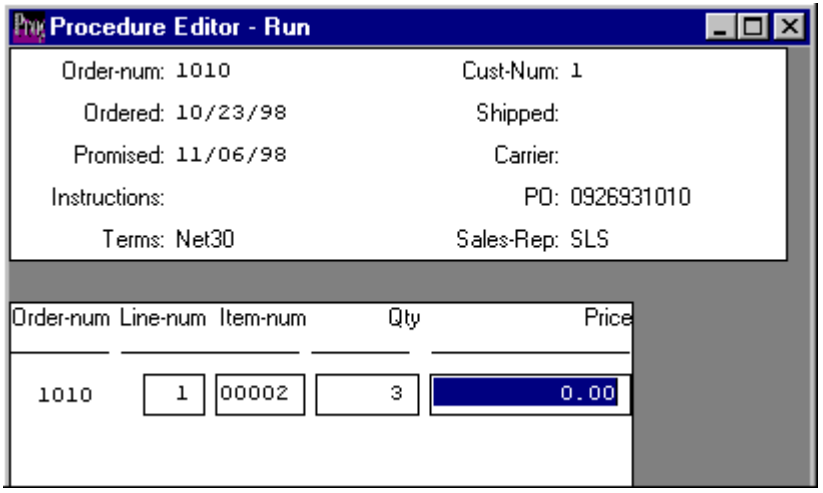
- **FOR EACH block —** An outermost block that directly updates the database (UPDATE region). Therefore, it is a transaction block. On each iteration of this block, Progress starts a transaction. If an error occurs before the end of the block, all work done in that iteration is undone.

Go ahead and run this procedure. Enter the data shown in the following display:



After you enter the order information, press **GO**, then enter the data for line number 1 of the order and press **GO**. The procedure prompts you for the next order–line. Press **END–ERROR** to tell the procedure that you are finished entering order–lines for order 1005. The procedure prompts you for information on the next order.

Enter the data shown on the following screen:



After you enter the quantity and are prompted for the price, press **STOP**. Progress returns you to the Procedure Editor.

Remember that either a system failure or pressing the **STOP** key causes Progress to back out the current transaction. Let's retrace our steps to determine which of the order and order–line data you entered should be in the database.

1. On the first iteration of the REPEAT block, you entered data for order number. Because Progress automatically starts a transaction for a REPEAT block if it is an outermost block containing database updates, a transaction was started at the start of that iteration.

2. In the inner REPEAT block, you entered a single order–line for order 1005. Even though this is a REPEAT block and does directly update the database, a transaction is already active. Therefore, Progress does not start a transaction for this inner REPEAT block. All of the updates done in this inner block are part of the transaction that is already active.

3. When you finished entering that first order–line and pressed `GO` to add the next order, Progress reached the end of the iteration of the transaction block and ended the transaction. Therefore, the order and order–line data you entered were written to the database. It should still be there, right? Run `p-check.p` to see if order 1005 and its order–line record exist.

p-check.p

```
PROMPT-FOR order.order-num.
FIND order USING order-num.
DISPLAY order WITH 2 COLUMNS.
FOR EACH order-line OF order:
    DISPLAY order-line.
END.
```

This procedure displays order 1005 and its single order–line, proving that the data is in the database.

The following steps trace the second iteration of the REPEAT block.

1. On the second iteration of the REPEAT block, you entered data for order 1010. Because Progress automatically starts a transaction for a REPEAT block if it is the outermost block containing database updates, a transaction was started at the start of that iteration.

2. In the inner REPEAT block, you started entering an order–line for order 32. Even though this is a REPEAT block and does directly update the database, a transaction is already active. Therefore, Progress does not start a transaction for this inner REPEAT block and you were still working within the initial transaction. In the middle of the transaction, you pressed `STOP`, which backed out the current transaction.

Run the `p-check.p` procedure again, this time supplying 1010 as the order number. Progress displays the message “order record not on file.”

This message confirms that the data for order 1010 was not stored in the database. But remember the rule about all-or-nothing processing. When working with multiple tables in a single transaction, it is important that either all the changes to all the tables are completed or none of the changes to any of the tables are completed.

When you pressed **STOP**, you were in the middle of adding an order–line record. Just to be sure that an order–line record corresponding to order 1010 was not stored in the database, run this procedure (supply 1010 as the order number and 1 as the order–line number).

p-check2.p

```
PROMPT-FOR order-line.order-num line-num WITH NO-VALIDATE.
FIND order-line USING order-num AND line-num.
DISPLAY order-line.
```

The Data Dictionary definition of the order–line field specifies a validation of CAN–FIND(order OF order–line). This means that when you supply an order–number, Progress checks to be sure that the order exists. Since you already know the order does not exist (we checked by running the `p-check.p` procedure), you can use the NO–VALIDATE Frame phrase option in the PROMPT–FOR statement. This option tells Progress to ignore any validation criteria defined in the Dictionary.

When you run the `p-check2.p` procedure and supply 1010 as the order number and 1 as the order–line number, Progress displays the message “order–line record not on file”.

Not only did Progress undo the order information you had entered for order 1010 but it also undid the partial order–line information you had entered.

If you did not press **STOP** but your system crashed, you would have seen exactly the same behavior.

12.5 How Much to Undo

You have seen how, when the system crashes or when you press **STOP**, Progress undoes the current transaction. Suppose you want to undo a transaction under program control or want to undo a smaller amount of work than that done since the beginning of the transaction.

Take a look at a different version of the `p-txn3.p` procedure. The outer REPEAT block is the transaction block:

p-txn3a.p

```
REPEAT:
    INSERT order WITH 2 COLUMNS.
    FIND customer OF order.
    REPEAT:
      CREATE order-line.
      order-line.order-num = order.order-num.
      DISPLAY order-line.order-num.
      UPDATE line-num order-line.item-num qty price.
    END.
END.
```

Suppose that you wanted to restrict the maximum amount of an order to $500. In the event that the value of an order exceeded that amount, you want to undo the current order–line entry and display a message to the user. For example, look at `p-txn5.p`:

p-txn5.p

```
DEFINE VARIABLE extension LIKE order-line.price.
DEFINE VARIABLE tot-order LIKE order-line.price.
REPEAT:
    INSERT order WITH 2 COLUMNS.
    tot-order = 0.
    o-l-block:
    REPEAT:
      CREATE order-line.
      order-line.order-num = order.order-num.
      DISPLAY order-line.order-num.
      UPDATE line-num order-line.item-num qty price.
      extension = qty * price.
      tot-order = tot-order + extension.
      IF tot-order > 500 THEN DO:
        MESSAGE "Order has exceeded $500".
        MESSAGE "No more order-lines allowed".
        UNDO o-l-block.
      END.
   END.
END.
```

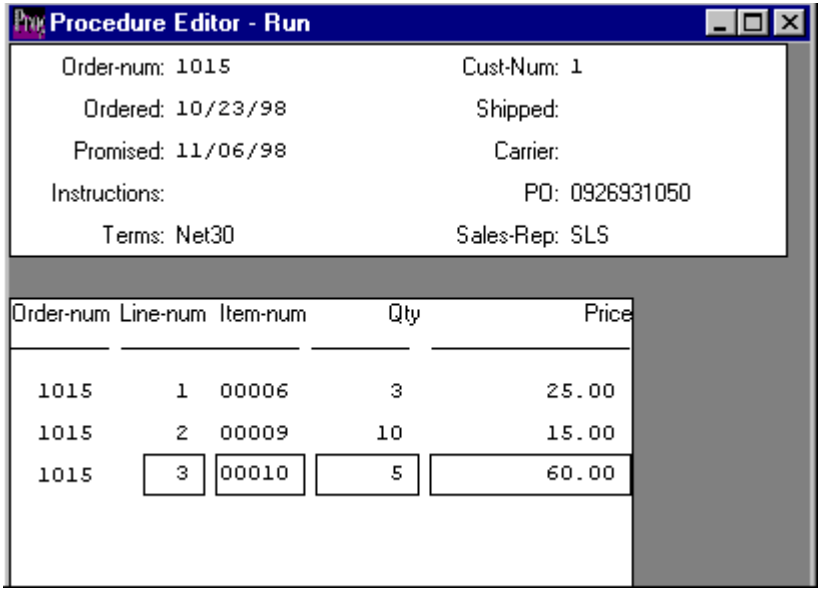
The outer REPEAT block is still the transaction block; it is the outermost block that contains direct updates to the database. However, in this example, Progress starts a subtransaction when it reaches the inner REPEAT block.

If an error occurs during a subtransaction, all the work done since the beginning of the subtransaction is undone. Subtransactions can be nested within other subtransactions.

A subtransaction is started when a transaction is already active and Progress encounters a *subtransaction block*. The following are subtransaction blocks:

- A procedure block that is run from a transaction block in another procedure.
- Each iteration of a FOR EACH block nested within a transaction block.
- Each iteration of a REPEAT block nested within a transaction block.
- Each iteration of a DO TRANSACTION, DO ON ERROR, or DO ON ENDKEY block. (These blocks are discussed later in this chapter.)

Now go ahead and run the `p-txn5.p` procedure. Enter the data shown in the following display:



After you enter the second order–line, the amount of the order is $225. When you try to enter the third order–line, the following happens:

- The MESSAGE statements display messages telling you that the limit has been exceeded.
- The UNDO statement undoes the o–l–block. The UNDO statement can undo only those blocks that are either transactions or subtransactions within the current transaction. In this case, the o–l–block is a subtransaction.

Table 12–1 shows when transactions and subtransactions are started.

Table 12–1: Starting Transactions and Subtransactions

Type of Block	Transaction Not Active	Transaction Active
DO transaction FOR EACH transaction REPEAT transaction Any DO ON ENDKEY, DO ON ERROR, FOR EACH, REPEAT, or procedure block that directly contains statements that modify database fields or records or that read records using an EXCLUSIVE–LOCK.	Starts a transaction	Starts a subtransaction
Any FOR EACH, REPEAT, or procedure block that does not directly contain statements that either modify the database or read records using an EXCLUSIVE–LOCK.	Does not start a subtransaction or a transaction	Starts a subtransaction

Note that data handling statements that cause Progress to automatically start a transaction for a database table do not cause Progress to automatically start a transaction for a work table or temporary table.

12.6 Controlling Where Transactions Begin and End

You may find that, because of the kind of work a procedure does, you want to start or end transactions in locations other than those Progress automatically chooses. You know that Progress automatically starts a transaction for each iteration of four kinds of blocks:

- FOR EACH blocks that directly update the database.
- REPEAT blocks that directly update the database.
- Procedure blocks that directly update the database.
- DO ON ERROR or DO ON ENDKEY blocks that contain statements that update the database.

A transaction ends at the end of the transaction block or when the transaction is backed out for any reason.

Sometimes you want a transaction to be larger or smaller depending on the amount of work you want undone in the event of an error. You can explicitly tell Progress to start a transaction by using the TRANSACTION option with a DO, FOR EACH, or REPEAT block header:

- DO TRANSACTION:
- FOR EACH TRANSACTION:
- REPEAT TRANSACTION:

When you explicitly tell Progress to start a transaction, it starts a transaction on each iteration of the block regardless of whether the block contains statements that directly update the database. Of course, Progress does not start a transaction if one is already active.

You can also give a DO block the TRANSACTION property by using the ON ERROR phrase (DO ON ERROR:) if it also directly contains statements that update the database. DO ON ERROR is discussed later in this chapter.

12.6.1 Making Transactions Larger

In the p-txn3a.p procedure, the outermost REPEAT block is the transaction block. That means that when Progress undoes the transaction, it undoes any work performed in the current iteration of the outer REPEAT block. Only the current order is undone, while all other orders are safely stored in the database:

p-txn3a.p

```
REPEAT:
    INSERT order WITH 2 COLUMNS.
    FIND customer OF order.
    REPEAT:
        CREATE order-line.
        order-line.order-num = order.order-num.
        DISPLAY order-line.order-num.
        UPDATE line-num order-line.item-num qty price.
    END.
END.
```

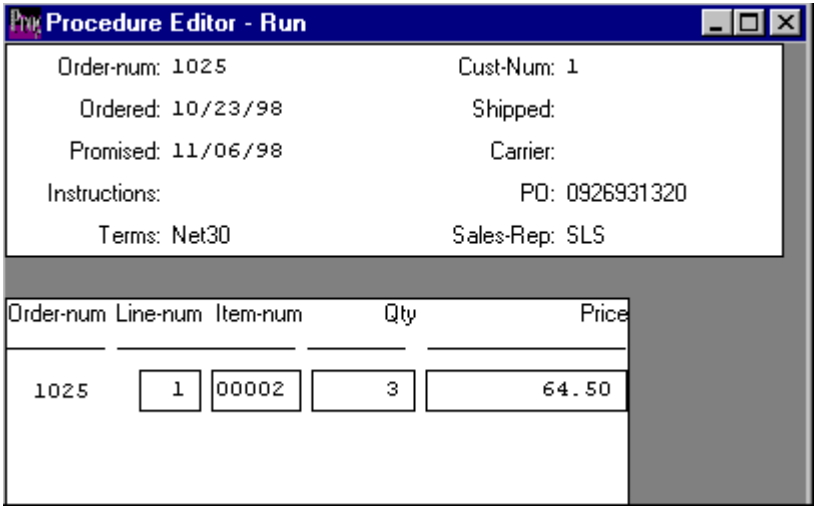
Suppose you wanted, in the event of a system crash, to undo all the orders entered since the start of the procedure. That is, you want to make the transaction block not just one iteration of the outer REPEAT block but rather you want the transaction block to encompass all iterations of the outer REPEAT block. To do this, you use a DO block together with the TRANSACTION option, as in p-txn4.p:

p-txn4.p

```
DO TRANSACTION:
    REPEAT:
        INSERT order WITH 2 COLUMNS.
        FIND customer OF order.
        REPEAT:
            CREATE order-line.
            order-line.order-num = order.order-num.
            DISPLAY order-line.order-num.
            UPDATE line-num order-line.item-num qty price.
        END.
    END.
END.
```

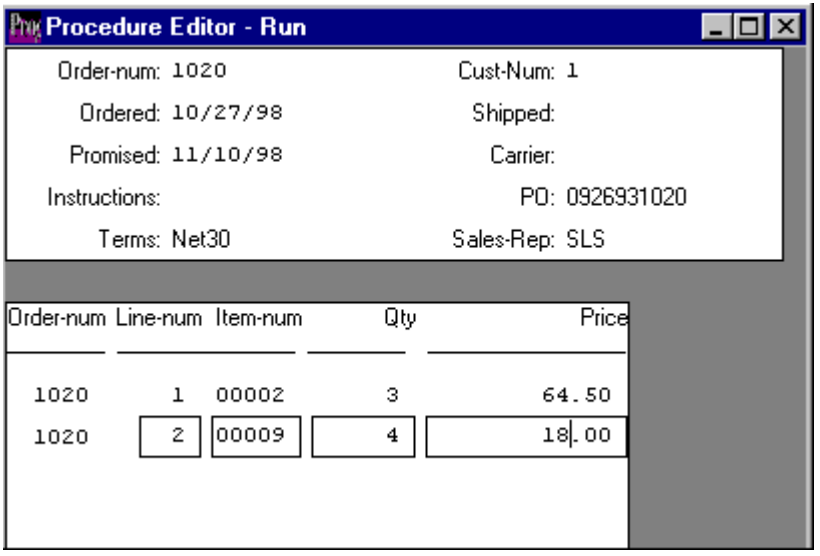
The TRANSACTION option on the DO block overrides the default transaction placement. Even though an outermost REPEAT, FOR EACH, or procedure block that contains direct database updates is normally made the transaction block, the TRANSACTION option overrides that default.

Now, Progress starts a transaction at the start of the DO block. That transaction does not end until the end of the DO block or until the transaction is backed out for any reason. Go ahead and run this procedure, entering the following data:



After you enter all the information for the first order–line, press **GO**. The procedure prompts you for more order–line information. Press **END–ERROR** to indicate that there are no more order–lines to enter for this order. The procedure prompts you for the next order.

Enter the following data:



Before entering the price for order–line 2, press **STOP**. Progress backs out the transaction, returning you to the procedure editor.

Because the transaction encompasses all iterations of the outer REPEAT block, you would expect that all work done in those iterations would have been backed out. Now, make sure that happened. Run `p-check.p` to check on the orders you entered:

p-check.p

```
PROMPT-FOR order.order-num.
FIND order USING order-num.
DISPLAY order WITH 2 COLUMNS.
FOR EACH order-line OF order:
    DISPLAY order-line.
END.
```

Progress displays the message "order record not on file" for both order number 1020 and 1025. This proves that all the work you did in the procedure has been backed out.

Note that any activity that occurs within a DO block that has no TRANSACTION or ON ERROR phrase in the block header is encompassed in an enclosing transaction or subtransaction and does not result in a transaction or subtransaction being started.

12.6.2 Making Transactions Smaller

Now imagine the reverse situation to the one in the last section. That is, in the event of a system crash or pressing **STOP**, you want to undo only the current order–line. Again, you use a DO block with the TRANSACTION option to tell Progress where to start transactions:

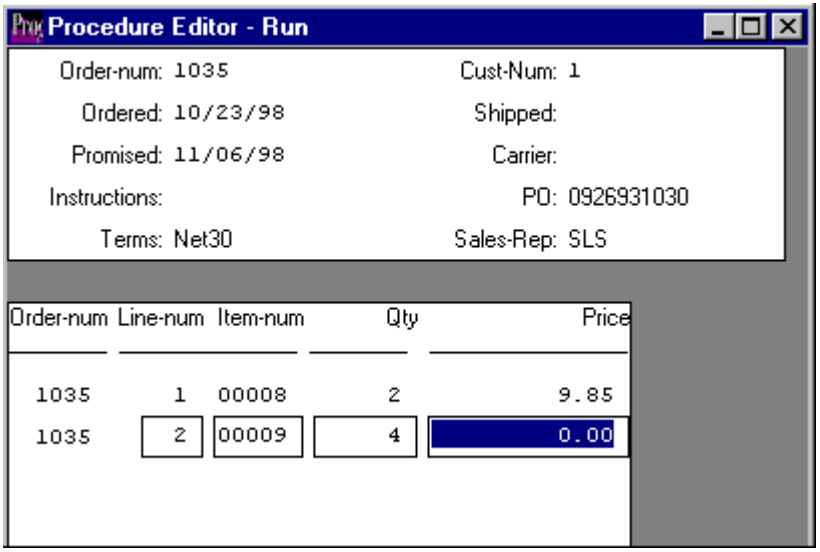
p-txn10.p

```
REPEAT:
    DO TRANSACTION:
        INSERT order WITH 2 COLUMNS.
        FIND customer OF order.
    END.
    REPEAT TRANSACTION:
        CREATE order-line.
        order-line.order-num = order.order-num.
        DISPLAY order-line.order-num.
        UPDATE line-num order-line.item-num qty price.
        FIND item OF order-line.
    END.
END.
```

In p-txn10.p, Progress starts a transaction for each order and also for each order–line you enter. There are two outermost blocks that contain direct updates to the database:

- The DO TRANSACTION block is an outermost block that contains direct updates to the database (INSERT order). The DO TRANSACTION statement and its corresponding END statement make the insertion of the order record a transaction.
- The inner REPEAT block is an outermost block that contains direct updates to the database.

The outer REPEAT block is not the transaction block because it does not contain any direct updates to the database. Run p-txn10.p, entering the following data:



Press **STOP** while entering the second order–line. Progress undoes the current transaction block, which means that only the work done on the second order–line is undone. Order 1030 and the first order–line should be in the database. Run p-check.p to find out:

p-check.p

```
PROMPT-FOR order.order-num.
FIND order USING order-num.
DISPLAY order WITH 2 COLUMNS.
FOR EACH order-line OF order:
  DISPLAY order-line.
END.
```

Note that any activity that occurs within a DO block that has no TRANSACTION or ON ERROR phrase in the block header is encompassed in an enclosing transaction or subtransaction and does not result in a transaction or subtransaction being started.

12.7 Transactions and Triggers

If a transaction begins with a trigger, it cannot extend beyond the end of the trigger. If the transaction is scoped to a block within the trigger, the normal transaction scoping rules apply. If the trigger is scoped to the trigger block itself, the transaction ends when the trigger ends.

In the procedure `p-txn12.p`, a transaction occurs within the CHOOSE trigger for the upd–cust button and within the CHOOSE trigger for the del–cust button:

p-txn12.p *(1 of 2)*

```
DEFINE BUTTON upd-cust LABEL "Update Customer".
DEFINE BUTTON del-cust LABEL "Delete Customer".
DEFINE BUTTON exit-app LABEL "Exit".

DEFINE VARIABLE curr-cust AS ROWID.
DEFINE VARIABLE changes-made AS LOGICAL.
DEFINE QUERY seq-cust FOR customer.
DEFINE BROWSE brow-cust
   QUERY seq-cust
   DISPLAY Cust-num Name
WITH 10 DOWN.

FORM
   upd-cust del-cust exit-app SKIP(1)
   brow-cust
   WITH FRAME main-frame.

OPEN QUERY seq-cust FOR EACH customer.

ON VALUE-CHANGED OF brow-cust
   DO:
      curr-cust = ROWID(customer).
   END.
```

p-txn12.p *(2 of 2)*

```
ON CHOOSE OF upd-cust
   DO: /* TRANSACTION */
      FIND customer WHERE ROWID(customer) = curr-cust EXCLUSIVE-LOCK.
      UPDATE customer WITH FRAME cust-frame VIEW-AS DIALOG-BOX
         TITLE "Customer Update".
      changes-made = brow-cust:REFRESH().
      RELEASE customer.
   END.

ON CHOOSE OF del-cust
   DO:
       MESSAGE "Delete" customer.name + "?" VIEW-AS ALERT-BOX
           QUESTION BUTTONS YES-NO UPDATE kill-it AS LOGICAL.

       IF kill-it
       THEN DO TRANSACTION:
          FIND customer WHERE ROWID(customer) = curr-cust EXCLUSIVE-LOCK.

          DELETE customer.
          changes-made = brow-cust:REFRESH().
       END.
   END.

ENABLE ALL WITH FRAME main-frame.
PAUSE 0 BEFORE-HIDE.

WAIT-FOR CHOOSE OF exit-app OR WINDOW-CLOSE OF DEFAULT-WINDOW.
```

The transaction for updating a customer is scoped to the trigger block for CHOOSE of upd–cust. The transaction begins and ends when the trigger begins and ends. The transaction for deleting a customer is scoped to a DO block within the CHOOSE trigger for del–cust. The transaction begins and ends when the DO block begins and ends.

If a transaction is started in the main code and is active when a trigger is invoked, the trigger becomes part of the transaction. You cannot end the transaction within a trigger. For example, the procedure `p-txn13.p` begins a transaction for each pass through the DO WHILE loop (trans–loop) in the main code. All updates and deletions performed within the loop are part of a single transaction. When you choose either the tran–com or tran–undo button, the loop either iterates or is undone. At that point, all changes made during that iteration are either committed or backed out.

p-txn13.p *(1 of 4)*

```
DEFINE BUTTON upd-cust LABEL "Update Customer".
DEFINE BUTTON del-cust LABEL "Delete Customer".
DEFINE BUTTON exit-app LABEL "Exit".
DEFINE BUTTON tran-undo LABEL "Undo Transaction".
DEFINE BUTTON tran-com LABEL "Commit Transaction".

DEFINE VARIABLE changes-made AS LOGICAL.
DEFINE VARIABLE curr-cust AS ROWID.
DEFINE VARIABLE exit-chosen AS LOGICAL.
DEFINE VARIABLE undo-chosen AS LOGICAL.

DEFINE QUERY seq-cust FOR Customer.
DEFINE BROWSE brow-cust QUERY seq-cust
   DISPLAY Cust-num Name WITH 10 DOWN.

FORM
   upd-cust del-cust exit-app SKIP(1)
   brow-cust SKIP(1)
   tran-com tran-undo
   WITH FRAME main-frame.

OPEN QUERY seq-cust FOR EACH Customer.

ON VALUE-CHANGED OF brow-cust
   DO:
      curr-cust = ROWID(Customer).
   END.
```

p-txn13.p *(2 of 4)*

```
ON CHOOSE OF upd-cust
   DO:
      FIND Customer WHERE ROWID(Customer) = curr-cust EXCLUSIVE-LOCK.
      UPDATE Customer WITH FRAME cust-frame VIEW-AS DIALOG-BOX
         TITLE "Customer Update".
      changes-made = brow-cust:REFRESH().
   END.
ON CHOOSE OF del-cust
   DO:
       MESSAGE "Delete" Customer.name + "?" VIEW-AS ALERT-BOX
           QUESTION BUTTONS OK-CANCEL UPDATE kill-it AS LOGICAL.

       IF kill-it
       THEN DO:
          FIND Customer WHERE ROWID(Customer) = curr-cust EXCLUSIVE-LOCK.
          DELETE Customer.
          changes-made = brow-cust:REFRESH().
       END.
   END.
```

p-txn13.p *(3 of 4)*

```
ON CHOOSE OF tran-undo
  DO:
     undo-chosen = TRUE.
  END.

ON CHOOSE OF exit-app
  DO:
     DEFINE BUTTON exit-commit LABEL "Commit" AUTO-GO.
     DEFINE BUTTON exit-undo LABEL "Undo" AUTO-GO.
     DEFINE BUTTON exit-cancel LABEL "Cancel" AUTO-ENDKEY.

     FORM
        "Do you want to commit or undo your changes?" SKIP
         exit-commit exit-undo exit-cancel
         WITH FRAME exit-frame VIEW-AS DIALOG-BOX TITLE "Exit".

     ON CHOOSE OF exit-undo
        DO:
           undo-chosen = TRUE.
        END.

     exit-chosen = TRUE.

     /* If changes have been made during the current transaction,
        then ask the user to either commit or undo them (or cancel
        the Exit operation).                                      */
     IF changes-made
     THEN UPDATE exit-commit exit-undo exit-cancel WITH FRAME exit-frame.
  END.
```

p-txn13.p *(4 of 4)*

```
ENABLE ALL WITH FRAME main-frame.
PAUSE 0 BEFORE-HIDE.

exit-chosen = FALSE.

trans-loop:
DO WHILE NOT exit-chosen TRANSACTION ON ENDKEY UNDO, LEAVE
                    ON ERROR UNDO, LEAVE:

   changes-made = brow-cust:REFRESH().

   ASSIGN changes-made = FALSE
          undo-chosen = FALSE.

   WAIT-FOR CHOOSE OF tran-com, tran-undo, exit-app.

   /* If the user chose UNDO (either from the main frame
      or from the Exit dialog), then undo the current
      transaction and either start a new one or exit.    */
   IF undo-chosen
   THEN DO:
      IF exit-chosen
      THEN UNDO trans-loop, LEAVE trans-loop.
      ELSE UNDO trans-loop, RETRY trans-loop.
   END.

   /* Make sure we don't hold any locks after committing a transaction. */
   IF AVAILABLE(Customer)
   THEN RELEASE Customer.

   END.
```

When you choose the exit–app button from the main screen, the DO WHILE loop completes and the last transaction is committed.

NOTE: Generally, you do not want transactions to span triggers. Instead, you should design your application so that each transaction occurs within a single trigger. Otherwise, undoing one action might cause other unrelated actions to be undone also.

Transactions are, by nature, modal. This creates conflicts when you try to write a modeless application. You can handle this is one of two ways:

- Make the modal nature of the transactions clear. For example, `p-txn12.p` uses a dialog box for the update of the customer record. This prevents the user from performing any other actions until the update is completed.
- Hide modality from the user but enforce it carefully within your code.

The procedure p-txn14.p allows what appears to be a modeless update:

p-txn14.p *(1 of 5)*

```
DEFINE BUTTON exit-app LABEL "Exit".
DEFINE BUTTON com-cust LABEL "Save Changes".
DEFINE BUTTON rev-cust LABEL "Revert to Saved".

DEFINE VARIABLE curr-cust AS ROWID.
DEFINE VARIABLE exit-chosen AS LOGICAL.
DEFINE VARIABLE rev-chosen AS LOGICAL.
DEFINE VARIABLE temp-hand AS WIDGET-HANDLE.

DEFINE BUFFER this-cust FOR Customer.
DEFINE QUERY seq-cust FOR this-cust SCROLLING.
DEFINE BROWSE brow-cust QUERY seq-cust DISPLAY Cust-num Name WITH 4 DOWN.

FORM
   exit-app SKIP
   brow-cust
   WITH FRAME main-frame.

FORM
   Customer.Cust-num Customer.Name Customer.Address Customer.Address2
   Customer.City Customer.State Customer.Postal-Code Customer.Country
   Customer.Phone Customer.Contact Customer.Sales-rep
   Customer.Credit-Limit Customer.Balance Customer.Terms
   Customer.Discount Customer.Comments
   SKIP
   com-cust AT 15 rev-cust AT 45
   WITH FRAME curr-frame SIDE-LABELS.

OPEN QUERY seq-cust FOR EACH this-cust.
FIND FIRST Customer NO-LOCK.
```

p-txn14.p *(2 of 5)*

```
ON ITERATION-CHANGED OF brow-cust
   DO:
      IF AVAILABLE(Customer)
      THEN DO:
         /* Determine whether any updates were made to previous record. */
         temp-hand = FRAME curr-frame:CURRENT-ITERATION.
         temp-hand = temp-hand:FIRST-CHILD.

         search-widgets:
         DO WHILE temp-hand <> ?:
            IF CAN-QUERY(temp-hand, "MODIFIED")
            THEN IF temp-hand:MODIFIED
                 THEN LEAVE search-widgets.
            temp-hand = temp-hand:NEXT-SIBLING.
         END.

         /* If a modification was made, assign the record. */
         IF temp-hand <> ?
         THEN DO:
            DO WITH FRAME curr-frame TRANSACTION:
               ASSIGN Customer.
            END.
            MESSAGE "Customer record updated.".
         END.
      END.

      /* Set curr-cust to the ROWID of the current query
         record. Find that record with NO-LOCK.           */
      curr-cust = ROWID(this-cust).
      FIND Customer WHERE ROWID(Customer) = curr-cust NO-LOCK.

      /* Display the current record. We have to disable the update
         fields and them re-enable them after the DISPLAY. Otherwise,
         the DISPLAY sets all the MODIFIED attributes to TRUE.*/
      DISABLE ALL WITH FRAME curr-frame.
      DISPLAY Customer WITH FRAME curr-frame.
      ENABLE ALL WITH FRAME curr-frame.
   END.

ON CHOOSE OF exit-app /* Exit application. */
   DO:
      /* Set a flag so we'll know why we exited the WAIT-FOR. */
      exit-chosen = TRUE.
   END.
```

p-txn14.p *(3 of 5)*

```
ON CHOOSE OF com-cust /* Commit changes to Customer record. */
   DO:
      /* Commit any changes made to the Customer record. */
      DO WITH FRAME curr-frame TRANSACTION:
         ASSIGN Customer.
         RELEASE Customer.
      END.

      MESSAGE "Customer record updated.".

      /* Release the exclusive lock. */
      CLOSE QUERY seq-cust.
      OPEN QUERY seq-cust FOR EACH this-cust.
      REPOSITION seq-cust TO ROWID curr-cust.

      /* Restore the MODIFIED attribute to FALSE. */
      HIDE FRAME curr-frame.
      DISABLE ALL WITH FRAME curr-frame.
      ENABLE ALL WITH FRAME curr-frame.
      VIEW FRAME curr-frame.
   END.

ON CHOOSE OF rev-cust /* Undo pending changes to Customer record. */
   DO:
      rev-chosen = TRUE.

      /* Undo any changes to the Customer screen buffer
         and reset the MODIFIED attributes to FALSE.      */
      DISABLE ALL WITH FRAME curr-frame.
      IF AVAILABLE(Customer)
      THEN DISPLAY Customer WITH FRAME curr-frame.
      ENABLE ALL WITH FRAME curr-frame.
END.
```

p-txn14.p *(4 of 5)*

```
ON ENTRY OF FRAME curr-frame
   DO:
      /* Upgrade from NO-LOCK to EXCLUSIVE-LOCK. */
      FIND Customer WHERE ROWID(Customer) = curr-cust EXCLUSIVE-LOCK
                  NO-WAIT NO-ERROR.

      /* If we didn't get the exclusive lock, then give a
         message and return focus to previous frame.         */
      IF LOCKED(Customer)
      THEN DO:
          MESSAGE "Record is currently locked by another user."
                  VIEW-AS ALERT-BOX ERROR BUTTONS OK.
      END.

      IF NOT AVAILABLE(Customer)
      THEN DO:
          APPLY "CHOOSE" TO rev-cust IN FRAME curr-frame.
          RETURN NO-APPLY.
      END.

      /* Pick up any updates that may have occured. */
      DISPLAY Customer WITH FRAME curr-frame.
   END.

ENABLE ALL WITH FRAME main-frame.
ENABLE ALL WITH FRAME curr-frame.
APPLY "ITERATION-CHANGED" TO brow-cust.
PAUSE 0 BEFORE-HIDE.
```

p-txn14.p *(5 of 5)*

```
main-loop:

DO WHILE TRUE ON ENDKEY UNDO, RETURN ON ERROR UNDO, RETRY:
   ASSIGN rev-chosen = FALSE
          exit-chosen = FALSE.

   WAIT-FOR CHOOSE OF exit-app OR WINDOW-CLOSE OF DEFAULT-WINDOW OR
         CHOOSE OF com-cust, rev-cust FOCUS brow-cust IN FRAME main-frame.

   IF AVAILABLE(Customer)
   THEN RELEASE Customer.

   IF exit-chosen
   THEN LEAVE main-loop.

   IF rev-chosen
   THEN UNDO main-loop, RETRY main-loop.
END.
```

When you run this procedure, focus is initially in the main–frame frame. When you move focus to the curr–frame, the ON ENTRY trigger applies an EXCLUSIVE–LOCK to the customer record. Subsequently, when you either choose the com–cust button or change the iteration of the browse, a small transaction assigns your changes to the customer record. The EXCLUSIVE–LOCK on the record is then released.

Although this example appears modeless when you run it, it is really modal. When you move focus to the curr–frame frame you enter a mode in which you hold an EXCLUSIVE–LOCK on the customer record (previously, you held the record NO–LOCK). This mode remains in effect until you choose either the com–cust or rev–cust button or change the iteration of the browse. At that point, the EXCLUSIVE–LOCK is released.

NOTE: While it is possible to present a modeless update, it requires somewhat complex coding for even a simple example, as in `p-txn14.p`. In a more realistic example where the user has more choices of actions, this technique is more difficult and more dangerous. Using modal updates with dialog boxes is safer and is the preferred approach.

12.8 Transactions and Subprocedures

If you start a transaction in a main procedure, that transaction remains active even while the main procedure runs called procedures. For example, `p-txn11.p` runs `p-txn11a.p` within a transaction:

p-txn11.p

```
DEFINE VARIABLE answer AS LOGICAL.
DEFINE NEW SHARED VARIABLE cust-num-var AS ROWID.

REPEAT WITH 1 DOWN:
    PROMPT-FOR customer.cust-num.
    FIND customer USING cust-num.
    DISPLAY name sales-rep.
    UPDATE credit-limit balance.
    SET answer LABEL "Do you want to do order processing?"
          WITH FRAME a NO-HIDE.
    IF answer THEN DO:
       cust-num-var = ROWID(customer).
       RUN p-txn11a.p.
    END.
END.
```

This procedure lets you update customer information and then asks if you want to process the customer's orders. If you say yes, the procedure runs a second procedure called `p-txn11a.p`.

p-txn11a.p

```
DEFINE SHARED VARIABLE cust-num-var AS ROWID.

HIDE ALL.
FIND customer WHERE ROWID(customer) = cust-num-var.
FOR EACH order OF customer:
    UPDATE order WITH 2 COLUMNS.
    FOR EACH order-line OF order:
      UPDATE order-line.
  END.
END.
```

The REPEAT block in the p-txn11.p procedure is the transaction block for that procedure: it contains a direct update to the database. The transaction begins at the start of each iteration of the REPEAT block and ends at the end of each iteration. That means that, when the p-txn11.p procedure calls the p-txn11a.p procedure, the transaction is still active. So all the work done in the p-txn11a.p subroutine is part of the transaction started by the main procedure, p-txn11.p. The following figure, illustrates the start and end of the full transaction.

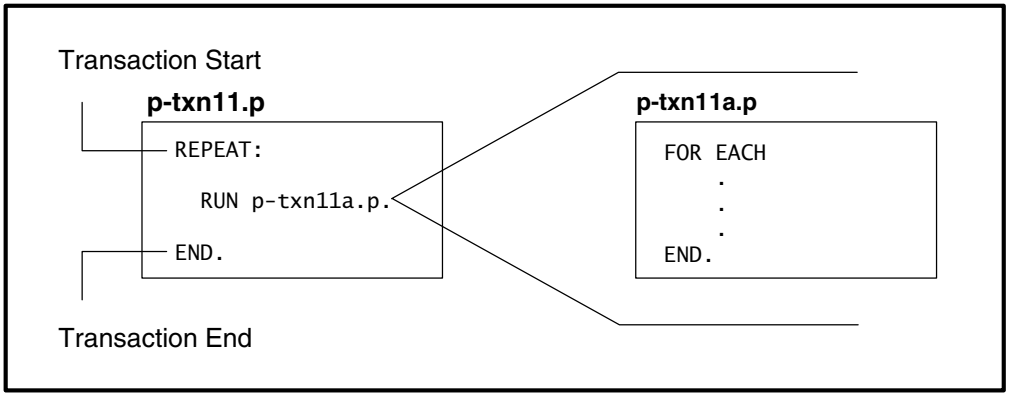


Figure 12–3: The Transaction Block for the p-txn11.p Procedure

If a system error occurs while you are processing orders for a customer, Progress undoes all the order processing work you have done for that customer, as well as any changes you made to the customer record itself.

12.9 Transactions with File Input

You must be especially careful when you process transactions that read input from a text file. If a crash occurs, the active transaction is backed out of the database, but you will not automatically know how far processing proceeded in the text file. To handle this, you must either:

- Restore the database and rerun the procedures that were running when the system failed.
- Run all data input processes as a single transaction (this has record locking implications as well as implications in terms of the size of the before-image file for the transaction).
- Have a way to determine how many of the input data lines were processed and committed to the database so that you do not rerun lines that were successfully processed by a procedure.

  One technique for doing this is to save the filename and the last line number in a database table, since changes in this status information can be synchronized with the corresponding database updates.

12.10 Transactions and Program Variables

You have read quite a bit about how transactions are backed out and how database changes are undone. But what happens to work done with variables?

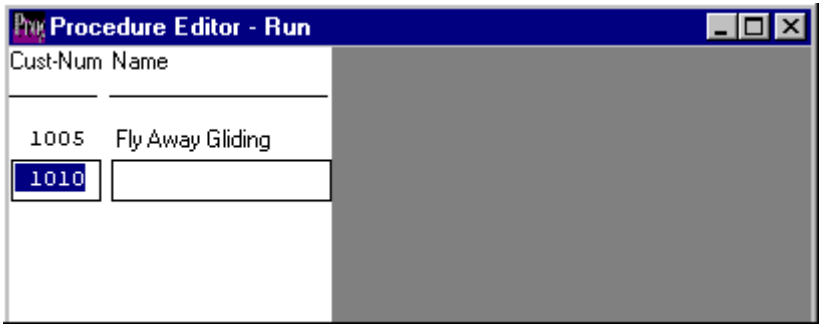
Any changes made to variables in a transaction or subtransaction block are undone whenever a transaction or subtransaction is backed out. The variables are restored to the values they had at the beginning of the transaction or subtransaction that is undone. Variables specifically defined as NO–UNDO are not undone in this case. However, changes to variables made outside a transaction are never undone since only transaction and subtransaction blocks can be undone.

Suppose you are creating customer records and using a variable to keep track of how many records have been created:

p-var.p

```
DEFINE VARIABLE ctr AS INTEGER INITIAL 0.

REPEAT:
  CREATE customer.
  ctr = ctr + 1.
  UPDATE cust-num name WITH NO-BOX.
END.
DISPLAY ctr "customer records were created"
  WITH NO-BOX NO-LABELS COLUMN 35.
```

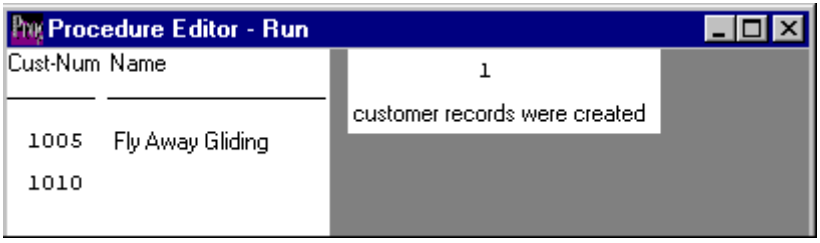
Run `p-var.p` and enter the data shown in the following display:



At this point, the following has happened:

- On the first iteration of the REPEAT block, the CREATE statement created a customer record.
- The ctr variable was incremented by 1, so its value became 1.
- You used the UPDATE statement to put the values of 500 and “Fly Away Gliding” in the cust–num and name fields of the new customer record.
- On the second iteration of the REPEAT block, the CREATE statement creates a second customer record.
- The ctr variable is incremented by 1 again, giving it a new value of 2.

Press **END–ERROR.** Progress undoes the work done in the current iteration of the REPEAT block and leaves that block. The customer record created on the second iteration, which had all its fields set to their initial values, is backed out and the value of ctr is reset from 2 to the value it had at the beginning of the iteration (transaction), which is 1. The DISPLAY statement tells you that just one customer has been created, as shown in the following display:



Within a transaction, any changes made to variables or to database fields are undone when that transaction is undone. However, any changes made to variables prior to the start of the transaction are not backed out. For example, consider `p-tnchp.p`.

p-tnchp.p

```
DEFINE VARIABLE i AS INTEGER.
DEFINE VARIABLE j AS INTEGER.

culoop:
REPEAT:
  i = i + 1.
  PROMPT-FOR customer.cust-num.
  FIND customer USING cust-num.
  DISPLAY name.
  FOR EACH order OF customer:
    j = j + 1.
    DISPLAY order-num.
    UPDATE odate.
    IF odate > TODAY THEN UNDO cusloop, RETRY cusloop.
  END.
END.
```

In this procedure, Progress starts a transaction at the beginning of each iteration of the FOR EACH block because the block contains an UPDATE statement to modify records. The UNDO cusloop statement undoes changes made during the current iteration of the FOR EACH block. When this occurs, Progress undoes the changes to the order being updated and to the variable j. Variable j is restored to its value at the beginning of the current iteration of the FOR EACH block (at the beginning of the transaction). However, Progress does not undo changes to the variable i because those changes were not done within the transaction.

Although backing out of variables is useful in many cases, keep the following in mind:

- There is a certain amount of overhead associated with undoing variables in transactions and subtransactions.
- If you are going to be doing extensive calculations involving variables or arrays, then consider using the NO–UNDO option on those variables if you have no need for the undo services on those variables.

12.10.1 Transactions in Multi-database Applications

In a multi-database application, you generally do not have to code any additional transaction handling. Multi-database transactions are handled in much the same way that single-database transactions are handled. The Progress two-phase commit mechanism ensures that any transaction is either committed to all affected databases or to none. You may, if you wish, check to see that all necessary databases are connected before you start a transaction.

Two-Phase Commit

During a transaction, Progress writes data to one or more databases as program control passes through database update statements in the transaction block. At the end of a transaction block, Progress tries to commit the changes to the databases. Progress uses a two-phase commit protocol to commit the changes to the databases. In the two-phase commit protocol, Progress polls all the databases affected by the transaction to see if they are reachable.

In the first phase of the two-phase commit, Progress checks whether it can reach each database and makes the appropriate validation checks for each database. If any one of the databases is unreachable or the validation checks fail for a database, Progress backs out the transaction and returns the databases to their pretransaction states using the before-image files. If all of the databases are reachable and their validation checks succeeded, Progress commits the changes to the databases.

For more information on two-phase commit, see the *Progress Database Administration Guide and Reference*.

Checking Database Connections

If you want to test database connections prior to entering a transaction, use the CONNECTED function.

```
IF CONNECTED("db1") AND CONNECTED("db2")
THEN RUN txnblk.p. /* transaction block */
ELSE MESSAGE "Unable to perform transaction".
```

You should connect to all databases affected by a transaction prior to entering a transaction block. As a general rule, do not execute a database connection in a transaction block. The database connection overhead could lock records in other databases affected by the transaction for considerable length of time. A database connection failure also causes a transaction error. Progress defers DISCONNECT statements in a transaction until the transaction completes or is undone.

For more information on connecting and disconnecting databases, see Chapter 9, “Database Access.”

12.10.2 Transactions in Distributed Applications

When a requesting application with an active transaction runs a remote procedure, Progress does not propagate the transaction to the remote procedure. Rather, the remote procedure acts as if it is the first procedure of the application, and follows the normal 4GL rules for starting and terminating transactions. If a requesting application and a remote procedure connect to the same database, each database connection comprises a separate transaction.

For more information on remote procedures, see *Building Distributed Applications Using the Progress AppServer*.

12.11 Determining When Transactions Are Active

You can use the TRANSACTION function to determine whether a transaction is active in a procedure. The TRANSACTION function can help you identify the transaction and subtransaction blocks within a procedure.

Run `p-nord2.p` to see how the TRANSACTION function works:

p-nord2.p

```
MESSAGE "A transaction" (IF TRANSACTION THEN "is" ELSE "is not")
        "active before the REPEAT block.".

ordblock:
REPEAT:
  MESSAGE "A transaction" (IF TRANSACTION THEN "is" ELSE "is not")
        "active in the outer REPEAT block.".
  CREATE order.
  UPDATE order-num cust-num odate.

  olblock:
  REPEAT:
    MESSAGE "A transaction" (IF TRANSACTION THEN "is" ELSE "is not")
        "active in the inner REPEAT block.".
    CREATE order-line.
    order-line.order-num = order.order-num.
    SET line-num qty item-num price.
  END.
END.
```

This procedure displays messages telling you whether a transaction is active or inactive in three places. You can get more information on transaction activity by using the LISTING option on the COMPILE statement. See the *Progress Language Reference* for more information on the COMPILE statement and the TRANSACTION function.

12.12 Progress Transaction Mechanics

So far, this chapter has explained the actions Progress takes in the event of different kinds of errors and how you can override those actions and specify your own. But there is another side to what Progress is doing during transactions and subtransactions. The next two sections summarize the mechanics of transactions and subtransactions.

12.12.1 Transaction Mechanics

During a transaction, information on all database activity occurring during that transaction is written to a *before-image* (BI) file. Progress maintains one BI file for each database. The information written to the before-image file is carefully coordinated with the timing of the data written to the actual database table. That way, if an error occurs during the transaction, Progress uses this before-image file to restore the database to the condition it was in before the transaction started. Information written to the before image file is not buffered. It is written to disk immediately.

Space in the before-image file is allocated in units called clusters. Progress automatically allocates new clusters as needed. (You can use the PROUTIL TRUNCATE BI utility to set the cluster size.) After all changes associated with a cluster have been committed and written to disk, Progress can reuse the cluster. Therefore the disk space used by the before-image file depends on several factors including the cluster size, the scope of your transactions, and when physical writes are made to the database (.db) file.

12.12.2 Subtransaction Mechanics

If a transaction is already active and Progress encounters a DO ON ERROR, DO TRANSACTION, FOR EACH, REPEAT, or procedure block, Progress starts a subtransaction. All database activity occurring during that subtransaction is written to a *local-before-image* (LBI) file. Progress maintains one LBI file for each user. If an error occurs during the subtransaction, Progress uses this local-before-image file to restore the database to the condition it was in before the subtransaction started. Progress uses the local-before-image file to back out variables and to back out subtransactions in all cases when an entire transaction is not being backed out.

Note that the **first** time a variable is altered within a subtransaction block, **all** of the variables in the procedure are written to the LBI file as a record.

Because the local-before-image information is not needed for crash recovery, it does not have to be written to disk in a carefully synchronized fashion as does the before-image information. This minimizes the overhead associated with subtransactions. The local-before-image file is written using normal, buffered I/O.

The amount of disk space required for each user's LBI file depends on the number of subtransactions started that are subject to being undone.

12.13 Efficient Transaction Processing

Here are a few guidelines to improve the efficiency of transaction processing procedures:

- If you are doing extensive calculations with variables, and you do not need to take advantage of undo processing for those variables, use the NO–UNDO option when defining the variables.
- If you are processing array elements, process them in a DO WHILE block rather than in a REPEAT WHILE block. That way, you will not start a separate transaction or subtransaction for each array element.
- When the logic of your application permits, do as much processing as possible directly at the transaction level rather than creating subtransactions. This principle should not restrict the way you implement your application, but you should use it whenever it is convenient.

13

Locks

If you write multi-user applications, there are additional issues and features you should address during your application development cycle.

This chapter explains:

- Sharing database records
- Resolving locking issues

The concepts explained in this chapter apply to applications running in a multi-user environment. If you are developing an application on a single-user system, you can use all the multi-user statements shown in this chapter. Progress simply disregards those statements when you run the application in single-user mode.

13.1 Applications in a Multi-user Environment

Suppose there are two departments, Shipping and Receiving, that are using the same database. When they ship an order, the Shipping Department subtracts the number of items shipped from that item's on–hand value. When it receives inventory from a vendor, the Receiving Department adds the number of items received to that item's on–hand value.

Consider the case where the number of parkas on hand is 67. The Shipping Department ships 20 parkas and the Receiving Department receives 10 parkas. The number of parkas on hand is now 57 (67 on–hand – 20 shipped + 10 received = 57).

Figure 13–1 shows how this simple situation, when performed with no special multi-user controls, can produce a very different result from the obvious one you expect.

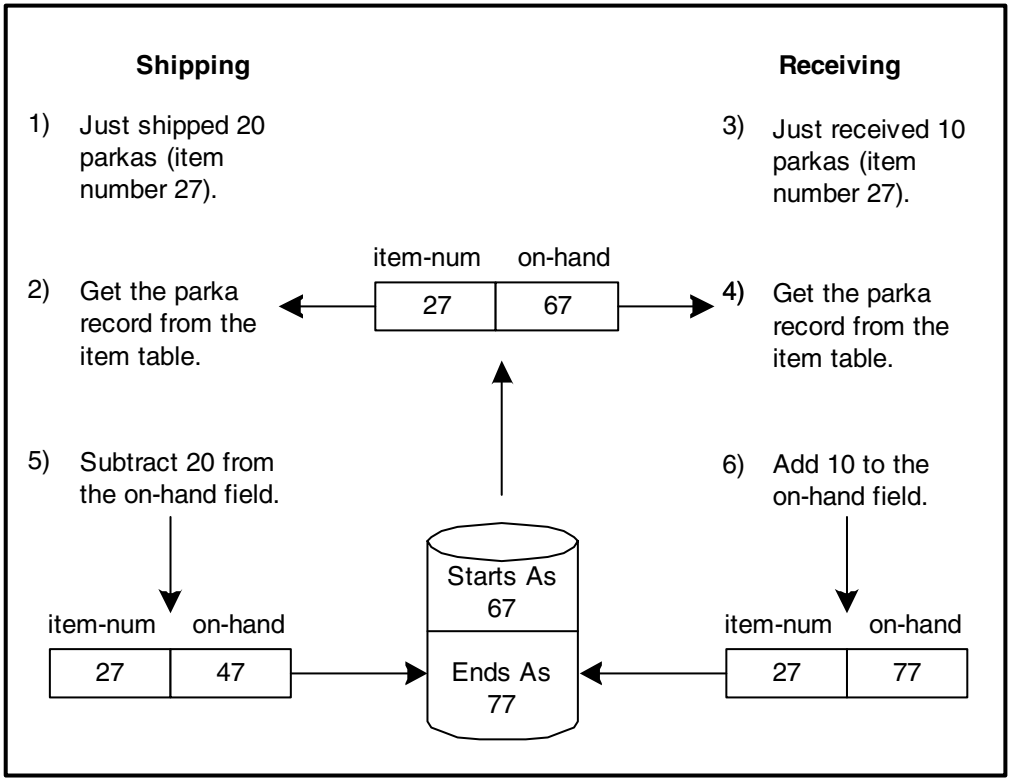


Figure 13–1: Multi-user Update Scenario Without Locks

Examine what happens if you do not take any special steps to handle multiple users accessing the same database. The points below correspond to the numbers in Figure 13–1.

1. The Shipping Department ships an order for 20 parkas (item number 27).
2. The Shipping Department gets the parka record from the item table. In that record, the value of the on–hand field is 67.
3. The Receiving Department receives 10 parkas (item number 27).
4. The Receiving Department also gets the parka record from the item table. In that record, the value of the on–hand field is 67.
5. The Shipping Department subtracts 20 from the on–hand value, and returns the record to the database. The on–hand value of the parka record in the database is now 47.
6. The Receiving Department adds 10 to the on–hand value of 67 and returns the record to the database. That record overwrites the record written by the Shipping Department, causing the on–hand value of the parka record in the database to be 77.

The database indicates that there are 77 parkas on hand instead of 57. It is as if the Receiving Department never updated the database.

Scenarios like this occur often in a multi-user environment and, for that reason, Progress supplies services to make sure that the data in your database is accurate.

13.2 Sharing Database Records

The situation described above has to do with multiple users who need access to the same record at the same time. In the illustration, things did not work as expected because both the Shipping and Receiving departments were making changes to the same record at the same time. But neither department could see the other department's changes.

The solution to this record sharing problem is to make sure that:

- Multiple users can read, or look at, the same record at the same time.

  BUT

- Only one user at a time can update, or make changes to, a record.

Progress uses record locks to enforce these rules.

13.2.1 Using Locks to Avoid Record Conflicts

Take another look at the Shipping and Receiving scenario. Figure 13–2 shows how you can use locks to manage concurrent use of the same database record.

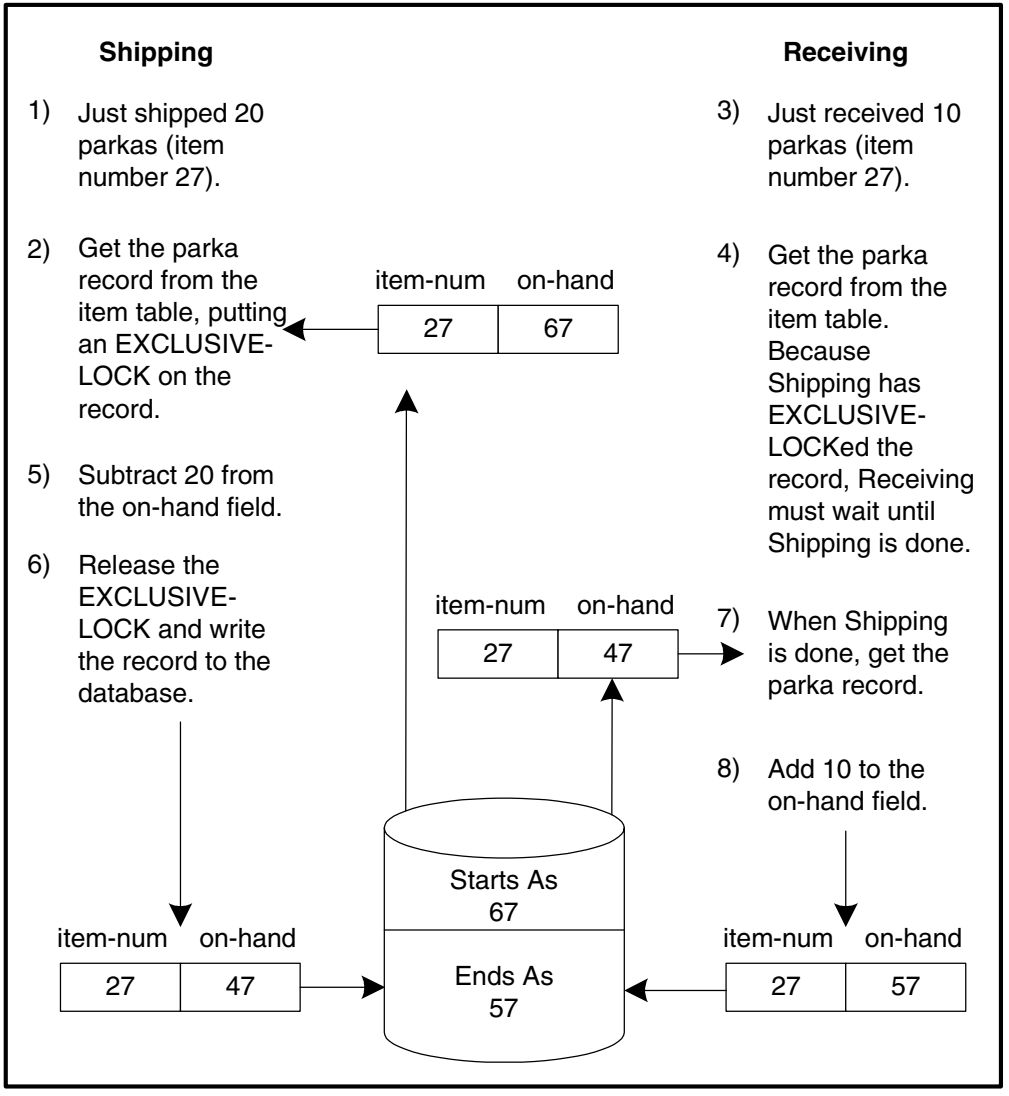


Figure 13–2: Multi-user Update Scenario with Locks

Here are the procedures that do this work. The numbers in the margin of the procedures correspond to the numbers in the list that follows the procedures:

p-lock1.p

```
                /* SHIPPING */

                DEFINE VARIABLE qty-shipped AS INTEGER
                  LABEL "Number Shipped".

                REPEAT:
/* 1 */               PROMPT-FOR item.item-num.
/* 2 */               FIND item USING item-num EXCLUSIVE-LOCK.
                      DISPLAY item-name on-hand.
                      SET qty-shipped.
/* 5 */              on-hand = on-hand - qty-shipped.
                      DISPLAY on-hand.
/* 6 */         END.
```

p-lock2.p

```
                /* SHIPPING */

                DEFINE VARIABLE qty-shipped AS INTEGER
                  LABEL "Number Received".

                REPEAT:
/* 3 */               PROMPT-FOR item.item-num.
/* 4, 7 */            FIND item USING item-num.
                      DISPLAY item-name on-hand.
                      SET qty-recvd.
/* 8 */               on-hand = on-hand - qty-recvd.
                      DISPLAY on-hand.
                END.
```

1. The Shipping Department enters and ships an order for 20 parkas (item number 27).

2. The Shipping Department gets the parka record from the item table, placing an EXCLUSIVE–LOCK on the record. That means that no other user can look at the record until Shipping is finished with it. In that record, the value of the on–hand field is 67.

   NOTE: You can read a locked record if you bypass all record locking using NO–LOCK. For more information, see the "Bypassing Progress Lock Protections" section.

3. The Receiving Department receives and enters 10 parkas (item number 27) into inventory.

4. The Receiving Department tries to get the parka record from the item table but cannot because Shipping has an EXCLUSIVE–LOCK on the record.

5. The Shipping Department subtracts 20 from the on–hand value, and returns the record to the database.

6. Once it returns the record to the database, the procedure releases the EXCLUSIVE–LOCK.

7. The Receiving Department can now successfully read the record from the database. The on–hand value of the parka record in the database is now 47.

8. The Receiving Department adds 10 to the on–hand value of 47 and returns the record to the database. Therefore, the final on–hand value for the parka record is 57–exactly what you would expect!

13.2.2 How Progress Applies Locks

You've just seen how you can apply a lock on your own. But if you do not apply any locks, Progress performs default locking. In particular:

- Whenever it reads a record, Progress puts a SHARE–LOCK on that record. (An exception is the browse widget. See the section "Bypassing Progress Lock Protections" for more information.) This means that other users can read the record but cannot update it until the procedure releases the SHARE–LOCK. If you try to read a record with a SHARE–LOCK when another user has that record EXCLUSIVE–LOCKed, Progress displays a message that the record is in use and you must wait to access it.

- Whenever Progress updates a record, it puts an EXCLUSIVE–LOCK on that record. This means that other users cannot read or update that record until the procedure releases EXCLUSIVE–LOCK. If you try to read a record with an EXCLUSIVE–LOCK, when another user has that record SHARE–LOCKed or EXCLUSIVE–LOCKed, you receive a message that the record is in use and you must wait to access it.

NOTE: SHARE–LOCKs and EXCLUSIVE–LOCKs use up entries in the lock table. The possible number of entries in the lock table defaults to 500. You can change this with the Lock Table Entries (–L) startup parameter. Progress stops a user program if it attempts to access a record that overflows the lock table.

In Figure 13–3, when the FIND statement reads the customer record, Progress puts a SHARE–LOCK on that record. Because the UPDATE statement lets you change the record, Progress upgrades the SHARE–LOCK to an EXCLUSIVE–LOCK after the UPDATE statement executes.

p-lock3.p

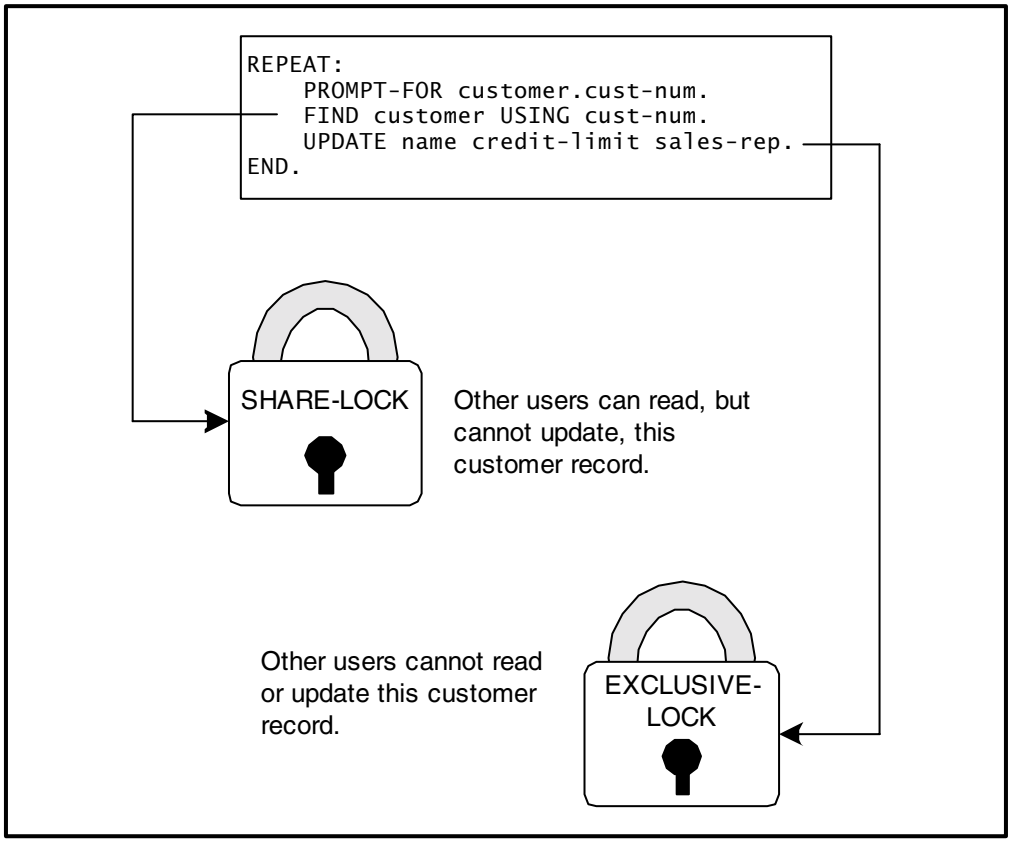


Figure 13–3: How Progress Applies Locks

13.3 Bypassing Progress Lock Protections

In some cases you might want to read a record even though another user holds an EXCLUSIVE–LOCK on that record. For example:

- You might want to produce a report and you do not want to be held up by locking conflicts.
- You might want to browse through a set of records and then select one to update. You do not want to lock all the records that you are browsing, but only lock the one you select.

For cases such as these, Progress allows you bypass the normal locking mechanism and access a record with NO–LOCK. For example, `p-lock6.p` produces a simple customer report:

p-lock6.p

```
FOR EACH customer NO-LOCK:
    DISPLAY cust-num name address city state postal-code.
END.
```

Because the FOR EACH statement in this example uses the NO–LOCK option, the procedure accesses every customer record without error even if another user holds an EXCLUSIVE–LOCK on one or more records.

For a browse widget, NO–LOCK is the default mode. If you want to update a record, then you must find that record again to apply a lock. For example, `p-lock7.p` browses records with NO–LOCK but allows you to select a record to update:

p-lock7.p

```
DEFINE BUTTON upd-cust LABEL "Update Customer".
DEFINE BUTTON exit-app LABEL "Exit".

DEFINE VARIABLE changes-made AS LOGICAL.
DEFINE VARIABLE curr-cust AS ROWID.
DEFINE QUERY seq-cust FOR customer.
DEFINE BROWSE brow-cust QUERY seq-cust DISPLAY Cust-num Name WITH 5 DOWN.

FORM
   upd-cust exit-app SKIP(1)
   brow-cust
   WITH FRAME main-frame.

OPEN QUERY seq-cust FOR EACH customer.

ON VALUE-CHANGED OF brow-cust
   DO:
      curr-cust = ROWID(customer).
   END.

ON CHOOSE OF upd-cust
   DO: /* TRANSACTION */
      FIND customer WHERE ROWID(customer) = curr-cust EXCLUSIVE-LOCK.
      UPDATE customer WITH FRAME cust-frame VIEW-AS DIALOG-BOX
         TITLE "Customer Update".
      DISPLAY Cust-num Name WITH BROWSE brow-cust.
      RELEASE customer.
   END.

ENABLE ALL WITH FRAME main-frame.
PAUSE 0 BEFORE-HIDE.

WAIT-FOR CHOOSE OF exit-app OR WINDOW-CLOSE OF DEFAULT-WINDOW.
```

When you run this procedure and choose the upd–cust button, the trigger finds the current record again and applies an EXCLUSIVE–LOCK to it. You can then update the record.

NOTE: If you read a record with NO–LOCK before a transaction starts and then read the same record with EXCLUSIVE–LOCK within the transaction, the EXCLUSIVE–LOCK is automatically downgraded to a SHARE–LOCK when the transaction ends. Progress does not automatically return you to NO–LOCK status. If you do not want to hold a lock beyond the end of the transaction, you must execute a RELEASE statement on the record within the transaction.

In p-lock7.p, because a RELEASE statement is executed in the transaction, the record lock is released after the transaction completes.

Because NO–LOCK bypasses the normal Progress lock protection, there are risks in using NO–LOCK. For example, in p-lock7.p, the customer names you see in the browse widget might not be up-to-date. Another user might have updated one or more records since you retrieved them. You receive no warning that your data is no longer current. However, when you execute the FIND statement in the upd–cust trigger, you retrieve the up-to-date record.

13.4 How Long Does Progress Hold a Lock?

Record locks and transactions are closely related. Chapter 12, "Transactions" explained that transactions are always related to blocks. To review, the following are transaction blocks:

- Any block that uses the TRANSACTION keyword on the block statement (DO, FOR EACH, or REPEAT).
- A procedure block, a trigger block, and each iteration of a DO ON ERROR, FOR EACH, or REPEAT block that directly updates the database or directly reads records with EXCLUSIVE–LOCK.

13.4.1 Locks and Transactions

To understand the relationship between record locks and transactions, see Figure 13–4, which shows a modified version of `p-lock3.p`.

p-lock4.p

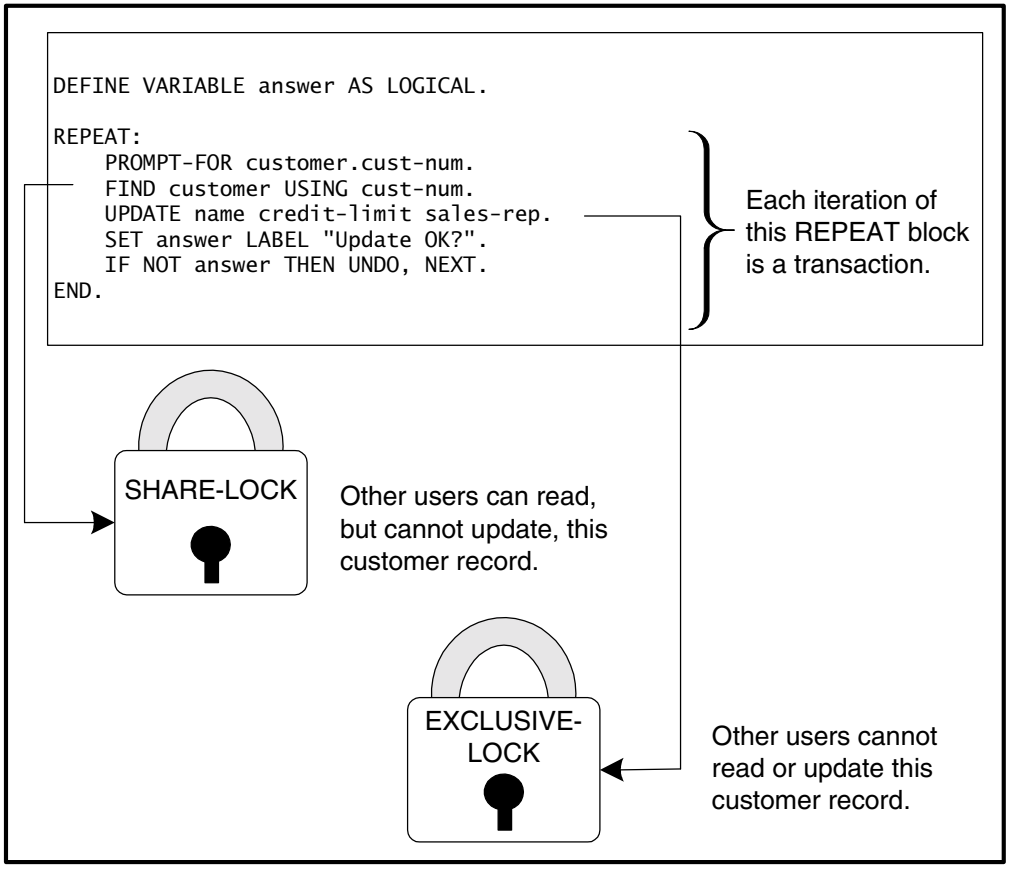


Figure 13–4: Locks and Transactions

The following hypothetical steps illustrate a common problem with record locks:

1. At the start of the first iteration of the REPEAT block, Progress starts a transaction.
2. You supply customer number 1 to the PROMPT–FOR statement.
3. The FIND statement reads the database record for customer 1 and Progress places a SHARE–LOCK on that record.

4. The UPDATE statement lets you make changes to the record and Progress places an EXCLUSIVE–LOCK on the record.

5. At the end of the UPDATE statement, Progress releases the EXCLUSIVE–LOCK.

6. Another user running the same procedure finds customer 1 and updates information for that customer (this user gets the record with the new data you entered in Step 4).

7. You answer no to the “Updates OK” prompt and Progress undoes your changes.

8. The other user answers yes to the “Updates OK” prompt and Progress writes the record back to the database, with the changes that you wanted undone.

Because of this problem and others like it, Progress uses some standards to determine how long to hold a lock. These standards produce behavior that matches what you want to happen in situations like the one just described. Table 13–1 lists the standards Progress uses to determine when to release record locks.

Table 13–1: When Progress Releases Record Locks

Type of Lock	Acquired During a Transaction[1]	Acquired Outside a Transaction[2]	Acquired Outside, but Held Going into a Transaction[3]
SHARE	Held until the transaction end or record release, whichever is later.[4]	Held until record release.	Held until the transaction end or record release, whichever is later.[4]
EXCLUSIVE	Held until transaction end. Then converted to SHARE[5] if record scope is larger than transaction and record is still active in any buffer.	N/A	N/A

1 A record acquires a lock during a transaction if Progress reads or rereads it after the start, and before the end of a transaction.

2 A record acquires a lock outside a transaction if Progress reads it when a transaction is not active and releases it before a transaction starts.

3 A record acquires a lock outside a transaction and holds it going into a transaction if Progress reads the record when a transaction is not active and releases the record when a transaction starts

4 Progress releases a record from a buffer at the end of the record scope, when Progress executes a RELEASE statement, or when Progress replaces the record in the buffer by a CREATE, FIND, or FOR EACH statement

5 This is true even if Progress read the record with NO–LOCK prior to the transaction and rereads it with EXCLUSIVE–LOCK during the transaction. By default, Progress converts the lock to SHARE–LOCK, not back to NO–LOCK. To avoid holding the record SHARE–LOCK in this case, execute a RELEASE statement during the transaction. Progress then releases the lock at the end of the transaction. You can then re-read the record with NO–LOCK.

Progress releases locks acquired within a transaction, (or changes them to SHARE–LOCK if it locked the record prior to the transaction) if it backs out the transaction. This does not occur when Progress backs out a subtransaction because Progress does not release the locks a record acquires within a transaction unless the transaction ends or Progress undoes the entire transaction.

How do the rules in Table 13–1 affect the `p-lock4.p` procedure?

1. At the start of the first iteration of the REPEAT block, Progress starts a transaction.

2. You supply customer number 1 to the PROMPT–FOR statement.

3. The FIND statement reads the database record for customer 1 and Progress places a SHARE–LOCK on that record. The SHARE–LOCK remains until the end of the transaction or until Progress releases the record, whichever occurs later. In this example, transaction end and record release both happen at the end of the REPEAT block.

4. The UPDATE statement lets you make changes to the record and Progress upgrades the SHARE–LOCK to an EXCLUSIVE–LOCK. This lock remains until the end of the transaction, which is the end of this iteration of the REPEAT block.

5. Another user running the same procedure tries to find customer 1. However, because the record for customer 1 is EXCLUSIVE–LOCKed, the FIND statement waits until the record is available. The user sees a message that the record is in use.

6. You answer NO to the “Updates OK” question, Progress undoes the changes you made to the record, reaches the end of the first iteration of the REPEAT block, the transaction ends and releases the EXCLUSIVE–LOCK on the record.

7. The other user is able to find the record and update it.

Because of the duration of the record locks, Progress processes your transactions consistently in a multi-user environment.

13.4.2 Locks and Multiple Buffers

Progress also uses default behaviors to prevent the condition known as bleeding record locks when multiple buffers use the same record. A *bleeding record lock* occurs when a buffer with NO–LOCK inherits a SHARE–LOCK from another buffer during a record disconnect.

For example, examine this code:

p-lock12.p

```
DEFINE BUFFER acust FOR customer.
DEFINE BUFFER bcust FOR customer.

FIND acust WHERE acust.cust-num = 1 NO-LOCK.
FIND bcust WHERE bcust.cust-num = 1 SHARE-LOCK.
RELEASE bcust.
```

In this code example, two customer buffers contain copies of the same record. One has it with a NO–LOCK and the other has it with a SHARE–LOCK. When the second FIND statement executes, the NO–LOCK state of acust is upgraded to SHARE–LOCK. When Progress executes the RELEASE statement, the customer record is disconnected from the bcust buffer and its SHARE–LOCK is downgraded to a NO–LOCK. What happens to the acust buffer lock status? If acust retains the SHARE–LOCK, then you can say that the bcust lock **bled** to acust. This is a bleeding record lock. Fortunately, Progress does not simply release the specific record and lock. It checks all other buffers to see what locks they require. In this case, the acust buffer is downgraded to NO–LOCK. Progress does not allow bleeding record locks to occur in any situation.

NOTE: Progress Version 7.3A and earlier did allow bleeding record locks as described in this section. To preserve application behavior that depended on bleeding locks, use the Bleeding Record Lock (–brl) startup parameter. You might have to use this startup parameter if a working Version 7.3A or earlier Progress application generates the following error messages when compiled under V7.3B or later:

```
table-name record has NO-LOCK status, update to field not allowed
```

13.5 Resolving Locking Conflicts

You can use the following options to resolve locking conflicts and to describe the way you want to lock records:

- EXCLUSIVE–LOCK
- SHARE–LOCK
- NO–LOCK

For example, if you know you are going update a record, you can use the EXCLUSIVE–LOCK option with the FIND statement, as shown in the following procedures:

p-lock4.p

```
                    /* USER 1 */

DEFINE VARIABLE answer AS LOGICAL.
REPEAT:
    PROMPT-FOR customer.cust-num.
    FIND customer USING cust-num.
    UPDATE name credit-limit sales-rep.
    SET answer LABEL "Update OK?".
    IF NOT answer THEN UNDO, NEXT.
END.
```

p-lock5.p

```
                    /* USER 2 */

FOR EACH customer:
    DISPLAY cust-num name address city state postal-code.
END.
```

Before running either of these procedures, follow these steps to set up the multi-user environment:

1 ♦ Start the multi-user server, using the command appropriate to your operating system.

Operating System	Syntax
UNIX Windows	`proserve` *database-name*

Here, *database–name* is the name of the database that you want to run multi-user Progress against.

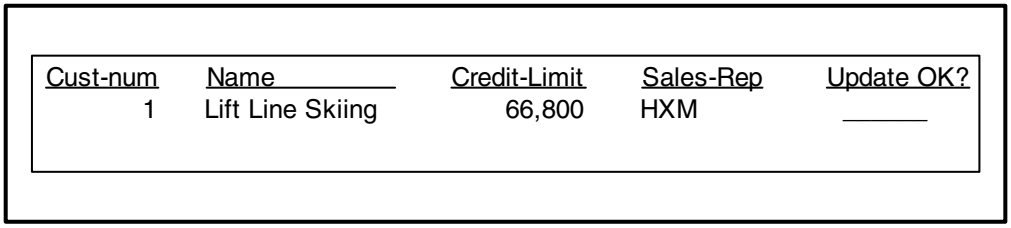
2 ♦ On one terminal (terminal 1), start multi-user Progress with the appropriate command for your operating system.

Operating System	Syntax
UNIX Windows	`mpro` *database-name*

3 ♦ On a second terminal (terminal 2), enter the same command to start another multi-user Progress session for the same database.

You are now running two users in a multi-user environment.

On terminal 1 run the p-lock4.p procedure. Enter **1** in response to the PROMPT–FOR statement. Make some change to the record for customer 1 and then press GO. The SET statement prompts you to verify the changes you made.



Cust-num	Name	Credit-Limit	Sales-Rep	Update OK?
1	Lift Line Skiing	66,800	HXM	

On terminal 2, run the p-lock5.p procedure. Progress displays a message telling you that the record is in use by another user.

```
customer in use by user1 on ttyi13. Wait or press CTRL-C to stop.
```

(This message is system specific; it names the appropriate user and device on each system.)

The UPDATE statement places an EXCLUSIVE–LOCK on the customer record 1. That lock is in effect until the end of the transaction, which means Progress locks the record while it prompts you to verify your updates. Meanwhile, the user on terminal 2 cannot read the record for customer 1 because the user on terminal 1 has that record EXCLUSIVE–LOCKed.

At this point, you can press STOP on terminal 2 if you do not want to wait for terminal 1 to release the record. But do not do that right now. Go to terminal 1 and answer YES. You now see customer information on terminal 2.

When you answer YES on terminal 1, Progress completes the SET statement and reaches the end of the REPEAT block, which is also the end of the transaction in the p-lock4.p procedure. Progress releases EXCLUSIVE–LOCKs at the end of the transaction, making the record available to other users.

Imagine that, instead of EXCLUSIVE–LOCKing the customer record, the user at terminal 1 SHARE–LOCKed the record. Meanwhile, the user at terminal 2 came along and SHARE–LOCKed the record. Now suppose that both users want to update the record. They both need an EXCLUSIVE–LOCK on the record. But the user at terminal 1 cannot EXCLUSIVE–LOCK the record because the other user has the record SHARE–LOCKed. And the user at terminal 2 cannot EXCLUSIVE–LOCK the record because the other user has it SHARE–LOCKed. This situation is known as *deadly embrace*. One of the two users must press `STOP` to resolve the situation.

There are ways to avoid this situation. For example, if you know you are going to update a record, you can read that record with EXCLUSIVE–LOCK. Alternatively, where you run a reporting procedure, such as `p-lock5.p`, you might not need to see the data that is in flux. You can use the NO–LOCK option to tell Progress to let you see the record even if it is EXCLUSIVE–LOCKed:

p-lock6.p

```
FOR EACH customer NO-LOCK:
    DISPLAY cust-num name address city state postal-code.
END.
```

Try running the `p-lock4.p` procedure again. Once again, enter **1** as the customer number, make any change to the `customer` record, and press **GO**. Now go to another terminal and run the `p-lock6.p` procedure.

This time, Progress lets the `p-lock6.p` procedure see customer 1 because the NO–LOCK option ignores the EXCLUSIVE–LOCK placed on that customer record by the `p-lock4.p` procedure.

There are two functions you can use when working with locking situations: AVAILABLE and LOCKED. See the *Progress Language Reference* for information and examples for using these functions to manage record-locking conflicts.

13.6 Managing Locks to Improve Concurrency

Rather than rely on default locking behavior, you can use a simple lock management algorithm to:

- Eliminate “deadly embrace”
- Decrease the scope of transactions
- Prevent users from tying up records for extended periods of time

Each time you want to present data to the user for inspection and possible update, follow this procedure:

1. Request and display the record with the NO–LOCK option.
2. Let the user manipulate the data until the user signals that they have finished (ON LEAVE OF *field,* ON CHOOSE OF *ok–button,* or similar).
3. Request the same record again using the FIND CURRENT or GET CURRENT statement with the EXCLUSIVE–LOCK option.
4. Use the CURRENT–CHANGED function to see if the original record has changed since you first presented it to the user.
5. If the record is the same, commit the change.
6. If the record has changed, inform the user, display the new data, and repeat the process.

The code example below demonstrates this technique:

p-lock13.p

```
FORM customer.name customer.balance WITH FRAME upd.

ON GO OF FRAME upd DO:
    DO TRANSACTION:
        FIND CURRENT customer EXCLUSIVE-LOCK.
        IF CURRENT-CHANGED customer THEN DO:
            MESSAGE "This record has been changed by another user"
                    SKIP
                    "Please re-enter your changes."
                    VIEW-AS ALERT-BOX.
            DISPLAY customer.name customer.balance with frame upd.
            RETURN NO-APPLY.
        END.
        ASSIGN customer.name customer.balance.
    END.
    FIND CURRENT customer NO-LOCK.
END.

FIND FIRST customer NO-LOCK.
DISPLAY customer.name customer.balance WITH FRAME upd.
DO ON ENDKEY UNDO, LEAVE:
    ENABLE customer.name customer.balance WITH FRAME upd.
    WAIT-FOR "GO" OF FRAME upd.
END.
```

The first problem this technique avoids is the classic “user at lunch” syndrome. If the user begins work on a SHARE–LOCK record and does not finish with it immediately, all other users are prevented from updating the record. The FIND CURRENT technique moves the scope of the transaction between possible user actions. In other words, no user interface manipulation can tie up a record.

Making transactions smaller, or atomic, also improves performance and promotes modular code design that can support many interface styles.

Finally, this technique avoids the “deadly embrace” problem. Deadly embrace occurs when two users have the same record with SHARE–LOCK and both are waiting for the other to release the record. Although this problem can be avoided by good design, using the FIND CURRENT technique means that this will never be a problem.

13.7 Changing the Size of a Transaction

Previous chapters showed how the size of a transaction affects how much work is undone or recovered in the event of a system failure or an error (see Chapter 6, "Handling User Input," and Chapter 12, "Transactions" for information). Now you know that transactions and record locks are closely related. Figure 13–5 shows how changing the size of a transaction affects record locks.

p-txn3a.p

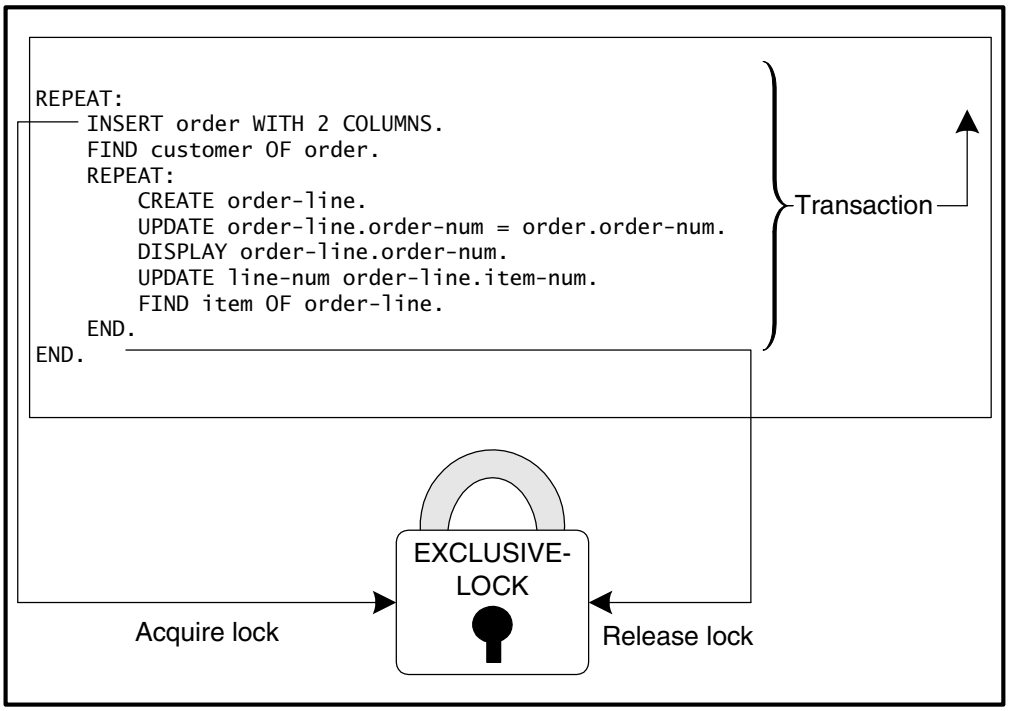


Figure 13–5: Transaction Size and Locks

In this procedure, the outer REPEAT block is the transaction block. The INSERT statement creates an order record and gets an EXCLUSIVE–LOCK on that record. Progress holds that lock until the end of the transaction block (the outer REPEAT block).

The UPDATE statement in the inner REPEAT block lets you update order–line records. While you update order–line records, you still have an EXCLUSIVE–LOCK on the order record you created at the start of the REPEAT block. Because of that lock, no other user can access that order record until you finish updating order–line records and release your EXCLUSIVE lock (and, in this case, the record as well) at the end of the outer REPEAT block.

It is also important to note that just prior to the end of each transaction, all of the order–line records created are EXCLUSIVE–LOCKed and all of the item records found are SHARE–LOCKed.

13.7.1 Making a Transaction Larger

In the last procedure, you EXCLUSIVE–LOCKed one order record at a time: you got an EXCLUSIVE–LOCK when you INSERTed the record at the start of the outer REPEAT block. At the end of the outer REPEAT block, Progress released that EXCLUSIVE–LOCK, making the record available to other users.

Suppose you want to EXCLUSIVE–LOCK all the order records you create, retaining those locks until you finish creating order records. You can do this by simply making the transaction larger, as shown in Figure 13–6.

p-txn4.p

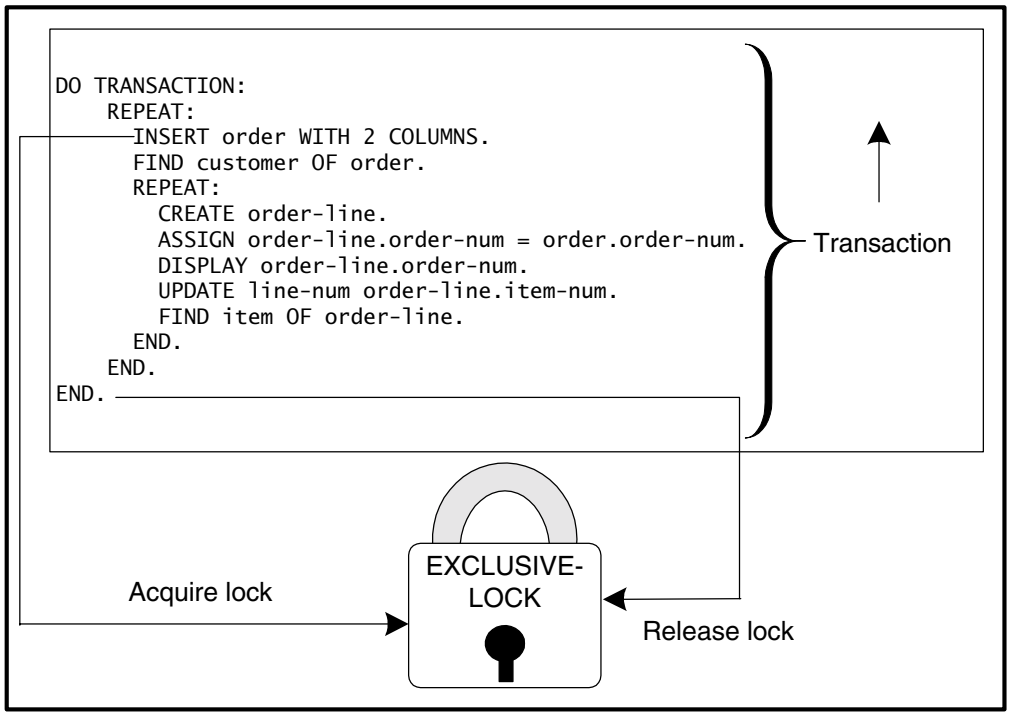


Figure 13–6: Making Transactions Larger

The TRANSACTION option on the DO block overrides the default transaction placement. Even though an outermost REPEAT, FOR EACH, or procedure block that contains direct database updates normally is made the transaction block, the TRANSACTION option overrides that default.

Now, you still get an EXCLUSIVE–LOCK when you INSERT an order record at the start of the outer REPEAT block. But remember that Progress releases EXCLUSIVE–LOCKs at the end of the transaction. When the REPEAT block ends, the transaction is not over, so Progress retains the EXCLUSIVE–LOCK. When you INSERT the next record, you get another EXCLUSIVE–LOCK on that record.

You can have many order and order–line records EXCLUSIVE–LOCKed, preventing other users from accessing those records. You can also have many item records SHARE–LOCKed, preventing other users from updating those records.

13.7.2 Making a Transaction Smaller

Now suppose that once you have EXCLUSIVE–LOCKed the order record at the start of the outer REPEAT block, you want to release that lock as quickly as possible (before the end of the block) so that other users can access that order record. You also want to release the SHARE–LOCKs on the item records as soon as possible You can do this by simply making the transaction smaller, as shown in Figure 13–7.

p-txn10.p

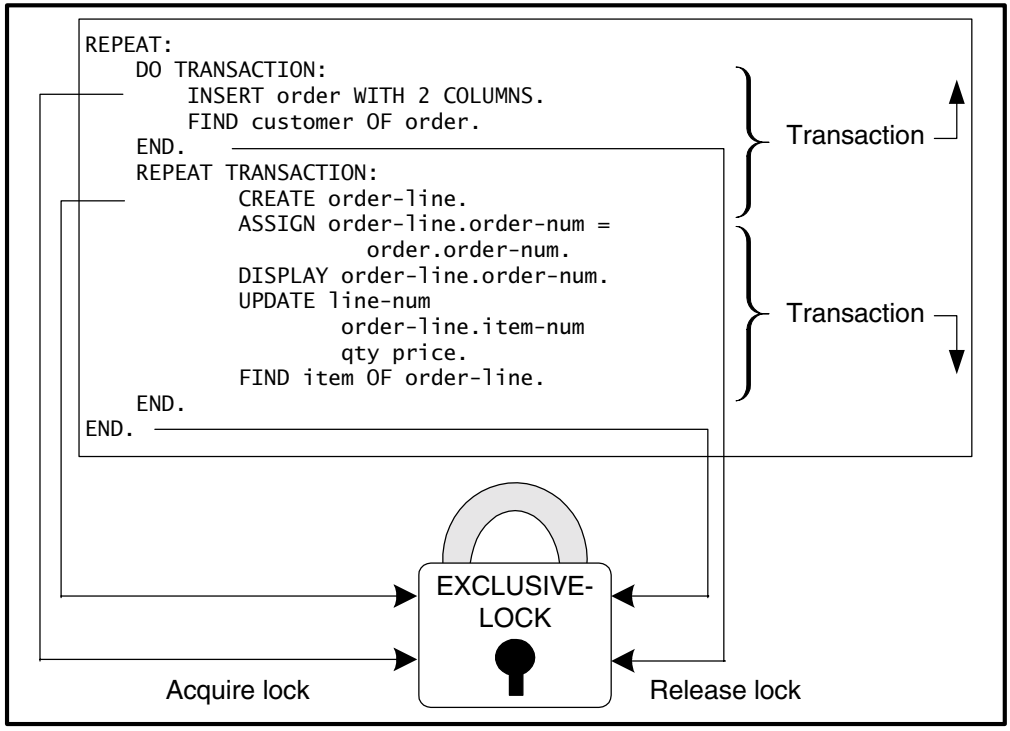


Figure 13–7: Making Transactions Smaller

Progress starts a transaction for each order and also for each order–line you enter. When you insert an order record at the start of the outer DO TRANSACTION block, Progress EXCLUSIVE–LOCKs that record. Progress releases that EXCLUSIVE–LOCK at the end of the transaction, which is at the end of the DO TRANSACTION block.

Now, other users can access the order record while you update order–line records. Each iteration of the REPEAT TRANSACTION block is a transaction. That means that the CREATE statement EXCLUSIVE–LOCKs the order–line record. Progress releases that lock at the end of the REPEAT TRANSACTION block, making the order–line record available to other users.

14

Providing Data Security

As a developer, you can provide your Progress applications with *application security* that prevents unauthorized users from running application procedures or accessing data in a database.

Progress provides the following levels of application security:

- *Connection security* ensures that only authorized users can connect to the database.
- *Schema security* ensures that only authorized users can modify table, field, and index definitions.
- *Compile-time security* ensures that only authorized users can compile procedures that perform specific data accesses.
- *Run-time security* ensures that only authorized users can run specific precompiled procedures.

The security administrator can define security for the first three levels (connection time, schema, and compile time) in the Data Dictionary. For more information on security administration, see the *Progress Database Administration Guide and Reference* and the *Progress Client Deployment Guide*. As the developer, you can provide run-time security as described in this chapter.

14.1 Progress User ID and Password

Each of the four levels of Progress application security requires user IDs and passwords. The following sections describe the conventions for Progress user IDs and passwords, as well as how to validate them.

14.1.1 User ID

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

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

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

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

14.1.2 Password

A *password* is a string of up to 16 characters that is associated with a user ID. When you add a password through the Data Dictionary, the password is encoded with the ENCODE function. Because ENCODE returns different values for uppercase and lowercase input, **all Progress passwords are case sensitive**.

When using ENCODE with passwords, PSC strongly recommends:

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

14.2 Validating Progress User IDs and Passwords

If the security administrator establishes a list of valid user IDs, then your application must prompt the user for a user ID and password at connection time. Typically, an application does this by running the standard Progress startup procedure, `_prostar.p`. This procedure, in turn, runs the standard Progress authentication procedure, `_login.p`, for each connected database. (*Authentication* is the process of verifying a user's identity.)

The `_prostar.p` procedure also prepares `_login.p` to run appropriately in the current application environment (character or graphical) and verifies that no connected databases have the logical name DICTDB. This allows `_prostar.p` to assign the same alias (DICTDB) to each connected database before calling `_login.p`. Then, `_login.p` can authenticate access as it is called for each different database using the same database name.

This is the _login.p procedure:

_login.p

```
DEFINE INPUT PARAMETER viewAsDialog AS LOGICAL NO-UNDO.

{ login.i }
DEFINE VARIABLE tries    AS INTEGER NO-UNDO.


IF USERID("DICTDB") <> "" OR NOT CAN-FIND(FIRST DICTDB._User) THEN
    RETURN.

DO ON ENDKEY UNDO, LEAVE:

    currentdb = LDBNAME("DICTDB").

    /* reset id and password to blank in case of retry */
    ASSIGN id = ""
           password = "".

    if viewAsDialog then do:

      DISPLAY currentdb WITH FRAME logindb_frame view-as dialog-box.

      UPDATE id password ok_btn cancel_btn help_btn
             WITH FRAME logindb_frame view-as dialog-box.
    end.
    else do:
      DISPLAY currentdb WITH FRAME login_frame.

      UPDATE id password ok_btn cancel_btn help_btn
             WITH FRAME login_frame.
    end.
    IF SETUSERID(id,password,"DICTDB") <> TRUE THEN DO:
        MESSAGE "Userid/Password is incorrect."
                VIEW-AS ALERT-BOX ERROR BUTTONS OK.
        IF tries > 1 THEN
            QUIT. /* only allow 3 tries*/
        tries = tries + 1.
        UNDO, RETRY.
    END.
END.
HIDE FRAME login_frame.
```

The _login.p procedure uses the Progress SETUSERID function to check the user ID and password that the user enters. The user has three tries to enter the correct user ID and password for each database. If the user fails to do so after three tries, Progress exits the user from the database. If the user ID and password combination is valid for the database, SETUSERID establishes that user ID for the connection.

The input parameter for _login.p allows it to display the authentication prompts either in a dialog box (viewAsDialog = TRUE) or in the frame of a separate window (viewAsDialog = FALSE). The _prostar.p procedure uses a separate window in graphical environments and the default window in character environments, so it always passes FALSE as an argument to _login.p.

As explained earlier, the _login.p procedure only works for a database with the DICTDB alias. (By default, this alias is assigned to the first database you connect to during a session.) If you want to avoid this restriction, you can create your own procedures, based on _prostar.p and _login.p, that pass an argument for the database name.

If the application does not run _prostar.p at connection time, or if the user bypasses _login.p (by pressing **END–ERROR** when prompted for the user ID and password), then the user is assigned the blank user ID. While blank user IDs can connect to the database, they cannot access data protected by compile-time and run-time security.

If you connect to a database dynamically using the CONNECT statement, you can use the User ID (–U) and Password (–P) connection parameters in the CONNECT statement, or you can use the SETUSERID function after the connection.

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

p-passts.p

```
DEFINE VARIABLE passwd AS CHARACTER FORMAT "x(16) LABEL "Password".
DEFINE VARIABLE success AS LOGICAL.
DEFINE VARIABLE user-id AS CHARACTER FORMAT  "x(32)" LABEL "User ID".

CONNECT mywork.

success = FALSE.
DO WHILE NOT success:
  MESSAGE "Enter a user ID and password for the database mywork.".
    SET user-id passwd BLANK.

  IF SETUSERID(user-id, passwd, "mywork")
  THEN success = TRUE.
  ELSE DO:
    BELL.
    MESSAGE "Invalid user ID and password; please try again.".
  END.
END.
```

14.3 Run-time Security

To ensure that only authorized users can run certain procedures in your application, you can provide run-time security to check the user ID of the user attempting to run a procedure.

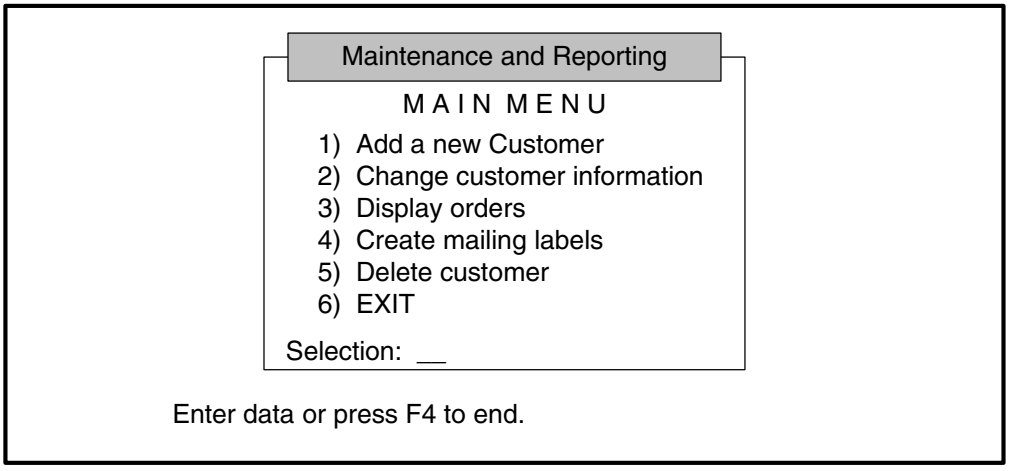
14.3.1 Checking for User IDs

This sections shows some examples of procedures that you can use to check for user IDs. The following procedure uses `_prostar.p`:

p-csmnu3.p

```
DEFINE VARIABLE selection AS INTEGER FORMAT "9".
RUN _prostar.p.

REPEAT:
  FORM  SKIP(2)    "    M A I N M E N U"
      SKIP(1)    "  1)  Add a new customer"
      SkIP(1)    "  2)  Change customer Information"
      SkIP(1)    "  3)  Display orders"
      SKIP(1)    "  4)  Create mailing labels"
      SKIP(1)    "  5)  Delete a customer"
      SKIP(1)    "  6)  EXIT"
  WITH CENTERED TITLE "Maintenance and Reporting".
  UPDATE SKIP(2) SPACE(1) selection AUTO-RETURN
      WITH SIDE LABELS.
HIDE.
      IF selection EQ 1 THEN RUN p-adcust.p.
  ELSE IF selection EQ 2 THEN RUN p-chcust.p.
  ELSE IF selection EQ 3 THEN RUN p-itlist.p.
  ELSE IF selection EQ 4 THEN RUN p-rept6.p.
  ELSE IF selection EQ 5 THEN RUN p-delcus.p.
  ELSE IF selection EQ 6 THEN QUIT.
  ELSE MESSAGE "Incorrect selection - please try again".
END.
```

This procedure defines user access by first running the _prostar.p procedure before displaying the following main menu in a character environment.



Suppose you want to define, on a per procedure basis, the individuals who can run each of the Maintenance and Reporting menu procedures. You can use the CAN–DO function to check the user ID(s) established by _prostar.p. The p-adcust.p procedure allows you to enter customer information:

p-adcust.p

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

If you want to limit the use of this procedure to users with a user ID of manager or salesrep, you can modify the procedure as follows to include security checking:

p-adcus2.p

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

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

Security Checking

p-adcust.p

The first part of `p-adcus2.p` handles security checking that ensures the user is authorized to run the procedure. The CAN–DO function compares the values listed in the parentheses against the user ID attempting to run the procedure. If the user ID does not match any of the values listed, the procedure displays a message and exits. If the user ID does match one of the values, the procedure continues executing.

The ID list you provide in the CAN–DO function is a comma-separated list of user ID tokens. You can use tokens to indicate specific users who have or do not have access. Table 14–1 lists the types of tokens you can specify.

Table 14–1: Values to Use for ID Lists

Value	Meaning
*	All users are allowed access.
user	This user has access.
!*user*	This user does not have access.
*string**	Users whose IDs begin with "*string*" have access
!*string**	Users whose IDs begin with "*string*" do not have access

For more information on the CAN–DO function, see the *Progress Language Reference*.

You can also use the USERID function to check user IDs in a procedure. Use this function when you want to allow **only one** user ID access to a procedure:

p-adcus3.p

```
IF USERID <> "manager"
THEN DO:
  MESSAGE "You are not authorized to run this procedure.".
  RETURN.
END.

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

Security Checking

p-adcust.p

If the user ID of the user running the procedure is not manager, the procedure displays a message and exits. If the user ID is manager, the procedure continues.

If you use either the CAN–DO function or the USERID function to compare the user ID of a user with one or more user IDs you include in a procedure, you must modify and recompile that procedure whenever you change the user IDs allowed access to it. You can avoid having to make these changes by building a permissions table for activities in your application.

NOTE: If you connect to more than one database, the USERID function requires a logical database name and the CAN–DO function requires a user ID qualification. For more information on multiple-database considerations, see Chapter 9, "Database Access."

14.3.2 Defining Activities-based Security Checking

Applications that you write probably break down into several areas or activities. For example, you may have one set of procedures that handles customer activities, and another set that handles inventory activities. For each set of activities, you may want to establish a valid group of users. To do this, you build a *permissions table*. To establish and use a permissions table, you must:

- Create a table that defines the kinds of access different users have to application activities.
- Include statements in application procedures to check the permissions table at run time.
- Write a procedure that can modify access permissions.

Creating an Application Activity Permissions Table

You create a permissions table within the Data Dictionary. This chapter calls the table permissions, but you can use any name you want. Each record in the permissions table contains at least two fields: an activity field and a can–run field. The activity field contains the name of the procedure and the can–run field contains a comma-separated list of the user IDs of those users who have permission to run the procedure. Normally, the primary index is built on the activity field.

After you create a permissions table, you must add a record for every application activity for which you want to provide security.

Adding Records to the Permissions Table

To create records for the permissions table, you must first determine the users and the activities. The users are identified by their user IDs, and the activities by specific procedures or subsystems.

In the following example, three user IDs—manager, salesrep, and inventory—are given permission to perform the following activities:

- Add new customers to the database by running `p-adcust.p`. Manager and salesrep have permission.
- Update records in the database by running `p-chcust.p`. Manager and salesrep have permission.
- Remove customer records from the database by running `p-delcus.p`. Manager has permission.
- The order report and mailing label procedures (`p-itlist.p` and `p-rept6.p`) are grouped into a subsystem called print. Manager and inventory have permission.

To add these application activities as records to the permissions table, you can write a simple Progress procedure:

p-prmsn.p

```
REPEAT:
  INSERT permissons.
END.
```

This procedure lets you add records of activities to the permissions table until you press **END–ERROR**.

Figure 14–1 shows records of application activities that you can add to the permissions table.

```
Activity        Can-Run
--------------  ------------------------
p-adcust.p      manager, salesrep
p-chcust.p      manager, salesrep
p-delcus.p      manager
print           manager, inventory
```

Figure 14–1: Sample Activity Permissions Entries

After you create these records of application activities, you must include statements in your procedures that check them at run time. After that, the security administrator is responsible for maintaining these records.

Including Security Checking in Procedures

When the user runs a procedure, you can check the permission for the activity associated with the procedure. Specifically, you:

- Find the activity record in the permissions table.
- Compare the permissions for the activity with the user ID of the user running the procedure.
- If there is a match for the user ID, allow access. Otherwise, display a message and exit from the procedure.

Figure 14–2 shows what happens when the user with user ID manager runs the `p-adcust.p` procedure.

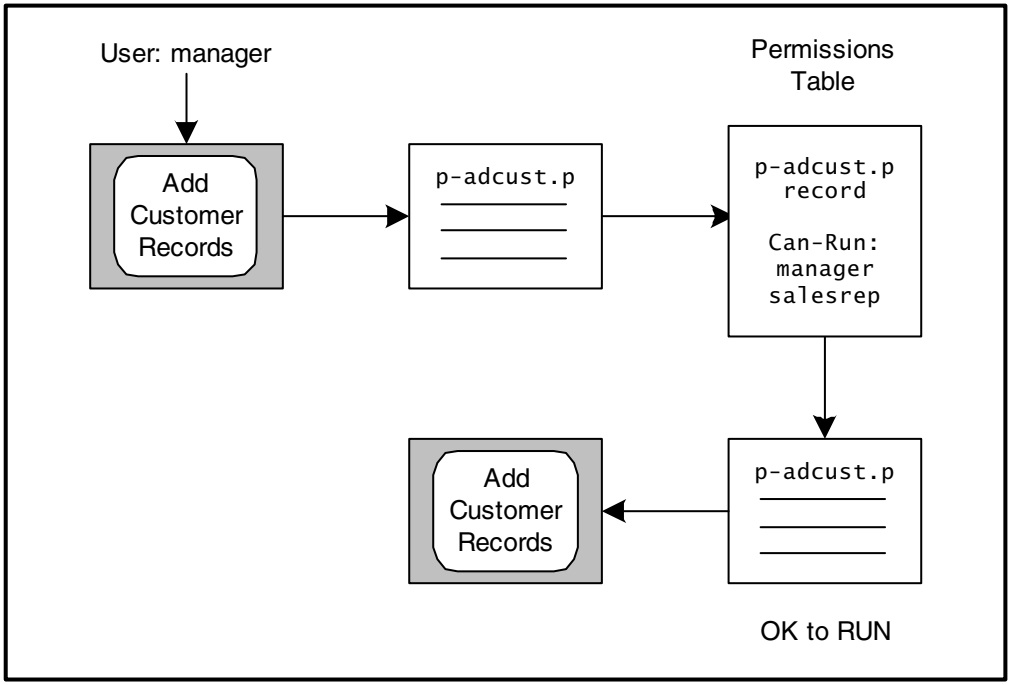


Figure 14–2: User with Permission to Run a Procedure

When the user ID and the permission defined in the `p-adcust.p` record in the permissions table match, the manager can run the procedure. Figure 14–3 shows what happens when the user with user ID inventory tries to runs `p-adcust.p`. Because there is no match, the procedure displays a message and the user cannot run the remaining code.

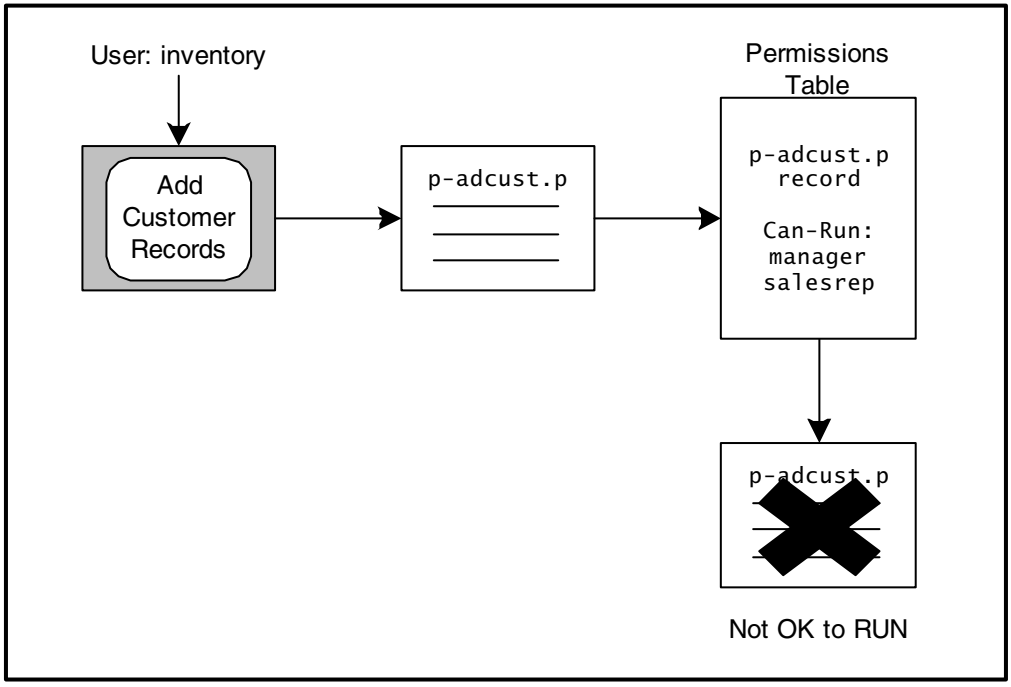


Figure 14–3: User Without Permission to Run a Procedure

You use the CAN–DO function to do security checking in your procedure. The procedure `p-adcus4.p` is a modified version of the `p-adcust.p` procedure that includes activity-based security checking:

p-adcus4.p

```
DO FOR permission:                                                    } Security
  FIND permission "p-adcust.p" NO-LOCK.                               } Checking
  IF NOT CAN-DO(can-run)                                              }
  THEN DO:                                                            }
    MESSAGE "You are not authorized to run this procedure.".          }
    RETURN.                                                           }
  END.                                                                }
END.

REPEAT:                                                               } p-adcust.p
  INSERT customer WITH 2 COLUMNS.                                     }
END.
```

The first part of this procedure makes sure the user is authorized to run the procedure. The FIND statement reads the permission record for the `p-adcust.p` procedure. The CAN–DO function compares the value of the can–run field in the record with the user ID of the user running the procedure. If the values do not match, the procedure displays a message and exits. If there is a match, the procedure allows the user to add customer records.

Progress checks privileges within a DO FOR block to ensure that the record read by the FIND statement is held only during that block, rather than during the entire procedure. In addition, the NO–LOCK option ensures that other users can access or update the permissions table while this procedure is running.

The part of the `p-adcust.p` procedure that does security checking is standard. For example, you could include the same security checking statements in the procedures `p-chcust.p` and `p-delcus.p`, if you change the name of the activity record being read in the permissions table:

p-delcs2.p

```
DO FOR permission:                                                    } Security
  FIND permission "p-delcs.p" NO-LOCK.                                } Checking
  IF NOT CAN-DO(can-run)                                              }
  THEN DO:                                                            }
    MESSAGE "You are not authorized to run this procedure.".          }
    RETURN.                                                           }
  END.                                                                }
END.

PROMPT-FOR customer cust-num.                                         } p-delcus.p
FIND customer USING cust-num.                                         }
DELETE customer.
```

The following procedure shows how you can modify a print procedure, such as `p-itlist.p`, and add security checking to it:

p-itlst2.p

```
DO FOR permission:                                                    }
  FIND permission "print" NO-LOCK.                                    }
  IF NOT CAN-DO(can-run)                                              }  Security
  THEN DO:                                                            }  Checking
    MESSAGE "You are not authorized to run this procedure.".          }
    RETURN.                                                           }
  END.                                                                }
END.

FOR EACH customer WHERE credit-limit > 50000 BY postal-code:          }
  DISPLAY name address city state postal-code credit-limit            }
    WITH SIDE-LABELS.                                                 }
  FOR EACH order OF customer:                                         }
    DISPLAY order WITH SIDE-LABELS.                                   }
    FOR EACH order-line OF order, item OF order-line:                 }  p-itlist.p
      DISPLAY line-num item.item-num item-name                        }
        qtyorder-line.price WITH SIDE-LABELS.                         }
    END.                                                              }
  END.                                                                }
END.
```

Here, the FIND statement reads the print record from the permissions table. The CAN–DO function compares the value of the can–run field with the user ID of the user running the procedure. If there is no match, the procedure displays a message and exits. If there is a match, the procedure displays order information.

Remember, there is no separate record in the permissions table for the `p-itlst2.p` procedure. However, you can use the record for the print activity to handle security for any procedure that you specify as a print activity. You can include the same security checking statements in any other procedure that you consider to be a print activity, such as `p-rept6.p`:

p-rept6.p

```
OUTPUT TO mail.lst.

FOR EACH customer WHERE curr-bal >= 1400 BY zip:
  PUT contact SKIP
      name SKIP
      address SKIP.
  IF address2 NE "" THEN PUT address2 SKIP.
  PUT city + ", " + st + "  " + STRING(zip," 99999") FORMAT "X(23)"
SKIP(1).
  IF address2 EQ "" THEN PUT SKIP(1).
END.
```

For application maintenance purposes, you might want to put security checking statements into an include file. Procedures that require security checking can simply include that file, passing the activity as an argument. The following is an example of a such an include file:

p-chkprm.i

```
DO for permissions:
  FIND permissions {1} NO-LOCK.
  IF NOT CAN-DO (can-run)
  THEN DO:
    MESSAGE "You are not authorized to run this procedure".
    RETURN.
  END.
END.
```

Protecting and Maintaining the Permissions Table

To protect the permissions established in the permissions table, you can provide the security administrator with the following procedures:

- A procedure that defines who can modify the permissions table. Initially, you can run this procedure and enter the user ID of the security administrator, as well as user IDs of those authorized to modify the permissions table.
- A security update procedure (for example, `p-secupd.p`) that the security administrator or other authorized users can run to modify permissions for specific procedures and functions.

To do security checking, these procedures require a record in the permissions table associated with the activity of maintaining security. Figure 14–4 shows an example security record in the permissions table.

Activity	Can-Run
security	*

Figure 14–4: Sample Security Record for Activity Permissions

When you create the security record, initialize the can–run field with an asterisk. This means that, initially, any user can run the security administration procedure (`p-secadm.p`). However, after you run it and enter the authorized user IDs, only the authorized users can change the security record:

p-secadm.p

```
FIND permission "security".
IF NOT CAN-DO(can-run)
  THEN DO:
    MESSAGE "You are not authorized to run this procedure.".
    RETURN.
END.

DISPLAY "Please enter one or more security ids"
  "for the security administrator,"
  "separating the ids with commas" SKIP(1).
UPDATE can-run.
```

Security Checking

The authorized users can also run the next procedure, `p-secupd.p`, which updates permissions for procedures and activities:

p-secupd.p

```
DO FOR permission:
  FIND permission "security" NO-LOCK.
  IF NOT CAN-DO(can-run)
  THEN DO:
    MESSAGE "You are not authorized to run this procedure.".
    RETURN.
  END.
END.

REPEAT FOR permission:
  PROMPT-FOR activity.
  FIND permission USING activity.
  UPDATE can-run.
END.
```

Security Checking

The first part of `p-secupd.p` checks the security record in the permissions table to make sure that the user is authorized to run the procedure. If the user is not authorized, the procedure displays a message and exits. Otherwise, the second part of `p-secupd.p` permits the user to modify the can–run field for a specified activity.

Figure 14–5 summarizes the security process developed for procedures and functions in an application.

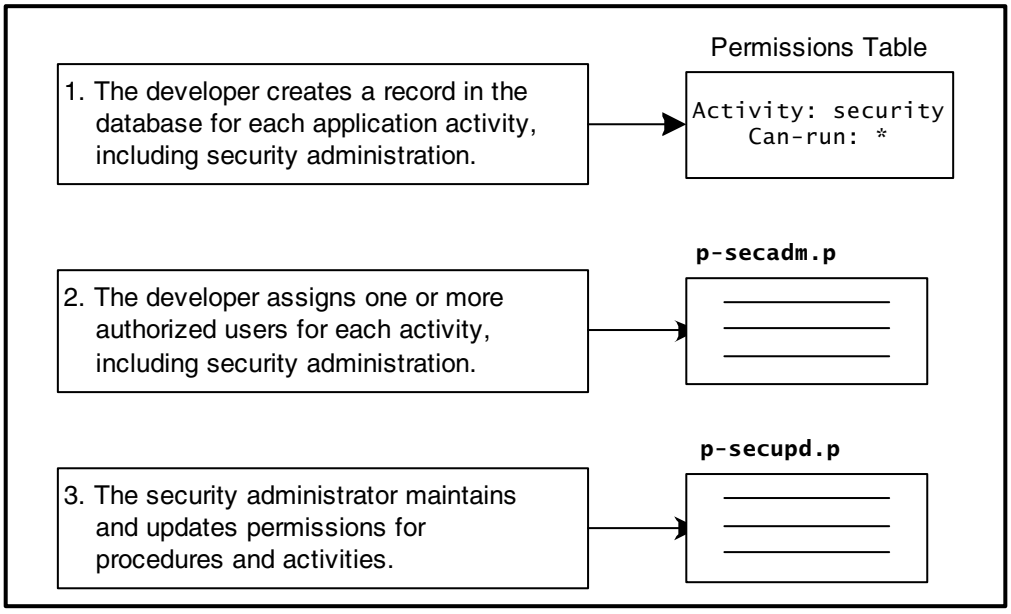


Figure 14–5: Summary of Run-time Security Process

15

Work Tables and Temporary Tables

The Progress 4GL provides two structures that can be used to manipulate table information: work tables and temporary tables. Temporary tables can be further sub-divided into static temporary tables and dynamic temporary tables.

This chapter explains:

- Work tables
- Temporary tables
- A comparison between these two temporary structures and database tables
- Dynamic temporary tables

15.1 Work Tables

This section explains how to use work tables to do the following:

- Produce categorized reports when you do not know the value of each possible category until you have made a pass through the table.
- Collect information from a number of records and then do calculations on that information.
- Do complex sorting on summarized information.
- Produce cross-tab reports.
- Simplify array manipulation, coding, and the use of dynamic variables.

NOTE: You cannot access work tables using SQL statements.

15.1.1 Characteristics of Work Tables

Work tables are to database tables what variables are to database fields. Like a variable, a work table:

- Must be defined in any procedure that uses it.
- Is a temporary structure stored in memory rather than in the database, and can either be local to a single procedure or shared among multiple procedures.
- Can be defined to be LIKE a database table (much like a variable can be defined to be LIKE a database field). Work tables also have the option to adopt the same dictionary validation specified for the corresponding database table.
- Does not have indexes, therefore you cannot do a FIND *work-table*. However, you can do a FIND FIRST *work-table*.

NOTE: Because Progress stores and processes work tables in memory, the number of work tables and the number of records in a work table that you can use is limited by the memory available on your system.

15.1.2 Producing Categorized Reports

Control break reports list information broken down into categories. For example, items in the sports database are assigned to catalog pages. Every item also has an on–hand value (how many items are in inventory) and a price. The `p-wrk1.p` procedure produces a report that lists the value of the inventory for each catalog page:

p-wrk1.p

```
FOR EACH item BREAK BY cat-page:
   ACCUMULATE price * on-hand (SUB-TOTAL BY cat-page).
   IF LAST-OF(cat-page)
   THEN DISPLAY cat-page ACCUM SUB-TOTAL BY cat-page (price * on-hand).
END.
```

This procedure produces the following report.

```
  Cat-Page          TOTAL
----------     ----------
         0     21,992.16
         1      8,489.92
         3        450.00
         4     12,377.40
         5     10,575.00
         6      1,445.52
         7      1,615.27
         8      1,367.10
        11      7,339.40
        12      2,250.00
        13        854.00
        14      2,074.83
        15        587.50
        16        256.86
        17        325.71
```

The control break (BREAK BY cat–page) in this procedure logically separates the item table into catalog pages. However, because you do not know what all the catalog pages are, or even how many catalog pages there are, the procedure must make two passes through the table:

- The BREAK keyword causes the first pass through the table. Progress sorts the item table and writes a sorted list to a temporary file (called a sort file). Progress determines at what points in the list the cat–page value changes.
- The second table pass occurs when Progress goes through the sort file, using the ROWIDs stored in that file to read item records from the item table. The ROWID is the internal database identifier that Progress associates with every database record.

Figure 15–1 shows how the procedure processes the item table.

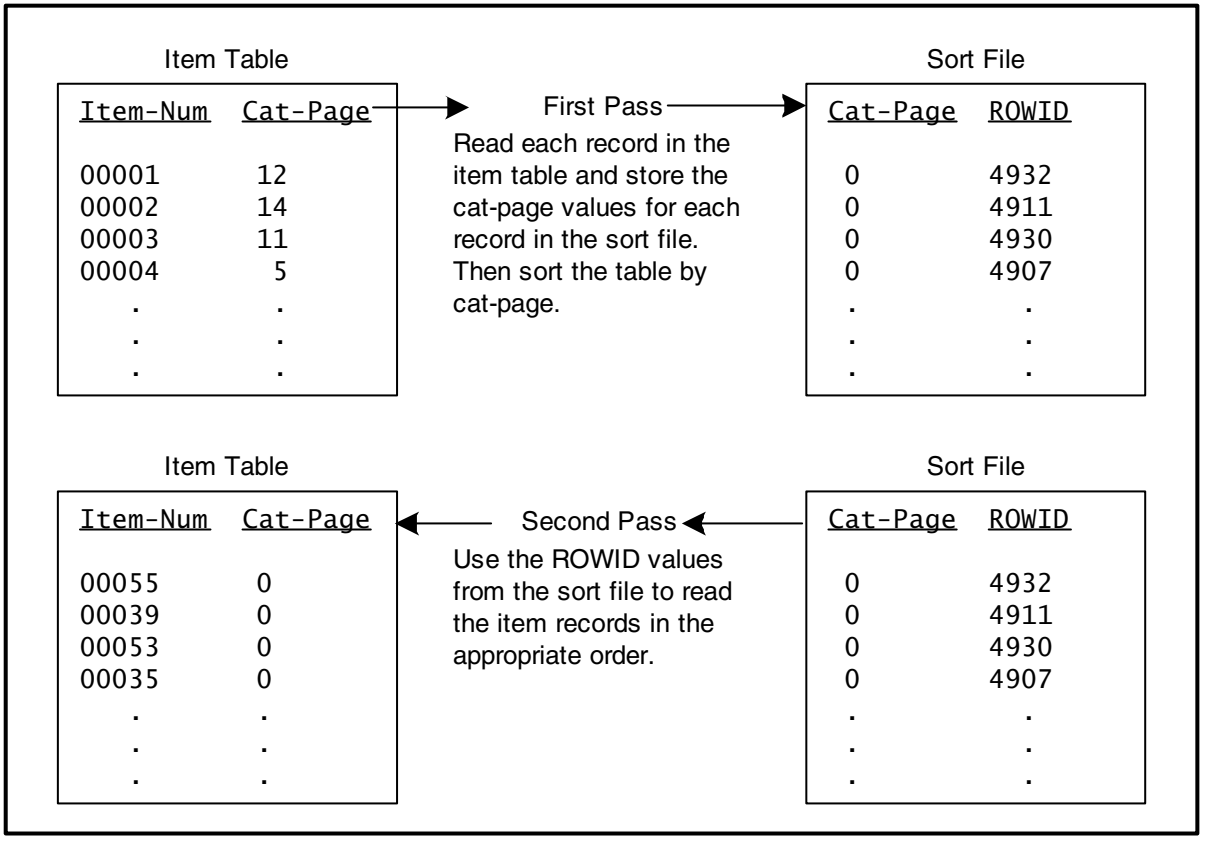


Figure 15–1: Processing a Table for a Categorized Report

The following procedure uses work tables to produce the same report:

p-wrk2.p

```
/* 1 */
DEFINE WORK-TABLE cpage
       FIELD w-cat-page LIKE item.cat-page
       FIELD w-inv-value AS DECIMAL FORMAT "->>>,>>>,>>9.99"
           LABEL "Inventory Value".

/* 2 */
FOR EACH item:
    FIND FIRST cpage WHERE cpage.w-cat-page = item.cat-page NO-ERROR.
    IF NOT AVAILABLE cpage
    THEN DO:
        CREATE cpage.
        cpage.w-cat-page = item.cat-page.
    END.
/* 3 */
    cpage.w-inv-value = cpage.w-inv-value + (item.price * item.on-hand).
END.

/* 4 */
FOR EACH cpage BY w-cat-page:
    DISPLAY w-cat-page w-inv-value.
END.
```

The numbers in the margin of the procedure correspond to the following activities:

1. The DEFINE WORK–TABLE statement defines a work table, cpage, that contains two fields, w–cat–page and w–inv–value. The w–cat–page field has the same definition as the cat–page field in the item table.

2. For each of the records in the item table, the procedure checks whether there is a cpage record for the page to which the item belongs. If there is not, the procedure creates a cpage record and stores the cat–page value in the cpage record.

3. The = (ASSIGNMENT) statement accumulates the value of the inventory for each catalog page by adding the inventory value of the current item to the existing inventory value for the page (w–inv–value).

4. The FOR EACH block sorts the cpage table and displays the inventory value of each catalog page.

Figure 15–2 shows the table processing this procedure performs.

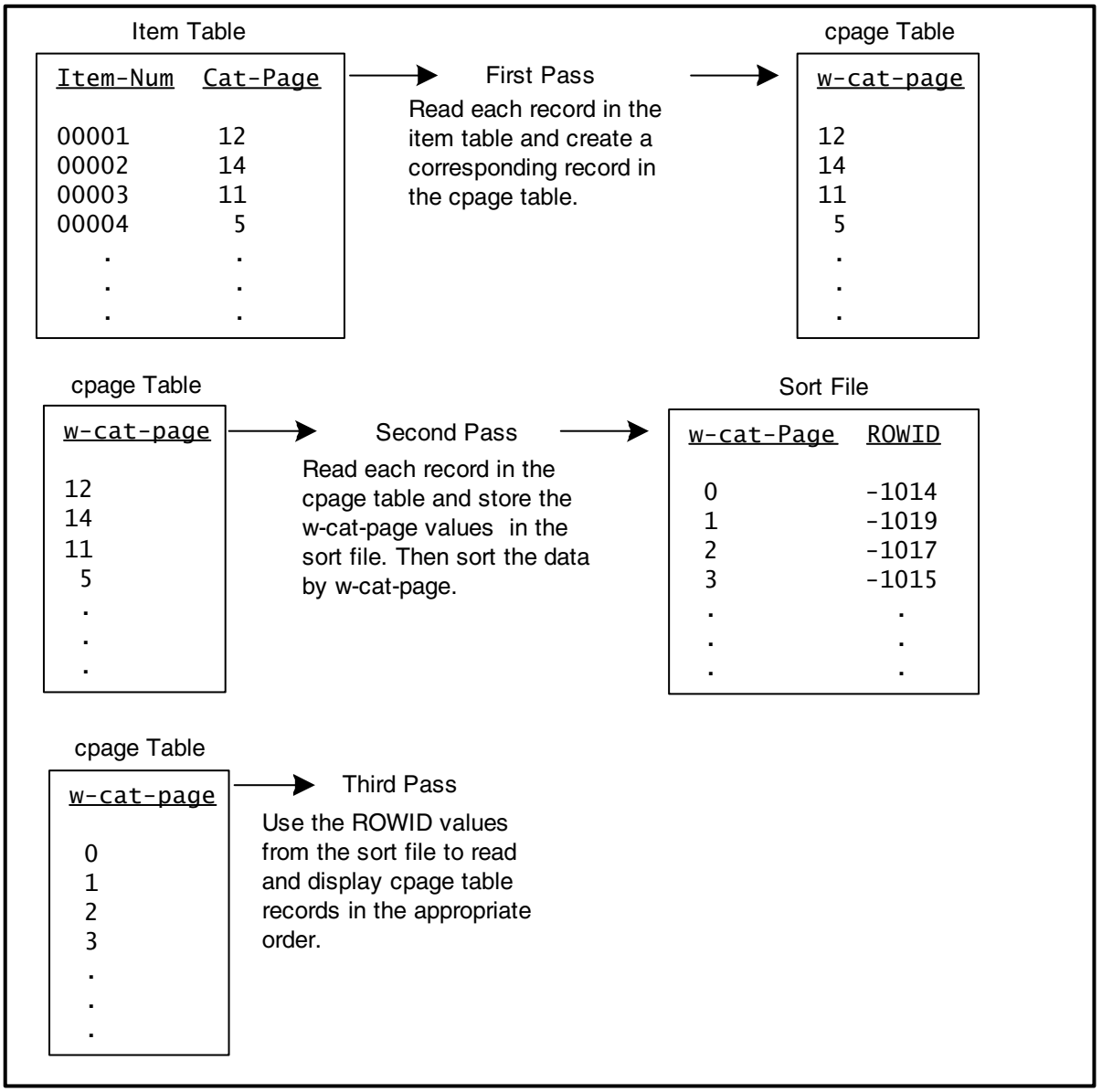


Figure 15–2: Using Work Tables to Produce a Categorized Report

The work table version of the procedure also makes a pass through the item table. But it does not need to first sort the entire item table since it can create new work table records for each new catalog page as it encounters that page. The procedure then makes a pass through the cpage

work table, building a sort file for that table. (Note that ROWIDs for work tables are always negative.) Finally, the procedure makes a second pass through the cpage table, retrieving the cpage records in the appropriate order.

You wonder: "If the `p-wrk1.p` procedure makes two table passes and the `p-wrk2.p` procedure makes three table passes, why is the work table procedure any better than the procedure that uses only database tables?" There are two reasons:

- The cpage table is far smaller than the item table (in a typical database). Therefore, a pass through the cpage table takes much less time than a pass through the item table.
- Progress performs all processing for work tables in memory so processing is faster than for database tables.

15.1.3 Using Work Tables to Collect and Manipulate Data

Suppose that you want to add another step to the report shown in the previous section. In the new report, you want to:

- Display the inventory value of each catalog page (as in the previous example).
- Display the inventory value for the item with the largest inventory value in each catalog page.

To prepare this report without using work tables, you have to use variables to keep track of the item quantities as well as the item inventory values. The number of variables you need depends on the number of pages in the catalog. You might not know this number when you write the procedure. When you use work tables, you can store item quantities and inventory values in fields in the work table.

p-wrk3.p

```
/* 1 */
DEFINE WORK-TABLE cpage
       FIELD w-cat-page LIKE item.cat-page
       FIELD w-inv-value AS DECIMAL FORMAT "->>>,>>>,>>9.99"
           LABEL "Inventory Value"
       FIELD w-item-value AS DECIMAL FORMAT ">>>,>>9.99"
           LABEL "Item Inv. Value"
       FIELD w-item-num LIKE item.item-num.

/* 2 */
FOR EACH item:
    FIND FIRST cpage WHERE cpage.w-cat-page = item.cat-page NO-ERROR.
    IF NOT AVAILABLE cpage
    THEN DO:
        CREATE cpage.
        cpage.w-cat-page = item.cat-page.
    END.

/* 3 */
    IF price * on-hand > w-item-value
    THEN DO:
        ASSIGN w-item-value = price * on-hand
               w-item-num = item.item-num.
    END.
/* 4 */
    cpage.w-inv-value = cpage.w-inv-value + (item.price * item.on-hand).
END.

/* 5 */
FOR EACH cpage BY w-cat-page:
    DISPLAY w-cat-page w-inv-value w-item-num w-item-value.
END.
```

The numbers in the margin of the procedure correspond to the following activities:

1. The DEFINE WORK–TABLE statement defines a work table, cpage, which contains four fields, w–cat–page (the work table catalog–page), w–inv–value (the work table inventory–value), w–item–value (the inventory value of a particular item), and w–item–num (the item number).

2. For each of the records in the item table, the procedure checks whether there is a cpage record for the catalog page to which the item belongs. If there is not, the procedure creates a record and stores the cat–page value in the cpage record.

3. If the value of the current item is greater than the value of any previous item for the same page, the ASSIGN statement stores the value and number of that item in the corresponding work table fields.

4. This statement accumulates the inventory value for the catalog page.

5. The FOR EACH block displays the inventory value of each page and the number and value of the highest value item on that page.

Using work tables allows you to sort on any value. For example, if you want to sort on the inventory value of the catalog pages, use the BY w–inv–value option on the FOR EACH block. Without work tables, you cannot do this sort as easily.

15.1.4 Using Work Tables for Complex Sorting

Progress uses indexes to order database records. Although work tables have no indexes, you can position work table records where you want because whenever you create a work table record, Progress places it after the record that it found last in that work table. This flexibility lets you keep the work table records in a particular order without having to sort later in the procedure. For example, `p-wrk4.p` is a modified version of the `p-wrk3.p` procedure:

p-wrk4.p

```
DEFINE WORK-TABLE cpage
   FIELD w-cat-page LIKE item.cat-page
   FIELD w-inv-value AS DECIMAL FORMAT "->>>,>>>,>>9.99"
       LABEL "Inventory Value"
   FIELD w-item-value AS DECIMAL FORMAT ">>>,>>9.99"
       LABEL "Item Inv. Value"
   FIELD w-item-num LIKE item.item-num.

FOR EACH item:
   FIND FIRST cpage WHERE cpage.w-cat-page >= item.cat-page NO-ERROR.

   IF NOT AVAILABLE cpage OR cpage.w-cat-page > item.cat-page
   THEN DO:
      FIND PREV cpage NO-ERROR.
      CREATE cpage.
      cpage.w-cat-page = item.cat-page.
   END.

   IF price * on-hand > w-item-value
   THEN DO:
      ASSIGN w-item-value = price * on-hand
             w-item-num = item.item-num.
   END.

   cpage.w-inv-value = cpage.w-inv-value + (item.price * item.on-hand).
END.

FOR EACH cpage:
   DISPLAY w-cat-page w-inv-value w-item-num w-item-value.
END.
```

This procedure uses the following logic to maintain the cpage work table in catalog page order:

1. The FIND FIRST statement tries to find a cpage record whose catalog page is the same or greater than that of the current item.

2. If the catalog page of the cpage record is larger than that of the current item, or if the cpage record does not exist, Progress finds the previous cpage record (if any) and creates the new cpage record after that record in the work table. If it finds a cpage record with the same product line as the item, it does not create a new cpage record, but uses the existing one.

3. The procedure computes the necessary field values for the current cpage record.

Figure 15–3 shows how this procedure keeps the cpage work table in order by product line.

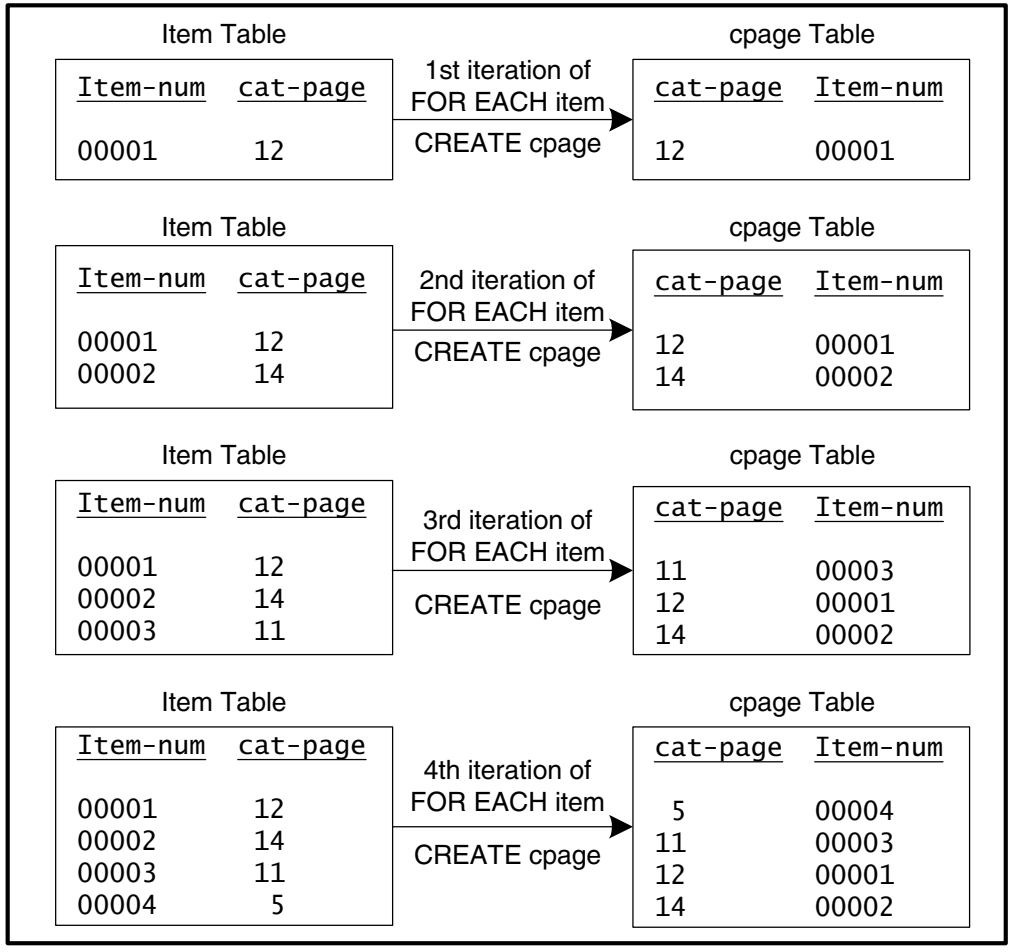


Figure 15–3: Maintaining Work Tables in Sorted Order

By maintaining the cpage table in catalog page order as the procedure creates new cpage records, there is no need to sort the table in the final FOR EACH block. When that block displays the cpage records, they are already sorted by catalog page.

There is another advantage to using this method of sorting as opposed to the method used in `p-wrk3.p`. Figure 15–4 illustrates this advantage.

p-wrk3.p
p-wrk4.p

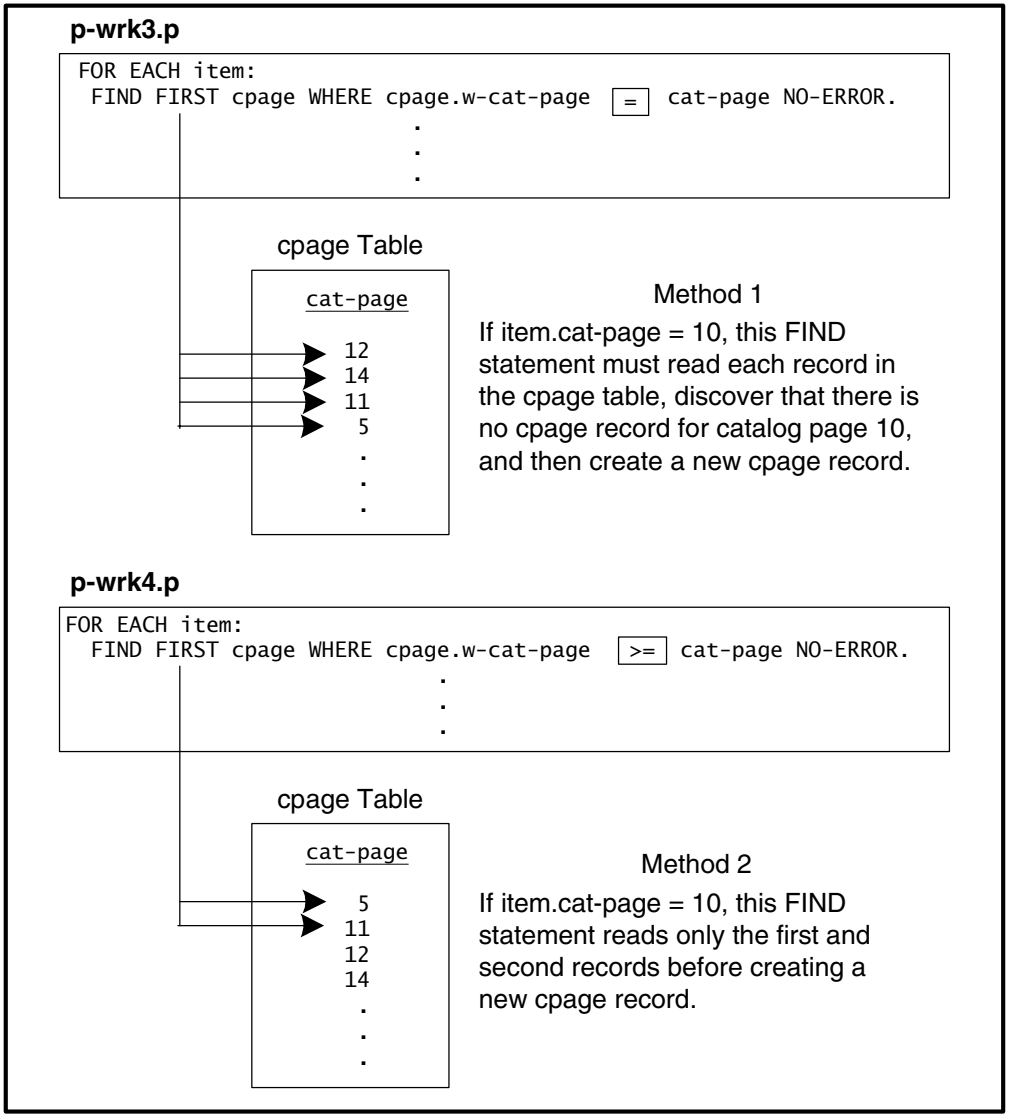


Figure 15–4: Comparing Two Sorting Methods

In Method 2, because the records in the cpage table are already in sorted order, the find is much more efficient than in Method 1. For an item with a catalog page that already exists in the cpage table, both methods are equally efficient, on average.

15.1.5 Using Work Tables for Cross-tab Reports

A *cross-tab* report typically lists information in tabular (or spreadsheet) format. For example, suppose you want a report that breaks down order values by catalog page and sales rep. The report might look like the following.

Order Value by Catalog Page and Sales Rep

Page	BBB	DKP	SLS	Others
0	3,685.65	167,655.18	52,984.94	269,074.77
1	0.00	112,699.96	45,227.75	232,079.66
3	0.00	10,305.00	3,195.00	14,130.00
4	11,830.00	98,959.22	27,976.06	232,669.67
5	0.00	10,200.00	1,125.00	24,375.00
6	0.00	7,073.82	0.00	12,946.92
7	0.00	10,536.06	1,757.90	14,556.49
8	0.00	3,027.15	0.00	6,779.70
11	0.00	14,299.75	4,025.90	27,031.15
12	0.00	29,247.55	15,594.30	204,010.05
13	0.00	14,213.00	11,956.00	35,380.00
14	0.00	23,880.32	6,070.77	53,188.68
15	0.00	4,162.50	0.00	8,387.50
16	0.00	442.37	4,523.59	7,434.67
17	0.00	7,274.19	496.32	4,079.13

This report lists the order value for each of the catalog page for each of three sales reps.

There are two ways to produce this report without using work tables:

- Usually, your report has more rows than columns or more columns than rows. If the number of rows is larger than the number of columns, sort the records represented by each row (for example, catalog page) and then accumulate values acquired during that sort into an array whose elements represent each of the columns (for example, the order value for each sales rep).

- Displays results whenever Progress encounters a new catalog page value.

Using work tables simplifies the task of producing cross-tab reports: you do not have to sort the data and you do not have to be concerned with the dimensions of the table as you do when using arrays. The `p-wrk5.p` procedure produces this report using work tables:

p-wrk5.p

```
DEFINE WORK-TABLE cpage
   FIELD w-cat-page LIKE item.cat-page FORMAT ">9"
   FIELD w-ord-value AS DECIMAL FORMAT "->>>,>>9.99" EXTENT 4.

DEFINE VARIABLE rep-code AS INTEGER.
DEFINE VARIABLE rep-names AS CHARACTER INITIAL "BBB,DKP,SLS".

FOR EACH order, customer OF order, EACH order-line OF order,
         item OF order-line:

   FIND FIRST cpage WHERE cpage.w-cat-page = item.cat-page NO-ERROR.

   IF NOT AVAILABLE cpage
   THEN DO:
      CREATE cpage.
      cpage.w-cat-page = item.cat-page.
   END.

   rep-code = LOOKUP(customer.sales-rep, rep-names).
   IF rep-code = 0
   THEN rep-code = 4.
   w-ord-value[rep-code] = w-ord-value[rep-code] +
                           (order-line.price * order-line.qty).
END.

FOR EACH cpage BY cpage.w-cat-page:
  FORM HEADER "Page" "BBB" AT 15 "DKP" AT 29 "SLS" AT 43 "Others" AT 54
      WITH TITLE "Order Value by Catalog Page and Sales Rep" NO-LABELS.

   DISPLAY w-cat-page w-ord-value.
END.
```

15.2 Temporary Tables

This section contains the following information on temporary tables:

- Characteristics of temporary tables
- Defining temporary tables
- Using database tables, temporary tables, and work tables
- Comparison of database tables, temporary tables, and work tables
- Defining indexes for temporary tables
- Shared and global shared temporary tables
- Temporary tables as parameters
- Example of a procedure that defines a temporary table

15.2.1 Characteristics of Temporary Tables

Temporary tables are database tables that Progress stores in a temporary database. You define temporary tables as you do work tables. Unlike work tables, temporary tables have indexes and perform at the speed of regular database tables. Unlike regular database tables, which are permanent and which multiple users can access simultaneously, temporary tables last only for the duration of the procedure that defines them (or for the duration of the Progress session, if you make them GLOBAL), and allow only one user at a time to access them. Finally, temporary tables are private, visible only to the user (process) that creates them. In short, if you want to sort, search, and process data for a duration not longer than the Progress session, use temporary tables.

Progress stores temporary tables and temporary files in the same directory—by default, your current working directory. You can change this directory by using the Temporary Directory (–T) startup parameter.

Temporary tables require less memory than large work tables. The Buffers for Temporary Tables (–Bt) startup parameter allows you to set the size of the buffer that Progress uses to control temporary tables. For more information on the –Bt parameter, see the *Progress Startup Command and Parameter Reference*.

NOTE: You cannot access temporary tables using SQL statements.

15.2.2 Defining a Temporary Table

You define a temporary table by using the DEFINE TEMP–TABLE statement. For example, the following statement defines temporary table Temp–Cust that inherits the field definitions of table Customer, which must belong to a database that is connected.

```
DEFINE TEMP-TABLE Temp-Cust LIKE Customer.
```

Using the DEFINE TEMP–TABLE statement, you can define a temporary table that:

- Inherits the field definitions of an existing database table, or has the fields you define explicitly
- Inherits the index definitions of an existing table, or has the indexes you define explicitly
- Lasts only as long as the procedure that creates it, or as long as the Progress session
- Is private to a single procedure, or shared among several procedures

For more information on the DEFINE TEMP–TABLE statement, see its reference entry in the *Progress Language Reference*.

15.2.3 Using Database, Temporary, and Work Tables

Database tables provide data storage in a permanent database and can be accessed by single or multiple users. However, if you want to process data for the duration of a procedure in complete privacy, use temporary tables or work tables. Temporary tables have several advantages over work tables:

- Temporary tables perform like database tables.
- Because temporary tables have indexes, you can use statements that require indexes, such as FOR EACH or a simple FIND.
- Temporary tables are not limited to the value of the buffer size defined by the –Bt parameter. Progress stores any overflow of temporary table records to disk. Work tables, however, are limited by the amount of memory available on your system.

- You can pass a temporary table as a parameter when you call a procedure. In fact, if you want to call a Progress AppServer passing it a table of data, you must pass the data as a temporary table. There is no other way. For more information on Progress AppServers, see *Building Distributed Applications Using the Progress AppServer*. For more information on temporary tables as parameters, see the "Temporary Tables As Parameters" section in this chapter.

There are some situations where you might want to use work tables:

- If you want to process a small number of records (for example, 50 or less) and you do not have to order or sort the data.
- If your existing applications use work tables and you want to maintain compatibility with these applications. Note that if you rewrite existing applications (that use work tables) so that they now use temporary tables, remember that temporary tables inherit index layouts defined for the LIKE table. If you do not want this result, you must define your own indexes.

Table 15–1 compares database tables, temporary tables, and work tables.

Table 15–1: Database, Temporary, and Work Tables *(1 of 2)*

Progress Feature	Database Table	Temporary Table	Work Table
Database manager	Progress uses the database manager in a single- or multi-user mode when working with database tables.	Progress uses the database manager in single-user mode to access the temporary database that contains the temporary tables.	Progress does not use the database manager when working with work tables. Work tables are stored in memory.
Indexes	You can define indexes for database tables.	You can define indexes for temporary tables.	You cannot define indexes for work tables.
Record deletion	To remove database records from the database, you must explicitly delete them with the DELETE statement.	If you do not explicitly delete records in the temporary table, Progress discards those records and the temporary table itself at the end of the procedure or session that initially defined the temporary table.	If you do not explicitly delete records in a work table, Progress discards those records and the work table itself at the end of the procedure that initially defined the work table.

Table 15–1: Database, Temporary, and Work Tables *(2 of 2)*

Progress Feature	Database Table	Temporary Table	Work Table
Record movement	A database table can move data between a record buffer and a database record.	A temporary table cannot move data between a record buffer and a database record.	Work tables cannot move data between a record buffer and a database record.
Multi-user record access	Multiple users can access the same database table at the same time.	Users do not have access to each other's temporary tables.	Users do not have access to each other's work tables.
Transactions	Progress automatically starts transactions for certain blocks and statements. Database tables use the before-image file to roll back changes.	You must start transactions explicitly. Temporary tables use the local before-image file to roll back changes.	You must start transactions explicitly. Work tables use the local before-image file to roll back transactions.
Buffer size	Use the –B parameter to specify the database table buffer size.	Use the –Bt parameter to specify the temporary table buffer size.	Available memory determines the work table buffer size.
Location	Database tables are stored directly in the database.	Temporary tables are stored in the directory where Progress stores temporary files.	Work tables are stored in memory.
Special data type support	You can store RAW and RECID fields, but not ROWID fields, in a database table.	You can store RAW, ROWID, and RECID fields in a temporary table.	You can store RAW, ROWID, and RECID fields in a work table.
Triggers and Events	Database tables support triggers and events.	Temporary tables do not support triggers and events.	Work tables do not support triggers and events.

The following sections describe in greater detail the similarities and differences between temporary tables and work tables, as well as how to define indexes for temporary tables.

15.2.4 Similarities Between Temporary and Work Tables

Temporary tables and work tables have the following similarities:

- You create temporary tables and work tables with a DEFINE statement, and they are scoped to the procedures that define them. When you specify the LIKE *table* option, both inherit the attributes of the specified database table (*table*). This optionally includes all Data Dictionary validation for the table (using the VALIDATE option). However, temporary tables and work tables do not inherit schema triggers from the database table.
- The NEW SHARED and SHARED options allow two or more procedures to access the same temporary tables and work tables.
- Update statements, such as UPDATE, CREATE, and DELETE, that reference a temporary table or a work table do not require a transaction to be active. If you want a temporary table or a work table to participate in a transaction, you must explicitly start a transaction for it.
- The UNDO and NO–UNDO options allow you to decide whether to incur the overhead associated with transaction and nested UNDO processing.
- Whenever a Progress session ends due to a crash or a QUIT statement, Progress deletes all records in temporary tables (and the temporary database itself) and all records in work tables.

15.2.5 Differences Between Temporary and Work Tables

Temporary tables and work tables have the following differences:

- When you define temporary tables with the LIKE option and do not explicitly specify indexes for them, they inherit index information from the database table. Work tables do not have index support.
- While temporary table records are stored in a temporary database on disk, work table records are stored in memory. Note that for a relatively small application with a sufficiently large buffer allocation, Progress does not have to perform any disk I/O for temporary tables. To specify buffer size for temporary tables, use the –Bt parameter.
- The performance of a temporary table is comparable to the performance of a database table. If you select the NO–UNDO option, UPDATE performance increases for temporary tables. If you do not use NO–UNDO, then changes made to the temporary table during a transaction are logged to the local before image (LBI) file.

- The FIND statement for temporary tables uses the same cursor concept and choose-index logic that are used for regular database tables. When a temporary table record is created, its logical position in the table depends on the values of its key fields. Its logical position in the table can be changed by changing the key values.

- Because temporary tables have indexes similar to regular database tables, they can perform fast sequential and random access to temporary table records using one or more indexes. Since work tables have no index support, all record access is performed with a sequential search.

- If you use the Record phrase OF option to join a work table, the other table in the join (database or temporary) must have a unique index for the common fields. For more information on joins and the OF option, see Chapter 9, "Database Access."

- You can pass temporary tables (but not work tables) as parameters to procedures. In fact, if you want to call a Progress AppServer passing it a table of data, you must pass the table of data as a temporary table. There is no other way. For more information on Progress AppServers, see *Building Distributed Applications Using the Progress AppServer*. For more information on temporary tables as parameters, see the "Temporary Tables As Parameters" section in this chapter.

The following program illustrates using a temporary table for non-database data:

p-ttnodb.p

```
/* p-ttnodb.p - static temp-table with no database */
DEFINE VARIABLE i AS INTEGER.
DEFINE VARIABLE btth AS HANDLE.
DEFINE VARIABLE browseHndl AS WIDGET-HANDLE.
DEFINE TEMP-TABLE t-field NO-UNDO
   FIELD fInt AS INTEGER
   FIELD fChar AS CHARACTER
   FIELD fDec AS DECIMAL
   FIELD fLog AS LOGICAL.

DO i=1 TO 10:
    CREATE t-field.
    ASSIGN fInt = i
           fChar = "char" + STRING(i)
           fDec = i
           fLog = NO.
END.

DEFINE QUERY qt-field FOR t-field SCROLLING.
OPEN QUERY qt-field FOR EACH t-field.
btth = BUFFER t-field:HANDLE.
DEFINE FRAME F1
    WITH SIZE 82 BY 29 .

CREATE BROWSE browseHndl
    ASSIGN
        FRAME = FRAME F1:HANDLE
        X = 2
        Y = 2
        WIDTH = 80
        DOWN = 18
        TITLE =  "Temp Table is not based on DB table"
        QUERY = QUERY qt-field:HANDLE
        SENSITIVE = YES
        VISIBLE = TRUE
        READ-ONLY = FALSE
        SEPARATORS = YES.
browseHndl:ADD-COLUMNS-FROM(btth).

ENABLE ALL WITH FRAME F1.
WAIT-FOR CLOSE OF CURRENT-WINDOW.
```

15.2.6 Defining Indexes for Temporary Tables

You can define indexes explicitly for temporary tables or you can inherit indexes from existing database tables. When you define a temporary table to be LIKE a database table, the temporary table inherits index layouts from the LIKE table according to these rules:

- If you do not specify any index information, the temporary table inherits all the index layouts defined for the LIKE table.
- If you specify one or more USE INDEX options, the temporary table inherits only the explicitly named index layouts. If one of these indexes is the primary index of the LIKE table, it becomes the primary index of the temporary table, unless you explicitly specify AS PRIMARY for another index.
- If you specify one or more INDEX options, the temporary table also inherits index layouts named by a USE INDEX option. If the new index is specified as PRIMARY, it becomes the primary index of the temporary table.
- If you specify any index information, then no more than one primary index can be specified. If you do not specify a primary index, then the first index specified is treated as the primary index.
- If you do not specify any index information, Progress creates a default primary index which sorts records in the order they are entered.

The following is a summary of the rules that determine the primary index of a temporary table.

1. If you specify AS PRIMARY or IS PRIMARY for an index, then that is the primary index.
2. If you do not specify AS PRIMARY or IS PRIMARY, but you inherit the primary index from a database table, that index is the primary index.
3. If neither Rule 1 nor Rule 2 applies, then the first index you specify, if any, is the primary index, unless it is a word index, in which case it can’t be primary.
4. If you have not specified any index information, then Progress creates a default primary index that sorts the records in entry order.

15.2.7 Tailoring Temporary Table Visibility and Life Span

Depending on how you specify the NEW, GLOBAL, and SHARED options of the DEFINE TEMP–TABLE statement, you can tailor the visibility and life span of the temporary table to the needs of your application.

You can define a temporary table that is visible to:

- Only the procedure that creates it
- The procedure that creates it, and the procedures that the creating procedure calls
- All procedures in the Progress session

You can define a temporary table to live as long as:

- The procedure that creates it
- The current Progress session

Table 15–2 displays the DEFINE TEMP–TABLE options you should specify for each type of temporary table, and the resulting temporary table's visibility and life span.

Table 15–2: Temporary Table Options, Visibility, and Life Span

Type	Options	Visibility	Life Span
Not Shared and Not Global	DEFINE...	The creating procedure	The creating procedure
Shared	DEFINE NEW SHARED... (in the creating procedure) DEFINE SHARED... (in the procedures that the creating procedure calls that want to share the temporary table)	The creating procedure, and the procedures that the creating procedure calls that want to share the temporary table	The creating procedure
Global	DEFINE NEW GLOBAL... (in the creating procedure, and in other procedures in the Progress session that want to access the temporary table)	All procedures in the Progress session	The Progress session

15.2.8 Temporary Tables As Parameters

Temporary table parameters strongly resemble other parameters, but have a few differences. This section discusses the similarities and differences.

Similarities

Temporary table parameters resemble other procedure parameters in several ways:

1. In the calling procedure, you define the parameter type as INPUT, OUTPUT, or INPUT–OUTPUT. INPUT is the default. Here is the syntax for a defining a temporary table parameter in the calling procedure's RUN statement.

   SYNTAX

   ```
   {
         [ INPUT ]  TABLE name
     |  { INPUT-OUTPUT | OUTPUT } TABLE name [ APPEND ]
   }
   ```

   NOTE: For information on the APPEND option, see the last item in the "Differences" section.

2. In the called procedure, you define the parameter type as INPUT, OUTPUT, or INPUT–OUTPUT. INPUT is the default. Here is the syntax for defining a temporary table parameter in the called procedure's DEFINE PARAMETER statement.

   SYNTAX

   ```
   DEFINE
     {     { [ INPUT ] | INPUT-OUTPUT | }
              PARAMETER TABLE FOR name [ APPEND ]
       |  OUTPUT PARAMETER TABLE FOR name
     }
   ```

   NOTE: For information on the APPEND option, see the "Differences" section.

3. The parameter types (INPUT, OUTPUT, or INPUT–OUTPUT) you define in the calling procedure and the called procedure must match.

4. Data moves between the calling procedure and the called procedure depending on the parameter type. INPUT parameter data moves from the calling procedure to the called procedure, OUTPUT parameter data moves from the called procedure to the calling procedure, and INPUT–OUTPUT parameter data moves from the calling procedure to the called procedure, then back to the calling procedure. Some parameters send data, and others receive data. Table 15–3 identifies which parameters send data and which receive data, based on parameter type (INPUT, OUTPUT, or INPUT–OUTPUT) and role (calling procedure or called procedure).

Table 15–3: Identifying Sender and Receiver Parameters

Parameter Type	Calling Procedure	Called Procedure
INPUT	Sender	Receiver
OUTPUT	Receiver	Sender
INPUT–OUTPUT	Sender and Receiver	Sender and Receiver

5. Data moves between the parameters of the calling procedure and the parameters of the called procedure at precise times. INPUT parameter data moves just before the called procedure begins to execute. OUTPUT parameter data moves just after the called procedure's RETURN statement executes. INPUT–OUTPUT parameter data moves like input parameter data, then like output parameter data.

6. You can only use the APPEND option for an INPUT parameter in the calling procedure or an OUTPUT parameter in the called procedure.

Differences

The following special rules apply to temporary table parameters:

1. The calling procedure and the called procedure must have separate temporary tables.

   NOTE: If the calling procedure and the called procedure use the same shared or global temporary table as a parameter (which might occur when code is automatically generated), then no data moves, and using the APPEND option (even when the compiler allows it) causes a run time error.

2. The *signatures* of the two temporary tables must match. This means that the calling procedure's temporary table and the called procedure's temporary table must match with respect to the number of columns, the data type of each column, and the number of extents of each column (for columns with extents).

3. The calling procedure's temporary table and the called procedure's temporary table need not have matching indexes.

4. Progress does not automatically convert data between "similar" data types, such as integer and decimal.

5. At data movement time, Progress overlays any existing data in the receiving temporary table (OUTPUT and INPUT–OUTPUT parameters in the calling procedure, and INPUT and INPUT–OUTPUT parameters in the called procedure) — unless you define the receiving temporary table with the APPEND option. If you use the APPEND option, Progress appends the incoming data to the existing data.

15.2.9 Temporary Table Example

The following procedure creates a temporary table (temp–item) that stores the total inventory value (the sum of item.price * item.on–hand) for each item's catalog page (item.cat–page). It builds a temp–item with two indexes: one that sorts the table in ascending order by catalog page and a second that sorts the table in descending order by inventory value.

This shows how you can use a temporary table to store a calculated result from the database, and efficiently report the same result according to different sorting and selection criteria:

p-tmptbl.p

```
DEFINE TEMP-TABLE temp-item
   FIELD cat-page LIKE item.cat-page
   FIELD inventory LIKE item.price LABEL "Inventory Value"
   INDEX cat-page IS PRIMARY cat-page ASCENDING
   INDEX inventory-value inventory DESCENDING.

DEFINE VARIABLE cutoff LIKE item.price.
DEFINE VARIABLE inv-price LIKE item.price.
DEFINE VARIABLE report-type AS INTEGER INITIAL 1.

FORM
   cutoff LABEL "Inventory Lower Cutoff for each Catalog Page"
      AT ROW 1.25 COLUMN 2
   report-type LABEL "Report Sorted ..."  AT ROW 2.25 COLUMN 2
      VIEW-AS RADIO-SET RADIO-BUTTONS
        "By Catalog Page", 1, "By Inventory Value", 2
   WITH FRAME select-frame SIDE-LABELS WIDTH 70 TITLE "Specify Report ..."
      VIEW-AS DIALOG-BOX.

FOR EACH item BREAK BY item.cat-page:
   ACCUMULATE price * on-hand (SUB-TOTAL BY item.cat-page).
   IF LAST-OF(item.cat-page) THEN DO:
      inv-price = ACCUM SUB-TOTAL BY item.cat-page (price * on-hand).
      CREATE temp-item.
      temp-item.cat-page = item.cat-page.
      inventory     = inv-price.
   END.
END. /* FOR EACH item */

REPEAT:

   UPDATE cutoff report-type WITH FRAME select-frame.

   IF report-type = 1 THEN
      FOR EACH temp-item USE-INDEX cat-page WITH FRAME rpt1-frame:
         IF inventory >= cutoff THEN
            DISPLAY temp-item.cat-page inventory.
      END.
   ELSE
      FOR EACH temp-item USE-INDEX inventory-value WITH FRAME rpt2-frame:
         IF inventory >= cutoff THEN
            DISPLAY temp-item.cat-page inventory.
      END.
END. /* REPEAT */
```

15.3 Dynamic Temporary Tables

It is sometimes difficult to know the exact nature of a temp–table that may be needed at runtime, especially since there are dynamic buffers for ordinary database tables. A dynamic temp–table allows users to create a temp–table on the fly at runtime without having to do a compile-time DEFINE TEMP–TABLE.

15.3.1 Static and Dynamic Temp Tables

A static temp–table is created with the DEFINE TEMP–TABLE statement and configured at compile time. A dynamic temp–table is created with the CREATE TEMP–TABLE statement which creates a dynamic handle to a temp–table object whose fields and buffers are configured at runtime. It is possible to associate a dynamic BUFFER object with a static temp–table which allows a certain amount of runtime manipulation. It is also possible to associate a temp–table object with a static temp–table as follows:

```
DEFINE VARIABLE tth AS WIDGET-HANDLE.
DEFINE TEMP-TABLE tmptblx LIKE customer.
tth = tmptblx:HANDLE.
```

Creating a handle to a static temp–table is especially useful if you want to pass it as a parameter. See the "Dynamic Temp–tables as Local and Remote Parameters" section for more information on temp–tables as parameters

15.3.2 Creating a Dynamic Temp–table

There are several steps to creating a dynamic temp–table:

- Creating the temp–table object
- Adding fields to the dynamic temp–table
- Adding indexes to the dynamic temp–table
- Completing the dynamic temp–table

Creating the Temp–table Object

You can create a dynamic temp–table by using the CREATE TEMP–TABLE statement which creates an empty temp–table object. You must use the ADD/CREATE type methods to add fields and indexes to the table. For example, the following statement creates an empty temp–table that can be referred to by its handle, tthandle:

```
CREATE TEMP-TABLE tthandle.
```

Adding Fields to the Dynamic Temp–table

Once you have created the empty temp–table object, you must add fields to it to make it useful. There are four methods you can use to add fields to the temp–table:

- CREATE–LIKE( ) — copies the field definitions from a source table and establishes all the indexes of the source table unless specified otherwise.
- ADD–FIELDS–FROM( ) — copies the specified fields from a source table. No indexes are established. This method can be used multiple times on the same temp–table and is especially useful when the rows of the temp–table need to represent a join.
- ADD–LIKE–FIELD( ) — copies the single field specified from a source table.
- ADD–NEW–FIELD( ) — adds a field with the specified properties.

The following code fragment illustrates creating a dynamic temp–table and adding fields to it:

```
DEFINE VARIABLE tthandle AS HANDLE.

/* create an "empty" undefined temp-table */
CREATE TEMP-TABLE tthandle.

/* give it customer table's fields and indexes */
tthandle:CREATE-LIKE("customer").

/* add all fields from salesrep table except for month-quota */
tthandle:ADD-FIELDS-FROM("salesrep","month-quota").

/* give it a single extra integer field */
tthandle:ADD-NEW-FIELD("f1","integer").
. . .
```

Adding Indexes to the Dynamic Temp–table

There are several ways of creating indexes for dynamic temp–tables. The CREATE–LIKE( ) method described above creates all the indexes of the source table by default. If you specify a single index in the CREATE–LIKE( ) method, however, you will get only that index. In addition, the following methods also create indexes in the dynamic temp–table:

- ADD–INDEX–FIELD( ) — adds the named field to the named index
- ADD–LIKE–INDEX( ) — copies the specified index from the specified source table
- ADD–NEW–INDEX( ) — adds a new index with the specified properties

The following code fragment adds an index to the previous example:

```
DEFINE VARIABLE tthandle AS HANDLE.

CREATE TEMP-TABLE tthandle.

tthandle:CREATE-LIKE("customer").
tthandle:ADD-FIELDS-FROM("salesrep","month-quota").
tthandle:ADD-NEW-FIELD("f1","integer").

/* add a new index "rep" and add the "rep-name" to the new index */
tthandle:ADD-NEW-INDEX("rep")
tthandle:ADD-INDEX-FIELD("rep","rep-name")
. . .
```

Completing the Dynamic Temp–table

Once you have added all the fields and indexes for your dynamic temp–table, you must signal the 4GL that the temp–table definition is complete by using the TEMP–TABLE–PREPARE( ) method. This method causes the pending list of fields and index definitions to become part of the actual temp–table object, making it ready for use. TEMP–TABLE–PREPARE( ) must be called before any non–ADD/CREATE method can be called.

The following code fragment completes the previous examples:

```
DEFINE VARIABLE tthandle AS HANDLE.

CREATE TEMP-TABLE tthandle.

tthandle:CREATE-LIKE("customer").
tthandle:ADD-FIELDS-FROM("salesrep","month-quota").
tthandle:ADD-NEW-FIELD("f1","integer").

/* add an index like the salesrep.sales-rep index and call it "srep" */
tthandle:ADD-LIKE-INDEX("srep","sales-rep","salesrep").

/* the definition of temp-table, "newcust", is complete */
tthandle:TEMP-TABLE-PREPARE("newcust").
. . .
```

15.3.3 Dynamic Buffers for Dynamic Temp–tables

All dynamic buffer attributes and methods are available to dynamic temp–tables. For more information on dynamic buffers, see Chapter 11, "Database Triggers" and the *Progress Language Reference* manual sections on Buffer Object Handle and Buffer–field Object Handle.

Every dynamic temp–table is created with at least one buffer, just like a static temp–table. This default buffer's object handle is returned by the DEFAULT–BUFFER–HANDLE( ) method. It is through this buffer handle that you have access to the records and fields in the temp–table object as follows:

```
dyn-buff-hdl = dyn-tt-hdl:DEFAULT-BUFFER-HANDLE.
dyn-buff-hdl:BUFFER-CREATE.
buff-fld = dyn-buff-hdl:BUFFER-FIELD(3).
```

The DEFAULT–BUFFER–HANDLE( ) method may not be called until the TEMP–TABLE–PREPARE( ) method has been called, since the default buffer is not created until then.

In addition, several new methods have been created to enhance the use of dynamic buffers for dynamic temp–tables:

- BUFFER–COPY — copies any common fields from one buffer to another, excluding specified fields and excluding fields not present in both tables.
- BUFFER–COMPARE — compares common fields in two buffers, excluding specified fields and excluding fields not present in both tables.

The CREATE BUFFER statement will now support both temp–tables and buffer handles, as follows:

```
CREATE BUFFER dyn-buff-hdl FOR TABLE dyn-tt-hdl.
```

The following program illustrates the use of buffers and buffer–fields in dynamic temp–tables. It uses a dynamic query which is described in Chapter 20, "Using Dynamic Widgets":

p-ttdyn2.p

```
/* p-ttdyn2.p - a join of 2 tables  */
DEFINE VARIABLE tth4 AS HANDLE.
DEFINE VARIABLE btth4 AS HANDLE.
DEFINE VARIABLE qh4 AS HANDLE.
DEFINE VARIABLE bCust AS HANDLE.
DEFINE VARIABLE bOrder AS HANDLE.
DEFINE VARIABLE i AS INTEGER.
DEFINE VARIABLE fldh AS HANDLE EXTENT 15.

bCust = BUFFER customer:HANDLE.
bOrder = BUFFER order:HANDLE.

CREATE TEMP-TABLE tth4.
tth4:ADD-FIELDS-FROM(bCust,"address,address2,phone,city,comments").
tth4:ADD-FIELDS-FROM(bOrder,"cust-num,carrier,instructions,PO,terms").
tth4:TEMP-TABLE-PREPARE("CustOrdJoinTT").
btth4 = tth4:DEFAULT-BUFFER-HANDLE.

FOR EACH customer WHERE cust.cust-num < 6, EACH order OF customer:
    btth4:BUFFER-CREATE.
    btth4:BUFFER-COPY(bCust).
    btth4:BUFFER-COPY(bOrder).
END.

/* Create Query  */
CREATE QUERY qh4.
qh4:SET-BUFFERS(btth4).
qh4:QUERY-PREPARE("for each CustOrdJoinTT").
qh4:QUERY-OPEN.

REPEAT WITH FRAME zz DOWN:
    qh4:GET-NEXT.
    IF qh4:QUERY-OFF-END THEN LEAVE.
    REPEAT i = 1 TO 15:
        fldh[i] = btth4:BUFFER-FIELD(i).
        DISPLAY fldh[i]:NAME FORMAT "x(15)"
        fldh[i]:BUFFER-VALUE FORMAT "x(20)".
    END.
END.

btth4:BUFFER-RELEASE.
DELETE OBJECT tth4.
DELETE OBJECT qh4.
```

15.3.4 Dynamic Temp–tables as Local and Remote Parameters

Using a temp–table as a parameter requires the ability to send the template of the temp–table, its index–information, and its field information locally to a local procedure or remotely to an AppServer.

The DEFINE PARAMETER and RUN statements support sending the table–handle as a parameter as shown below:

```
/* calling program */

RUN myprog.p(INPUT TABLE-HANDLE newhdl).
```

```
/* called program */

DEFINE INPUT PARAMETER TABLE-HANDLE newhdl.
```

For more information on parameter syntax, see the DEFINE PARAMETER statement and the RUN statement in the *Progress Language Reference* manual.

INPUT

If the parameter is INPUT TABLE–HANDLE, the definition behind the handle plus the contents of the temp–table are sent from the caller to the called routine. The called routine can have either a dynamic INPUT TABLE–HANDLE or a static INPUT TABLE as a matching parameter.

In the called routine, a new instance of the temp–table is created along with its handle, completely separate from the caller's table and populated with the contents from the caller's table. This instance is used during the called routine. If the called routine has a static table as the matching parameter, the caller temp–table data will be loaded into this table. In no case is the caller's table affected by the called procedure.

OUTPUT

If the parameter is OUTPUT TABLE–HANDLE, the handle plus the definition behind the handle are sent from the caller to the called routine. If the handle is NULL, then no definition is sent. The called routine can have either a table handle for a static OUTPUT parameter or a dynamic OUTPUT TABLE–HANDLE as a matching parameter. The called routine returns the definition behind the handle along with the contents of the output temp–table. In the caller, a new instance of the table is created if the original handle was NULL and populated with the returned temp–table contents. This new table replaces any pre-existing table in the caller, either static or dynamic. If the APPEND keyword was used, the returned contents are appended to the pre-existing table, rather than replacing it.

NOTE: For local cases, there is usually no need to use anything more than a simple HANDLE type parameter to allow routines to share a temp–table. The TABLE–HANDLE parameter is only for cases where either one or both sides of a remote communication lack a static definition of the temp–table.

INPUT–OUTPUT

If the parameter is INPUT–OUTPUT TABLE–HANDLE, a combination of the above will occur.

The following programs illustrate using a temp–table handle as a parameter. The called program, getdbtable.p, is used to create a temp–table from any database table whose name is passed to it and to pass back the handle to this temp–table. The calling program, passdbname.p, passes the database table name and receives back the temp–table handle which it uses to access the created temp–table.

These programs use dynamic queries which are described in Chapter 20, “Using Dynamic Widgets”:

p-passdb.p

```
DEFINE VARIABLE tth AS HANDLE.
DEFINE VARIABLE dbtabname AS CHARACTER.
DEFINE VARIABLE q AS HANDLE.
DEFINE VARIABLE bh AS HANDLE.
DEFINE VARIABLE fh1 AS HANDLE.
DEFINE VARIABLE fh2 AS HANDLE.

dbtabname = "item".
RUN p-getdb.p (INPUT dbtabname ,OUTPUT TABLE-HANDLE tth).

IF tth = ? THEN RETURN.
bh = tth:DEFAULT-BUFFER-HANDLE.
fh1 = bh:BUFFER-FIELD(1).
fh2 = bh:BUFFER-FIELD(2).

CREATE QUERY q.
q:SET-BUFFERS(bh).
q:QUERY-PREPARE("FOR EACH " + dbtabname).
q:QUERY-OPEN.

REPEAT:
    q:GET-NEXT.
    IF q:QUERY-OFF-END THEN LEAVE.
    MESSAGE fh1:STRING-VALUE fh2:STRING-VALUE.
END.
```

p-getdb.p

```
DEFINE INPUT PARAMETER tabname AS CHARACTER.
DEFINE OUTPUT PARAMETER TABLE-HANDLE tth.
DEFINE VARIABLE bh AS HANDLE.
DEFINE VARIABLE fh AS HANDLE.
DEFINE VARIABLE q AS HANDLE.
DEFINE VARIABLE dbh AS HANDLE.
DEFINE VARIABLE i AS INTEGER.

CREATE BUFFER dbh FOR TABLE tabname.
CREATE TEMP-TABLE tth.
tth:ADD-FIELDS-FROM(tabname).
tth:TEMP-TABLE-PREPARE(tabname).
bh = tth:DEFAULT-BUFFER-HANDLE.

CREATE QUERY q.
q:SET-BUFFERS(dbh).
q:QUERY-PREPARE("FOR EACH " + tabname).
q:QUERY-OPEN.

REPEAT i = 1 TO 5:
    q:GET-NEXT.
    bh:BUFFER-CREATE.
    bh:BUFFER-COPY(dbh).
END.

DELETE OBJECT dbh.
DELETE OBJECT q.
DELETE OBJECT tth.  /* delete is delayed until the parameter is passed */
```

15.3.5 Error Handling and Messages

Errors for dynamic objects do not automatically raise Progress 4GL errors since they occur inside a widget expression. Rather, all the methods that can have errors return FALSE if an error occurs, so they must be tested. If NO–ERROR is in effect in the statement containing the widget reference, no messages display, but they can be retrieved from the ERROR–STATUS system handle.

16

Widgets and Handles

A *widget* is a visual element of a user interface. For example, a widget might be an image, a menu, a frame, a window, or a fill-in field. Widgets are often called *user interface components* (UICs) or *controls*.

Progress supports the following types of widgets:

- Windows
- Frames
- Dialog boxes
- Browse
- Fill-in fields, text, sliders, selection lists, combo boxes, toggle boxes, radio sets, and editors for data representation
- Buttons, images, rectangles, and literals
- Menus, submenus, menu bars, and menu items

Progress provides handle and widget-handle variables that you can use to reference widgets or the context of procedures. Progress also supports system handles. A *system handle* is a built-in handle that references a particular UIC (such as the current window), a Progress procedure context, session information (such as error status), or a special Progress or system function (such as the system clipboard).

You can also specify triggers for widgets and procedures. *Triggers* are blocks of code that execute in response to a user-specified or program-generated event.

This chapter explains the functions and inter-relationships of user-interface widgets, triggers, widget handles, and system handles, and describes how to use them.

16.1 Overview

Widgets exist in a hierarchy consisting of container and atomic widgets. A *container* widget can contain one or more other widgets. An *atomic* widget represents data or other visual information but does not contain any other widgets. Most container and atomic widgets can have a menu widget (a type of container widget) associated with them, and many atomic widgets can have an associated literal widget that represents a side label.

16.1.1 Widget Types

The basic container widget of a user interface is the window. In a character environment, the screen itself is the only window. In a graphical environment, the screen can contain many windows.

In Progress, a window can contain one or more frames. A frame can contain one or more field-level widgets and child frames organized into field groups. *Field-level* widgets are atomic widgets that can represent variables, database fields and records, or visual objects such as buttons, images and rectangles. Figure 16–1 shows a simple user-interface display. The display contains two windows labeled Window 1 and Window 2. Window 1 contains two frames labeled Frame A and Frame B. Each of these frames contains two field-level widgets. Window 2 contains one frame, which contains a text widget and a slider widget.

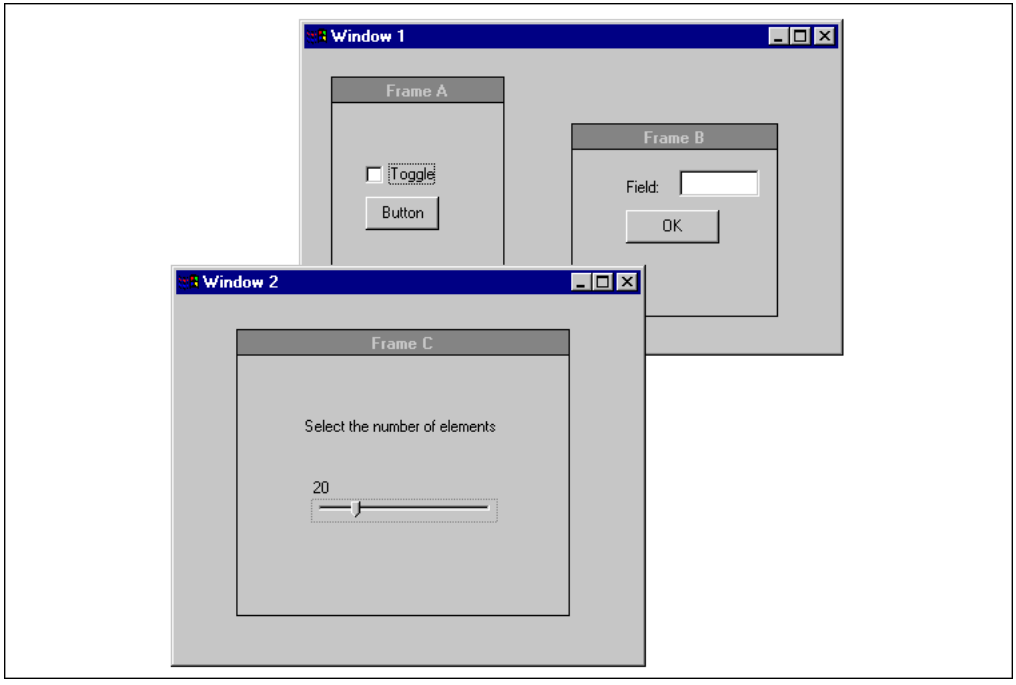


Figure 16–1: User-interface Display

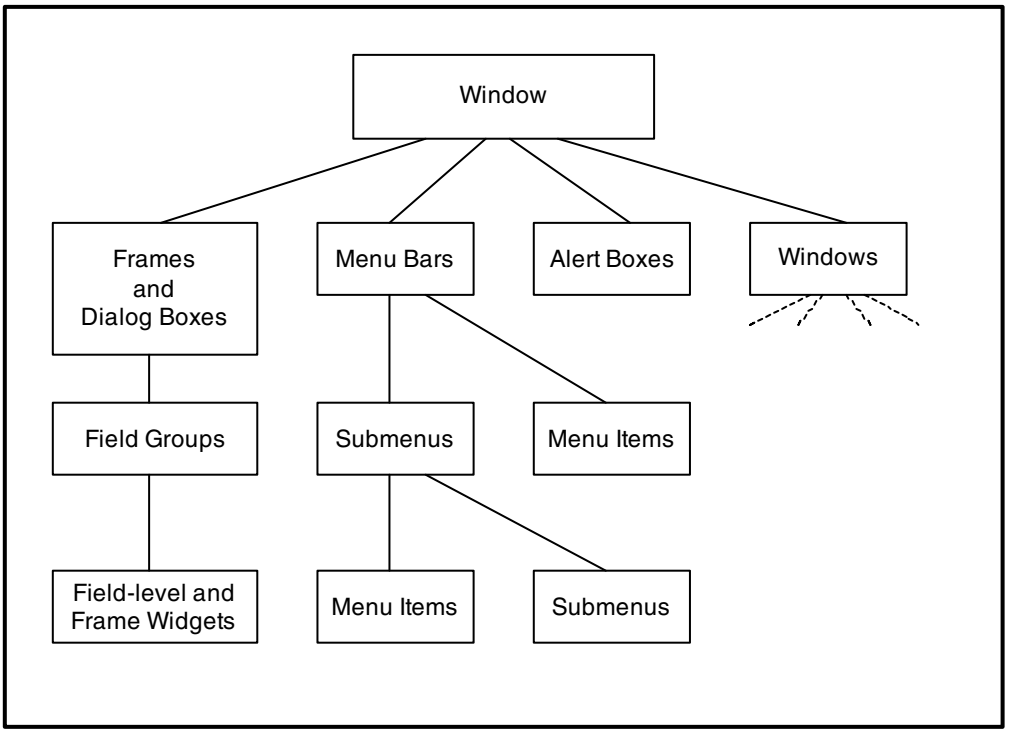


Figure 16–2: A Widget Hierarchy

As shown in Figure 16–2, a window can contain frames, dialog boxes, a menu bar, and an alert box. A frame or dialog box contains field groups. A field group can contain one or more field-level widgets and child frames, creating frame families. *Alert boxes* are special widgets for displaying messages only. A window can also parent (own) another window, creating a window family. For more information on windows and window families, see Chapter 21, "Windows," For information on frames, frame families, field groups, and dialog boxes, see Chapter 19, "Frames." For more information on windows, frames, and dialog boxes, see Chapter 25, "Interface Design."

Some field-level widgets are data-representation widgets. *Data-representation widgets*, such as fill-ins, sliders, and selection lists, can represent database fields or variables. The browse widget can represent multiple records in a database. Other field-level widgets, such as buttons, images, and rectangles represent visual objects and perform special functions. For more information on field-level widgets, see Chapter 10, "Using the Browse Widget," Chapter 17, "Representing Data" (for other data-representation widgets), and Chapter 18, "Buttons, Images, and Rectangles."

It is important to distinguish between a data-representation widget and its underlying data storage. For example, a fill-in widget and the variable it represents each exist independently of the other. In fact, you can have a data-representation widget that represents different data storage at different times or does not represent any storage at all.

A menu widget is a special type of container widget that contains only submenu and menu item widgets. Progress supports two types of menus. As shown in Figure 16–2, a window can contain a menu bar, which can contain pull-down submenus. The second type of menu (not shown in Figure 16–2) is a pop-up menu. A pop-up menu can be associated with a data-representation widget, a button, a frame, a dialog box, or a window. For more information on menus, see Chapter 22, "Menus."

Progress lets you specify triggers for all user-interface widgets. For more information, see the "User-interface Triggers" section.

Progress also supports a special type of field-level widget, the control–frame, that provides both a design-time and run-time interface to ActiveX controls. ActiveX controls offer a rich set of user interface extensions to Progress. Many of the most popular ActiveX controls are also available through the Crescent Division of Progress Software Corporation. For more information on control–frames and ActiveX controls, see the *Progress External Program Interfaces* manual.

The next section describes the basic types of widgets and how you can create them.

16.2 Static Versus Dynamic Widgets

Most widgets are either static or dynamic. *Static widgets* are defined when a procedure is compiled. They are created when the procedure is invoked and remain available until the procedure terminates or for the scope of their definition, whichever is longer. *Dynamic widgets* can be created and deleted during execution of a procedure and are globally scoped. (*Scope* is the duration that an object is available to the application.)

NOTE: All windows, except the static default window created by Progress, are dynamic. For more information on creating dynamic Windows, see Chapter 21, "Windows," and the "Dynamic Widgets" section in this chapter. Alert boxes and field groups are neither static nor dynamic but are created automatically by Progress. Progress creates an alert box for a window when you invoke the MESSAGE statement with the VIEW–AS ALERT–BOX option. For more information on using alert boxes, see Chapter 25, "Interface Design." Progress automatically creates field groups when you create and use a frame or dialog box. For more information on working with field groups, see Chapter 19, "Frames."

16.2.1 Creating Widgets

Creating any widget, static or dynamic, involves two actions—describing and then instantiating the widget. You *describe* a widget when you define its visual and behavioral characteristics. You *instantiate* a widget when you actually create an instance of the widget that you describe. Thus, instantiation makes a widget handle available to reference the widget. The process of describing and instantiating a widget differs depending on whether the widget is static or dynamic.

16.2.2 Using Widget Handles

To allow you to reference instantiated widgets within an application, Progress assigns each widget a unique *widget handle*. A widget handle is a pointer to the widget. Progress provides the WIDGET–HANDLE data type to support widget-handle variables and fields (in temporary tables and work tables).

NOTE: Progress also provides handles to procedures with the HANDLE data type. For more informatigon, see Chapter 3, "Block Properties."

You can find the handle for a specific widget using the HANDLE attribute. An *attribute* is a named widget characteristic. You can reference a handle attribute for a widget using this syntax:

SYNTAX

```
widget-reference :handle-attribute [ IN container-widget ]
```

For a static widget, the initial *widget–reference* is the name you use to describe the widget, possibly qualified by the widget type. For a dynamic widget, it is the widget-handle variable you use to describe the widget. The *handle–attribute* is any attribute that returns a widget handle value. The *container–widget* parameter specifies the container widget for *widget–reference*, if there is more than one. (For more information on referencing and using attributes, see the "Widget Attributes" section.)

For example, if you want to find the widget handle for a database field displayed in a named frame, you can use a code fragment like the following to retrieve the value of the HANDLE attribute for the widget used to display the field.

```
DEFINE VARIABLE myhand AS WIDGET-HANDLE.

FIND FIRST customer.

DISPLAY customer.name WITH FRAME name-frame.

myhand  = customer.name:HANDLE IN FRAME name-frame.
```

Many other attributes, such as PARENT or FRAME, return widget-handle values. For more information on the HANDLE attribute and other widget-handle attributes, see the *Progress Language Reference*.

You can obtain the handle for any widget that is instantiated. For information on describing and instantiating static widgets, see the "Static Widgets" section. For information on dynamic widgets, see the "Dynamic Widgets" section.

16.2.3 Static Widgets

You can describe and instantiate a static widget in one or two statements, depending on the widget type.

Describing a Static Widget

You can describe a static widget in several ways, depending on the widget:

- **Frame** — You can use the Frame phrase as part of many other statements, including a FORM, DEFINE FRAME, I/O (DISPLAY, UPDATE, ENABLE, etc.), or block header (FOR, REPEAT, etc.) statement.
- **Dialog box** — You specify the VIEW–AS DIALOG-BOX option in the Frame phrase.
- **Field-level data-representation widget (EDITOR, SELECTION–LIST, etc.)** — You can use the VIEW–AS phrase as part of some other phrase or statement—in a Format phrase for variables and in the Data Dictionary View–As phrase dialog for fields.
- **Browse, menu, submenu, menu-item, or a field-level widget not used for data representation, such as a button, image, or rectangle** — You can use the DEFINE *widget–type* statement. Note that you describe menu-item widgets directly in a static menu or submenu description. You can also use this statement to define frames, as noted above.

Instantiating a Static Widget

Progress automatically instantiates any static widget whose unique identity is known when you describe or use it. Thus:

- When you start a non-batch-mode client session, Progress instantiates the default window.
- When you describe or use any static frame or dialog box widget, Progress instantiates that frame or dialog box, unless it has already been instantiated by a previous description or use. You *use* a named or unnamed frame when you execute a frame-oriented I/O statement (DISPLAY, UPDATE, etc.) with or without a frame name. For more information on named and unnamed frames, see Chapter 19, "Frames."
- When you describe any static menu widget, Progress instantiates that menu, including its menu-items, submenus, and descendents.

Name references to some widgets specify a unique widget. These include static frame, dialog box, menu, or menu-item of menu widgets. Progress creates only one such widget in a procedure at a time. Thus, Progress can instantiate the widget when you first describe it.

However, to instantiate the description for any other static widget type, including a field-level, browse, submenu, or menu-item of a submenu widget, you must instantiate a static widget that contains it. The reason for this is that when you describe a widget that requires a container, you are only specifying the qualifying attributes of the widget. You can only establish a unique identity (widget handle) for a contained widget by associating it with an instantiated container widget. For example, this code fragment describes an editor widget associated with the variable xvar, and creates an instance of the editor in frame a.

```
DEFINE VARIABLE xvar AS CHARACTER VIEW-AS EDITOR SIZE 40 BY 10.

ENABLE xvar WITH FRAME a.
```

Also, the widget description for a static field-level, submenu, or submenu menu-item widget might specify more than one instance of that widget. When you include a reference to a field-level widget in the descriptions of more than one frame or dialog box, you instantiate a different, but identical, field-level widget for each frame or dialog box in which it appears.

This code fragment describes a button, b1, and creates two instances of it—one in frame x and one in frame y.

```
DEFINE BUTTON b1.

ENABLE b1 WITH FRAME x.
ENABLE b1 WITH FRAME y.
```

Similarly, when you include a reference to a submenu widget in the descriptions of more than one instantiated menu or submenu, you instantiate a different, but identical, submenu widget for each menu and submenu in which it appears.

This code fragment describes a submenu, sub1, and creates two instances of it—one in menu m1 and one in menu m2.

```
DEFINE SUB-MENU sub1
    MENU-ITEM sub-item1
    MENU-ITEM sub-item2.

DEFINE MENU m1
    MENU-ITEM item1
    SUB-MENU sub1.

DEFINE MENU m2
    MENU-ITEM item1
    SUB-MENU sub1.
```

Thus:

- When you specify a static browse or field-level widget within the description of a static frame or dialog box, Progress creates a unique instance of that browse or field-level widget. Although a browse widget requires a container to be instantiated, you can only instantiate a single browse from a single browse description. Also, you can reference a single browse widget in only one frame description at a time.

  On the other hand, all instantiations of a static field-level data-representation widget represent the same underlying variable or database field. This example shows two fill-in fields—one in frame x and the other in frame y—each for the same cust–num field in the current customer record.

  ```
  DEFINE FRAME x customer.cust-num.
  DEFINE FRAME y customer.cust-num.
  ```

  For more information on browse widgets, see Chapter 10, “Using the Browse Widget.” For more information on data-representation widgets, see Chapter 17, “Representing Data.”

- When you specify a static submenu widget within a static menu description, Progress creates a unique instance of that submenu widget and each descendent submenu and menu-item widget.

- When you specify a static menu-item widget within a static menu description, Progress creates a unique instance of that menu-item.
- When you specify a static submenu or menu-item widget within another static submenu description, Progress creates a unique instance of the named submenu or menu item **only when** the containing submenu is instantiated by a succeeding menu description.

Working with Static Widgets

The VIEW–AS phrase for describing static data-representation widgets has the following general syntax:

SYNTAX

```
VIEW-AS widget-type [ options ]
```

The *widget–type* parameter is the name of the widget type, such as FILL–IN, TOGGLE–BOX, or SLIDER. The *options* parameter is a set of options that further describe the widget.

The DEFINE statement for describing other static widgets (buttons, images, rectangles, menus, submenus, and frames) has the following general syntax:

SYNTAX

```
DEFINE widget-type name [ options ]
```

The *name* parameter specifies a name for the widget, much like a variable name for data-representation widgets.

Again, frames and dialog boxes are described using the Frame phrase in many statements. Some of these statements, such as the FORM, I/O, and block header statements can describe a named or unnamed frame and apply scoping to the frame. The DEFINE FRAME statement, however, describes a named, unscoped frame. For more information on frame scoping, see Chapter 19, “Frames.”

NOTE: The VIEW–AS DIALOG–BOX option is a special case of VIEW–AS applied to the Frame phrase only.

For example, this code fragment describes an editor widget and a frame. It instantiates the frame and editor widget in the frame description.

```
DEFINE VARIABLE e AS CHARACTER VIEW-AS EDITOR INNER-LINES 5
                  INNER-CHARS 30.

DEFINE FRAME main-frame
      e
      WITH NO-LABELS.
```

You can also specify triggers for all static widgets when they are defined using the Trigger phrase, except the browse, frame, menu, and submenu widgets. For more information, see the "User-interface Triggers" section.

Referencing Static Widgets

In general, you can reference static widgets using either the widget name or a widget-handle variable to which its handle has been assigned. For more information on widget handles, see the "Widget Handles" section. The widget name for data-representation widgets is the field or variable name. For frames, it is the frame name; and for all other static widgets, it is the *name* used in the DEFINE *widget–type* statement.

The following widgets also require that you qualify name references with the *widget–type*:

- Frames and dialog boxes

  NOTE: A dialog box is a type of frame. Therefore, you qualify all name references to it as a FRAME widget.

- Menus
- Submenus
- Menu items

For example, this code fragment shows name references to menu and menu-item widgets:

```
DEFINE MENU medit1
  MENU-ITEM icut LABEL "Cut"
  MENU-ITEM icopy LABLE "Copy"
  MENU-ITEM ipaste LABLE "Paste".

DEFINE MENU medit2 LIKE MENU medit1.

ON CHOOSE OF MENU-ITEM icopy IN MENU medit2 RUN copy2.
```

This code fragment shows name references to field-level and frame widgets, including the browse widget, custbrow; the fill-in field, order.order–num; and the frame, x:

```
DEFINE QUERY q FOR customer.
DEFINE BROWSE custbrow QUERY q DISPLAY cust-num name WITH SIZE 25 by10.
DEFINE FRAME x custbrow order.order-num.

ON VALUE-CHANGED OF custbrow IN FRAME x DO: END.
ON RETURN OF order.order-num IN FRAME x DO: END.
```

16.2.4 Dynamic Widgets

You can describe and immediately instantiate any dynamic widget with a CREATE Widget statement, which has the following general syntax:

SYNTAX

```
CREATE widget-type handle-variable
  [ IN WIDGET-POOL pool-name ]
  [ ASSIGN attribute = value [ attribute = value ] ... ]
  { [ trigger-phrase ] }
```

The *widget–type* parameter is the name of the widget type, such as WINDOW, FRAME, or BUTTON, and the *handle–variable* parameter is the name of a variable of type WIDGET–HANDLE that receives the widget handle value for the widget.

The *pool–name* parameter specifies a previously created widget pool. A widget pool allows you to manage a set of dynamic widgets as a group. For more information on creating and using widget pools, see Chapter 20, “Using Dynamic Widgets.”

The *attribute* parameter is the name of a widget attribute that is valid for the widget. Unlike static widgets, which you describe using compile-time options, you must describe a dynamic widget by setting its attributes at run time. You can set attributes when or after you create the widget. For more information on setting widget attributes, see the “Widget Attributes” section.

The *trigger–phrase* specifies one or more triggers for the widget. Note that the ON statement provides a more flexible technique for specifying triggers. The scoping is more reliable and allows the CREATE Widget statement to be shorter and more readable. For more information, see the “User-interface Triggers” section.

You can use this syntax to create any dynamic widget. You can create any type of widget dynamically except literal, down frame, and browse widgets. Literal widgets are created only by Progress to hold side labels for static data-representation widgets. Down frames and browse widgets provide different techniques for displaying multiple iterations of data. Progress requires compile-item information to create these widgets. For more information on down frames, see Chapter 19, "Frames." For more information on browse widgets, see Chapter 10, "Using the Browse Widget."

Referencing Dynamic Widgets

Note that unlike static widgets, a dynamic widgets are created and referenced only with widget-handle variables. This is because you are describing an actual instance of the widget, independent of any other widget. A dynamic widget stands on its own, whether or not it requires a container widget to be part of the user interface. Note also that dynamic data-representation widgets, such as fill-in fields, have no underlying field or variable for data storage. You must explicitly associate a dynamic widget with a field or variable by assigning data between the widget's SCREEN–VALUE attribute and any appropriate field or variable.

Deleting Dynamic Widgets

You can explicitly delete a dynamic widget with the DELETE WIDGET statement:

SYNTAX

```
DELETE WIDGET widget-handle
```

If you do not explicitly delete a widget, it remains allocated until the end of the session or until the widget pool that contains it is deleted. If you delete a dynamic container widget, such as a frame or window, any dynamic widgets that it contains are deleted also.

An advantage of dynamic widgets is that you can determine the number of widgets to create at run time. For example, you might create a button for each table in a database or each record in a table. A disadvantage is that you must do more of the work for dynamic widgets. For example, Progress can automatically lay out static widgets in a frame, but you must explicitly position dynamic widgets.

NOTE: The Procedure Editor optionally deletes all dynamic widgets that you create in a session each time you return to the Editor after running an application. For more information, see the on-line help for the Procedure Editor, and *Progress Basic Development Tools* manual (Character only).

You can also delete groups of dynamic widgets in a widget pool using the DELETE WIDGET–POOL statement. For more information, see Chapter 20, "Using Dynamic Widgets."

The following procedure creates a dynamic button for each record in the salesrep table:

p-dybuts.p

```
DEFINE VARIABLE num-buts  AS INTEGER.
DEFINE VARIABLE temp-hand AS WIDGET-HANDLE.

DEFINE FRAME butt-frame
   WITH WIDTH 60 CENTERED TITLE "Sales Representatives".

FORM
   salesrep
   WITH FRAME rep-frame.

num-buts = 0.
FOR EACH salesrep:
   CREATE BUTTON temp-hand
         ASSIGN LABEL = salesrep.sales-rep
                FRAME = FRAME butt-frame:HANDLE
                ROW = TRUNC(num-buts / 3, 0) + 1
                COLUMN = ((num-buts MOD 3) * 20) + 1
                SENSITIVE = TRUE
         TRIGGERS:
            ON CHOOSE
               DO:
                 FIND salesrep WHERE salesrep.sales-rep = SELF:LABEL.
                 DISPLAY salesrep WITH FRAME rep-frame.
               END.
         END TRIGGERS.
   num-buts = num-buts + 1.
END.

FRAME butt-frame:HEIGHT-CHARS = (num-buts / 3) + 2.

VIEW FRAME butt-frame.

WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

Note that within the CREATE BUTTON statement, the procedure assigns the button to butt–frame, then calculates a ROW and COLUMN value within that frame. The procedure must explicitly set the width of the frame in the DEFINE FRAME statement and set the height of the frame after all the buttons are created. By contrast, Progress automatically positions the static widgets within rep–frame and makes that frame the proper size.

For more information on creating and using dynamic widgets, see Chapter 20, “Using Dynamic Widgets.”

16.3 Handles

Handles are pointers to Progress objects. You generally define variables or fields to hold handle values with the HANDLE, WIDGET–HANDLE, or COM–HANDLE data type. Note that HANDLE and WIDGET–HANDLE have a weak type relationship; that is, one can be assigned directly to the other. The COM–HANDLE data type provides ActiveX support. Progress supports following types of handle, depending on the object referenced:

- Widget
- Procedure
- Query, buffer, and buffer-field
- Server
- Component
- Transaction
- System

16.3.1 Widget Handles

Progress assigns every static or dynamic widget a handle value once it is created. This handle serves a different purpose, depending on whether the widget is static or dynamic:

- For a static widget, the handle provides a unique reference to the widget besides the widget name. The widget name, by itself, can be ambiguous if the widget is specified in more than one container widget. Progress resolves this ambiguity differently depending on the widget. However, the handle to a widget is always unambiguous.
- For a dynamic widget, it provides the only means to reference and manage the widget that you create.

You generally store widget handle values in variables or fields defined with the WIDGET–HANDLE data type. For more information on widget handles and how to use them, see the "Static Versus Dynamic Widgets" section.

16.3.2 Procedure Handles

Every external procedure that is in scope within an application has a handle that identifies the context of the procedure. Using a valid procedure handle, you can execute the internal procedures and procedure triggers of the specified external procedure from any other procedure in your application. This capability is especially useful with persistent procedures. Thus, you can encapsulate related functionality and make it available to your entire application.

You generally store procedure handle values in variables or fields defined with the HANDLE data type. For more on procedure handles and how to use them, see the information on procedures in Chapter 3, “Block Properties.”

16.3.3 Query, Buffer, and Buffer-field Handles

Query, buffer, and buffer-field object handles let you manipulate static and dynamic versions of these objects. For more information, see the section on query, buffer, and buffer-field objects" in section of Chapter 9, “Database Access.”

16.3.4 Server Handles

A server handle references the connection that you create to a Progress AppServer using the CREATE SERVER statement. You generally store server handle values in variables or fields defined with the HANDLE data type. For more information on server handles and how to use them, see the *Building Distributed Applications Using the Progress AppServer* manual.

16.3.5 Component Handles

A component handle references a COM object (ActiveX Automation object or ActiveX control). Unlike all other handles, you store component handle values in variables defined with the COM–HANDLE data type. For more information on component handles and how to use them, see the *Progress External Program Interfaces* manual.

16.3.6 Transaction Handles

A transaction object handle provide access to the current transaction object. For more information, see the Transaction Object Handle reference entry in the “Handle Reference” chapter of the *Progress Language Reference*.

16.3.7 System Handles

Progress maintains several global handles that describe the current context of an application. Some system handles reference the widget handle of a context-determined user-interface widget. Others reference data and facilities available for and about the current session. Table 16–1 lists the Progress system handles.

Table 16–1: System Handles *(1 of 2)*

System Handle	Description
ACTIVE–WINDOW	A handle to the Progress window that has most recently received input focus during the session.
CLIPBOARD	A handle to the system clipboard.
COLOR–TABLE	A handle to information on the current color table.
COMPILER	A handle to information on the most recently executed COMPILE statement.
COM–SELF	A component handle to the ActiveX control that generates an OCX event. Accessible within the OCX event procedure that handles the event.
CURRENT–WINDOW	A setable handle to the default window for the Progress session.[1,2]
DEBUGGER	A handle to the Progress Application Debugger.
DEFAULT–WINDOW	A handle to the static window created by Progress for the session. Every session has one static window.[1]
ERROR–STATUS	A handle to information on the last statement executed with the NO–ERROR option.
FILE–INFO	A handle to information on an operating system file.
FOCUS	A handle to the field-level widget that currently has keyboard focus (that is, the current field).
FONT–TABLE	A handle to information on the current font table.
LAST–EVENT	A handle to the last event received by the program.
RCODE–INFO	A handle to information on a Progress r-code file.

Table 16–1: System Handles *(2 of 2)*

System Handle	Description
SELF	A handle for the widget associated with the currently executing user-interface trigger.
SESSION	A handle to information on the current Progress session.
SOURCE–PROCEDURE	A handle to the procedure file that contains the original invocation (RUN statement or function invocation) of the current internal procedure or user-defined function.
TARGET–PROCEDURE	From within an internal procedure: a handle to the procedure file mentioned, explicitly or implicitly, by the original RUN statement that invoked (perhaps through a chain of super procedures) the current internal procedure. From within a user-defined function: a handle to the procedure file mentioned, explicitly or implicitly, by the original function invocation that invoked (perhaps through a chain of super versions of functions) the current user-defined function.
THIS–PROCEDURE	A handle to the executing external procedure in which the handle is referenced.
WEB–CONTEXT	Provides access to information on the current connection to the Web server.

[1] The initial setting of the CURRENT−WINDOW handle is the Progress static window. CURRENT−WINDOW can also be set to the handle of any dynamic window.

[2] If the THIS−PROCEDURE:CURRENT−WINDOW attribute is set to the handle of a valid window, this window becomes the default window for the executing procedure (overriding the setting of the CURRENT−WINDOW handle). The setting of THIS−PROCEDURE:CURRENT−WINDOW changes the default window only for the current external procedure block

The pointer specified by a system handle has a data type appropriate to the object to which it points. Thus, the THIS–PROCEDURE handle and most other system handles return a HANDLE data type. System handles that reference widgets, such as CURRENT–WINDOW, return a WIDGET–HANDLE data type.

You can change some system handles within an application. For example, you can change the current window by assigning another handle value to the system CURRENT–WINDOW handle. You can establish a menu bar for the default window by assigning a menu bar handle to the DEFAULT–WINDOW:MENUBAR attribute.

For more information on each system handle, see the appropriate reference entry in the *Progress Language Reference*.

16.3.8 Handle Management

In general, handles allow you to pass around references to widget, procedure, server, ActiveX, and system objects both within procedures (using variables) and between procedures (using parameters). Depending on the object and what you do with it, a handle might or might not be valid at different points in your application. To ensure that the handle you reference is valid, you can test it using the VALID–HANDLE function. This function returns TRUE if the specified handle points to a valid object that is still available to your application.

Progress also reuses handle values. For example, if you delete a dynamic button, Progress might reuse the widget handle for the deleted button to point to another widget, procedure, or system object. To ensure that a valid handle you have stored still points to a widget of a particular type, you can check the value of the TYPE attribute for the handle. For example, if the handle points to a window, the TYPE attribute is set to "WINDOW". For more information on referencing attributes, see the "Widget Methods" section.

There are occasions where you might want to store a handle value as a string. Thus, you can convert any widget handle value to an integer string using the STRING function. To convert this string to its original handle value, use the WIDGET–HANDLE function. This function converts handle values for all types of handles. (Progress automatically converts values between component handles and any other data type.) For more information on the WIDGET–HANDLE function, see the *Progress Language Reference*.

CAUTION: The WIDGET–HANDLE function can only convert a string to a handle value that was originally converted from a handle value. Any other string causes a system error.

16.4 Widget Attributes

Every widget or handle has certain attributes. Each attribute stores a value associated with the widget or handle. This value represents an aspect, state, or capability of the widget. For example, most widgets have a VISIBLE attribute that is either TRUE or FALSE (that is, the widget is either visible or invisible).

Widget attributes fall into several basic categories:

- Geometry (that is, the size and location of the widget)
- Appearance (color, font, and cursor)
- I/O related
- Relationships to other widgets (such as parent, siblings, or owner)

Each type of widget has its own attributes. For example, the attributes supported for a button are different than those supported for a frame. For a list of the attributes supported by each widget, see the reference entry for the widget in the *Progress Language Reference*.

Handle attributes correspond to the type of object the handle references:

- Widget
- Procedure
- Pseudo-widget or system function

Some attributes are set when the widget or object is created and are inaccessible to the Progress programmer. Other attributes can be read and, in some cases, updated within the application. For example, you can enable or disable input on a menu item by changing the value of its SENSITIVE attribute.

16.4.1 Attribute References

You reference an attribute the same way, whether for reading or setting, and this is the syntax for an *attribute reference*:

SYNTAX

```
widget:attribute [ IN { FRAME | MENU | SUB-MENU } name ]
```

The *widget* parameter is the name of a widget (UIC), or a widget, procedure, or system handle. The *attribute* parameter specifies the name of the widget or handle attribute. You must leave no space between the colon (:) and its parameters.

The *name* parameter specifies the name of a container widget appropriate for the static UIC specified by *widget*. If the *widget* appears in more than one frame or menu, the default frame or menu is the most recently defined container whose description includes the *widget*. If the *widget* appears in more than one submenu, the default submenu is the first submenu in the menu tree whose description includes the *widget*.

Note that specifying a container *name* is only appropriate where *widget* specifies a name reference to a static widget. Handles are unique by definition and take no container name as part of the attribute reference.

Whenever possible, use unique names for your static widgets. Otherwise, specify the container widget in the attribute reference. Thus, if you add widget definitions later, you can avoid bugs caused by referencing the wrong default widget in your current attribute references.

16.4.2 Referencing Attribute Values

To reference the value of an attribute, you can either assign the attribute reference directly to a field or variable or include the attribute reference in an expression or statement option, like a function. For example, this is the syntax to assign an attribute value to a field or variable (*field*):

SYNTAX

```
field = attribute-reference
```

The *attribute–reference* includes the syntax specified in the "Attribute References" section. For information on referencing attribute values in expressions, see the "Attributes in Expressions" section.

16.4.3 Setting Attribute Values

This is the syntax to set an attribute value:

SYNTAX

```
attribute-reference = expression
```

The *attribute–reference* includes the syntax specified in the "Attribute References" section. The *expression* must produce a value compatible with the data type of the attribute.

16.4.4 Attributes in Expressions

An attribute reference can appear in an expression to yield an attribute value only. The attribute separator (:) is not an expression operator. It can only separate an attribute name from the name of the widget or handle variable to which it applies. Thus, it cannot reference an attribute of a widget handle value that is, itself, the result of an expression. This example shows a valid attribute reference in an expression.

```
DEFINE VARIABLE WidthTotal AS DECIMAL.
DEFINE BUTTON bChoose LABEL "Choose Me".
DEFINE FRAME Aframe bChoose.
WidthTotal = 5 * (bChoose:WIDTH-CHARS + 1) + 2.
```

This example shows an invalid attribute reference.

```
DEFINE VARIABLE WidthTotal AS DECIMAL.
DEFINE BUTTON bChoose LABEL "Choose Me".
DEFINE FRAME Aframe bChoose.
WidthTotal = (bChoose:FRAME):WIDTH-CHARS.
```

You cannot use the widget handle value returned by bChoose:FRAME to directly reference the WIDTH–CHARS attribute. You can, however, assign bChoose:FRAME to a widget-handle variable and use that variable for your attribute reference. For more information on widget-handle variables, see the "Handles" section.

16.4.5 Attribute Example

The following code fragment sets an attribute based on the Progress user ID, tests a logical expression that includes the result, and displays a message that includes a value assigned from another attribute.

```
DEFINE VARIABLE SecurityCode AS INTEGER.
DEFINE VARIABLE LastCodeDisplay AS CHARACTER.
DEFINE FRAME SecurityFrame SecurityCode WITH SIDE-LABELS.
                  .
                  .
                  .
IF USERID ("sports") <> "SUPERVISOR" THEN
    SecurityCode:VISIBLE = FALSE.

IF NOT SecurityCode:VISIBLE THEN DO:
    LastCodeDisplay = SecurityCode:SCREEN-VALUE.
    MESSAGE "You cannot continue with security code " LastCodeDisplay.
    QUIT.
END.
```

16.5 Widget Methods

In addition to attributes, some widgets have methods. A *method* is an operation that performs a specific action related to a widget. It might modify or return information on a specific widget, change a widget's relationship to other widgets, or affect other system resources related to the widget. Methods also return a value, like a function.

16.5.1 Method References

The syntax for a *method reference* is similar to the syntax for an attribute reference:

SYNTAX

```
widget:method ( [ arg [ , arg ] ... ] ) [ IN FRAME frame ]
```

The *widget* parameter is the name of a widget (UIC), or a widget or system handle. The *method* parameter specifies the name of the widget or handle method. You must leave no space between the colon (:) and its parameters.

Methods typically take one or more arguments. For example, the method ADD–FIRST(*new–item*) adds *new–item* as the first value in a selection list or combo box. Note that if a method takes no arguments, you must include an empty set of parentheses in the method reference.

Methods exist for windows, frames, and many widgets contained in frames. There are also specific methods for certain system handles. For more information on a specific method, see the *Progress Language Reference*.

16.5.2 Invoking Methods

You can invoke a method in two ways:

- Indirectly, by returning its value in an expression
- Directly as a statement, ignoring its value

Thus, the syntax to invoke a method is similar to the syntax for invoking a user-defined function:

SYNTAX

```
[ field = ] method-reference
```

The *method–reference* includes the syntax specified in the "Method References" section. You can return its value by assigning the method reference to a field or variable (*field*) or include the method reference in an expression, like a function.

16.5.3 Methods in Expressions

A method reference can appear in an expression to yield the method result only. The method separator (:) is not an expression operator. It can only separate a method name from the name of a widget or handle variable to which it applies. Thus, it cannot reference a method for a widget handle value that is, itself, the result of an expression. This example shows a valid method reference in an expression.

```
DEFINE VARIABLE WidthTotal AS DECIMAL.
DEFINE BUTTON bChoose LABEL "Choose Me".
DEFINE FRAME Aframe bChoose.
WidthTotal = 5 * (FONT-TABLE:GET-TEXT-WIDTH-CHARS(bChoose:LABEL) + 1) + 2.
```

This example shows an invalid method reference.

```
DEFINE VARIABLE WidthTotal AS DECIMAL.
DEFINE BUTTON bChoose LABEL "Choose Me".
DEFINE FRAME Aframe bChoose.
WidthTotal = 5 * ((FRAME Aframe:GET-TAB-ITEM(1)):WIDTH-CHARS + 1) + 2.
```

You cannot use the widget handle value returned from the GET–TAB–ITEM( ) method to directly reference the WIDTH–CHARS attribute. For more information on widget-handle variables, see the"Handles" section.

16.5.4 Method Example

The following procedure assigns an editor widget to the contents of a file using the INSERT–FILE( ) method. It writes the contents back to the file using the SAVE–FILE( ) method after the user has a chance to change it. Messages are displayed depending on user's input and the success or failure of the methods, which return TRUE if they succeed:

p-method.p

```
DEFINE VARIABLE To-Do AS CHARACTER VIEW-AS EDITOR SIZE 60 BY 10.
DEFINE FRAME DoFrame SPACE(3) To-Do WITH SIDE-LABELS SIZE 76 BY 12.

ON GO OF To-Do IN FRAME DoFrame DO:
    IF To-Do:SAVE-FILE("to-do.lst") THEN
        MESSAGE "TO DO list saved."
            VIEW-AS ALERT-BOX INFORMATION BUTTONS OK.
    ELSE DO:
        BELL.
        MESSAGE "TO DO list NOT saved!!"
            VIEW-AS ALERT-BOX WARNING BUTTONS OK.
    END.
END.

ON END-ERROR OF To-Do IN FRAME DoFrame
    MESSAGE "TO DO list NOT changed."
        VIEW-AS ALERT-BOX INFORMATION BUTTONS OK.

ENABLE ALL WITH FRAME DoFrame.

IF To-Do:INSERT-FILE("to-do.lst") THEN
    WAIT-FOR GO OF To-Do IN Frame DoFrame.
ELSE DO:
    BELL.
    MESSAGE "TO DO list NOT available."
        VIEW-AS ALERT-BOX MESSAGE BUTTONS OK.
END.
```

16.6 Widget Realization

When you create or define a widget, Progress creates an internal data structure associated with that widget. Before the widget can be displayed on the screen, the window system must also create a data structure for the widget. When this second data structure exists, the widget is *realized*.

You can modify some widget attributes at any time. Other attributes are initially modifiable, but become fixed while the widget is realized. This means that you cannot modify these attributes again unless the widget is *derealized*. A widget becomes derealized when the associated data structure within the windowing system is deleted.

There are two reasons it is important for you to understand when realization and derealization occur. First, because you cannot change some attributes while the widget is realized and, second, because some attributes must be set to a valid value before realization takes place.

16.6.1 Becoming Realized

In general, a widget becomes realized when the application needs to make it visible on the screen. This happens, for example, when a field-level widget is referenced in a screen I/O statement such as DISPLAY or UPDATE, or when the VISIBLE attribute of a frame is set to TRUE. Also, widgets typically become realized when a parent, owner, or child widget becomes realized. References to specific attributes and methods can also cause realization.

Table 16–2 lists the specific actions that cause realization for each type of widget.

Table 16–2: Realizing Widgets *(1 of 3)*

Widget Type	Action that Causes Realization
Browse	The browse becomes visible on the screen.
Button	The button becomes visible on the screen.
	The frame that contains the button becomes visible on the screen.
Combo box	The combo box becomes visible on the screen.
	The frame that contains the combo box becomes visible on the screen.
	An application references any method of the combo box.
Dialog box	See the information for frames.

Table 16–2: Realizing Widgets *(2 of 3)*

Widget Type	Action that Causes Realization
Editor	The editor becomes visible on the screen.
	The frame that contains the editor becomes visible on the screen.
	An application references any method of the editor.
	An application queries any of the following attributes: CURSOR–CHAR, CURSOR–LINE, CURSOR–OFFSET, MODIFIED, LENGTH, SELECTION–END, SELECTION–START, SELECTION–TEXT.
Field group	The frame that contains the field group becomes realized.
Fill-in field	The fill-in becomes visible on the screen.
	The frame that contains the fill-in becomes visible on the screen.
Frame[1]	The frame becomes visible on the screen. (This happens anytime the frame's VISIBLE attribute is set to TRUE.)
	Any field-level widget within the frame becomes realized.
	An UPDATE or CHOOSE statement is executed that references the frame or any field in the frame.
	An application reads any of the following attributes: BORDER–BOTTOM, BORDER–LEFT, BORDER–RIGHT, BORDER–TOP, MAX–COLUMNS, MAX–HEIGHT, MAX–ROWS, MAX–WIDTH.
Image	The image becomes visible on the screen.
Menu bar	The window that owns the menu bar becomes realized.
	The menu bar is attached to a window that is already realized.
Menu item	The parent menu becomes realized.
	The menu item is added to a menu that is already realized.
Pop-up menu	The owner of the pop-up menu becomes realized.
	The pop-up menu is attached to a widget that is already realized.

Table 16–2: Realizing Widgets *(3 of 3)*

Widget Type	Action that Causes Realization
Radio set	The radio set becomes visible on the screen.
	The frame that contains the radio set becomes visible on the screen.
	An application sets any of the following attributes: HEIGHT, HEIGHT–ROWS, WIDTH, WIDTH–COLUMNS.
Rectangle	The rectangle becomes visible on the screen.
	The frame that contains the rectangle becomes visible on the screen.
Selection list	The selection list becomes visible on the screen.
	The frame that contains the selection list becomes visible on the screen.
	An application references any method of the selection list.
	An application reads the selection list's MULTIPLE or VALUE attribute.
	An application reads or sets the selection list's INNER–CHARS or INNER–LINES attribute.
Slider	The slider becomes visible on the screen.
	The frame that contains the slider becomes visible on the screen.
Submenu	The parent menu is realized.
	The submenu is attached to a parent menu that is already realized.
Text	The text widget becomes visible on the screen.
	The frame that contains the text widget becomes visible on the screen.
Toggle box	The toggle box becomes visible on the screen.
	The frame that contains the toggle box becomes visible on the screen.
Window[2]	Any frame in the window is realized.

[1] If a frame is shared, it is realized in all procedures as soon as it is realized in any procedure.

[2] In batch mode, the window is realized as soon as it is created.

16.6.2 Becoming Derealized

Realization is not a permanent state. Under conditions listed in Table 16–3, a previously realized widget becomes derealized.

Table 16–3: Derealizing Widgets

Widget Type	Action that Causes Derealization
All dynamic widgets	The widget is deleted.
All static widgets	The widget goes out of scope
Any field-level widget	The containing field group is derealized.
Field group	The containing frame is derealized.
	The associated iteration is scrolled off the screen.
Frame	The containing window is deleted.
Menu bar	The window that owns the menu bar is deleted.
	The menu bar is detached from the window.
Menu item	The parent menu is derealized.
Pop-up menu	The widget that owns the pop-up menu is derealized.
	The pop-up menu is detached from its owner.
Submenu	The parent menu is derealized.

16.7 Resizing Widgets Dynamically

Certain run-time conditions can make the size of a widget inappropriate:

- The font of a button, combo box, editor, fill-in, frame, radio button, selection list, slider, toggle box, or window changes.
- The format of the text displayed within a fill-in changes.
- The label of a button, toggle box, or radio button changes.
- The size of the image displayed within a button changes.
- A selection list's or editor's INNER–CHARS or INNER–LINES attribute changes.

By default, Progress resizes the widget if any of the above conditions occur. It determines the size that is most appropriate under the new conditions. This automatic resizing is governed by the value of the the Boolean AUTO–RESIZE attribute, which by default is on, or TRUE. If AUTO–RESIZE is on for a particular widget, then the widget will resize based on its physical characteristics (for example, font, label, and format).

Any of the following actions will stop Progress from automatically resizing widgets at run time:

- You explicitly turn AUTO–RESIZE off, either at widget definition or at run time.
- You explicitly specify the size of a widget, either at widget definition or at run time, by changing any of the following characteristics:
  - Width
  - Height
  - Columns
  - Rows
  - INNER–CHARS
  - INNER–LINES

Note that if you turn AUTO–RESIZE on for a widget whose size you have been controlling, Progress will compute a new size immediately; it will not wait until one of the previously mentioned conditions, such as a font change, occurs.

For a programming example that demonstrates the AUTO–RESIZE attribute, see the information on fill-ins in Chapter 17, "Representing Data."

16.8 User-interface Triggers

For each widget you define or create, you can establish one or more *user-interface triggers*. A user-interface trigger is a block of code associated with an event-widget pair (that is, with a specific event and a specific widget). The trigger executes whenever that event is applied to that widget. User interface triggers allow you to write event-driven applications.

NOTE: Progress also provides procedure triggers that allow you to apply events to a procedure handle to execute code in the specified procedure. For more information, see Chapter 3, "Block Properties."

16.8.1 User-interface Events

Progress supports the following types of events:

- Keystrokes
- Mouse events
- Direct manipulation events
- Other, widget-specific events

Each type of widget supports a set of user-interface events. You can define triggers only on those events supported by the widget.

Keystrokes

You can set up triggers for raw keystrokes or for key functions. Raw keystroke events include all the characters in the character set, including control sequences such as CTRL–X and key labels such as F1 and DEL–CHAR. Key functions include GO, HELP, END–ERROR, and DELETE, among others.

For more information on Progress key labels, key functions, and key codes, see Chapter 6, "Handling User Input." In the same chapter, see the the section on monitoring keystrokes during data entry for examples of triggers on specific keystrokes.

Mouse Events

Progress supports a conceptual portable mouse model (that is, SELECT, MENU, and EXTEND buttons) and a physical two-button mouse model (that is, LEFT, and RIGHT buttons). The portable model provides better portability. However, using any mouse events may cause problems when porting to a system that does not have a mouse. Consider using functional events such as CHOOSE, ENTRY, and MENU–DROP instead.

To map the portable model to a two-button mouse, a physical button may have two meanings, or a modifier key, such as CTRL, may be used with a physical button. For example, with a two-button mouse on Windows, the LEFT button maps to the SELECT and MOVE buttons, the RIGHT button maps to the MENU button, and the RIGHT button with **CTRL** maps to the EXTEND button.

Table 16–4 lists the portable and two-button mouse events supported by Progress.

Table 16–4: Mouse Events

Portable Mouse Events	Two-button Mouse Events
MOUSE–SELECT–UP	LEFT–MOUSE–UP
MOUSE–SELECT–DOWN	LEFT–MOUSE–DOWN
MOUSE–SELECT–CLICK	LEFT–MOUSE–CLICK
MOUSE–SELECT–DBLCLICK	LEFT–MOUSE–DBLCLICK
MOUSE–MENU–UP	RIGHT–MOUSE–UP
MOUSE–MENU–DOWN	RIGHT–MOUSE–DOWN
MOUSE–MENU–CLICK	RIGHT–MOUSE–CLICK
MOUSE–MENU–DBLCLICK	RIGHT–MOUSE–DBLCLICK
MOUSE–MOVE–UP	N/A
MOUSE–MOVE–DOWN	N/A
MOUSE–MOVE–CLICK	N/A
MOUSE–MOVE–DBLCLICK	N/A
MOUSE–EXTEND–UP	N/A
MOUSE–EXTEND–DOWN	N/A
MOUSE–EXTEND–CLICK	N/A
MOUSE–EXTEND–DBLCLICK	N/A

Direct Manipulation Events

Progress supports events related to marking, moving, and resizing widgets on the screen. For more information on these events, see Chapter 24, "Direct Manipulation."

Other Events

In addition to normal keystrokes, mouse events, and direct manipulation events, Progress supports the additional events listed in Table 16–5.

Table 16–5: Other Progress Events

Event	Description
CHOOSE	Occurs when the user chooses a widget, such as a button or most menu items.
ENTRY	Occurs when the user moves focus into a frame or field-level widget.
DEFAULT–ACTION	Occurs when the user executes the environment-specific action for completing a selection list.
ITERATION–CHANGED	Occurs when the user changes the current iteration of a browse widget.
LEAVE	Occurs when the user moves focus out of a frame or field-level widget.
MENU–DROP	Occurs when the user selects a pull-down menu from a menu bar.
VALUE–CHANGED	Occurs when the user changes the value of a combo box, toggle box, selection list, radio set, or toggle box menu item.
WINDOW–CLOSE	Occurs when the user closes a window.
WINDOW–MAXIMIZED	Occurs when the user maximizes a window.
WINDOW–MINIMIZED	Occurs when the user minimizes a window.
WINDOW–RESTORED	Occurs when the user restores the size of a window.

16.8.2 Trigger Definition

You can set up a user-interface trigger when you define or create the widget or subsequently in an ON statement.

For static widgets, triggers can be either of two types: definitional or run-time. A *definitional trigger* is specified in the statement that defines the widget and is set up at compile time. A *run-time trigger* is specified in an ON statement and is set up at run time when that statement is executed.

The following procedure defines two static buttons. It sets up a definitional trigger for the Quit button and a run-time trigger for the Next button:

p-trig1.p

```
DEFINE BUTTON quit-button  LABEL "QUIT"
    TRIGGERS:
       ON CHOOSE
          QUIT.
    END TRIGGERS.

DEFINE BUTTON next-button LABEL "NEXT".

FORM
   next-button quit-button
   WITH FRAME butt-frame.

FORM
   Customer.Cust-num Customer.Name
   WITH FRAME cframe 10 DOWN USE-TEXT.

OPEN QUERY all-cust FOR EACH Customer.

ON CHOOSE OF next-button
   DO:
      GET NEXT all-cust.
      IF NOT AVAILABLE Customer
      THEN RETURN NO-APPLY.

      DOWN WITH FRAME cframe.
      DISPLAY Customer.Cust-num Customer.Name WITH FRAME cframe.
   END.

ENABLE ALL WITH FRAME butt-frame.

GET FIRST all-cust.

DISPLAY Customer.Cust-num Customer.Name WITH FRAME cframe.

WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

In an ON statement, you can specify more than one event-widget pair for the same trigger code.

In the following example, the items in the sounds menu all share the same trigger:

p-says.p

```
DEFINE SUB-MENU sounds
   MENU-ITEM sound-cat LABEL "Cat"
   MENU-ITEM sound-cow LABEL "Cow"
   MENU-ITEM sound-dog LABEL "Dog".

DEFINE SUB-MENU exit-all
   MENU-ITEM exit-confirm LABEL "Really?".

DEFINE MENU mainbar  MENUBAR
   SUB-MENU sounds   LABEL "Sounds"
   SUB-MENU exit-all LABEL "Exit".

ASSIGN MENU-ITEM sound-cat:PRIVATE-DATA = "Meow"
       MENU-ITEM sound-cow:PRIVATE-DATA = "Mooo"
       MENU-ITEM sound-dog:PRIVATE-DATA = "Bow-wow".

ON CHOOSE OF MENU-ITEM sound-cat, MENU-ITEM sound-cow,
               MENU-ITEM sound-dog
   DO:
      MESSAGE SELF:PRIVATE-DATA.
      ASSIGN SELF:SENSITIVE = FALSE.
   END.

ON CHOOSE OF MENU-ITEM exit-confirm
   DO:
      QUIT.
   END.

CURRENT-WINDOW:MENUBAR = MENU mainbar:HANDLE.

DISPLAY "BARNYARD FUN" WITH FRAME title-frame CENTERED.

WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

Note that, by using SELF within the trigger, the behavior is customized for each widget. The trigger displays the value of the PRIVATE–DATA attribute for the selected widget and then disables that widget.

If you use the LIKE option in defining a variable or widget, the widget you are defining inherits any triggers defined on that other widget. If triggers are inherited in this way, you **cannot** specify additional definitional triggers. However, if there are no triggers to inherit, you may specify definitional triggers.

For dynamic widgets, triggers are always run-time triggers. You can specify the run-time trigger in the statement that creates the widget or in a subsequent ON statement.

16.8.3 Trigger Execution

A trigger is executed only if all of the following are true:

- The widget is enabled for input.
- The trigger is currently active.
- The user performs the action associated with the trigger, or the application executes an APPLY statement that applies the trigger event to the widget.
- If the APPLY statement is not used, one of the following must also be true:
  - Execution is blocked for input (for example, by a WAIT–FOR or UPDATE statement).
  - The PROCESS EVENTS statement is executed.

16.8.4 Trigger Scope

The scope of a trigger is that part of an application during which the trigger can be executed. The scope of a definitional trigger is the same as the scope of the widget.

By default, the scope of a run-time trigger is the nearest containing procedure, subprocedure, or trigger block in which it is defined. You might want a run-time trigger to remain active beyond the procedure or trigger that defines it. You can do this by setting up a persistent trigger. Like a definitional trigger, a *persistent trigger* remains active as long as the widget exists. A persistent trigger allows you to create the equivalent of a definitional trigger for a dynamic widget.

Table 16–6 summarizes the scope for each type of trigger.

Table 16–6: Trigger Scopes

Trigger Type	Scope
Definitional	The same as the widget for which it is defined.
Run-time, nonpersistent	The procedure, subprocedure, or trigger block in which it is defined.
Run-time, persistent	The same as the widget for which it is defined.

A persistent trigger must consist only of a single RUN statement. The RUN statement can invoke an internal or external procedure. The procedure can have one or more input parameters, but must not have any output or input/output parameters. You must ensure that the specified procedure is available whenever the trigger is invoked. For example, if you use an internal procedure, you must ensure that the trigger is not invoked from another external procedure.

NOTE: If you pass parameters to a persistent trigger procedure, the parameter values are evaluated once when the trigger is defined. They are **not** reevaluated each time the trigger executes. You **cannot**, for example, pass the SELF system handle as a parameter to the persistent trigger.

The following procedure creates a button within the CHOOSE trigger for make–butt. Normally, any trigger assigned to the new button remains active only until execution of the CHOOSE trigger ends. To maintain the new trigger beyond the CHOOSE trigger, you must use a persistent trigger:

p-pers1.p

```
DEFINE VARIABLE num-butts    AS INTEGER INITIAL 0.
DEFINE VARIABLE temp-hand    AS WIDGET-HANDLE.

DEFINE BUTTON make-butt      LABEL "Make new button".

DEFINE FRAME  butt-frame     WITH WIDTH 44.

ENABLE make-butt WITH FRAME make-frame.

ON CHOOSE OF make-butt
   DO:
      if num-butts >= 100
      THEN DO:
          MESSAGE "Too many buttons.".
          RETURN NO-APPLY.
      END.
      num-butts = num-butts + 1.
      ASSIGN FRAME butt-frame:HEIGHT-CHARS =
                     TRUNC(((num-butts - 1) / 10), 0) + 2.

      CREATE BUTTON temp-hand
           ASSIGN FRAME = FRAME butt-frame:HANDLE
                  ROW = TRUNCATE((num-butts - 1) / 10, 0) + 1
                  COLUMN = (((num-butts - 1) * 4) MODULO 40) + 1
                  LABEL = STRING(num-butts)
                  SENSITIVE = TRUE
                  VISIBLE = TRUE
           TRIGGERS:
              ON CHOOSE
                 PERSISTENT RUN but-mess.
           END TRIGGERS.
   END.

VIEW FRAME butt-frame.

WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.


PROCEDURE but-mess.

   MESSAGE "You have selected button number" SELF:LABEL
           "of" num-butts.

END PROCEDURE.
```

The example allows the user to create up to 100 dynamic buttons. Each time the user chooses make–butt, the value of num–butts is incremented, the height of the frame is recalculated to ensure the new button will fit, and the new button is created. When the user chooses one of the dynamic buttons, the procedure displays a message.

If after defining a persistent trigger for an event-widget pair you define a nonpersistent trigger for that same pair, the nonpersistent trigger overrides the persistent trigger. However, after the nonpersistent trigger goes out of scope (or is reverted), the persistent trigger becomes active again.

NOTE: Because persistent triggers remain active beyond the scope of the defining procedure, you must ensure that any variables, frames, or buffers referenced by the trigger are available whenever it is active.

16.8.5 Reverting Triggers

To cancel a trigger definition before it goes out of scope, use the REVERT option of the ON statement. The REVERT option cancels any trigger for the event-widget pair defined in the current procedure or trigger.

In the following procedure, `p-rev.p`, two triggers are defined for the selection of hello–button: one in the outer procedure and one in the internal procedure, inner. At the first WAIT–FOR statement within inner, the local trigger is active. Therefore, when the user selects the button, “Inner message” is displayed. Before the second WAIT–FOR, this trigger is reverted. The trigger from the outer procedure becomes active again. Therefore, when the user selects the button again, “Outer message” is displayed.

p-rev.p

```
DEFINE BUTTON hello-button LABEL "Greeting".
DEFINE BUTTON revert-button LABEL "Revert".

FORM
   hello-button revert-button
   WITH FRAME butt-frame.

ON CHOOSE OF hello-button IN FRAME butt-frame
   DO:
      MESSAGE "Outer message".
   END.

ENABLE hello-button WITH FRAME butt-frame.
RUN inner.

PROCEDURE inner:
   ON CHOOSE OF hello-button IN FRAME butt-frame
      DO:
         MESSAGE "Inner message".
      END.

   ENABLE revert-button WITH FRAME butt-frame.

   WAIT-FOR CHOOSE OF revert-button.

   ON CHOOSE OF hello-button IN FRAME butt-frame
      REVERT.

   DISABLE revert-button WITH FRAME butt-frame.
   WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
END PROCEDURE.
```

NOTE: You cannot use REVERT to cancel a persistent trigger. Instead, you must set up a replacement persistent trigger where the procedure consists only of a RETURN statement.

16.8.6 Universal Triggers

You can use the ANYWHERE option of the ON statement to set up a trigger that applies to all widgets in an application. This is equivalent to listing every widget in the ON statement.

The `p-utrig.p` procedure sets up a universal ENTRY trigger. This trigger is executed every time the user moves the focus to a new widget. If the Verbose menu item is off, the trigger does nothing. If Verbose is on, the trigger displays a message about the widget that receives focus:

p-utrig.p

```
DEFINE VARIABLE mess-string AS CHARACTER.
DEFINE VARIABLE text-str    AS CHARACTER FORMAT "x(12)" LABEL "String".
DEFINE VARIABLE verb-mode   AS LOGICAL.
DEFINE VARIABLE widget-type AS CHARACTER.

DEFINE BUTTON ok-butt LABEL "OK".
DEFINE BUTTON cancel-butt LABEL "CANCEL" AUTO-ENDKEY.

DEFINE SUB-MENU opt-menu
   MENU-ITEM opt-verbose TOGGLE-BOX LABEL "Verbose".

DEFINE MENU mainbar MENUBAR
   SUB-MENU opt-menu LABEL "Options".

CURRENT-WINDOW:MENUBAR = MENU mainbar:HANDLE.
ENABLE ok-butt cancel-butt WITH FRAME x TITLE "Frame 1".
ENABLE text-str WITH FRAME y TITLE "Frame 2".

ON VALUE-CHANGED OF MENU-ITEM opt-verbose
   verb-mode = NOT verb-mode.

ON ENTRY ANYWHERE
   DO:
      IF verb-mode
      THEN DO:
         widget-type = SELF:TYPE.

         mess-string = "You are entering a " + widget-type + " (".

         IF CAN-QUERY(SELF, "TITLE")
         THEN mess-string = mess-string +
              (IF SELF:TITLE <> ? THEN SELF:TITLE ELSE " ") + ")".
         ELSE IF CAN-QUERY(SELF, "LABEL")
         THEN mess-string = mess-string +
              (IF SELF:LABEL <> ? THEN SELF:LABEL ELSE " ") + ")".
         ELSE mess-string = mess-string + ")".

         MESSAGE mess-string.
      END.
   END.

WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

16.8.7 Applying Events

You can use the APPLY statement to explicitly send an event to any widget from within a program, including insensitive widgets. If a trigger is currently active for that event-widget pair, the trigger executes. Progress handles the event as if it came from the user (provides default handling) or not, depending on the event-widget pair and whether the trigger returns NO–APPLY.

The following example applies the VALUE–CHANGED event to a browse widget:

p-apply.p

```
DEFINE QUERY custq FOR customer.
DEFINE BROWSE custb QUERY custq DISPLAY cust-num name WITH 15 DOWN.

FORM
   customer EXCEPT comments
   WITH FRAME y SIDE-LABELS.

ON VALUE-CHANGED OF BROWSE custb
  DO:
     DISPLAY customer EXCEPT comments WITH FRAME y.
  END.

OPEN QUERY custq FOR EACH Customer.
ENABLE custb WITH FRAME x.

APPLY "VALUE-CHANGED" TO BROWSE custb.

ENABLE ALL WITH FRAME y.

WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

Without the APPLY statement in `p-apply.p`, the first customer record would not initially appear in frame y.

NOTE: In an APPLY statement, the event name must be enclosed in quotes (“ ”).

Default Processing of Events

In addition to executing any trigger you have defined for an event, the APPLY statement might also cause the default processing for the event-widget pair by Progress. For example, applying “A” to a fill-in field inserts the letter “A” at the cursor position in the field. However, some event-widget pairs do not get Progress default processing using the APPLY statement. For example, applying the CHOOSE event to a button executes any trigger on that button, but does not give focus to the button.

Triggers Versus Internal Procedures

If you only want your trigger to execute, consider making that trigger code an internal procedure and running the procedure rather than using the APPLY statement. For example, in the following procedure, the CHOOSE trigger for the fore–one button fetches the next customer record (or refetches the current record if there are no more records). This same action is required within several other triggers and within the main code:

p-repos3.p *(1 of 2)*

```
DEFINE BUTTON back-one  LABEL "<".
DEFINE BUTTON back-fast LABEL "<<".
DEFINE BUTTON fore-one  LABEL ">".
DEFINE BUTTON fore-fast LABEL ">>".

DEFINE QUERY scroll-cust FOR Customer SCROLLING.

FORM
  Customer.Cust-num Customer.Name SKIP
  back-fast back-one fore-one fore-fast
  WITH FRAME cust-frame.


OPEN QUERY scroll-cust FOR EACH Customer NO-LOCK.

ENABLE back-fast back-one fore-one fore-fast WITH FRAME cust-frame.

ON CHOOSE OF back-fast
  DO:
    /* Position back 10 records (or to beginning) and
       fetch  the next record from there.            */
    REPOSITION scroll-cust BACKWARD 10.
    RUN next-one.
  END.

ON CHOOSE OF back-one
  DO:
     /* Position to previous record and fetch it. */
     REPOSITION scroll-cust BACKWARD 2.
     RUN next-one.
  END.

ON CHOOSE OF fore-one
   DO:
      RUN next-one.
   END.
```

p-repos3.p *(2 of 2)*

```
ON CHOOSE OF fore-fast
   DO:
      /* Position forward 10 records (or to last record) and fetch. */
      REPOSITION scroll-cust FORWARD 9.
      RUN next-one.

   END.

/* Fetch the first record. */
RUN next-one.

WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.


PROCEDURE next-one:
      /* Fetch the next record. */
      GET NEXT scroll-cust.
      IF QUERY-OFF-END("scroll-cust") THEN
        GET PREV scroll-cust.
      DISPLAY Customer.Cust-num Customer.Name WITH FRAME cust-frame.
END PROCEDURE.
```

However, rather than applying CHOOSE to fore–one to execute the action, the fetch code has been moved to an internal procedure (next–one). That procedure is then run from within triggers and the main code as needed. Running an internal procedure is more efficient than applying an event because you avoid the overhead of event handling.

Event–Widget Independence

With the APPLY statement, you can actually send any event to any widget for which you have defined a trigger for that event. Progress executes the trigger even if there is no default handling for that event associated with the widget. Thus, you can use this feature to extend the repertoire of developer events (U1 to U10) to include any event (*non-standard event*) not normally associated with a widget.

NOTE: If you use a nonstandard event-widget association, remember that future Progress versions might make that event-widget association standard and thus introduce unexpected default handling.

For more information on the APPLY statement, see the *Progress Language Reference*.

17

Representing Data

Database fields and Progress procedure variables can be represented on the display in a variety of ways: as text, fill-ins, sliders, selection lists, combo boxes, toggle boxes, radio sets, and editors. To choose a representation for a variable or a database field, you should consider the following factors:

- The data type and extent of the value
- The graphical representation that best conveys information to end users and most easily allows them to update the value

17.1 Using the VIEW–AS Phrase

You specify a representation for a database field or variable with the VIEW–AS phrase:

- **Database fields** — You use the VIEW–AS phrase as a modifier to a screen I/O statement, such as an UPDATE or DISPLAY.
- **Variables** — You can use the VIEW–AS phrase as a modifier to a screen I/O statement, or you can use it as a modifier to the DEFINE VARIABLE statement. In many cases, it is preferable to use the VIEW–AS phrase when defining the variable; this means that code that manipulates the variable, such as an UPDATE statement, does not have to specify the representation. It also allows you to isolate your user-interface code, which can make your applications easier to port to different user-interface environments.

17.1.1 Multiple Representations for a Single Value

Both procedure logic and your deployment strategy determine whether you need more than one representation for a single value. Two common scenarios are:

- A value needs just one representation within a single user interface, but you need to alter the data representation when deploying the same application in a different environment with a different user interface. For example, you view a database character field as a fill-in in one interface, but as an editor in another.
- A value needs more than one representation within a single interface. For example, the user chooses a file to delete from a selection list, and the application displays a dialog that asks them if they really want to delete the file, giving them the opportunity to change the file in a fill-in. Note that two representations of the same underlying value cannot appear in the same frame; if a single interface has more than one representation for a value, they must appear in different frames.

17.1.2 A Value's Default Representation

If you do not use a VIEW–AS phrase with a variable or database field, Progress displays it as a fill-in that can be made into a CHOOSE field and that, when disabled, loses the border decoration typical of fill-ins in the Windows graphical environment. The same is true if you specify simply VIEW–AS FILL–IN.

However, if you specify VIEW–AS FILL–IN NATIVE, Progress displays a fill-in that is more characteristic of graphical environments: the fill-in cannot be made into a CHOOSE field, and it does not lose its border decoration when it is disabled.

17.2 Fill-ins

A fill-in is a rectangular field that contains a character representation of a value of any data type, with or without extents. The user enters new characters to update the value. The number of characters the user can enter, and hence the size of the fill-in, is usually governed by the format you specify. You can also specify an external size for the field or you can let Progress determine the size based on the format. The default formats are:

- Eight alphanumeric characters for character fields.
- Seven digits and an optional sign for integer and decimal values. For decimal values, two of the digits follow the decimal point.
- Three characters for logical values. By default, the value must be either yes or no.
- A format of 99/99/99 for date values.

Because a fill-in is the default representation for fields, variables, and expressions, you usually do not have to specify it. However, you can use the following syntax to explicitly view a value as a fill-in:

SYNTAX

```
VIEW-AS FILL-IN [ NATIVE ] [ size-phrase ]
  [ TOOLTIP tooltip ]
```

Progress supports two subtypes of fill-ins: default Progress fill-ins and native graphical fill-ins. For full backwards compatibility, default Progress fill-ins provide support for the CHOOSE FIELD statement. Native graphical fill-ins look and behave like other fill-ins within the Windows environment.

When a non–NATIVE (Progress 4GL) fill–in is disabled, the border disappears but the text does not gray out. When a NATIVE fill–in is disabled, the text grays out.

If you want a native fill-in, you can specify VIEW–AS FILL–IN NATIVE for the field or set the widget's SUBTYPE attribute to "NATIVE" (the default is "PROGRESS"). You cannot change this attribute after the fill-in is realized.

17.2.1 Format and Size

By default the size of a fill-in on the screen is determined by its format and the font used when the code is compiled. For example, if a fill-in has a format of x(8) then 8 characters are visible on the screen. The external width of the fill-in might be 9 or 10 characters to allow for the side borders.

For proportional fonts on Windows, Progress considers the width of the widest character in the font (*max–char–width*) and the average width of all characters in the font (*ave–char–width*) to determine the width of a fill-in. Also, the rules differ for the default system font versus any other font.

The rules for the default system font are as follows:

- If the value is 3 characters or less, Progress multiples the length of the string times the width of the widest character in the font: *string–length * max–char–width*
- If the value is more than 3 characters, but less than 13, Progress uses the widest character in the font for 3 characters and the average character for the remaining characters: (3 * *max–char–width*) + ((*string–length* – 3) * *ave–char–width*)
- If the value is 13 characters or more, Progress multiplies the length of the string times the average character width for the font: *string–length * ave–char–width*.

For all other fonts than the default system font, Progress multiplies the length of the string (for all string lengths) times the average character width for the font: *string–length * ave–char–width*.

Sometimes you want to explicitly set the size of the widget regardless of format. To prevent Progress from resizing the field to fit the format, either set the AUTO–RESIZE attribute to FALSE or explicitly specify a size within the VIEW–AS FILL–IN phrase or by setting the fields WIDTH and HEIGHT attributes.

The following example defines two variables that are viewed as fill-ins. For the first, filename, the procedure specifies a large format but a smaller size. For the second, no_scroll, the default format, x(8), and size are used:

p-fillin.p

```
DEFINE VARIABLE filename AS CHARACTER FORMAT "X(80)"
                            VIEW-AS FILL-IN SIZE 42 BY 1.
DEFINE VARIABLE no-scroll AS CHARACTER.

DEFINE FRAME f
       filename no-scroll.

UPDATE filename no-scroll WITH FRAME f.
```

Because the procedure specifies a width of 42 character for filename, only about 40 characters are visible on the screen (the other character positions are used for side borders). Because the format is greater than the number of visible characters, filename becomes a horizontally scrolling field. When updating the field, when you type past the right edge of the field, the value automatically scrolls so that you can enter more characters.

For more information, see Chapter 16, "Widgets and Handles."

17.2.2 Fill-in Attributes

At run time, a procedure can query or set fill-in specific attributes in these ways:

- Set AUTO–ZAP to TRUE to allow the initial screen value of the fill-in to be erased when the user begins typing in the field. The fill-in must have input focus to set AUTO–ZAP.
- Query or set the FORMAT attribute. The format determines the number of spaces and special characters (such as dollar signs and decimal points) Progress uses when displaying the fill-in. Note that the format also establishes a constraint on the data that the user can enter. However, changing the format has no effect on the data type of the field. For more information, see "Applying Formats When Displaying Widgets" later in this chapter.
- Enable or disable automatic tabbing with the AUTO–RETURN attribute. If this attribute is true, Progress tabs to the next widget when the fill-in is full. If you set AUTO–RETURN to true for the last widget in a frame, then when the field is full, Progress issues a GO event for the frame.
- Enable or disable character-echo during data entry with the BLANK attribute.
- Query the FRAME–NAME attribute to find the name of the frame that contains the field. This is equivalent to using the FRAME–NAME function and is useful for context-sensitive help.
- Query the DBNAME, INDEX, and TABLE attributes to find the name and index of the corresponding database field and the table in which that database field resides.

NOTE: The value of the SESSION handle's DATA–ENTRY–RETURN attribute determines the meaning of the **RETURN** key in a fill-in. If the attribute is TRUE, Progress interprets a **RETURN** as a TAB event for most fill-ins and as a GO event for a fill-in if that fill-in is the last tab item in a frame family (Version 6 behavior). In this case, the GO event is propagated to all frames of the frame family. If the attribute is FALSE, the window system determines the meaning of the **RETURN** key. The default value is TRUE for character interfaces and FALSE for graphical interfaces. For more information on frame families, see Chapter 19, "Frames." For more information on tab order, tab items, and DATA–ENTRY–RETURN, see Chapter 25, "Interface Design."

17.2.3 Fill-in Fonts on Windows in Graphical Interfaces

On Windows, in graphical interfaces, if no font is specified for the fill-in, Progress uses two default fonts:

- A fixed font for date fields, numeric fields, and for character fields that contain fill characters (such as the parentheses surrounding the area code of a telephone number).
- A proportional font for character fields that do not contain fill characters.

Progress looks for definitions of these fonts in the current environment, which might be the registry or an initialization file. If the current environment does not define these fonts, Progress uses the system default fixed and proportional fonts. For more information on environments, see the Chapter 23, "Colors and Fonts." In this book and the chapter on user interface environments in the *Progress Client Deployment Guide*.

17.3 Text

On Windows, fill-in fields use extra space on the screen for border decorations. To save space for read-only text, you can view the value as a text widget rather than a fill-in. (If you want to preserve the native look, you can view it as a native fill-in but set the SENSITIVE attribute to FALSE.)

NOTE: You cannot update a field that is displayed as a text widget. Use the text widget only for read-only data (for example, report data or field labels).

You can use the following form of the VIEW–AS phrase to view a value as text:

SYNTAX

```
VIEW-AS TEXT [ size-phrase ] [ TOOLTIP tooltip ]
```

The following code displays customer names as fill-ins.

```
FOR EACH customer:
  DISPLAY customer.name.
END.
```

Notice the spacing between customer names in the resulting screen:



To eliminate this space, view the customer name as text.

```
FOR EACH customer:
  DISPLAY customer.name VIEW-AS TEXT.
END.
```

When you run this code, the screen appears as follows:



You can specify that all fields in a frame be displayed as text rather than as fill-ins by specifying USE–TEXT in the frame phrase. This is especially useful for reports.

17.4 Sliders

A *slider* is a rectangular scale that represents a range of INTEGER values. A pointer that resides on the trackbar within the scale indicates the current value. The current value is also displayed in character format near the scale. Only values of type INTEGER can be viewed as sliders.

In a graphical interface, a slider takes on the native look of the windowing system. In a character interface, the location of the pointer on the slider is indicated by reverse video; if the slider has focus, and the current value option is set, square brackets appear around the current value.

The following form of the VIEW–AS phrase defines a slider:

SYNTAX

```
VIEW-AS SLIDER
  MAX-VALUE max-value MIN-VALUE min-value
  [ HORIZONTAL | VERTICAL ]
  [ NO-CURRENT-VALUE ]
  [ LARGE-TO-SMALL ]
  [ TIC-MARKS { NONE | TOP | BOTTOM | LEFT | RIGHT | BOTH }
    [ FREQUENCY n ] ]
  [ TOOLTIP tooltip ]
  [ size-phrase ]
```

NOTE: The options TIC–MARKS, FREQUENCY, and TOOLTIP presented in the the slider syntax box are supported on Windows only.

In both character and graphical interfaces, the slider can be oriented horizontally (with low values to the left and high values to the right) or vertically (with low values at the bottom and high values at the top). The default orientation is horizontal. You can set the orientation by specifying HORIZONTAL or VERTICAL within the VIEW–AS phrase or by setting the HORIZONTAL attribute to TRUE or FALSE before the slider is realized.

Also, in both character and graphical interfaces, you can change the default numeric range from small (minimum) to large (maximum) in addition to changing the default horizontal orientation of the slider. Using the LARGE–TO–SMALL option, you can override the default numeric range as follows:

- When the slider is positioned horizontally, the left-most position on the trackbar displays the maximum value and the right-most position displays the minimum value.
- When the slider is positioned vertically, the bottom-most position on the trackbar displays the maximum value and the top-most position displays the minimum value.

In both interfaces, the NO–CURRENT–VALUE option provides additional flexibility in terms of the slider's presentation. By default, the current value for a given position on the trackbar displays. However, if you use the NO–CURRENT–VALUE option, you can override this default behavior to indicate that the slider will not automatically display the current value as the pointer moves along the trackbar.

17.4.1 Manipulating a Slider

In a character interface, you can change the value of a slider by using the left and right arrow keys (for a horizontal slider) or the up and down arrow keys (for a vertical slider).

To change a slider value when a slider displays on Windows:

- Point to the pointer, press the SELECT or EXTEND mouse button, and drag or slide the pointer.
- Use the left and right arrow keys (for a horizontal slider) or the up and down arrow keys (for a vertical slider).
- Click on the trackbar, but not on the pointer. The pointer will move in larger increments in the direction in which you clicked.

17.4.2 Range and Size

When defining a slider, you must minimally specify the slider's range. That is, you must specify a minimum and maximum value for the widget. Specify the range using the MIN–VALUE and MAX–VALUE options, each of which takes an integer constant. You can change these values by setting the MIN–VALUE and MAX–VALUE attributes of the slider before the slider is realized. Depending on the windowing system in use, the minimum value, maximum value, or both may be displayed with the slider.

The range you specify for a slider has no effect on the size of the slider. For example, a horizontal slider with range from 0 to 100 has the same default width as a slider with a range for 0 to 10. If a slider has a very wide range, you might want to increase its size so that you can precisely position the marker with the mouse. Also, in a graphical interface, you might consider using the TIC–MARKS option to display short hash marks on the outside of the slider. Used with the FREQUENCY option, TIC–MARKS help reference specific increments for these hash marks along the trackbar to provide a more precise indication of the value selected on your slider.

In a character interface, a horizontal slider is only 6 positions wide by default and a vertical slider slider is only 3 positions high. If you want the marker to accurately reflect the current value for a slider with a large range, you must make the slider bigger.

You can set the size of a slider by using the SIZE–CHARS or SIZE–PIXELS option of the VIEW–AS phrase. Each of these options take a width and height value. You can also adjust the size by setting the HEIGHT–CHARS and WIDTH–CHARS or HEIGHT–PIXELS and WIDTH–PIXELS attributes.

NOTE: In a character interface, specify a SIZE that is a multiple of the range (MAX–VALUE – MIN–VALUE) to ensure that the marker moves in even increments.

When adjusting the size of a slider you are most concerned with the width for a horizontal slider and the height for a vertical slider. However, you should be careful about changing the other dimension. For example, if you make the height of a horizontal slider too small, the value displayed with the slider may be partially obscured.

17.4.3 Example Procedure

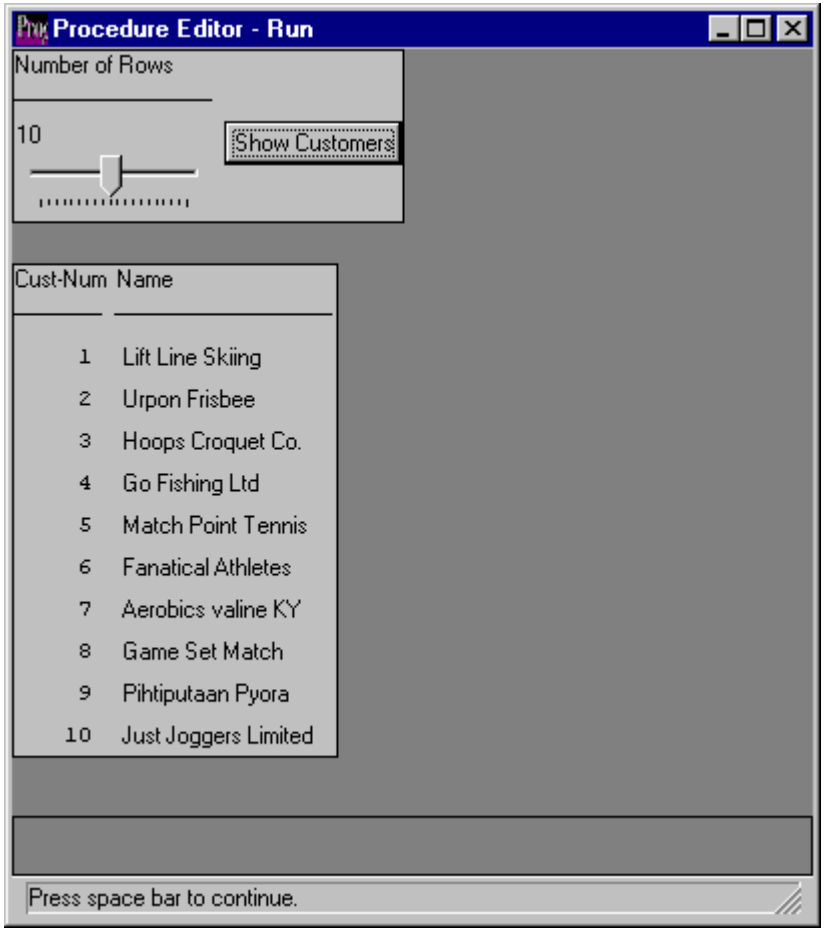
The following example procedure, p-slider.p, defines an INTEGER variable, num–rows, which is viewed as a slider. The slider's range is initially set as 1 to 20. However, before the widget is realized, the maximum value is adjusted based on the current window size.

p-slider.p

```
DEFINE VARIABLE num-rows AS INTEGER LABEL "Number of Rows"
                        VIEW-AS SLIDER MIN-VALUE 1 MAX-VALUE 20
                        TIC-MARKS BOTTOM FREQUENCY 1.
DEFINE BUTTON show-custs LABEL "Show Customers".

FORM
   num-rows HELP "Choose the number of customers per screen."
   show-custs
   WITH FRAME main-frame THREE-D.

DEFINE FRAME cust-frame
   customer.cust-num customer.name
   WITH DOWN THREE-D.

ON CHOOSE OF show-custs
   DO:
      CLEAR FRAME cust-frame ALL.
      FRAME cust-frame:DOWN =
           INTEGER(num-rows:SCREEN-VALUE IN FRAME main-frame).
      FOR EACH customer:
         DISPLAY cust-num name WITH FRAME cust-frame.
         DOWN WITH FRAME cust-frame.
      END.

      HIDE FRAME cust-frame.
   END.

num-rows:MAX-VALUE IN FRAME main-frame = SCREEN-LINES - 2.

ENABLE ALL WITH FRAME main-frame.

WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

When you choose the Show Customers buttons, the size of the cust–frame is adjusted based on the current value of the slider. The customer records are then displayed in that frame.

For example, if you set the slider to 10 and press **GO**, the screen appears as follows:



17.5 Toggle Boxes

A toggle box, sometimes called a check box, represents a logical value. The presence (true) or absence (false) of filling in the toggle box indicates the current value. In a graphical interface, a toggle box and its filling style take on the native look of the windowing system. You can change a toggle value by positioning to it and pressing **SPACEBAR**. In a graphical interface, you can point to the box and click the mouse SELECT button.

In a character interface, a toggle box is represented by a pair of square brackets ([ ]) that are either empty (False) or contain an X (True).

Note that setting the FORMAT option or attribute does not affect the the toggle box widget. You see the effects of the FORMAT change only if you display the value as a fill-in or text.

The following form of the VIEW–AS phrase defines a toggle box:

SYNTAX

```
VIEW-AS TOGGLE-BOX [ size-phrase ] [ TOOLTIP tooltip ]
```

There are no required options to use with the VIEW–AS phrase when defining a toggle box. You can optionally specify a label for the toggle box. Specify the label as a quoted character string.

If you do not use the LABEL option, the field or variable name is used. For an array, the default label includes the subscript in brackets ([ ]).

The following example, p-tbox.p, uses toggle boxes to represent an array of extent 3:

p-tbox.p *(1 of 2)*

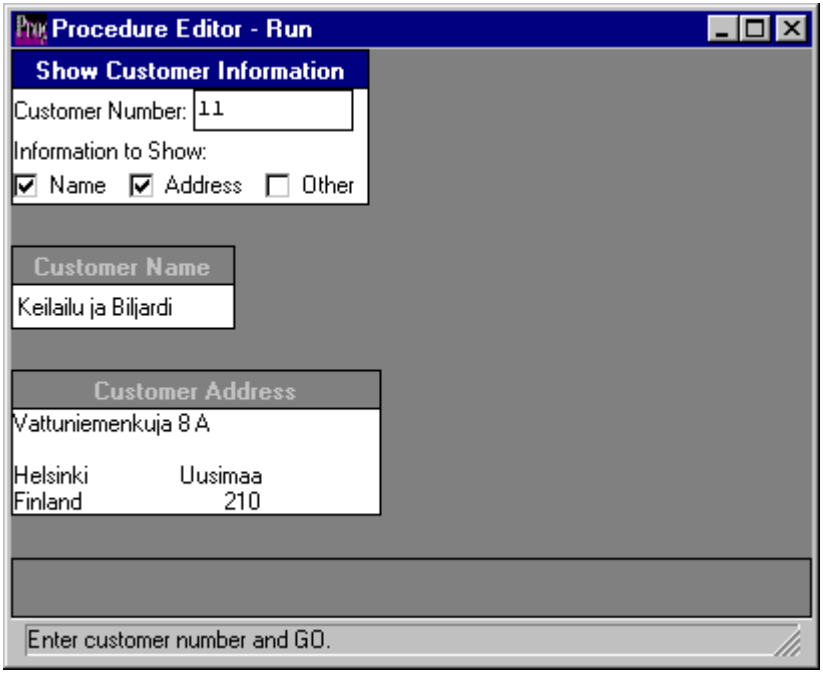
```
DEFINE VARIABLE choices   AS LOGICAL EXTENT 3
                          LABEL "Name", "Address", "Other"
                          INITIAL [yes, yes, no]
                          VIEW-AS TOGGLE-BOX.
DEFINE VARIABLE curcust AS INTEGER.

FORM
    curcust LABEL "Customer Number" SKIP
    "Information to Show:" SKIP
    choices SKIP
    WITH SIDE-LABELS FRAME choice-frame TITLE "Show Customer Information".

FORM
    customer.name
    WITH FRAME name-frame NO-LABELS TITLE "Customer Name".FORM
    customer.address SKIP customer.address2 SKIP
    customer.city customer.state SKIP
    customer.country customer.postal-code
    WITH FRAME addr-frame NO-LABELS USE-TEXT TITLE "Customer Address".

FORM
    customer
    EXCEPT name cust-num address address2 city state country postal-code
    WITH FRAME oth-frame SIDE-LABELS USE-TEXT TITLE "Other Customer Data".
```

p-tbox.p *(2 of 2)*

```
ON GO OF FRAME choice-frame OR RETURN OF curcust
    DO:
        HIDE FRAME name-frame FRAME addr-frame FRAME oth-frame.

        FIND  customer WHERE cust-num = INPUT curcust NO-ERROR.
        IF AMBIGUOUS customer OR NOT AVAILABLE customer
        THEN RETURN NO-APPLY.

        IF INPUT choices[1] /* name */
        THEN DISPLAY customer.name WITH FRAME name-frame.
        IF INPUT choices[2] /* address */
        THEN DISPLAY customer.address customer.address2
                     customer.city customer.state country postal-code
             WITH FRAME addr-frame.
        IF INPUT choices[3] /* other */
        THEN DISPLAY customer EXCEPT name cust-num address
                     address2 city state country postal-code
             WITH FRAME oth-frame.
        RETURN NO-APPLY.
    END.

DISPLAY choices WITH FRAME choice-frame.
ENABLE ALL WITH FRAME choice-frame.
STATUS INPUT "Enter customer number and GO.".

WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

When you run the example, you are prompted for a customer number. You can also set the three toggle boxes labeled cust–num, address, and other. You can set the toggle boxes to choose what information to display about the customer. When you press **GO**, the trigger finds the customer record and displays the information you specified.

For example, if you accept the default values for choices and press GO, the screen appears as follows:



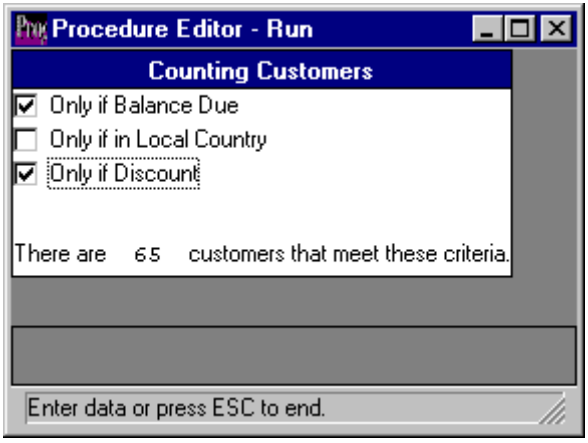
In `p-tbox.p`, changing the value of a toggle box has no immediate effect. The toggle boxes are used to set a state that is then read when you press **GO**. By writing a trigger for the the VALUE–CHANGED event, you can also use a toggle box to trigger an immediate action.

The following example, `p-tbox2.p`, displays three toggle boxes that specify search criteria for customers. It displays the number of customers that meet the current criteria. Each time you change the value of a toggle box, the procedure recalculates the number of customers that meet the specified criteria.

p-tbox2.p

```
DEFINE VARIABLE cnt AS INTEGER FORMAT ">>9".
DEFINE VARIABLE must-have-balance AS LOGICAL INITIAL no
        VIEW-AS TOGGLE-BOX LABEL "Only if Balance Due".
DEFINE VARIABLE must-be-local AS LOGICAL INITIAL no
        VIEW-AS TOGGLE-BOX LABEL "Only if in Local Country".
DEFINE VARIABLE must-get-discount AS LOGICAL INITIAL no
        VIEW-AS TOGGLE-BOX LABEL "Only if Discount".
DEFINE VARIABLE local-country AS CHARACTER INITIAL "USA".

FORM
   must-have-balance SKIP
   must-be-local SKIP
   must-get-discount SKIP(1)
   "There are" cnt "customers that meet these criteria."
   WITH FRAME count-frame TITLE "Counting Customers" NO-LABELS.

ON VALUE-CHANGED OF must-have-balance, must-be-local, must-get-discount
   DO:
     DEFINE VARIABLE meets-criteria AS LOGICAL.
     cnt = 0.
     FOR EACH customer:
         meets-criteria = yes.
         IF INPUT must-have-balance AND customer.balance <= 0
         THEN meets-criteria = no.
         ELSE IF INPUT must-be-local AND customer.country <> local-country
         THEN meets-criteria = no.
         ELSE IF INPUT must-get-discount AND customer.discount = 0
         THEN meets-criteria = no.

         IF meets-criteria
         THEN cnt = cnt + 1.
     END.
     DISPLAY cnt WITH FRAME count-frame.
   END.

cnt = 0.
FOR EACH customer:
    cnt = cnt + 1.
END.

DISPLAY must-have-balance must-be-local must-get-discount cnt
        WITH FRAME count-frame.
ENABLE must-have-balance must-be-local must-get-discount
       WITH FRAME count-frame.

WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

The VALUE–CHANGED event occurs every time you alter the value of a toggle box. In `p-tbox2.p`, each time you change the value of one of the toggle boxes, the procedure recalculates the number of customers that meet the criteria specified.



17.6 Radio Sets

A *radio set* is a series of buttons (radio items) that represent a single field or variable. The term radio set derives from the preset tuning buttons on many car radios. When one button is pressed, any previously selected button pops out. Each of the buttons represents a different value for the field or variable. At any one time, exactly one of the buttons is chosen. If the user chooses a different button, the previously chosen button is automatically deselected.

In a graphical interface, a radio set takes on the native look of the windowing system. In the character interface, each radio item is represented by a set of parentheses. The parentheses either contain a space character (FALSE) or the letter X (TRUE).

A radio set can represent a LOGICAL, CHARACTER, INTEGER, DECIMAL, or DATE value. Radio sets are appropriate for fields that can have only one of a limited number of values.

17.6.1 Defining a Radio Set

The following form of the VIEW–AS phrase defines a radio set:

SYNTAX

```
VIEW-AS RADIO-SET
  [ HORIZONTAL [ EXPAND ] | VERTICAL ]
  [ size-phrase ] RADIO-BUTTONS label , value
      [, label , value ] ...
  [ TOOLTIP tooltip ]
```

When viewing a value as a radio set, you must minimally specify one label/value pair for each possible value of the underlying variable or field (the label is the label that appears with the button). You can optionally specify an orientation for the set (the default is vertical) and a size. For a horizontal radio set, you can also specify the EXPAND option. This option spaces the buttons uniformly based on the length of the longest button label. If you do not give this option, the spacing between buttons varies depending on the size of the labels.

17.6.2 Example Procedures

The following example defines an integer variable, pay–stat, that can have a value of 1, 2, or 3. It prompts the user to update the variable by displaying it as a radio set. It then redisplays the variable as a fill-in.

By viewing the variable as a radio set, the user is constrained to selecting one and only one of the meaningful values for the variable. Since each integer value represents a different state for an outstanding invoice, the procedure defines meaningful labels that describe these states and displays these in each button, rather than displaying the integer values:

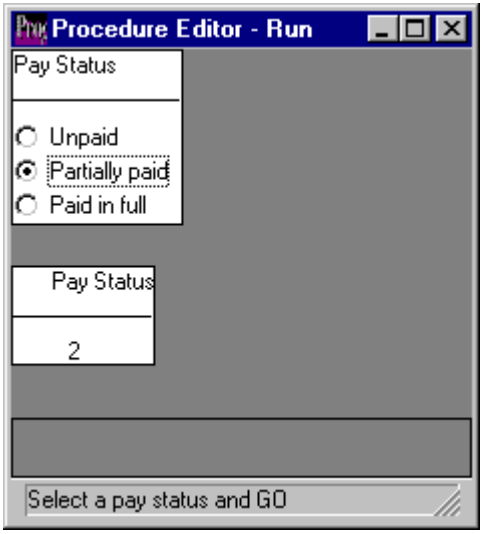
p-radio1.p

```
DEFINE VARIABLE pay-stat AS INTEGER INITIAL 1 LABEL "Pay Status"
                   VIEW-AS RADIO-SET RADIO-BUTTONS "Unpaid", 1,
                                                   "Partially paid", 2,
                                                   "Paid in full", 3.
FORM
   pay-stat
   WITH FRAME in-frame.FORM
  pay-stat VIEW-AS TEXT
  WITH FRAME out-frame.

ON GO OF FRAME in-frame
   DO:
      ASSIGN pay-stat.
      DISPLAY pay-stat WITH FRAME out-frame.
   END.

DISPLAY pay-stat WITH FRAME in-frame.
ENABLE pay-stat WITH FRAME in-frame.
STATUS INPUT "Select a pay status and GO".

WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

NOTE: When you view a field or variable as a radio set, if it is not set to a valid value, it defaults to the first radio button. In the example, pay–stat is explicitly initialized to 1.

When you run this procedure, because pay–stat is initialized to 1, the first radio button is initially chosen. You can either accept that value or choose another button. When you press `GO`, the value is assigned to pay–stat and then viewed as a text widget.

For example, if you select the second radio button and press GO, the screen appears as follows:



Note that when you select the second button, the first button is automatically deselected.

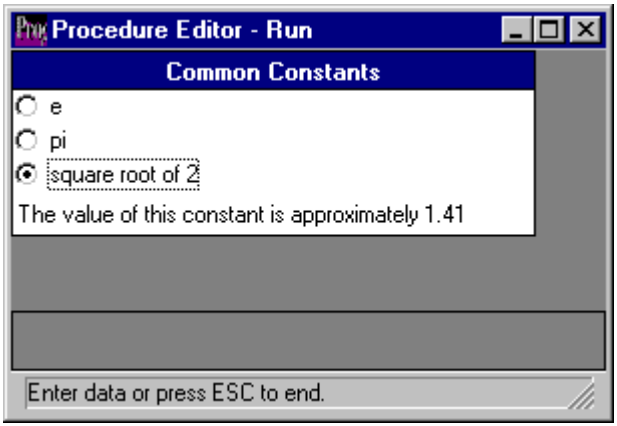
As with toggle boxes, you can also use the VALUE–CHANGED event with a radio set. This gives the user instant feedback when a new radio button is chosen, as in the following example:

p-radio2.p

```
DEFINE VARIABLE math-const AS DECIMAL FORMAT "9.999"
        VIEW-AS RADIO-SET RADIO-BUTTONS "e", 2.72,
                                        "pi", 3.14,
                                        "square root of 2", 1.41
        INITIAL 2.72.

FORM
   math-const SKIP
   description AS CHARACTER FORMAT "x(50)"
   WITH FRAME const-frame NO-LABELS TITLE "Common Constants".

ON VALUE-CHANGED OF math-const
   DO:
      DISPLAY "The value of this constant is approximately " +
              SELF:SCREEN-VALUE @ description WITH FRAME const-frame.
   END.

DISPLAY math-const WITH FRAME const-frame.
ENABLE math-const WITH FRAME const-frame.
APPLY "VALUE-CHANGED" TO math-const.

WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

When you run this procedure, the description message changes immediately each time you choose a different radio button.

For example, if you select the second radio button, the screen appears as follows:



17.7 Editors

An editor allows full editing of large character fields. The editor widget supports features such as cut, copy, paste, and word-wrap.

17.7.1 Defining an Editor

The following form of the VIEW–AS phrase defines an editor:

SYNTAX

```
VIEW-AS EDITOR
  {     size-phrase
      | INNER-CHARS char INNER-LINES lines
  }
  [ BUFFER-CHARS chars ]
  [ BUFFER-LINES lines ]
  [ LARGE ]
  [ MAX-CHARS characters ]
  [ NO-WORD-WRAP ]
  [ SCROLLBAR-HORIZONTAL ]
  [ SCROLLBAR-VERTICAL ]
  [ TOOLTIP tooltip ]
```

When viewing a character string as an editor, you must specify a size for the editor in one of two ways:

- Specify a width and height for the editor using the SIZE phrase.
- Specify the number of lines visible within the editor using the INNER–LINES option and the number of characters visible in each line using the INNER–CHARS option.

You can optionally limit the number of characters that the user can enter or that can be displayed within the editor by specifying the MAX–CHARS option or setting the MAX–CHARS attribute. You can also enable horizontal scrolling (thereby disabling word-wrap) and display scroll bars.

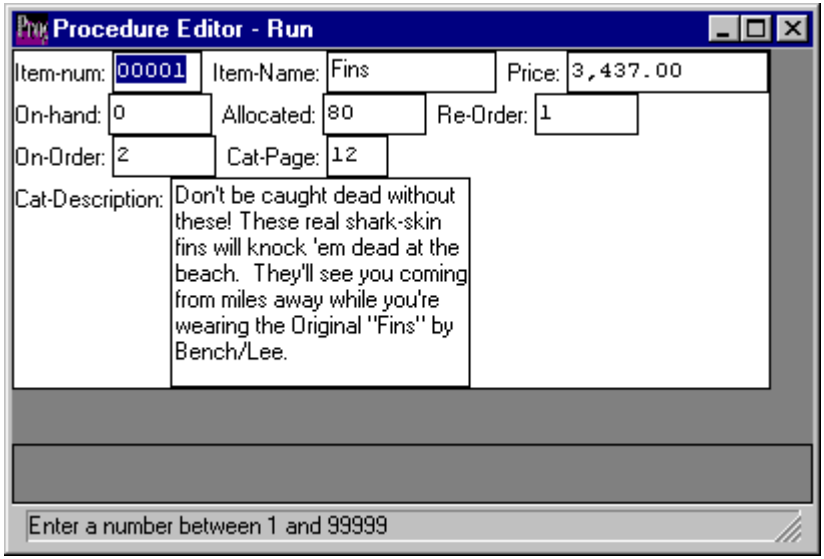
On Windows, you can specify an editor that handles larger quantities of text than normal using the LARGE option. For more information, see the "Using an Editor" section.

The p-edit.p procedure displays the cat–description field of the item table as an editor that is 30 characters wide and 5 characters long:

p-edit.p

```
FORM
   item.item-num item-name item.price on-hand
   allocated re-order on-order cat-page
   item.cat-description
      VIEW-AS EDITOR SIZE-CHARS 30 BY 5
   WITH FRAME x SIDE-LABELS.

FOR EACH item:
   UPDATE item WITH FRAME x.
END.
```

When you run p-edit.p, the screen appears as follows:



17.7.2 Using an Editor

By default, editors support vertical scrolling with the up and down arrow keys, whether or not you specify scroll bars for the editor. Therefore, in the example, the user can type more than four lines of text within the editor. When the user moves beyond the last line shown, the first line scrolls up and out of the visible editing area; a new line scrolls into bottom of the visible area. If the user later moves the cursor up to the first line, the last line scrolls down and out of the visible editing area and the first line becomes visible again. To allow the user to scroll vertically using a scroll bar, specify the SCROLLBAR–VERTICAL option in the VIEW–AS phrase.

Word Wrap

By default, the editor also supports word-wrap. This means that when you reach the end of a line within the editor, text wraps to the next line rather than scrolling to the right. You can choose to enable horizontal scrolling instead by specifying the NO–WORD–WRAP option in the VIEW–AS phrase. You can also specify SCROLLBAR–HORIZONTAL along with NO–WORD–WRAP to add a horizontal scroll bar to the editor. If you specify NO–WORD–WRAP, but not SCROLLBAR–HORIZONTAL, the user can scroll horizontally by using the left and right arrow keys at the edge of the displayed text.

Graphical Editors

Within an editor in a graphical environment, the user can select a range of text by holding down the SELECT mouse button and dragging the cursor. The user can delete the selected text by pressing **DELETE**. You can program further functionality for the editor by manipulating the editor's attributes and methods.

Also, when a graphical editor is disabled (made insensitive to input), Progress grays out the contents and restores it when the editor is re-enabled.

Windows Editors

On Windows, editor widgets are limited to 20K of data, by default. To support more data, you must specify the LARGE option in the VIEW–AS phrase or set the LARGE attribute to TRUE. This raises the limit according to available system resources. If you try to enter more than 20K of data into a normal editor widget on Windows, or set MAX–CHARS greater than 20K, Progress displays a warning message.

NOTE: The LARGE option of the VIEW–AS phrase and the LARGE attribute apply only to graphical interfaces on Windows.

Character Editors

In a character interface, the user can move to the beginning or end of an editor by using the **TOP–COLUMN** or **BOTTOM–COLUMN** keys. The **DELETE–FIELD** key deletes a word to the right of the cursor. The **DELETE–COLUMN** keys deletes to the end of the current line. The **INSERT–COLUMN** key inserts a new line above the current line. The **RETURN** key creates a new line after the current line. The user can toggle between inserting and overtyping text using the **INSERT–MODE** key.

NOTE: In character interfaces, the editor does not support the tab character. When Progress reads a file that contains tabs into a character editor widget, it replaces the tabs with eight space characters. When it writes out the file, the tabs are not restored, and the file is permanently changed.

17.7.3 Attributes and Methods

At run time, you can query or set editor-specific attributes to accomplish the following:

- Enable automatic indentation when the user presses **RETURN**
- Select text or place the text cursor
- Determine where the user has placed the text cursor
- Determine whether the user has selected text and the location of the selection
- Determine whether the user has modified the contents of the editor and find the length
- Change the compile-time specifications of INNER–CHARS, INNER–LINES, MAX–CHARS, WORD–WRAP, SCROLLBAR–HORIZONTAL, and SCROLLBAR–VERTICAL

NOTE: Although you can store a large amount of data in an editor widget, the SCREEN–VALUE attribute might not work as expected for values longer than 32K.

In addition to attributes, the editor also supports methods that allow a procedure to:

- Delete individual characters, text selections, or entire lines
- Insert strings or files at the current cursor position
- Search for, search-and-replace, or select text
- Convert a row and column position to a character offset
- Save the contents of the editor to a file

For more information on each attribute and method, see the *Progress Language Reference*.

17.7.4 Adding Functionality to an Editor

The following example, `p-edit2.p`, provides a more functional editor than `p-edit1.p`. This example lets you read an operating system file into the editor, make modifications to it, and then save the results. To do this, the example uses a menu bar and the READ–FILE and SAVE–FILE methods of the editor widget:

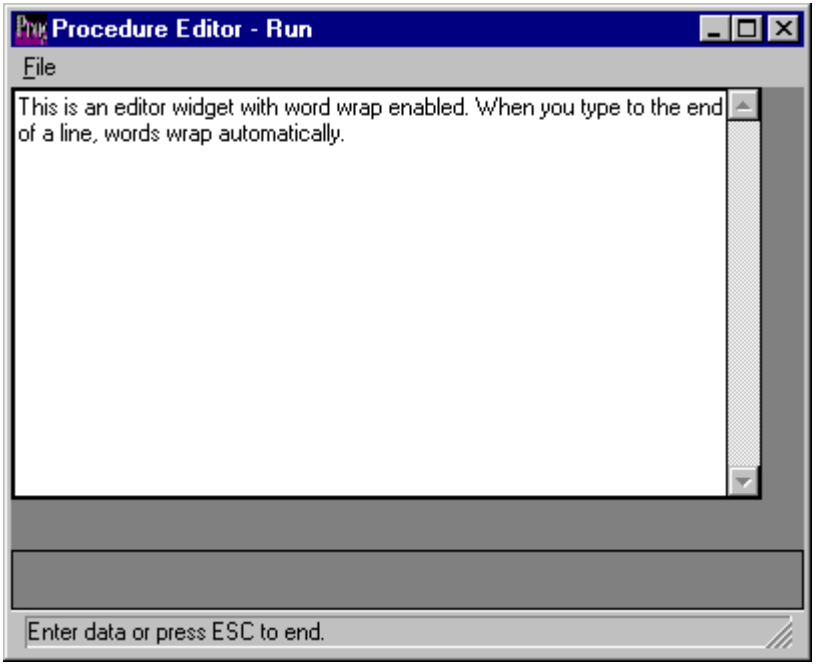
p-edit2.p *(1 of 2)*

```
DEFINE VARIABLE e AS CHARACTER VIEW-AS EDITOR
                     INNER-CHARS 70 INNER-LINES 15 SCROLLBAR-VERTICAL.
DEFINE VARIABLE filename AS CHARACTER FORMAT "x(60)".
DEFINE VARIABLE status-ok AS LOGICAL.

DEFINE SUB-MENU filemenu
   MENU-ITEM fopen LABEL "Open..."
   MENU-ITEM fsave LABEL "Save As..."
   MENU-ITEM fexit LABEL "E&xit".

DEFINE MENU mainbar MENUBAR
   SUB-MENU filemenu LABEL "File".

FORM
  e
  WITH FRAME edit-frame.

FORM
  "Enter Filename:" SKIP
  filename
  WITH FRAME file-spec NO-LABELS VIEW-AS DIALOG-BOX.

ON RETURN OF filename IN FRAME file-spec
   APPLY "GO" TO filename.

ON CHOOSE OF MENU-ITEM fopen
   DO:
      FRAME file-spec:TITLE = "Open File".
      UPDATE filename WITH FRAME file-spec.
      status-ok = e:READ-FILE(filename) IN FRAME edit-frame.
      IF NOT status-ok
      THEN MESSAGE "Could not read" filename.
   END.
```

p-edit2.p *(2 of 2)*

```
ON CHOOSE OF MENU-ITEM fsave
   DO:
      FRAME file-spec:TITLE = "Save File".
      UPDATE filename WITH FRAME file-spec.
      status-ok = e:SAVE-FILE(filename) IN FRAME edit-frame.
      IF NOT status-ok
      THEN MESSAGE "Could not write to" filename.
   END.

CURRENT-WINDOW:MENUBAR = MENU mainbar:HANDLE.

ENABLE e WITH FRAME edit-frame NO-LABELS.

WAIT-FOR CHOOSE OF MENU-ITEM fexit OR
         WINDOW-CLOSE OF CURRENT-WINDOW.
```

When you run this example, you can use the menu items to read from a file or type text directly into the editor.



You can increase the functionality of this editor further by adding additional options to the menu bar and using additional editor methods. For example, you can add an Edit submenu that includes a search–and–replace option using the REPLACE method.

NOTE: When you save the editor text to a file, some characters save differently, depending on your operating system. For example, **RETURN** key input writes out on Windows as x0d0d0a. On UNIX, this writes out as x0d0a. Similarly, the end-of-line on Windows writes out as <CR><LF>. On UNIX, this writes out as <CR> alone. Future versions of Progress might address these issues, allowing for improved portability among platforms.

17.8 Selection Lists

A *selection list* is a scrollable list of values for a CHARACTER field or variable. You can use a selection list to allow an end user to select one or more items from a finite list of character values. You can move within a selection list using the up and down arrow keys. In graphical and character interfaces, you can position to a value in the list by typing the first letter of the value.

On Windows, the user can select a value by pointing at it and clicking the SELECT mouse button. In a character interface, the user can select a value by positioning to it and pressing either **RETURN** or **SPACEBAR**. When an item is selected, the VALUE–CHANGED event occurs.

Typically, after the user selects one or more items from the list, you want to perform some action. For example, if a selection list contains a list of tasks, you might want to run a procedure to perform the selected task. Running the associated procedure is the *default action* for that selection list. You establish the default action by coding a trigger for the DEFAULT–ACTION event. This event occurs when the user:

- Double-clicks the SELECT mouse button on an item or positions to the item and presses **RETURN** on Windows.
- Presses the **ENTER** key in character mode.

17.8.1 Defining a Selection List

The following form of the VIEW–AS phrase defines a selection list.

SYNTAX

```
VIEW-AS SELECTION-LIST
  [ SINGLE | MULTIPLE ]
  [ NO-DRAG ]
  [ LIST-ITEMS item-list ]
  [ SCROLLBAR-HORIZONTAL ]
  [ SCROLLBAR-VERTICAL ]
  { size-phrase | INNER-CHARS cols INNER-LINES rows }
  [ SORT ]
  [ TOOLTIP tooltip ]
```

When viewing a value as a selection list, you must minimally specify the size of the selection list in one of the following ways:

- Using the SIZE, SIZE–CHARS, or SIZE–PIXELS option
- Using the INNER–CHARS and INNER–LINES options
- Using the LIKE option of the DEFINE VARIABLE statement to define a variable similar to another variable or field defined as a selection list

You can optionally specify:

- The LIST–ITEMS option to populate the list with items.
- The SINGLE option to indicate that the user can select only one of the values or the MULTIPLE option to indicate that the user can select more than one value. If more than one item is selected, then the field or variable value is a comma-separated list of the selected items.
- The NO–DRAG option to indicate that the user cannot select items by simultaneously holding down the mouse SELECT button and dragging the mouse through the list.

17.8.2 Scrolling

If a list is not long enough to display all the values on the screen, you can scroll up and down within the list by positioning with the arrow keys. Optionally, you can establish a vertical scroll bar for the selection list. You can do this by specifying the SCROLLBAR–VERTICAL option in the VIEW–AS phrase or by setting the SCROLLBAR–VERTICAL attribute to TRUE.

Similarly, the selection list may not be as wide as some values in the list. You can establish a horizontal scroll bar by specifying the SCROLLBAR–HORIZONTAL option in the VIEW–AS phrase or setting the SCROLLBAR–HORIZONTAL attribute to TRUE.

By default, no scroll bars are displayed with a selection list. After the selection list is realized, you cannot add or remove scroll bars.

17.8.3 Character Mode

If you use a selection list in a character interface, you need to be aware of several considerations:

- You can select or deselect an item in a selection list by pressing the **SPACE BAR**. This causes the VALUE–CHANGED event.
- In a single selection list, the **ENTER** key performs the default action. In a multiple selection list, the **ENTER** key first toggles the selection of the current item (causing a VALUE–CHANGED event) and then performs the default action. Therefore, to perform the default action on a multiple selection list, select all but one value with the **SPACE BAR**, position to the remaining value, and then press **ENTER**.
- If drag is enabled for a selection list, then each time you move to a new item within the list, the new item is selected. This means that the VALUE–CHANGED event might occur many times as you navigate through the list. If drag is disabled, then an item is selected only when you press the **SPACE BAR** or **ENTER–KEY**.

  For a single selection list, drag is enabled by default in the character interface. You can optionally specify NO–DRAG or change the DRAG–ENABLED attribute value. A multiple selection list in the character interface always behaves as though drag is disabled.

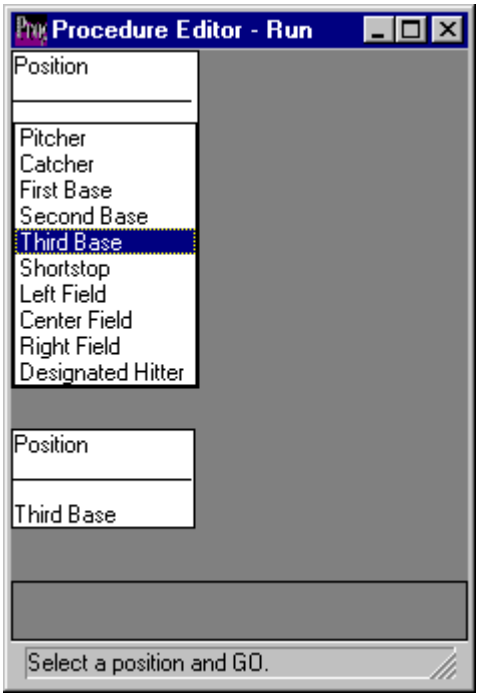
17.8.4 Example Procedure

The following example defines a variable, position, to be viewed as a selection list:

p-sel1.p

```
DEFINE VARIABLE position AS CHARACTER LABEL "Position" INITIAL "Pitcher"
   VIEW-AS SELECTION-LIST INNER-CHARS 18 INNER-LINES 10
           LIST-ITEMS "Pitcher", "Catcher", "First Base", "Second Base",
                      "Third Base", "Shortstop", "Left Field",
                      "Center Field", "Right Field", "Designated Hitter".

FORM
   position
   WITH FRAME sel-frame.

FORM
   position FORMAT "x(18)" VIEW-AS TEXT
   WITH FRAME text-frame.

ON GO OF FRAME sel-frame
   DO:
      ASSIGN position.
      DISPLAY position WITH FRAME text-frame.
   END.

DISPLAY position WITH FRAME sel-frame.
ENABLE position WITH FRAME sel-frame.
STATUS INPUT "Select a position and GO.".

WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

The INNER–CHARS option specifies that items in the selection list are 18 characters wide. The INNER–LINES option specifies that 10 of the values are visible at a time. In this example, position has 10 possible values, so all values are always visible. When you press **GO**, the second frame displays position as a text field, as in the following example:



17.8.5 Attributes and Methods

At run time, a procedure can set or query selection list attributes to accomplish the following:

- Change the delimiter Progress uses in a multiple selection list to separate the values stored in the SCREEN–VALUE attribute. Do this with the DELIMITER attribute.
- Display all list items in sorted order by setting the SORT attribute.
- Change the size of the list with the INNER–CHARS or INNER–LINES attributes.
- Change the contents of the list with the LIST–ITEMS attribute.
- Change the selection policy of the list with the MULTIPLE attribute, but only before the selection list is realized.
- Change the DRAG–ENABLED attribute to allow or disallow the user selecting items by dragging over them with the mouse. You can change this attribute only before realization.
- Query the total number of items in the list with the NUM–ITEMS attribute.

In addition to attributes, the selection list also supports methods that allow a procedure to:

- Add an item to the beginning or end of the list with the ADD–FIRST( ) or ADD–LAST( ) methods.
- Delete an item (by value or index) with the DELETE( ) method.
- Insert an item or delimiter-separated item list (by value or index) with the INSERT( ) method.
- Replace an existing item (by value or index) with a new item (or delimiter-separated item list) with the REPLACE( ) method.
- Retrieve the item specified by an index with the ENTRY( ) method.
- Return the index of a specified item with the LOOKUP( ) method.
- Query whether an item is currently selected with the IS–SELECTED( ) method.
- Scroll an item to the top of the list with the SCROLL–TO–ITEM( ) method.

For example, the following procedure reads filenames into a selection list and uses the LIST–ITEMS attribute to change the contents of the list dynamically. To compare it with a similar procedure using a combo box, see the `p-combo2.p` procedure in the "Combo Boxes" section:

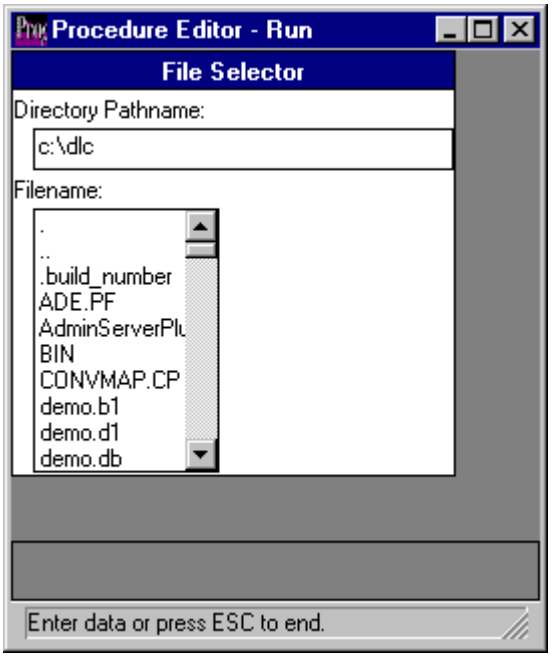
p-sel2.p *(1 of 2)*

```
DEFINE STREAM dirlist.
DEFINE VARIABLE ok-status AS LOGICAL.
DEFINE VARIABLE f-name AS CHARACTER FORMAT "x(14)".
DEFINE VARIABLE list-contents AS CHARACTER FORMAT "x(200)".
DEFINE VARIABLE dir AS CHARACTER FORMAT "x(40)".
DEFINE VARIABLE sl AS CHARACTER VIEW-AS SELECTION-LIST
          INNER-CHARS 15 INNER-LINES 10 SORT SCROLLBAR-VERTICAL.
FORM
   "Directory Pathname:"  SKIP
   dir AT 3 SKIP
   "Filename:" SKIP
   sl  AT 3
   WITH FRAME sel-frame NO-LABELS TITLE "File Selector".

FORM
   Readable AS LOGICAL Writable AS LOGICAL
   WITH FRAME file-status SIDE-LABELS.

ON GO, MOUSE-SELECT-DBLCLICK OF dir
   DO:
      ASSIGN dir.
      RUN build-list.
   END.

ON DEFAULT-ACTION OF sl
   DO:
       FILE-INFO:FILENAME = dir + "/" + SELF:SCREEN-VALUE.
       IF INDEX(FILE-INFO:FILE-TYPE, "D") > 0
       THEN DO:
          HIDE FRAME file-status.
          dir = FILE-INFO:PATHNAME.
          RUN build-list.
          DISPLAY dir WITH FRAME sel-frame.
       END.
```

p-sel2.p *(2 of 2)*

```
       ELSE DO:
          ASSIGN Readable = (INDEX(FILE-INFO:FILE-TYPE, "R") > 0)
                 Writable = (INDEX(FILE-INFO:FILE-TYPE, "W") > 0)

          FRAME file-status:TITLE = "Attributes of " + SELF:SCREEN-VALUE.
          DISPLAY Readable Writable WITH FRAME file-status.
       END.
   END.
dir = OS-GETENV("DLC").
DISPLAY dir WITH FRAME sel-frame.
RUN build-list.

ENABLE dir sl WITH FRAME sel-frame.

WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.

PROCEDURE build-list:
   ok-status = SESSION:SET-WAIT-STATE("General").

   INPUT STREAM dirlist FROM OS-DIR (dir).

   IMPORT STREAM dirlist f-name.
   list-contents = f-name.
   REPEAT:
      IMPORT STREAM dirlist f-name.
      list-contents = list-contents + "," + f-name.
   END.
   INPUT CLOSE.

   sl:LIST-ITEMS IN FRAME sel-frame = list-contents.
   ok-status = SESSION:SET-WAIT-STATE("").
END PROCEDURE.
```

This code allows you to select from a selection list of all the files and subdirectories within the specified directory. The DLC directory is the initial directory. The following screen appears:



When you double-click on an item in the list (or press **RETURN** in character mode), the DEFAULT–ACTION trigger executes. If the item is a file, the trigger displays your access to that file. If the item is a directory, the trigger rebuilds the selection list with the contents of that directory.

Note that within the internal procedure, build–list, the list of file and directory names is built first, and then the LIST–ITEMS method is used to change the contents of the selection list. You could use the ADD–LAST( ) method within the REPEAT loop to add items to the selection list one at a time, but using LIST–ITEMS once is more efficient.

For more information on each attribute and method, see the *Progress Language Reference*.

17.9 Combo Boxes

A *combo box* consists of a fill-in field or variable, a button, and a selection list. A combo box can display any single value from the selection list in the fill-in field. This single value can also be assigned to the underlying field or variable.

Progress provides the following types of combo-box widgets:

- A SIMPLE combo-box widget with a read/write edit control and a selection list that is always visible. This widget is supported in graphical interfaces only, and only in Windows. If you specify a SIMPLE combo–box widget in a character interface, Progress treats it as a DROP–DOWN–LIST combo–box widget.
- A DROP–DOWN combo-box widget with a read/write edit control and a selection list that appears when you click the drop-down button. This option is supported in graphical interfaces only, and only in Windows. If you specify a DROP–DOWN combo–box widget in a character interface, Progress treats it as a DROP–DOWN–LIST combo–box widget.
- A DROP–DOWN–LIST combo–box widget with a read-only edit control and a selection list that appears when you click the drop-down button. This is the default.

You can use the combo–box widget with a CHARACTER, INTEGER, DECIMAL, LOGICAL, or DATE field or variable. The value representations in the drop-down list conform to the data type of the underlying field or variable. Like radio sets and selection lists, combo boxes are useful for representing fields or variables that have a limited number of possible values. One advantage of combo boxes is that they take up less screen space than radio sets and selection lists.

The end user can position to a value in the drop-down list by using the arrow keys in all interfaces, or in graphical interfaces by using the scroll bar.

To select a value in the character interface, the end user can position to the value in the drop-down list using the arrow keys and press **SPACEBAR** or **RETURN** to confirm the selection.

To select a value in the graphical interface, the end user can apply the following techniques:

- Position to the value in the drop-down list using the scroll bar and click on the value in the drop-down list using the SELECT mouse button.
- Enter text in the fill-in and allow the edit control to complete keyboard input to the combo-box, based on a potential or unique match, by searching through the items in the drop-down list.
- Position to the value in the drop-down list using the arrow keys and press **SPACEBAR** or **RETURN** to confirm the selection.

When the user selects an item, it triggers the VALUE–CHANGED event.

Thus, a combo box combines the functionality of a fill-in field, selection list, and radio set. It uses a fill-in field to display the selected item, and like a selection list displays a list of available values. Like a radio set, it supports sets of values for any Progress data type and allows selection of any one value at a time.

17.9.1 Defining a Combo Box

The following form of the VIEW–AS phrase defines a combo box:

SYNTAX

```
VIEW-AS COMBO-BOX
  [ LIST-ITEMS item-list | LIST-ITEM-PAIRS item-pair-list ]
  [ INNER-LINES lines ] [ size-phrase ] [ SORT ]
  [ TOOLTIP tooltip ]
  [ SIMPLE | DROP-DOWN | DROP-DOWN-LIST ]
  [ MAX-CHARS characters ]
  [ AUTO-COMPLETION [ UNIQUE-MATCH ] ]
```

When viewing a value as a combo box, only the fill-in field displaying the value appears in a box along with a button to indicate that it is combo box. The drop-down list appears only when you choose the fill-in or button in the box. The definition options have these effects:

- The LIST–ITEMS option specifies a comma-separated list of values for the drop-down list. The representation for each value must conform to the data type and format of the underlying field or variable. In Windows, there is no default selected value for the combo box; the fill-in is empty. The combo box fill–in remains empty until you assign an item in the list to the screen value of the widget.

- The LIST–ITEM–PAIRS option specifies a list of label-value pairs. Each pair represents the label and value of a field or variable. When the drop-down list appears, it displays each pair's label. When the user selects a label, Progress assigns the corresponding value to the field or variable.

- The INNER–LINES option specifies the number of lines (items) displayed in the drop-down list at one time.

- In Windows, the *size-phrase* option specifies the outside width of the combo box. This includes the drop-down list width. The fill-in height is always the height of one character unit for a fill-in. This option also has no effect on the drop-down list height or the format of data in the list.

- The SORT option causes items in the drop-down list (LIST–ITEMS) to be displayed in sorted order no matter what order they are assigned. This option is supported in character interfaces only, and only in Windows.

- The TOOLTIP option allows you to define a help text message for a text field or text variable. Progress automatically displays this text when the user pauses the mouse button over a text field or text variable for which a tooltip is defined. No tooltip is the default. This option is supported in Windows only.

- The SIMPLE option specifies a combo–box widget with a read/write edit control and a selection list that is always visible. This option is supported in graphical interfaces only, and only in Windows.

- The DROP–DOWN option specifies a combo–box widget with a read/write edit control and a selection list that appears when you click the drop-down button. This option is supported in graphical interfaces only, and only in Windows.

- The DROP–DOWN–LIST option specifies a combo–box widget with a read-only edit control and a selection list that appears when you click the drop-down button. This is the default.

- The MAX–CHARS option specifies the maximum number of characters the edit control can hold. Use MAX–CHARS with only SIMPLE and DROP–DOWN combo–boxes. It is ignored for DROP–DOWN–LIST combo–boxes. This option is supported in graphical interfaces only, and only in Windows.

- The AUTO–COMPLETION option specifies that the edit control automatically complete keyboard input to the combo–box, based on a potential match, by searching through the items in the drop-down list. This option is supported in graphical interfaces only, and only in Windows.

- The UNIQUE–MATCH option specifies that the edit control complete keyboard input to the combo–box, based on a unique match, by searching through the items in the drop-down list. This option is supported in graphical interfaces only, and only in Windows.

17.9.2 Working with Combo Boxes

In general, it is important to keep in mind that a combo box maintains three types of data:

- The drop-down list, which is the available set of input values in character string form
- The screen buffer value displayed in the fill-in, which is the currently selected character string from the drop-down list
- The record buffer value, which is the current value of the underlying field or variable stored according to the field or variable data type

For an end user, it is only possible to select a value for the combo box from the available drop-down list. As with any data-representation widget, the application must assign the combo box with the ASSIGN statement to move this value into the record buffer.

It is possible for the application to assign any value to the underlying field or variable consistent with its data type. However, the application can only display a value from the field or variable (move the record buffer value to the screen buffer) that is also available in the combo box drop-down list.

NOTE: If you set an initial value for the underlying field or variable (through assignment or the INITIAL option), you must explicitly display as well as enable the combo box widget in order to move the initial value to the screen buffer for display in the fill-in. Otherwise when you enable the widget, nothing is displayed.

17.9.3 Example Procedure

The procedure p-combo1.p illustrates several ways to interact with a combo box. It displays a frame containing a combo box labeled Finish and several buttons. Each button modifies the combo box and displays the current value of the underlying variable, finish–var, in the message area.

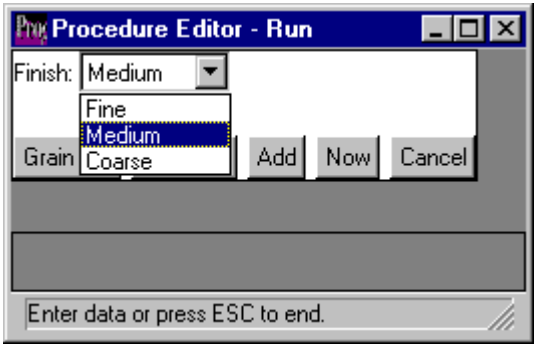
p-combo1.p *(1 of 2)*

```
DEFINE VARIABLE finish-var AS CHARACTER INITIAL "Medium"
  FORMAT "x(10)" LABEL "Finish" VIEW-AS COMBO-BOX INNER-LINES 3
  LIST-ITEMS "Fine", "Medium", "Coarse".

DEFINE BUTTON bGrain LABEL "Grain List".
DEFINE BUTTON bColor LABEL "Color List".
DEFINE BUTTON bCancel LABEL "Cancel".
DEFINE BUTTON bAdd LABEL "Add".
DEFINE BUTTON bNow LABEL "Now".
DEFINE VARIABLE stat AS LOGICAL.
DEFINE VARIABLE cnt AS INTEGER.
DEFINE VARIABLE type-var AS INTEGER INITIAL 2.
DEFINE VARIABLE type-char AS CHARACTER FORMAT "x(10)".

DEFINE FRAME a finish-var SKIP(1) bGrain bColor bAdd bNow bCancel
  WITH SIDE-LABELS.

ON CHOOSE OF bNow IN FRAME a
DO:
  MESSAGE "finish-var value is" finish-var.
END.

ON CHOOSE OF bAdd IN FRAME a
DO:
  REPEAT cnt = 1 to 3:
    type-char = ENTRY(cnt, finish-var:LIST-ITEMS) + STRING(type-var).
    stat = finish-var:ADD-LAST(type-char).
  END.
  type-var = type-var + 1.
END.

ON CHOOSE OF bGrain IN FRAME a
DO:
  finish-var:LIST-ITEMS = "Fine,Medium,Coarse".
  type-var = 2.
  APPLY "VALUE-CHANGED" TO finish-var.
END.
```

p-combo1.p *(2 of 2)*

```
ON CHOOSE OF bColor IN FRAME a
DO:
  finish-var:LIST-ITEMS = "Light,Neutral,Dark".
  type-var = 2.
  APPLY "VALUE-CHANGED" TO finish-var.
END.

ON VALUE-CHANGED OF finish-var IN FRAME a
DO:
  ASSIGN finish-var.
  MESSAGE "Assigning" finish-var.
END.

ENABLE ALL WITH FRAME a.
DISPLAY finish-var WITH FRAME a.

WAIT-FOR CHOOSE OF bCancel IN FRAME a.
```

When you choose the combo box at startup, it displays the drop-down list with the initial value selected, as in the following screen:



Note that the procedure explicitly displays the combo box to show the initial assigned value. Each time you change the value (either directly with the drop-down list or indirectly with a button), the VALUE–CHANGED event is triggered to force assignment of the new value.

The procedure also illustrates the combo box LIST–ITEMS attribute and ADD–LAST( ) method. Progress provides additional attributes and methods to increase the usability of combo boxes.

17.9.4 Attributes and Methods

At run time, a procedure can set or query combo box attributes to accomplish the following:

- Change the size of the list by setting the INNER–LINES attribute (Windows and character interfaces only). The value must be at least 3.
- Display all list items in sorted order by setting the SORT attribute (Windows and character interfaces only).
- Change the contents of the list with the LIST–ITEMS attribute, and change the delimiter that Progress uses to separate items in the list using the DELIMITER attribute.
- Change the format of combo box fill-in and list data with the FORMAT attribute. If you change the format after realization, then all items in the list are converted to the new format.
- Query or set the item currently selected in the list with the SCREEN–VALUE attribute. If nothing is selected, this attribute returns the null string (" "). If you set the attribute to an item not in the list, Progress ignores the setting and displays a warning message.
- Query the total number of items in the list with the NUM–ITEMS attribute.
- Query the data type of the combo box with the DATA–TYPE attribute. You can set this attribute before realization, but you lose all items currently in the list.

In addition to attributes, the combo box also supports list manipulation methods that allow a procedure to:

- Add an item to the beginning or end of the list with the ADD–FIRST( ) or ADD–LAST( ) methods.
- Delete an item (by value or index) with the DELETE( ) method.
- Insert an item or delimiter-separated item list (by value or index) with the INSERT( ) method.
- Replace an existing item (by value or index) with a new item (or delimiter-separated item list) with the REPLACE( ) method.
- Retrieve the item specified by an index with the ENTRY( ) method.
- Return the index of a specified item with the LOOKUP( ) method.

For example, the following procedure reads filenames into a combo box and uses the LIST–ITEMS attribute to change the contents of the list dynamically. To compare it with a similar procedure using a selection list, see the `p-sel2.p` procedure in the "Selection Lists" section.

p-combo2.p *(1 of 2)*

```
DEFINE STREAM dirlist.
DEFINE VARIABLE ok-status AS LOGICAL.
DEFINE VARIABLE f-name AS CHARACTER FORMAT "x(14)".
DEFINE VARIABLE list-contents AS CHARACTER FORMAT "x(200)".
DEFINE VARIABLE dir AS CHARACTER FORMAT "x(40)".
DEFINE VARIABLE flcombo AS CHARACTER FORMAT "x(15)"
                VIEW-AS COMBO-BOX INNER-LINES 10 SORT.
FORM
   "Directory Pathname:"  SKIP
   dir AT 3 SKIP
   "Filename:" SKIP
   flcombo  AT 3
   WITH FRAME sel-frame NO-LABELS TITLE "File Selector".

FORM
   Readable AS LOGICAL Writable AS LOGICAL
   WITH FRAME file-status SIDE-LABELS.

ON GO, MOUSE-SELECT-DBLCLICK, RETURN OF dir
   DO:
      ASSIGN dir.
      RUN build-list.
   END.

ON VALUE-CHANGED OF flcombo
   DO:
       FILE-INFO:FILENAME = dir + "/" + SELF:SCREEN-VALUE.
       IF INDEX(FILE-INFO:FILE-TYPE, "D") > 0
       THEN DO:
          HIDE FRAME file-status.
          dir = FILE-INFO:PATHNAME.
          RUN build-list.
          DISPLAY dir WITH FRAME sel-frame.
       END.
       ELSE DO:
          ASSIGN Readable = (INDEX(FILE-INFO:FILE-TYPE, "R") > 0)
                 Writable = (INDEX(FILE-INFO:FILE-TYPE, "W") > 0)

          FRAME file-status:TITLE = "Attributes of " + SELF:SCREEN-VALUE.
          DISPLAY Readable Writable WITH FRAME file-status.
       END.
   END.
```

p-combo2.p *(2 of 2)*

```
dir = OS-GETENV("DLC").
DISPLAY dir WITH FRAME sel-frame.

ENABLE dir flcombo WITH FRAME sel-frame.

WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.

PROCEDURE build-list:
   ok-status = SESSION:SET-WAIT-STATE("General").

   INPUT STREAM dirlist FROM OS-DIR (dir).

   IMPORT STREAM dirlist f-name.
   list-contents = f-name.
   REPEAT:
      IMPORT STREAM dirlist f-name.
      list-contents = list-contents + flcombo:DELIMITER IN FRAME sel-frame
                      + f-name.
   END.
   INPUT CLOSE.

   flcombo:LIST-ITEMS IN FRAME sel-frame = list-contents.
   ok-status = SESSION:SET-WAIT-STATE("").
END PROCEDURE.
```

This code allows you to select from a combo box of all the files and subdirectories within the specified directory. The DLC directory is the initial directory.



When you double click, press **RETURN**, or press GO on the Directory Pathname field, the procedure builds a list of the files in the combo box from the specified directory. When you select a new item in the list, the VALUE–CHANGED trigger executes. If the item is a file, the trigger displays your access to that file. If the item is a directory, the trigger rebuilds the combo box with the contents of that directory.

Note that within the internal procedure, build–list, the list of file and directory names is built first, and then the LIST–ITEMS attribute is used to change the contents of the combo box. You could use the ADD–LAST( ) method within the REPEAT loop to add items to the combo box one at a time, but using LIST–ITEMS once is more efficient. Note, also, the use of the DELIMITER attribute to ensure that the list is built using the current list delimiter.

For more information on each attribute and method, see the *Progress Language Reference*.

17.10 Applying Formats When Displaying Widgets

When you view a value as text, a fill-in, a toggle box, combo box, or a radio set, you can specify a format. Progress supports character, numeric, logical, and date formats for these widgets:

- Character, numeric, and date formats determine how many spaces and special characters (such as dollar signs and decimal points) Progress uses when displaying the text that appears in text widgets, fill-ins and radio sets. For the fill-in, the format also establishes a constraint on the data that the user can enter.

- A logical format determines the strings that Progress uses when it displays the logical values TRUE and FALSE. Logical formats are used with text, fill-ins, and toggle boxes. For fill-ins, the format also establishes a constraint on the data the user can enter.

You can specify more than one format for a widget as long as you display the widget in a new frame each time you introduce a new format.

This section explains how to specify a display format within a Progress procedure. See the *Progress Database Design Guide* for information on specifying a format in the database.

17.10.1 Using the FORMAT Option

You can use the FORMAT option in two ways:

- With a screen I/O statement, such as an UPDATE or DISPLAY to specify a format for a database field

- With either a screen I/O statement or the DEFINE VARIABLE statement to specify a format for a variable

The following example introduces some of the formatting syntax explained in the remaining sections. It also demonstrates the flexibility you have in defining multiple formats for a single widget:

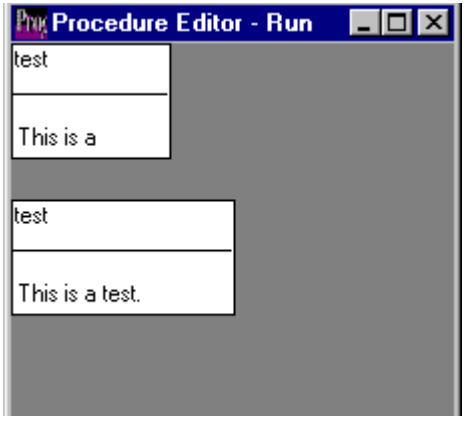
p-ftchp2.p

```
DEFINE VARIABLE test AS CHARACTER FORMAT "x(10)" VIEW-AS FILL-IN.

REPEAT:
  SET test FORMAT "x(20)".
  DISPLAY test WITH FRAME aaa.
  DISPLAY test FORMAT "X(20)" WITH FRAME bbb.
END.
```

The code first defines a variable named test, then displays it three times using two different character formats:

- The DEFINE VARIABLE statement specifies a format of x(10). This means that subsequent screen I/O statements that do not specify a format will allow I/O of up to ten characters.
- The SET statement specifies a format of x(20), allowing the user to supply up to 20 characters.
- The first DISPLAY statement does not specify a format, so Progress uses the FORMAT specified in the DEFINE VARIABLE statement. This means that if the previous SET statement inputs more than 10 characters, this DISPLAY statement displays only the first 10.
- The second DISPLAY statement specifies a format of x(20), allowing the display of all the characters input by the SET statement.

For example, if the user supplies 15 characters when the SET statement executes, the first DISPLAY statement makes it appear as if only 10 of those are stored, but the second DISPLAY statement shows that Progress has stored all 15 characters.



From the preceding example, we can see that a format does not affect a value already in a variable or database field; a format can only constrain the type of data the user enters into a field. Once in the field, the data can be displayed in any format, regardless of the original format.

If neither the Progress procedure nor the database specifies a format, Progress uses a default format based on the data type of the variable or field. Table 17–1 shows these default formats.

Table 17–1: Default Display Formats

Data Type	Data Format	Example	
		Stored Value	Displayed Value
CHARACTER	x(8)	Now is the time	Now is t
DATE	99/99/99	3/10/199	03/10/93
DECIMAL	–>>,>>9.99	12345	12,345.00
INTEGER	–>,>>>,>>9	–12345	–12,345
LOGICAL	yes/no	on	yes

The following sections explain the syntax to use when specifying a format for each of these data types.

17.10.2 Character Formats

Table 17–2 shows some examples of how Progress displays a character value using different formats. Following the table is a description of the syntax.

Table 17–2: Character Display Format Examples *(1 of 2)*

Format	Value in Field	Display
xxxxxxx	These are characters	These a
x(9)	These are characters	These are
x(20)	These are characters	These are characters
xxx	These are characters	The
AAA–9999	abc1234	abc–1234

Table 17–2: Character Display Format Examples *(2 of 2)*

Format	Value in Field	Display
! ! ! –9999	abc1234	ABC–1234
(999) 999–9999	6176635000	(617) 663–5000

The symbols used to construct character formats have the following definitions. In the definitions, the word *digit* means 0 through 9. The word *letter* means a–z, A–Z, and foreign alphabetic characters.

X

Represents any character.

N

Represents a *digit* or a *letter*. A blank (space) is not allowed.

A

Represents a *letter*. A blank is not allowed.

!

Represents a *letter* that is converted to uppercase during input. A blank is not allowed.

9

Represents a *digit*. A blank is not allowed.

(*n*)

A number that indicates how many times to repeat the previous format character. For example, !(5) is the same as !!!!! and represents five characters that are to be converted to uppercase when entered.

fillchar

You can use any character or characters you want to "fill" a display. For example, if you display the value abc with a format of x(3)***, the displayed value is abc***.

NOTE: To use X, N, A, !, or 9 as a fill character, you must precede that character with a tilde (~). To use a left parenthesis (( ) as a fill character after a nonfill character,

you must precede it with a tilde. If you use these five characters as fill characters in a format specification in a procedure, then enter two tildes (~~) so that the Progress Procedure Editor interprets the tilde literally and not as an escape lead-in for the following character (for example, you specify a format of x999, where the x is a fill character, as FORMAT "~~x999").

The database does not store fill characters; Progress supplies them when it formats the field for display. This allows you to display a field with different fill characters in different contexts.

Progress truncates the trailing blanks (spaces) in character fields. If a character field contains only one blank, the value is truncated to the null string. You can use the LEFT–TRIM, RIGHT–TRIM, or TRIM functions to truncate leading blanks, trailing blanks, or both.

17.10.3 Integer and Decimal Formats

The following is the syntax you can use when specifying the format of an integer or decimal value.

SYNTAX

```
[ ( ] [ string1 ]
  [ + | - | ( ]
  [ > | , ] ...
  [ 9 | Z | * | , ] ...
  [ . ]
  [ 9 | < | , ] ...
  [ + | - | ) | DR | CR | DB ]
  [ string2 ]
```

( )

Parentheses are displayed if the number is negative. If you use one parenthesis (left or right), you must use the other.

string1

A string made up of any characters except plus (+), minus (–), greater than (>), less than (<), comma (,), digits (0–9), letter z (z or Z), asterisk (*) or period (.).

`+`

Progress replaces this character with a plus sign (+) if the number is positive and a minus (–) sign if the number is negative. You can use only one plus or minus sign or CR, DR, or DB or one set of parentheses in a numeric data format.

`–`

When you use this character to the left of the decimal point, Progress replaces it with a minus sign if the number is negative and a blank or null if the number is positive. When you use this character to the right of the decimal point, Progress replaces this character with a minus sign if the number is negative and a blank if the number is positive.

`>`

This character is replaced with a digit if that digit is not a leading zero. If the digit is a leading zero, this character is replaced with a null and any characters to the left are moved one space to the right if you are using top labels. They are left justified if you are using side labels. See also <.

`,`

This character is displayed as a comma unless it is preceded by >, Z, or *. If the comma is preceded by >, and the > is replaced by a leading zero, the comma is replaced with a null. If the comma is preceded by Z, and the Z is replaced by a blank, the comma is replaced with a blank. If the comma is preceded by *, and the * replaces a leading zero, the comma is replaced by *.

`9`

Progress replaces this character with a digit, including cases where the digit is a leading zero.

`Z or z`

This character is replaced with a digit. If the digit is a leading zero, Z suppresses that digit, putting a blank in its place.

`*`

This character is replaced with a digit. If the digit being replaced is a leading zero, that zero is replaced with an asterisk.

`.`

This character represents a decimal point and is displayed as a period.

<

Used in conjunction with > to implement “floating decimal” format. The < symbol (up to 10) must appear to the right of the decimal and be balanced by an equal or greater number of > symbols left of the decimal. A digit is displayed in a position formatted with < when the corresponding > is a leading zero (and the stored value has the required precision). See Table 17–3.

DR, CR, DB

These characters are displayed if the number is negative. If the number is positive, Progress displays blanks in place of these characters. Progress does not treat these characters as sign indicators when you specify *string2*; Progress considers them part of *string2*.

string2

A string made up of any characters except plus (+), minus/hyphen (–), greater than (>), comma (,), any digit (0–9), letter z (z or Z), or asterisk (*).

When specifying a numeric data format, you must use at least one of the following characters: 9, z, *, or >.

Table 17–3 shows some examples of how Progress displays numeric values using different formats.

Table 17–3: Numeric Display Format Examples *(1 of 2)*

Format	Value	Display
9999	123	0123
9,999	1234	1,234
$zzz9	123	$ 123
$ > > >9	123	$123 [1]
$–>, > > 9.99	1234	$1,234.00
$–>, > > 9.99	1234	$1,234.00
#–zzz9.999	–12.34	#– 12.340
Tot= > > >9Units	12	Tot=12Units

Table 17–3: Numeric Display Format Examples *(2 of 2)*

Format	Value	Display
$>, >>9.99	–12.34	????????? [2]
$>,>>9.99	1234567	??????? [3]
>>,>99.99<<< [4]	12,345.6789	12,345.68
>>,>99.99<<<	1,234.5678	1,234.568
>>,>99.99<<<	123.45	123.45
>>,>99.99<<<	12.45678	12.45678
HH:MM:SS[5]	123456	12:34:56
HH:MM AM[5]	123456	12:34 AM

1 This display value is right justified if it has a column label, left justified is it has a side label.

2 In this example, there is a negative sign in the value –12.34, but the display format of $>,>>9.99 does not accommodate that sign. The result is a string of question marks.

3 In this example, the value 1234567 is too large to fit in the display format of $>,>>9.99. The result is a string of question marks.

4 This is a floating-decimal display format. The < symbols must follow the decimal point and be balanced by an equal or greater number of > symbols.

5 The time format is used with the STRING function. For more information, see the STRING function in the *Progress Language Reference* manual.

NOTE: If you use the European Numeric Format (–E) startup parameter, Progress interprets commas as decimal points and decimal points as commas when displaying or prompting for numeric values. However, you must enter data in procedures and the Data Dictionary as described above.

17.10.4 Logical Formats

A logical format specifies the strings you want to use to represent the values true and false. To specify a logical format, specify the true string, followed by a forward slash (/), followed by the false string. For example, to specify the strings on and off for TRUE and FALSE, respectively, use the following format option.

```
FORMAT "on/off"
```

If input is coming from a file and you have defined a format for a logical field or variable that is something other than TRUE/FALSE or yes/no, you can still use TRUE/FALSE or yes/no as input to that logical field or variable.

Table 17–4 shows some examples of how Progress displays a logical value with different formats.

Table 17–4: Logical Display Format Examples

Format	TRUE	FALSE
yes/no	yes	no
Yes/no	Yes	no
true/false	true	false
shipped/waiting	shipped	waiting

NOTE: You cannot use “no”, “n”, “false”, or “f” to represent true, and you cannot use “yes”, “y”, “true”, or “t” to represent false.

NOTE: If you use the MESSAGE statement to display a logical value, Progress always uses the default format “yes/no” and disregards any format you may have specified.

17.10.5 Date Formats

The default format for a date field or variable is“99/99/99”. Here is the syntax for specifying a date format.

```
9_/9_/9_ | 9_-9_-9_ | 9_.9_.9_ | 999999 | 99999999
```

Progress determines where to put the month, day, and year values based on any Date Format (–d) startup parameter you may have used. That startup parameter lets you specify a date format for your application. The default format is mm/dd/yy. Table 17–5 shows some examples of how Progress displays a date value using different formats. When you want the user to enter a date in an application, it is best to use the default date display format.

Table 17–5: Date Display Format Examples

Format	Value	Display
99/99/99	3/10/1990	03/10/90
99/99/9999	3/10/2090	03/10/2090
99–99–99	3/10/1990	03–10–90
99–99–99	3/10/2090	???????? (1)
999999	3/10/1990	031090
999999	03/10/90	031090
99999999	03/10/1990	03101990

1 In this example, the value of 3/10/2090 is too large to fit into the display format. The year part of the display format is “99", while the value being displayed is 2090, which requires a year format of “9999".

18

Buttons, Images, and Rectangles

In addition to representations of fields and variables, frames can also contain buttons, images, and rectangles. Buttons can be enabled for input; rectangles and images are display-only. Because these widgets occur in the widget hierarchy at the same level as field representations, Progress refers to them as field-level widgets.

18.1 Buttons

A button is a field-level widget that can be chosen by the user. Typically, you define a trigger to execute when the button is chosen. You can specify a text label for the button. You can also specify an image to be displayed in the button. Optionally, you can specify a second image to be displayed when the button is pressed down, and a third image to be displayed when the button is insensitive.

Also, in a graphical interface such as Windows, you can optionally specify ToolTips, a brief text message string that automatically displays when the mouse pointer pauses over a button widget for which a ToolTip value is defined. Although ToolTips can be defined for a variety of field-level widgets, including images and rectangles discussed later in this chapter, they are most commonly defined for button widgets. For more information on ToolTips and other Windows interface design options, see Chapter 25, "Interface Design."

In a character interface, a button appears as the label enclosed in angle brackets (< >). Any image you specify for the button is ignored in a character interface.

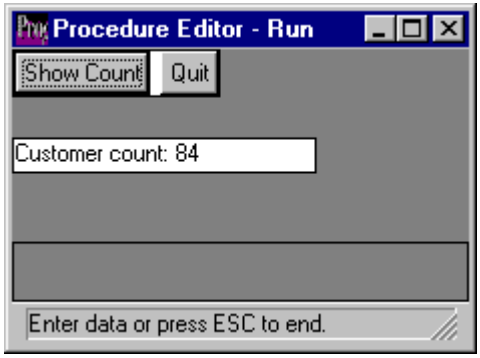
The user chooses a button by clicking on it with the mouse SELECT button or by positioning to it with the **TAB** or **ARROW–KEYS** and pressing either **SPACEBAR** or **ENTER**.

The following procedure defines two buttons:

p-but1.p

```
DEFINE VARIABLE cnt AS INTEGER.
DEFINE BUTTON count-button LABEL "Show Count".
DEFINE BUTTON quit-button LABEL "Quit".

FORM
   count-button quit-button
   WITH FRAME button-frame.
FORM
   cnt VIEW-AS TEXT LABEL "Customer count"
   WITH FRAME cnt-frame SIDE-LABELS.

ON CHOOSE OF count-button DO:
     HIDE FRAME cnt-frame.
     cnt = 0.
     FOR EACH customer:
         cnt = cnt + 1.
     END.
     DISPLAY cnt WITH FRAME cnt-frame.
END.

ENABLE count-button quit-button WITH FRAME button-frame.
WAIT-FOR CHOOSE OF quit-button
      OR WINDOW-CLOSE OF CURRENT-WINDOW.
```

The example defines two buttons, puts them in a frame, and defines a trigger to execute when the first button is chosen. The second button is referenced in the WAIT–FOR statement. When you run this code and choose count–button, the trigger counts the number of customers.



When you choose QUIT, the WAIT–FOR statement completes terminating the procedure.

At run time, you can use attributes such as AUTO–ENDKEY, AUTO–GO, and LABEL to query or change the button's characteristics.

18.1.1 Button Images

In a graphical interface, you can specify one, two, or three images to be displayed in a button. The primary image for a button is called the *up image*. It is displayed when the button is in the up or unpressed state. You can also specify a *down image* to be displayed when the button is in the down or pressed state. Progress displays the down image only momentarily when you press the button. Typically, the down image is similar to the up image but with a different color or shading. If you specify an up image but not a down image, Progress continues to display the up image when you press the button. Finally, you can specify an *insensitive* image to be displayed in place of the up image when the button is disabled.

NOTE: In a graphical interface the label is ignored and the image is used instead. In a character interface, the image is ignored and the label is used. Specifying both a label and an image makes your code portable between graphical and character interfaces.

For a button, you can use any image that is supported for the image widget. See the section "Images" later in this chapter.

NO–FOCUS Option

In addition to the types of images that a button can display, you can specify an attribute to indicate that a button with an image will not accept focus. The purpose of providing the NO–FOCUS attribute is to simulate in Progress the standard Windows toolbar button behavior; it is supported on Windows only. Therefore, a button for which the NO–FOCUS attribute is defined will not take focus when the mouse is clicked on it, and it will not accept keyboard input. Also, Progress will not generate ENTRY or LEAVE events for the button.

The status of the NO–FOCUS option of a button is a factor in determining the thickness of a button's border when no button size is defined. See the section "Specifying Button Size" later in this chapter.

For more information on the NO–FOCUS attribute, see the DEFINE BUTTON Statement reference entry and the NO–FOCUS Attribute reference entry in the *Progress Language Reference*.

Pre-defined Built-in Button Images

Frequently, you want to use an arrow as an image on a button. Therefore, Progress provides the following predefined built-in button images:

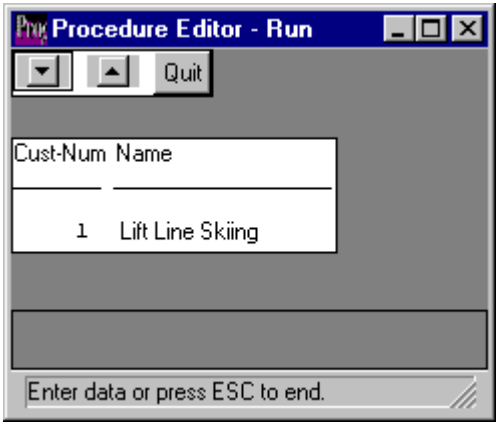
- BTN–UP–ARROW
- BTN–DOWN–ARROW
- BTN–LEFT–ARROW
- BTN–RIGHT–ARROW

The p-arrows.p procedure allows you to scroll forward and backward through the customer records until you choose the QUIT button. The example enables three buttons and uses built-in images for two of them:

p-arrows.p

```
DEFINE BUTTON next-but LABEL "Next" IMAGE-UP FILE "BTN-DOWN-ARROW"
                       SIZE-CHARS 6 BY 1.
DEFINE BUTTON prev-but LABEL "Prev" IMAGE-UP FILE "BTN-UP-ARROW"
                       SIZE-CHARS 6 BY 1.
DEFINE BUTTON quit-but LABEL "Quit".
DEFINE VARIABLE title-string AS CHARACTER INITIAL "Customer Browser".

FORM
    next-but prev-but quit-but
    WITH FRAME button-frame.
FORM
    customer.cust-num customer.name
    WITH FRAME name-frame.
ON CHOOSE OF next-but DO:
    FIND NEXT customer NO-ERROR.
    IF NOT AVAILABLE(customer)
    THEN DO:
        MESSAGE "This is the last customer."
                  VIEW-AS ALERT-BOX ERROR BUTTONS OK
                  TITLE title-string.
        FIND LAST customer.
    END.
    DISPLAY customer.cust-num customer.name
                WITH FRAME name-frame.
END.
ON CHOOSE OF prev-but DO:
    FIND PREV customer NO-ERROR.
    IF NOT AVAILABLE(customer) THEN DO:
        MESSAGE "This is the first customer."
                VIEW-AS ALERT-BOX ERROR BUTTONS OK
                TITLE title-string.
        FIND FIRST customer.
    END.
    DISPLAY customer.cust-num customer.name
            WITH FRAME name-frame.
END.

ENABLE next-but prev-but quit-but WITH FRAME button-frame.
FIND FIRST customer NO-LOCK.
DISPLAY customer.cust-num customer.name
      WITH FRAME name-frame.

WAIT-FOR CHOOSE OF quit-but OR WINDOW-CLOSE OF CURRENT-WINDOW.
```

When you run this procedure in a graphical environment, arrow images appear on next–but and prev–but. In a character environment, the labels NEXT and PREV appear instead. The SIZE–CHARS option on next–but and prev–but ensure that they are the same size as quit–but.



Specifying Button Size

When you assign an image to a button, you can specify two sizes: the size of the button and the size of the image. To size the image, use the IMAGE–SIZE, IMAGE–SIZE–CHARS, or IMAGE–SIZE–PIXELS option of the Image phrase. See the section "Images" later in this chapter.

To size the button, you can specify the outside dimensions of the button widget using the size–phrase. If no size is specified, Progress calculates a default size for the button. This calculation adds the button's border thickness (that is, the combination of 3–D shadows and highlights, and the focus rectangle) to the up image size defined by the IMAGE | IMAGE–UP *image–phrase* option. However, the thickness of the border depends on whether the button has dual images (up and down images) and whether it is a NO–FOCUS button. For detailed information on how button image and the NO–FOCUS status of a button determine the button's border thickness, see the section on the DEFINE BUTTON statement in the *Progress Language Reference*.

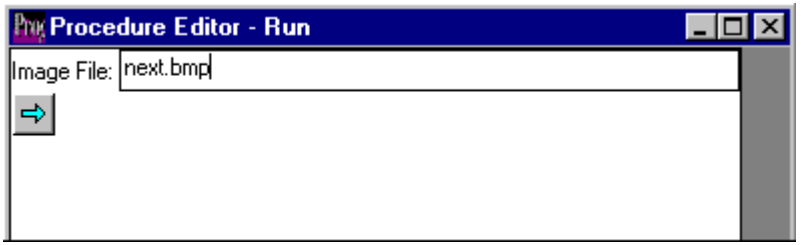
You can also change the images for a button at run time by using the LOAD–IMAGE–UP, LOAD–IMAGE–DOWN, and LOAD–IMAGE–INSENSITIVE methods.

The following code example lets you enter the name of an image file. When you enter a name and press **GO**, the contents of the image file are loaded as the up image for the button. If you do not enter the image file extension, Progress will only search for .bmp or .ico files. You must include the file extension if you wish to use a different type of image file:

p-but2.p

```
DEFINE BUTTON quit-button LABEL "Quit".
DEFINE VARIABLE image-file AS CHARACTER FORMAT "x(60)" LABEL "Image File".
DEFINE VARIABLE status-ok AS LOGICAL.

FORM
   image-file SKIP
   quit-button SKIP(13)
   WITH FRAME choices-frame SIDE-LABELS.

ON GO OF image-file OR RETURN OF image-file
   DO:
      ASSIGN image-file.
      status-ok = quit-button:LOAD-IMAGE-UP(image-file).
      IF status-ok = NO
      THEN MESSAGE "Cannot load this image." VIEW-AS ALERT-BOX
                                             MESSAGE BUTTONS OK.
      STATUS INPUT "Enter another filename or press the button to quit.".
   END.

ENABLE image-file quit-button WITH FRAME choices-frame.
STATUS INPUT "Enter filename of image for Quit button.".

WAIT-FOR CHOOSE OF quit-button OR WINDOW-CLOSE OF CURRENT-WINDOW.
```

If you run this example on Windows and load the Progress `adeicon\next.bmp` file, the following screen appears:



NOTE: The `adeicon\next.bmp` file resides in the Progress distribution in the library `gui\adeicon.pl`. For more information on unloading files from libraries, see the preface.

The size of the button changes depending on the size of the image you load. If the image is very large, the button might not fit in the frame. When this happens, Progress displays a warning message.

Preserving Original Color Specifications for Button Images

In a graphical interface such as Windows, you can specify colors for the various areas of a button. To ensure that colors associated with the button's images (that is, up, down, and insensitive) are not converted to the system 3–D colors, you can use the NO–CONVERT–3D–COLORS option. By default, Progress converts shades of grey in an image to the corresponding system 3–D color. Using this option allows you to override the default behavior to preserve your original color selections for the image on a button.

For detailed information on color conversions for 3–D colors, see the DEFINE BUTTON Statement reference entry in the *Progress Language Reference*.

18.1.2 Default Buttons in Dialog Boxes

In a graphical interface, dialog boxes typically have a default button. The default button receives **RETURN** key events whether or not it has focus, thereby allowing the user to execute the button's action without having to tab to it or move the mouse to it. However, if the widget that currently has focus normally takes a **RETURN** press event, that widget, and not the default button, will receive the RETURN event. For example, another button in the frame can still receive the RETURN event.

A default button is recognizable to the user by an extra border. For example, in many editors, a dialog box appears if the user attempts to close a file without saving it, prompting the user to either save the file or close the file and lose any changes. When programming this type of application, the Save button is typically the default button in the dialog.

When defining a button, specify the DEFAULT option to indicate that it can be a default button. To make it the default button for the frame, specify the button in the frame's DEFAULT–BUTTON option.

If you need to change a frame's default button at run time, make sure that you specify the DEFAULT option for each button that potentially may become the default button. Then, at run time, change the frame's DEFAULT–BUTTON attribute.

Note that if you define a default button in a frame that contains fill-ins, pressing **RETURN** does not move the cursor to the next field even if the SESSION:DATA–ENTRY–RETURN attribute is true.

18.2 Images

An image widget is a container in which you can display an image from a bitmap file. You create a static image widget at compile time with the DEFINE IMAGE statement.

The DEFINE IMAGE statement creates an image widget and optionally makes an association between the widget and an operating system image file. The DEFINE IMAGE statement takes an Image phrase, which has the following options:

- **FILE —** A character expression that specifies the name of the image file to associate with the image widget.
- **FROM** — Two integer values that specify position of the upper-left corner of the image within the image file. Use X and Y to specify pixel offsets and ROW and COLUMN to specify character offsets.
- **IMAGE–SIZE | IMAGE–SIZE–CHARS** — Two integer values, separated by the keyword BY, that specify the width and height of the image in character units.

- **IMAGE–SIZE–PIXELS** — Two integer values, separated by the keyword BY, that specify the width and height of the image in pixel units.
- **STRETCH–TO–FIT** — This option/attribute forces the image to expand or contract to fit within the image widget's boundaries.
- **RETAIN–SHAPE** — This option/attribute is used only when the STRETCH–TO–FIT option/attribute is TRUE. It indicates that the image should retain its aspect ratio.
- **TRANSPARENT** — This option/attribute indicates that the image's background color should become transparent.

When you use one of the image-size options in conjunction with the FILE options, Progress simply makes a compile-time association between the image file and the image widget; the image file does not have to exist at this point.

If you leave out the file extension when referencing an image file from within a procedure, Progress looks through your PROPATH for files that have the .bmp extension and then for files that have the .ico extension. You must supply the file extension if you reference an image file that does not have the the .bmp or .ico extension.

NOTE: Progress supports the display of many image file formats, including JPEG (.jpg) and Graphics Interchange Format (.gif). See the *Progress Language Reference* for a complete list of supported image file formats.

Omit the FILE option if you want to create an image widget that is not associated with an image file at compile time, but instead want to make the association at run time.

Use the FILE option without one of the image-size options if you do not know the size of the image and want Progress to determine the size at compile time. If you do this, Progress uses the entire image. Also note that the image file must exist at compile time or a compiler error will occur.

The following code fragment uses the DEFINE IMAGE statement to create an image widget. The FILE option makes an association between the image widget and the `pro` bitmap file. The FROM and the IMAGE–SIZE–PIXELS options instruct Progress to read a 64x64 pixel portion of the file beginning at pixel location 5, 5. The code fragment then displays the image.

```
DEFINE IMAGE myicon
  FILE "images\pro.gif"
  FROM X 5 Y 5
  IMAGE-SIZE-PIXELS 64 BY 64.

DISPLAY myicon WITH FRAME icon-frame.
```

To associate an image file with an image widget at run time, use the LOAD–IMAGE method. You can do this either to change an existing association or to create an association if you did not use the FILE option when defining the image widget. The LOAD–IMAGE method returns a logical value and has the following parameters:

filename

A character-string expression of a full or relative pathname of a file that contains an image.

x-offset

An integer that specifies the pixel along the x-axis at which to begin reading from the image file.

y-offset

An integer that specifies the pixel along the y-axis at which to begin reading from the image file.

width

An integer that specifies the number of pixels along the x-axis to read from the image file.

height

An integer that specifies the number of pixels along the y-axis to read from the image file.

NOTE: When you use the LOAD–IMAGE method, Progress uses pixels as the unit of measurement for the *x-offset*, *y-offset*, *width*, and *height*; you cannot specify character units.

The following code fragment creates a dynamic image widget and lets you choose an image file to load into the widget:

p-ldimg.p *(1 of 2)*

```
DEFINE VARIABLE filename AS CHARACTER.
DEFINE VARIABLE filter-string AS CHARACTER.
DEFINE VARIABLE got-file AS LOGICAL.
DEFINE VARIABLE status-ok AS LOGICAL.
DEFINE VARIABLE myimage AS WIDGET-HANDLE.

DEFINE BUTTON get-image LABEL "Get Image...".

FORM
   WITH FRAME img-frame TITLE "IMAGE" WIDTH 40.

FORM
   get-image WITH FRAME but-frame.

ON CHOOSE OF get-image
   DO:
      SYSTEM-DIALOG GET-FILE filename
               TITLE "Choose an Image to Display"
               FILTERS "Image files" filter-string
               MUST-EXIST
               UPDATE got-file.

      IF got-file
      THEN DO:
         status-ok = myimage:LOAD-IMAGE(filename).
         ASSIGN FRAME img-frame:HEIGHT-CHARS = myimage:HEIGHT-CHARS + 1
                FRAME img-frame:WIDTH-CHARS =
                              MAX(myimage:WIDTH-CHARS + 2, 15).
         VIEW FRAME img-frame.
      END.
   END.
```

p-ldimg.p *(2 of 2)*

```
CREATE IMAGE myimage
       ASSIGN FRAME = FRAME img-frame:HANDLE.

CASE SESSION:WINDOW-SYSTEM:
   WHEN "TTY"
      THEN MESSAGE "Images are not supported for this interface.".
   OTHERWISE
       ASSIGN filter-string = "*.ico, *.bmp".
END CASE.

IF filter-string = ""
THEN RETURN.

ENABLE get-image WITH FRAME but-frame.

WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

Because images are supported only for Windows, this procedure determines whether you are running in a character interface. If so, it displays a message and returns. Otherwise, you can choose the Get Image button and select an image to load.

For example, if you load the `admin.bmp` image shipped with Progress, the following screen appears:



NOTE: Each time you load a new image, the procedure adjusts the size of the frame to ensure the image widget still fits in the frame.

Converting Original Color Specifications for Images to System 3-D Colors

The CONVERT-3D-COLORS option enables colors that are associated with an image to be converted to system 3-D colors when the image is loaded. The color conversion process is based on mapping white, grey, and black colors to their original red-green-blue color values. Subsequently, these original color values are converted to the new corresponding system colors. During a session, if Windows notifies Progress that the system colors are changed, all images that have the CONVERT-3D-COLORS option are reloaded and converted to the new system colors.

For detailed information on color conversion for 3–D colors, see the DEFINE IMAGE Statement reference entry in the *Progress Language Reference*.

Transparency

If you set the TRANSPARENT attribute of an image to TRUE, portions of one image can show through another image. The background color of an image which is TRANSPARENT becomes transparent, allowing any image beneath it to show through. This feature does not apply to icon images since they can be drawn with transparent backgrounds.

The TRANSPARENT attribute overrides the CONVERT–3D–COLORS attribute. If both TRANSPARENT and CONVERT–3D–COLORS are set to TRUE, the latter will be ignored.

Specifying Tooltip Information for Images

On Windows, you can optionally specify ToolTips, a brief text message string that automatically displays when the mouse pointer pauses over a widget for which a ToolTip value is defined. Although ToolTips can be set up for a variety of field-level widgets, including buttons, images and rectangles discussed in this chapter, they are most commonly defined for button widgets. For more information on ToolTips and other Windows interface design options, see Chapter 25, “Interface Design.”

18.3 Rectangles

Rectangles serve as decoration and can be displayed in a frame or in a frame background. Rectangles are display-only, and you cannot tab to a rectangle. Rectangles can receive user input only if they are movable or resizable. See Chapter 24, "Direct Manipulation," for more information on moving and resizing widgets. Also, as previously mentioned, you can define ToolTip values for rectangles.

In a rectangle, the foreground color (FGCOLOR) is used for the edge and the background color (BGCOLOR) is used to fill the interior. In a character interface, the display color (DCOLOR) is used to fill the interior. The default values for unspecified colors are the owning frame's foreground and background colors. You can use the EDGE–PIXELS or EDGE–CHARS attributes to set the thickness of the rectangle's edge. The default edge is 1 pixel in graphical interfaces and 1 character in character interfaces. Use the NO–FILL option to create a hollow, transparent rectangle.

NOTE: The minimum size for a rectangle is character mode is 1 character high and 1 character wide. If you specify the size in pixels, Progress rounds down to a lower character. If the result is a dimension of less than 1 character, Progress does not display the rectangle.

If you specify an edge width of 0 and you do not specify a background color, you will not see the rectangle because it inherits the background color from its frame. To see the rectangle, you must either set the background color or specify an edge width that is greater than 0.

Table 18–1 shows the various effects that you can achieve by using different combinations of the FGCOLOR, BGCOLOR, and NO–FILL options in a graphical interface. The table assumes that the application's color table contains red in the first location and blue in the second.

Table 18–1: Using NO–FILL and Color with Rectangles *(1 of 2)*

Options	Resulting Rectangle
No options	A rectangle filled with the frame's background color and bordered in the frame's foreground color.
FGCOLOR 0	A rectangle with a red border filled with the frame's background color.
NO–FILL	A hollow rectangle bordered with the frame's foreground color.
FGCOLOR 0 NO–FILL	A hollow rectangle with a red border.

Table 18–1: Using NO–FILL and Color with Rectangles *(2 of 2)*

Options	Resulting Rectangle
FGCOLOR 0 BGCOLOR 1	A rectangle with a red border filled with blue.
BGCOLOR 1	A rectangle filled with blue and bordered with the frame's foreground color.

For example, the following code fragment defines a hollow rectangle with a border that is either one pixel (in graphical interfaces) or one character (in character interfaces) thick. The border is drawn in the frame's foreground color. If you want a solid fill rectangle without a border, specify 0 for the EDGE–PIXELS or EDGE–CHARS attribute.

```
DEFINE RECTANGLE poly
  SIZE-CHARS 30 BY 3
  NO-FILL

DISPLAY poly.
```

NOTE: In a character interface, a rectangle's proportions are determined by the proportions of the character set. For example, if a character set's height is two times the width, specify a SIZE option of 2 BY 1 if you want a square.

Some character interfaces support line graphics characters. If you want Progress to use these characters when drawing a rectangle, specify the GRAPHIC–EDGE option in the DEFINE RECTANGLE statement or set the GRAPHIC–EDGE attribute to TRUE. The GRAPHIC–EDGE attribute is ignored for a minimum-sized character rectangle (1 character by 1 character). In a graphical interface, Progress ignores the GRAPHIC–EDGE option and attribute setting.

At run time, an application can query or set the rectangle's fill or edge width using the FILLED and EDGE–PIXELS or EDGE–CHARS attributes.

Usually, you should put any rectangles into the frame background. If you put a rectangle in the foreground in character mode, you should display the rectangle before displaying anything else in the frame. This prevents the rectangle from overlaying other widgets.

19

Frames

This chapter describes frames, and how Progress allocates and manages frames. It also describes the major 4GL components used to control frames. For more information on frame design options, see Chapter 25, "Interface Design."

This chapter includes the following topics:

- What is a frame?
- Why Progress uses frames
- Static and dynamic frames
- Frame allocation
- Types of frames
- Frame scope
- Triggers and frames
- Frame flashing
- Frame families
- Frame services

- Using shared frames
- Field-group widgets
- Validation of user input and records

19.1 What Is a Frame?

A *frame* is a rectangular display area within a window that Progress uses to display field-level widgets and other frames. To see a frame, run this procedure:

p-frm1.p

```
FOR EACH customer:
  DISPLAY name address credit-limit.
END.
```

Figure 19–1 shows the output of this procedure.



Figure 19–1: Frame

Progress allocates frames automatically to hold field-level widgets referenced by screen display and input statements. Except for the PUT statement, all statements that display data or require user interaction place widgets in frames. Two or more statements can use the same frame.

You can also define and reference frames explicitly and contain frames within other frames. For more information on containing frames within frames, see the "Frame Families" section.

19.2 Why Progress Uses Frames

Progress uses frames to ease the task of laying out your data, so you do not have to individually position every field-level widget that you want to display. Progress automatically lays out frames according to predetermined rules, or defaults. One default is to display a label for each field in the frame. Another is to place a solid box around every frame. You must become familiar with these defaults to use frames effectively. Once you know the defaults, you can override them to change the appearance and placement of your data.

19.3 Static and Dynamic Frames

Progress supports static frames and dynamic frames. Static frames are allocated at compile time. Dynamic frames are created at run time.

Progress can produce frames in three ways:

- By allocating a default static frame
- By allocating an explicitly defined static frame
- By creating a dynamic frame

The sections in this chapter focus on static frames. For more information on dynamic frames, see Chapter 20, "Using Dynamic Widgets."

19.4 Frame Allocation

This section describes how Progress allocates static frames and how you can control their allocation.

19.4.1 How Progress Allocates Frames

Progress automatically allocates frames to a selected group of blocks. In general, the following blocks receive default frames:

- REPEAT blocks
- FOR EACH blocks
- DO WITH FRAME blocks
- Procedure blocks

However, Progress does not always allocate a frame to these blocks. For example, if a block displays no widgets, Progress doesn't allocate a frame. Progress recognizes when frames are needed and when they are not.

The Progress Compiler, in a top-to-bottom pass of a procedure, determines which frames to allocate. In Figure 19–2, there are three blocks. Block A and the procedure block both contain statements that display data, so Progress allocates a frame for each of those blocks. Block B does not display data, so Progress does not allocate a frame to it.

p-frm2.p

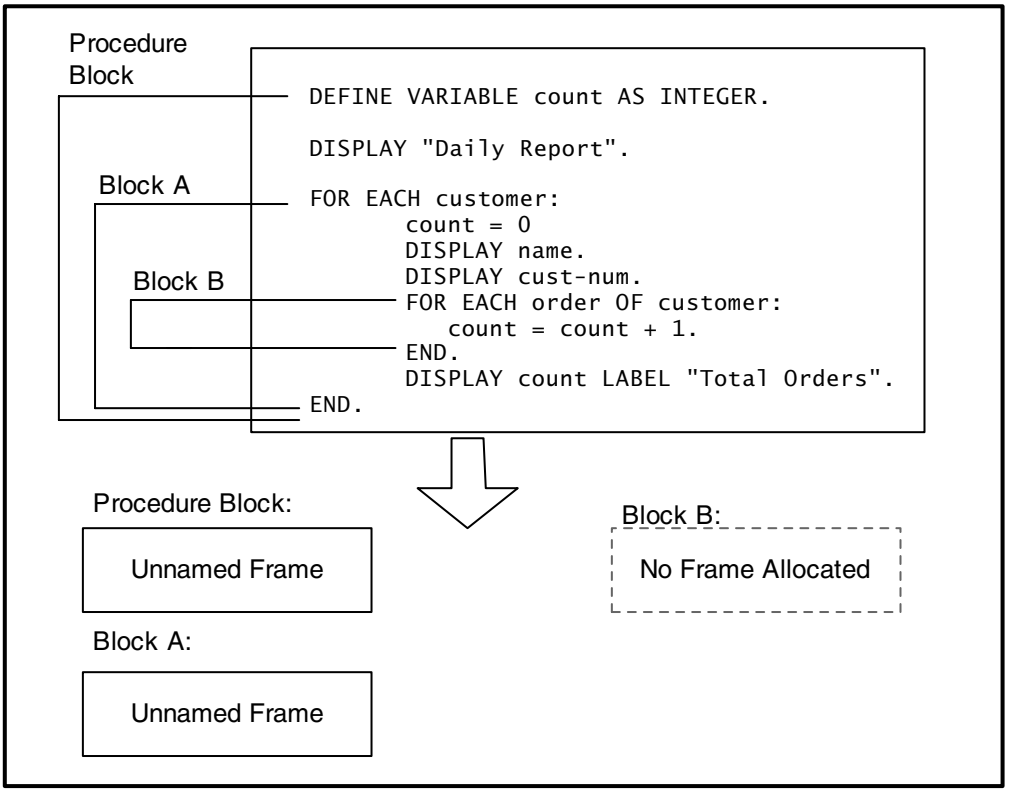


Figure 19–2: Frame Allocation

The allocated frames are all *unnamed* frames. An unnamed frame is a frame that Progress allocates by default, but that you do not explicitly name.

19.4.2 Default Frames

Progress allocates default frames when you display field-level widgets without naming a frame. For example, in Figure 19–2, this statement appears in the procedure block.

```
DISPLAY "Daily Report".
```

This statement displays a text string, but does not name a frame. Progress therefore allocates a default unnamed frame, using it to display the text string “Daily Report”. You can replace the above statement with the statement shown below.

```
DISPLAY "Daily Report" WITH FRAME a.
```

This statement explicitly names a frame (frame a), causing Progress to allocate that frame instead of a default frame. A default frame is not necessary, because no other statements in the procedure block display data (frame a displays the text string “Daily Report”). Progress allocates a default frame only for field-level widgets that are not explicitly placed in another frame.

19.4.3 Specifying Default Frames

Whenever you name a frame within a block’s header statement, the frame that you name becomes the default frame for that block.

For example, the header statement in the following FOR EACH block names a frame, so Progress allocates that frame instead of an unnamed frame. Frame a is the default frame for the block. Because the DISPLAY statement does not explicitly name another frame, it uses frame a:

p-frm3.p

```
FOR EACH customer WITH FRAME a:
  DISPLAY cust-num name.
END.
```

The header statement in the following FOR EACH block doesn’t name a frame, so Progress allocates a default unnamed frame:

p-frm4.p

```
FOR EACH customer:
  DISPLAY cust-num name.
END.
```

In these two examples, naming the frame does not affect the appearance of the output, which is the same for both procedures. However, in many cases, naming the default frame can affect the appearance of the output. Whether it does or not depends on the scope of the frame, the type of frame, where you reference the frame, and how you reference it. These issues are described in detail later in this chapter. However, the following two procedures illustrate how naming the default frame can affect the appearance of your output.

These procedures are identical, except that the first uses a FORM statement and the second uses a DEFINE FRAME statement to describe the frame. For more information on the FORM and DEFINE FRAME statements, see Chapter 25, "Interface Design."

p-form.p

```
FORM
   customer.cust-num customer.name
   WITH FRAME a 15 DOWN USE-TEXT.

FOR EACH customer WITH FRAME a:
   DISPLAY cust-num name.
END.
```

p-deffrm.p

```
DEFINE FRAME a
   customer.cust-num customer.name
   WITH 15 DOWN USE-TEXT.

FOR EACH customer WITH FRAME a:
   DISPLAY cust-num name.
END.
```

When you run the `p-form.p` procedure, all of the customer records flash on screen. For more information, see the "Frame Flashing" section. When you run the `p-deffrm.p` procedure, 15 records are displayed at a time, and there is no flashing.

Flashing occurs in the first procedure because of how the frame is *scoped*. The scope of a frame is the range or extent of the frame's availability within a procedure. For more information on frame scope, see the "Frame Scope" section.

The FORM statement scopes the frame to the procedure block; the DEFINE FRAME statement does not scope the frame. In both procedures, the frame is the default frame for the FOR EACH block. However, Progress provides different frames services, depending on whether the frame is scoped to an iterating block. If it is scoped to an iterating block, Progress advances and clears the frame (avoiding any flashing). If it is not scoped to an iterating block, Progress does not advance for each iteration.

19.4.4 Controlling Frame Allocation

By explicitly naming a frame, you can direct Progress to allocate a new frame, or you can direct Progress to use a named frame as the default frame for a block. The example procedure in Figure 19–3 performs these actions.

p-frm5.p

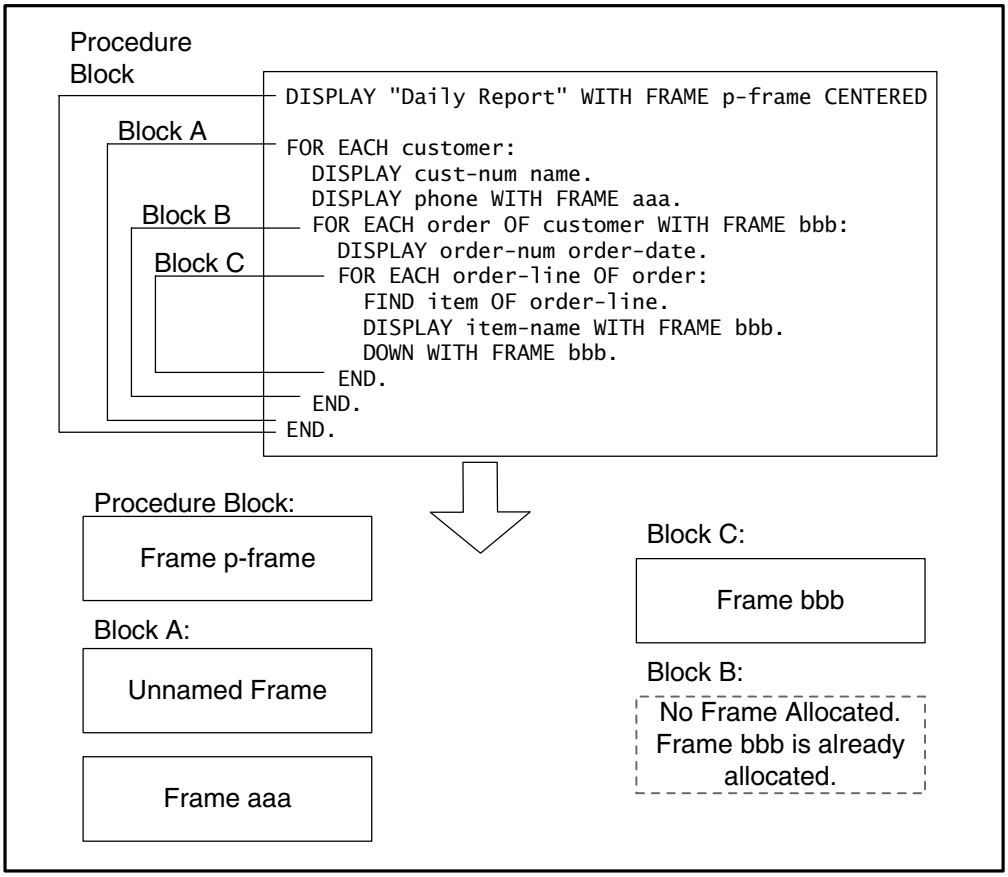


Figure 19–3: Controlling Frame Allocation

In Figure 19–3, Progress allocates four frames.

19.4.5 The Procedure Block

The DISPLAY statement in the procedure block creates one frame (p–frame).

```
DISPLAY "Daily Report" WITH FRAME p-frame CENTERED.
```

Progress uses this frame to display the text string “Daily Report”. No other statements in the procedure block display data, so Progress does not allocate a default unnamed frame.

If you add the following statement to this block, Progress allocates a default unnamed frame (because the statement does not explicitly name a frame).

```
DISPLAY "This statement uses a default frame".
```

19.4.6 Block A

Progress allocates two frames (an unnamed frame and frame aaa) to Block A. The first DISPLAY statement doesn’t explicitly name a frame, so Progress creates a default unnamed frame. The second DISPLAY statement explicitly names frame aaa.

```
DISPLAY phone WITH FRAME aaa.
```

19.4.7 Block B

The header statement in Block B names a frame (frame bbb), so Progress does not allocate a default unnamed frame. Frame bbb is the default frame for the block.

```
FOR EACH order OF customer WITH FRAME bbb.
```

19.4.8 Block C

No frame is allocated to Block C. The lone DISPLAY statement explicitly names frame bbb, so Progress uses that frame to display the item description.

```
DISPLAY item-name WITH FRAME bbb.
```

19.5 Types of Frames

In addition to controlling which frames Progress allocates, you can also control the type of frames that Progress allocates. There are two types of frame: down and one-down. A down frame displays multiple iterations of data. A one-down frame displays single iterations of data.

Static frames can be down or one-down. Dynamic frames can be one-down only.

p-frm6.p

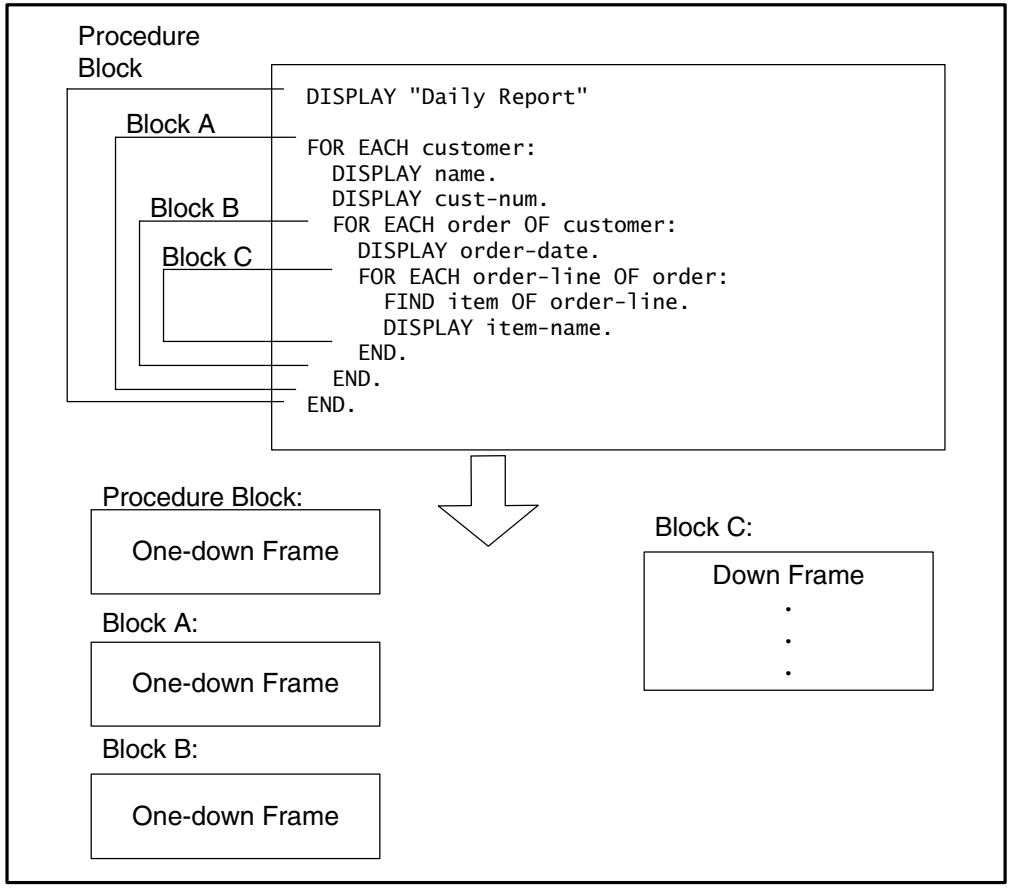


Figure 19–4: Types of Frames

Progress allocates four frames to the procedure in Figure 19–4. Three of the frames are one-down; the remaining frame is a down frame. The Progress compiler examines each block in the procedure and tests the following conditions to determine what type of frame to allocate to each block.

1. Is the block an iterating block?
2. Is the default frame for the block scoped to the block?
3. Does the block contain a nested block that has a frame scoped to it?

To determine the types of frames that the Progress Compiler allocates, see Table 19–1.

Table 19–1: Determining Frame Type

Is the Block an Iterating Block?	Is the Default Frame Scoped to the Block?	Does the Block Contain a Nested Block with a Frame Scoped to It?	Frame Type
Yes	Yes	No	Down
All other combinations			One-down

19.5.1 Example Procedures

The following examples show how Progress allocates different kinds of frames to different kinds of blocks.

In the procedure `p-frm7.p`, Progress allocates a down frame to the outer FOR EACH block. The inner FOR EACH block does not display data, so Progress does not allocate a frame to it, and therefore no frame is scoped to it:

p-frm7.p

```
DEFINE VARIABLE cnt AS INTEGER INITIAL 0.

FOR EACH customer:
  DISPLAY cust-num name.
  FOR EACH order OF customer:
    cnt = cnt + 1.
  END.
  DISPLAY cnt LABEL "Total Orders".
  cnt = 0.
END.
```

In the procedure `p-frm8.p`, Progress allocates a one-down frame to the outer FOR EACH block, a one-down frame to inner FOR EACH block, and a one-down frame to the DO WITH FRAME block. The DO WITH FRAME block has a frame scoped to it, so the inner FOR EACH block does not receive a down frame:

p-frm8.p

```
FOR EACH customer:
    DISPLAY cust-num name.
    FOR EACH order of customer:
        DISPLAY order-num.
        DO WITH FRAME a:
            FIND FIRST order-line OF order.
            DISPLAY item-num.
        END.
    END.
END.
```

In the procedure p-frm9.p, Progress allocates two one-down frames to the procedure block (an unnamed frame and frame aaa). Although frame aaa is the default frame for the REPEAT block, it is not scoped to the REPEAT block. Therefore, it is a one-down frame:

p-frm9.p

```
DISPLAY "Customer Report".

FORM customer.cust-num customer.name WITH FRAME aaa.

REPEAT WITH FRAME aaa:
  FIND NEXT customer.
  UPDATE cust-num name.
END.
```

19.6 Frame Scope

The *scope* of a static frame is the range or extent of the frame's availability within a procedure, which is equivalent to the area within a particular block. (This includes any and all nested blocks within the block.) For example, if a frame is scoped to the procedure block, it means that the frame is available throughout the entire procedure. You can display data in the frame from anywhere within the procedure. Figure 19–5 shows another example of frame scope.

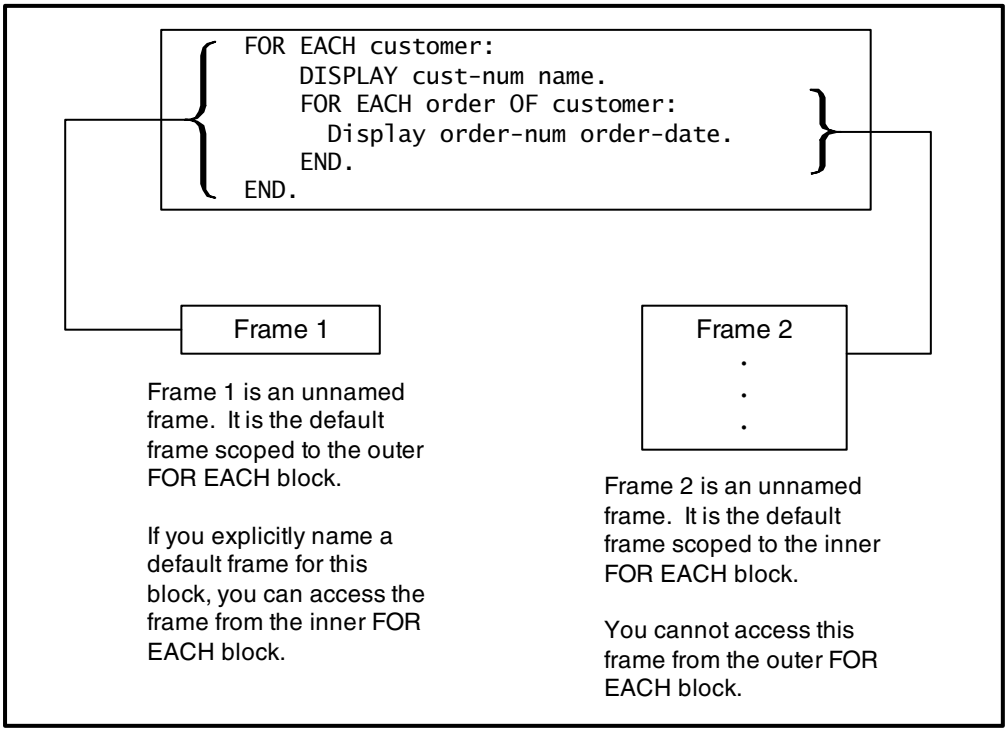


Figure 19–5: Frame Scope

The scope of a static frame is the first REPEAT, FOR EACH, procedure, or DO WITH FRAME block that references the frame. In this context, HIDE and VIEW statements do not count as frame references.

The DEFINE FRAME statement defines a frame but does not scope it. The scope of the frame is the next REPEAT, FOR EACH, procedure, or DO WITH FRAME block that references the frame.

NOTE: Dynamic frames are globally scoped. For more information, see Chapter 20, "Using Dynamic Widgets."

19.6.1 Example Procedures

The following examples show different ways that Progress scopes frames.

In the procedure `p-frm10.p`, the scope of frame aaa is the procedure block because that is the first block that references frame aaa. The scope of frame bbb is the REPEAT block. The DISPLAY statement in the REPEAT block uses frame bbb because it is the default frame for the REPEAT block. (The DISPLAY statement could override this by explicitly naming a frame: DISPLAY WITH FRAME.)

p-frm10.p

```
DISPLAY "Customer Display" WITH FRAME aaa.
REPEAT WITH FRAME bbb.
  PROMPT-FOR customer.cust-num WITH FRAME aaa.
  FIND customer USING cust-num.
  DISPLAY customer WITH 2 COLUMNS.
END.
```

In the procedure `p-frm11.p`, the scope of frame bbb is the procedure block because that is where it is first referenced. The FORM statement counts as a use of a frame. The default frame of the REPEAT block is bbb, since bbb is specified in the block's header statement. However, frame bbb is not scoped to the REPEAT block:

p-frm11.p

```
FORM WITH FRAME bbb.
DISPLAY "Customer Display" WITH FRAME aaa.
REPEAT WITH FRAME bbb:
  PROMPT-FOR customer.cust-num WITH FRAME aaa.
  FIND customer USING cust-num.
  DISPLAY customer WITH 2 COLUMNS.
END.
```

Progress can scope a frame to only one block. You cannot reference or use a frame outside its scope except in HIDE and VIEW statements.

In the procedure p-frm12.p, the scope of frame a is the REPEAT block, since that is the first block that references frame a. If you try to run this procedure, Progress displays as error saying that you cannot reference a frame outside its scope. The FOR EACH block names frame a, but the FOR EACH block is outside the scope of frame a. You could explicitly scope frame a to the procedure and use it in both the REPEAT and FOR EACH blocks. To do this, you could add a DO WITH FRAME block that encompasses both the REPEAT and FOR EACH blocks:

p-frm12.p

```
/* This procedure will not run */

REPEAT WITH FRAME a:
  PROMPT-FOR customer.cust-num.
  FIND customer USING cust-num.
  DISPLAY name.
END.
FOR EACH customer WITH FRAME a:
  DISPLAY name.
END.
```

19.7 Triggers and Frames

User-interface and session triggers can reference frames in the procedure that defines them and can use these frames to display widgets or character strings. In addition, triggers can use default frames to display data (if no frames are named within the trigger), and can name frames that do not exist within the defining procedure.

If a trigger displays data without specifying a frame, the trigger receives a default frame. Each time the trigger executes, Progress creates a new default frame. The trigger does not reuse a frame created during a previous execution.

If the trigger names a frame not named in the defining procedure, the new frame is created when the trigger executes.

Whenever a trigger creates a frame, the frame is local to the trigger and its scope ends when the trigger finishes executing. Each time the trigger executes, a new frame is created.

19.8 Frame Flashing

A phenomenon called frame flashing occurs when an iterating block displays to a frame that is scoped to an enclosing block. To see frame flashing, run this procedure:

p-frm13.p

```
/* This procedure produces flashing */

FORM customer.cust-num customer.name
     customer.credit-limit
     WITH FRAME cust-frame.

FOR EACH customer:
    DISPLAY cust-num name credit-limit
    WITH FRAME cust-frame.
END.
```

Iterating blocks repeatedly display data to the screen. If the data is displayed to a down frame that iterates and advances for each pass through the block, no frame flashing occurs. However, since cust–frame is not scoped to the iterating block, Progress does not provide the appropriate frame services. (For more information on frame services, see the "Frame Services" section.) Therefore, each iteration of the block overwrites the data from the previous iteration, which produces flashing.

You must take two steps to correct frame flashing. First, make the flashing frame a down frame with the DOWN frame phrase (for more information on frame phrases, see the Frame phrase reference entry in the *Progress Language Reference*). Second, use a DOWN statement to advance the cursor to the next iteration of the frame. When you apply these steps to the previous procedure, frame flashing no longer occurs:

p-frm14.p

```
FORM customer.cust-num customer.name
     customer.credit-limit
     WITH FRAME cust-frame DOWN.

FOR EACH customer:
    DISPLAY cust-num name credit-limit
    WITH FRAME cust-frame.
    DOWN WITH FRAME cust-frame.
END.
```

19.9 Frame Families

By default, when Progress allocates a frame or when you display and accept input from an explicitly defined frame, Progress parents that frame to either the current window or the window you specify.

NOTE: In character interfaces Progress provides only the current window. You cannot create additional windows.

For example, the following procedure parents frame cust–frame to the current window with the first DISPLAY statement. It then parents cust–frame to the window specified by hwin with the second DISPLAY statement:

p-frmwin.p

```
DEFINE VARIABLE hwin AS WIDGET-HANDLE.
CREATE WINDOW hwin
    ASSIGN TITLE = "New Window".
CURRENT-WINDOW:TITLE = "Current Window".

FIND FIRST customer.

DISPLAY name balance WITH FRAME cust-frame.
DISPLAY name balance WITH FRAME cust-frame IN WINDOW hwin.

PAUSE.
```

Note that a frame can be parented to only one window at a time. So, when the second DISPLAY statement parents cust–frame to the new window, it is removed from the current window. For more information on window management, see Chapter 21, “Windows.”

Frame Relationships

You can also parent a frame (*child frame*) to another frame (*parent frame*). This causes Progress to view the child frame within the display area of the parent frame. Frames that are parented by a frame, which in turn is parented by a window, form a *frame family*. The frame parented by the window is the *root frame* of the frame family. Frames parented by any child frame, in turn, form a *child frame family*. All frames in a frame family are viewed within the display area of the root frame, and all frames of a child frame family are displayed within the display area of the parenting child frame. As with root frames and their parent windows, a child frame can have only one parent frame at a time.

Figure 19–6 and Figure 19–7 show a typical frame family with three child frames. The root frame is the Customer Data frame, which is contained by the Update window.

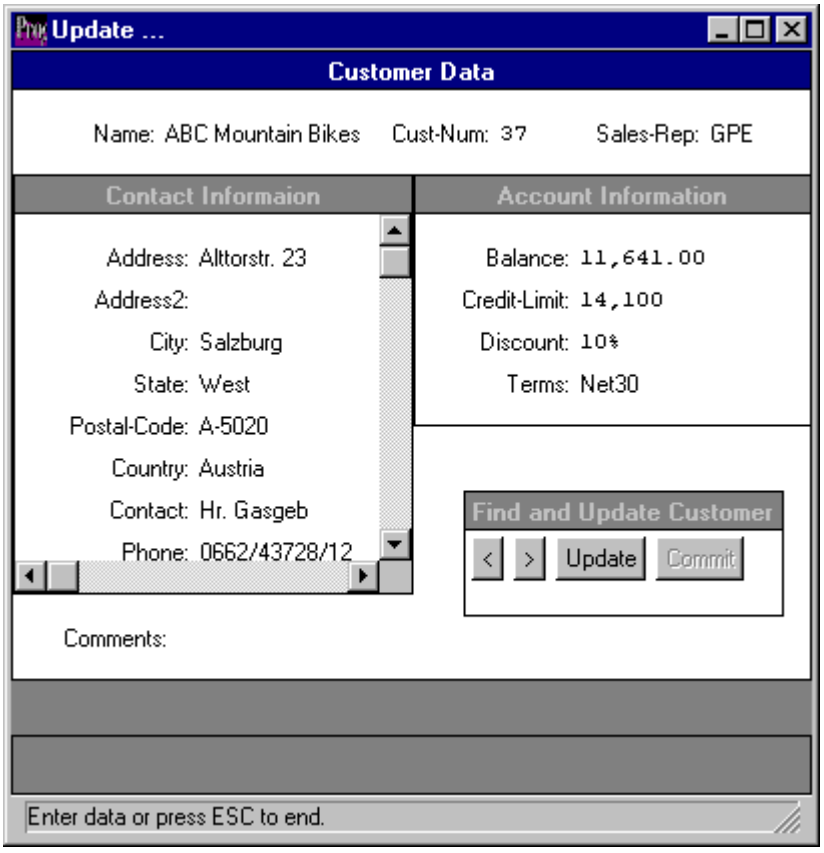


Figure 19–6: Frame Family (Windows Graphical Interface)

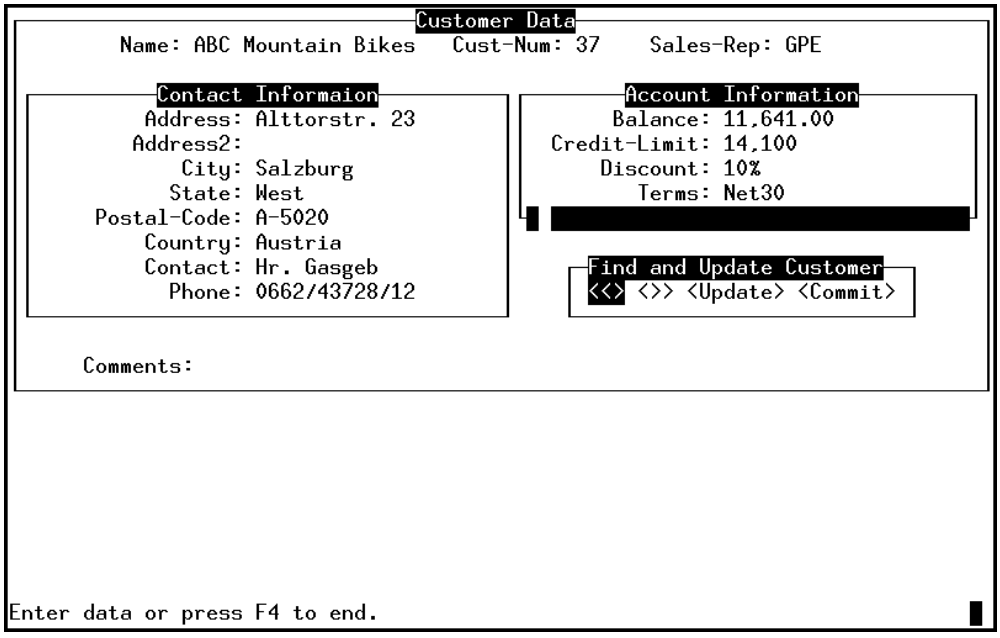


Figure 19–7: Frame Family (Character Interface)

The three child frames include the Contact Information, Account Information, and Find and Update Customer frames. Note how these three frames are nested between the top and bottom fields of the parent Customer Data frame.

Frame Family Behavior

Child frames behave like field-level widgets in a frame, as the remaining sections in this chapter explain. In fact, a frame is parented to a frame in exactly the same way that a field-level widget is parented to a frame, using a field group. For more information on field groups, see the "Field-group Widgets" section.

19.9.1 Defining a Frame Family

The first step in defining a frame family is to define a child frame of a root frame. To define a child frame of any frame, you set the child frame's FRAME attribute to the widget handle of the parent frame, as shown in the following procedure (`p-fof1.p`):

p-fof1.p

```
DEFINE FRAME fparent WITH SIZE 60 by 10 TITLE "Parent".
DEFINE FRAME fchild WITH SIZE 40 by 5 TITLE "Child".
FRAME fchild:FRAME = FRAME fparent:HANDLE.

DISPLAY WITH FRAME fparent.
ENABLE ALL WITH FRAME fparent.
ENABLE ALL WITH FRAME fchild.

WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

By default, Progress positions a child frame at row 1, column 1 (the upper–left corner) of the parent frame's display area. Thus, when you specify position options for a child frame, they are relative to the parent frame's display area, not the window's. In this way, you can define and position child frames of child frames (*descendant frames*) to any practical depth.

19.9.2 Using Frame Families

Child frames help to organize the display area of the parent frame. They can contain their own field-level widgets, and like any frame parented to a window, they can be referenced in frame I/O statements. Child frames can be viewed and hidden like field-level widgets and also participate in the tab order of the parent frame.

You can choose a default button or Cancel button from a child frame. If a child frame does not have the chosen button, invoking that button in the child frame will cause the first default or Cancel button found in the frame family (and in no special order) to be chosen.

You can also parent child frames to a dialog box, which can serve as a root frame for a frame family. However, child frames cannot be viewed as dialog boxes, themselves. For more information on dialog boxes, see Chapter 25, "Interface Design."

Child frames have some additional restrictions:

- You cannot use them as down frames; they can only be 1 down.
- They must fit at least partially within the virtual display area of their parent frame.
- They do not inherit the attributes of their parent frame.
- If you set SESSION:DATA–ENTRY–RETURN to TRUE, the last fill-in of a child frame can trigger a GO event for the frame if it is the last fill-in field in the frame family tab order. For more information, see the section on fill–in fields in Chapter 17, "Representing Data."
- You must display, enable, and disable the field-level widgets of child frames explicitly. Doing so for a parent frame has no effect on the fields displayed in its child frames.

The following procedure (`p-fof2.p`) uses a frame family to organize a window for updating customer records. It displays the Update window shown previously in Figure 19–6 and Figure 19–7.

p-fof2.p *(1 of 3)*

```
DEFINE QUERY custq FOR customer.
DEFINE VARIABLE cnt AS INTEGER.
DEFINE BUTTON bprev LABEL "<".
DEFINE BUTTON bnext LABEL ">".
DEFINE BUTTON bupdate LABEL "Update".
DEFINE BUTTON bcommit LABEL "Commit".
DEFINE FRAME cust-fr SKIP(.5)
    SPACE(8) customer.name customer.cust-num customer.sales-rep
    customer.comments AT COLUMN 6 ROW 13.5
    WITH SIDE-LABELS TITLE "Customer Data" KEEP-TAB-ORDER
&IF "{&WINDOW-SYSTEM}" <> "TTY" &THEN
        SIZE 80 BY 15.
&ELSE
        SIZE 80 BY 16.
&ENDIF
DEFINE FRAME cont-fr SKIP(.5)
    customer.address COLON 17 SKIP
    customer.address2 COLON 17 SKIP
    customer.city COLON 17 SKIP
    customer.state COLON 17 SKIP
    customer.postal-code COLON 17 SKIP
    customer.country COLON 17 SKIP
    customer.contact COLON 17 SKIP
    customer.phone COLON 17
    WITH SIDE-LABELS TITLE "Contact Informaion"
        SIZE 40 BY 10 AT COLUMN 1 ROW 3.
DEFINE FRAME ctrl-fr SKIP(.12)
    SPACE(.5) bprev bnext bupdate bcommit
    WITH TITLE "Find and Update Customer"
&IF "{&WINDOW-SYSTEM}" <> "TTY" &THEN
       SIZE 27.5 BY 2 AT COLUMN 46 ROW 10.5.
&ELSE
       SIZE 29 BY 3 AT COLUMN 45 ROW 10.
&ENDIF
DEFINE FRAME acct-fr SKIP(.5)
    customer.balance COLON 15 SKIP
    customer.credit-limit COLON 15 SKIP
    customer.discount COLON 15 SKIP
    customer.terms COLON 15
    WITH SIDE-LABELS TITLE "Account Information"
        SIZE 39.8 BY 6 AT COLUMN 41 ROW 3.

ON CHOOSE OF bnext IN FRAME ctrl-fr DO:
    GET NEXT custq.
    IF NOT AVAILABLE customer THEN GET FIRST custq.
    RUN display-proc IN THIS-PROCEDURE.
END.
```

p-fof2.p *(2 of 3)*

```
ON CHOOSE OF bupdate IN FRAME ctrl-fr DO:
    DISABLE ALL WITH FRAME ctrl-fr.
    ENABLE bcommit WITH FRAME ctrl-fr.
    ENABLE ALL EXCEPT customer.cust-num WITH FRAME cust-fr.
    ENABLE ALL WITH FRAME cont-fr.
    ENABLE ALL EXCEPT customer.balance WITH FRAME acct-fr.
END.

ON CHOOSE OF bcommit IN FRAME ctrl-fr DO:
    DISABLE ALL WITH FRAME cust-fr.
    DISABLE ALL WITH FRAME cont-fr.
    DISABLE ALL WITH FRAME acct-fr.
    ENABLE ALL WITH FRAME ctrl-fr.
    DISABLE bcommit WITH FRAME ctrl-fr.
    GET CURRENT custq EXCLUSIVE-LOCK.
    IF NOT CURRENT-CHANGED(customer) THEN DO:
        ASSIGN customer EXCEPT customer.cust-num customer.balance.
        RELEASE customer.
    END.
    ELSE DO:
        GET CURRENT custq NO-LOCK.
        RUN display-proc IN THIS-PROCEDURE.
        DO cnt = 1 TO 12: BELL. END.
        MESSAGE "Customer" customer.name SKIP
                "was changed by another user." SKIP
                "Please try again..." VIEW-AS ALERT-BOX.
    END.
END.

FRAME cont-fr:FRAME = FRAME cust-fr:HANDLE.
FRAME acct-fr:FRAME = FRAME cust-fr:HANDLE.
FRAME ctrl-fr:FRAME = FRAME cust-fr:HANDLE.

IF NOT SESSION:WINDOW-SYSTEM = "TTY"    THEN CURRENT-WINDOW:TITLE = "Update
...".
OPEN QUERY custq PRESELECT EACH customer NO-LOCK BY customer.name.
GET FIRST custq.
RUN display-proc IN THIS-PROCEDURE.
ENABLE ALL WITH FRAME ctrl-fr.
DISABLE bcommit WITH FRAME ctrl-fr.

WAIT-FOR WINDOW-CLOSE OF FRAME cust-fr.
```

p-fof2.p *(3 of 3)*

```
PROCEDURE display-proc:
   DISPLAY
        customer.name customer.cust-num customer.sales-rep
        customer.comments WITH FRAME cust-fr.
    DISPLAY
        customer.address customer.address2 customer.city customer.state
        customer.postal-code customer.country customer.contact
        customer.phone WITH FRAME cont-fr.
    DISPLAY
        customer.balance customer.credit-limit
        customer.discount customer.terms WITH FRAME acct-fr.
END.
```

Note how each frame is managed individually for input and output. Displaying frame cust–fr displays its child frames, but not their field values. Likewise, enabling the fields of cust–fr for input has no effect on the fields in cont–fr, acct–fr, or ctrl–fr. Each of these child frames must have their field-level widgets enabled and disabled individually.

When enabled, you can tab among all the field-level widgets in the cust–fr frame family. For more information on frame family layout and tabbing behavior, see the remaining sections of this chapter and Chapter 25, “Interface Design.”

19.10 Frame Services

Progress provides the following services for each frame in a block:

- Viewing and hiding
- Advancing and clearing
- Retaining previously entered data during RETRY of a block
- Creating field-group widgets

19.10.1 Viewing and Hiding Frames

Progress brings a frame into view when you display data in that frame. You can also use the VIEW statement (or you can set the frame's VISIBLE attribute to TRUE) to explicitly bring a frame into view. You can use the HIDE statement to explicitly hide a frame (or you can set the frame's VISIBLE attribute to FALSE).

NOTE: If a parent or ancestor window of a frame has its HIDDEN attribute set to TRUE, you can set the frame's attributes (directly or indirectly with VIEW and data display statements) to display the frame, but Progress does not display the frame until all ancestor windows have their HIDDEN attributes set to FALSE.

When displaying frames within a window, Progress tiles the frames, starting at the top of the window and proceeding to the bottom until the window is full. By default, every frame begins at column 1. If there are more frames than fit in the window, Progress hides (erases) the frames closest to the bottom of the window until enough space is cleared to display the next frame. If ready to hide a frame, Progress pauses, by default, and displays the message "Press space bar to continue".

Within a window, you can display one frame over another using the OVERLAY option, and you can make a frame fit in a display area that is smaller than the frame itself, using the SIZE option of the Frame phrase. For more information on the SIZE option, see Chapter 25, "Interface Design."

Bringing Frames into View

You can bring a frame into view by displaying data in that frame, using the VIEW statement, or setting the frame's VISIBLE attribute to true. In the following procedure, the DISPLAY statement brings the frame into view:

p-form1.p

```
FORM customer.name contact AT 40 SKIP
     customer.address credit-limit AT 40 SKIP
     customer.city customer.state NO-LABEL
     customer.postal-code NO-LABEL balance AT 40 SKIP(1)
     phone
     HEADER "Customer Maintenance" AT 25 SKIP(1)
     WITH SIDE-LABELS NO-UNDERLINE FRAME a.

FOR EACH customer WITH FRAME a:
  DISPLAY curr-bal.
  UPDATE name address city state postal-code phone contact credit-limit.
END.
```

The FORM statement describes frame a. To bring the frame into view before the DISPLAY statement, you can use the VIEW statement. In the next example, the VIEW statement brings frame b into view.

p-form2.p

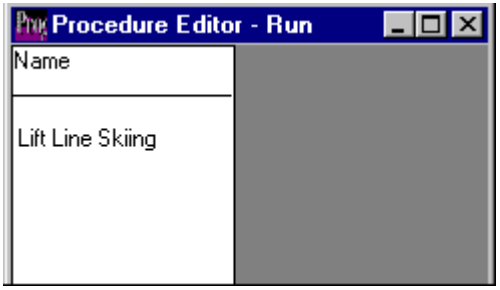
```
DEFINE FRAME b WITH ROW 7 CENTERED SIDE-LABELS
       TITLE "Customer Info".

VIEW FRAME b.

REPEAT:
  PROMPT-FOR customer.cust-num WITH FRAME a ROW 1.
  FIND customer USING cust-num NO-LOCK.
  DISPLAY customer EXCEPT comments WITH FRAME b.
END.
```

In the next example, Progress automatically removes frames from the screen by hiding them:

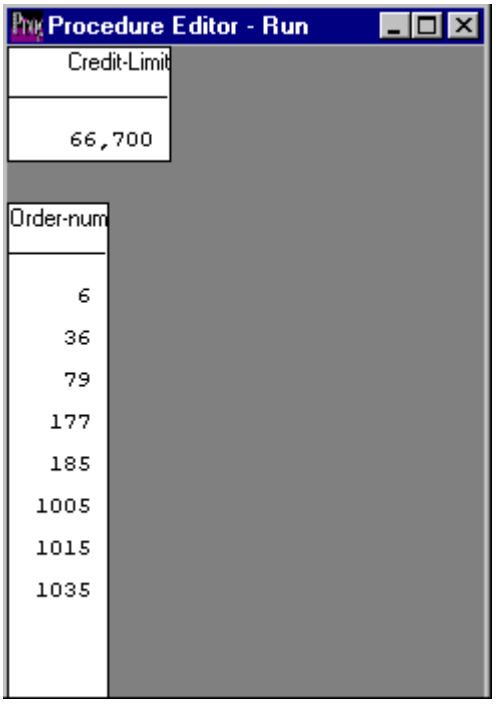
p-hide.p

```
FOR EACH customer:
  DISPLAY name WITH FRAME f1 DOWN.
  DISPLAY credit-limit WITH FRAME f2.
  FOR EACH order OF customer WITH FRAME f3:
    DISPLAY order-num.
  END.
END.
```

The output of this procedure is as follows:



At this point, the procedure has displayed the customer name. But there is no more vertical space on the screen in which to display the max–credit and order numbers. Press **SPACEBAR**.



You can see that Progress automatically hides frame f1 to make room for frames f2 and f3. When Progress runs out of space, it hides frames starting from the bottom of the window.

Using Overlay Frames

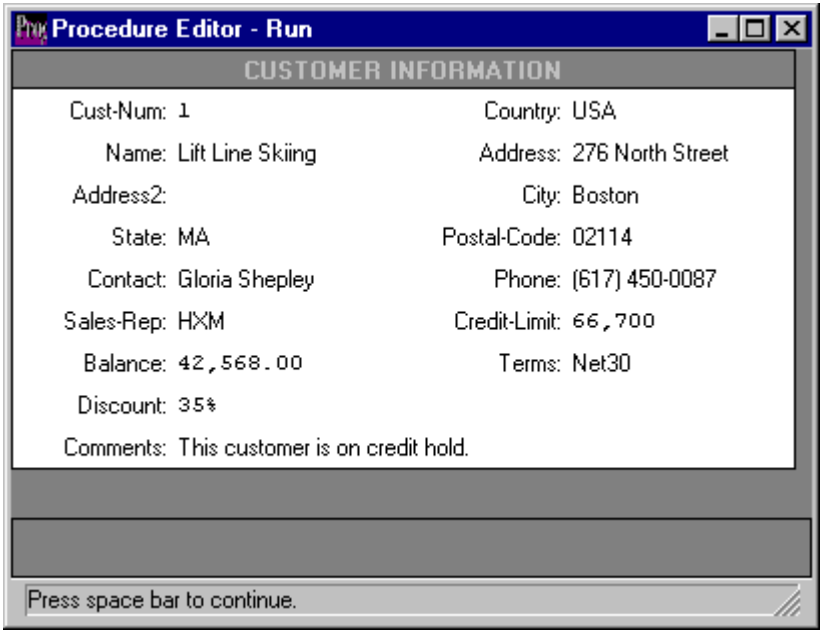
Progress allows you to create a frame that overlays another frame. To display one frame over another, set the top frame's OVERLAY attribute to true (or use the OVERLAY frame phrase option). Also, make sure that the bottom frame's TOP–ONLY attribute is set to false. Progress displays a message before it displays the overlay frame.

For example, you might want to display all the information for a customer, then see that customer's orders one at a time while keeping at least some of the customer information in view. The following procedure uses an overlay frame to accomplish this:

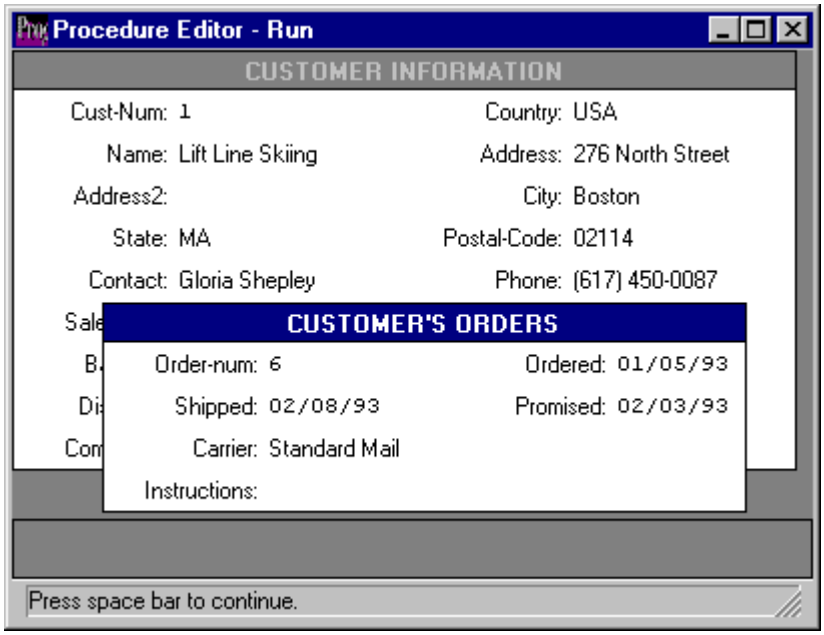
p-ovrlay.p

```
FOR EACH customer:
    DISPLAY customer WITH 2 COLUMNS TITLE "CUSTOMER INFORMATION".
    FOR EACH order OF customer:
        DISPLAY order-num order-date ship-date promise-date carrier
                instructions WITH 2 COLUMNS 1 DOWN OVERLAY
                TITLE "CUSTOMER'S ORDERS" ROW 7 COLUMN 10.
    END.
END.
```

This procedure produces the following output:



When you press **SPACEBAR**, if the customer has orders, you see the customer's first order in a frame that overlays the first frame on your screen, as shown in the following figure:



When you press **SPACEBAR**, the next customer appears, the order frame clears, and the next order for the customer appears. Progress automatically refreshes a covered frame when an overlay frame clears.

The `p-ovrlay.p` procedure displays the frame with the order information over the frame with the customer information because the frame phrase for the order frame includes the OVERLAY option. You can use the ROW and COLUMN options to control where the overlay frame is placed on the screen. If you do not include the ROW and COLUMN options, the overlay frame displays with the upper left corner in row 1, column 1 (unless you use some other option to affect its placement, such as the CENTERED option).

Working with Multiple Overlay Frames

If every frame is an overlay frame and no frame has its TOP–ONLY attribute set to TRUE, you can bring any overlay frame to the top by giving focus to one of its field-level widgets. You can also **prevent** all overlay frames from changing their current overlay position (*Z order*) by setting the window's KEEP–FRAME–Z–ORDER attribute to TRUE. No matter how this attribute is set, you can always change the Z order of an overlay frame using these methods:

- MOVE–TO–TOP( ) — Moves the frame to the top of the Z order.
- MOVE–TO–BOTTOM( ) — Moves the frame to the bottom of the Z order.

The default Z order of multiple overlay frames is the order in which they are enabled.

Child frames are actually permanent overlay frames of their parent frame. When you overlay several child frames, they maintain their collective Z order on top of the parent frame. However, you can change the relative Z order of the child frames within the parent using these methods.

For more information on the OVERLAY option of the Frame phrase and the methods changing frame Z order, see the *Progress Language Reference*.

NOTE: Dialog boxes also appear on top of other frames and may be preferable to overlay frames. For more information on dialog boxes, see Chapter 25, "Interface Design," and the *Progress Language Reference*.

Viewing and Hiding Frame Families

When you view and hide a parent frame, all of its child frames and field-level widgets whose HIDDEN attributes are set to FALSE are viewed and hidden with it. The viewing and hiding of child frames also works as it does for field-level widgets.

When you use the VIEW statement to view a child frame, Progress views the child frame and all of its ancestor frames, even if their HIDDEN attributes are set to TRUE. If necessary, Progress resets their HIDDEN attributes to FALSE. However, if you use the DISPLAY or ENABLE statement to view a child frame, Progress sets the attributes appropriately to display the child frame, but only displays it on the screen if none of its ancestor frames have their HIDDEN attributes set to TRUE. In this case, when all ancestor frames have their HIDDEN attributes set to FALSE, the previously displayed or enabled child frame is then displayed on the screen.

When you hide a child frame, like a field-level widget, its HIDDEN attribute is set to TRUE. Thus, if you later hide and redisplay an ancestor frame, the hidden child frame stays hidden until you explicitly view or display it.

When you run the sample procedure `p-fof3.p`, the following window appears, which lets you view and hide frame families:

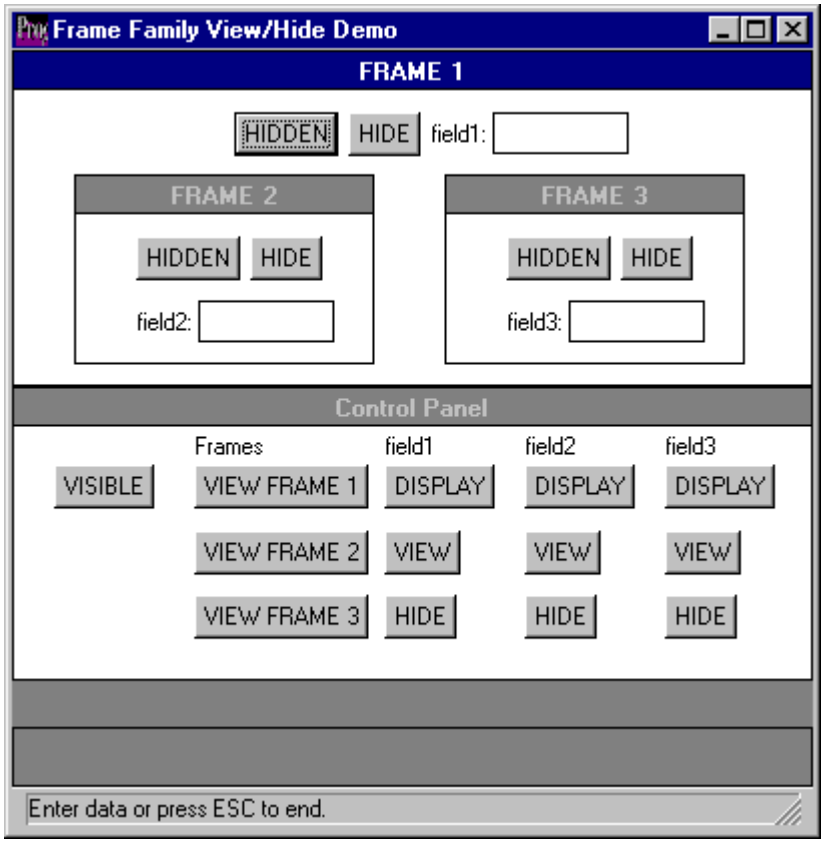


Figure 19–8: Frame Family Viewing Demo

You can hide and view each frame and fill-in, displaying the HIDDEN and VISIBLE attribute values in the message area as appropriate.

19.10.2 Advancing and Clearing Frames

When a block iterates, Progress advances any down frames scoped to the block to the next iteration. Progress also clears the field values in the screen buffer for all one-down frames scoped to the block. These actions take place only if you displayed or entered data into at least one of the fields in the frame during the iteration.

When a down frame is full, Progress clears the entire frame before displaying more data. You can use the RETAIN frame phrase option to specify a number of iterations of data that you do not want Progress to clear.

19.10.3 Retaining Data During Retry of a Block

When the procedure retries a block, either because of an implicit or explicit UNDO or RETRY, Progress does not clear the frame scoped to that block. That way, the data are still available when the user enters new data or changes the data entered on the first try.

19.10.4 Creating Field-group Widgets

For each frame, Progress automatically creates one or more *field-group widgets* to hold the field-level widgets. For more information on field groups, see the "Field-group Widgets" section.

19.11 Using Shared Frames

Sometimes you want several procedures to use the same frame. You might use the same frame throughout an application or with a few different procedures in an application. Progress lets you define a frame as shared, so that many procedures can use it. You use the DEFINE SHARED FRAME statement to define a shared frame. You must describe the shared frame in a FORM statement.

You can write a procedure to display customer information, another procedure to update order information, and you can display all the information in the same frame. One way to do this is to define a shared frame.

Figure 19–9 shows how two procedures can use the same frame.

p-dicust.p
p-updord.p
p-dicust.i

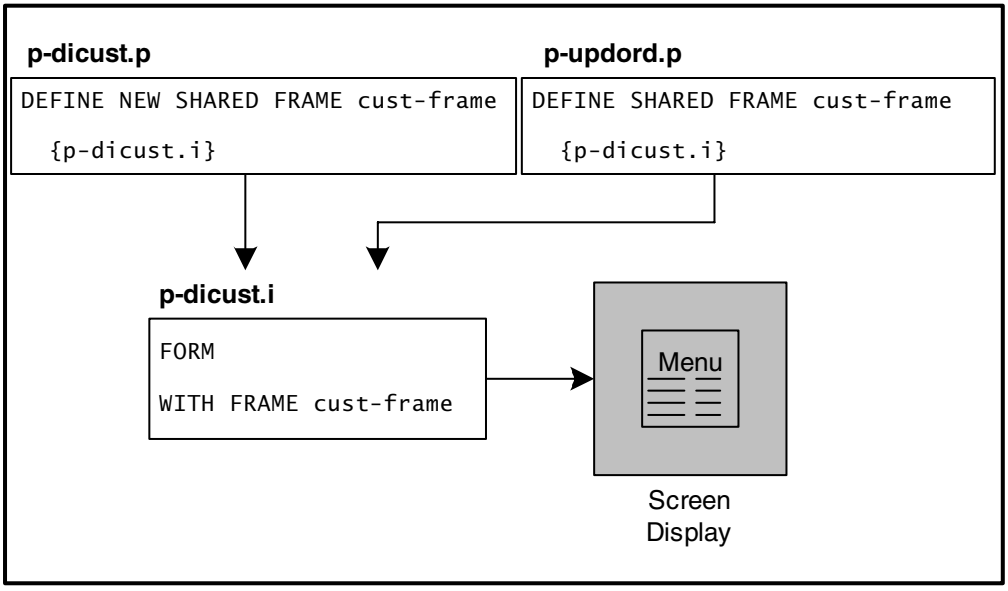


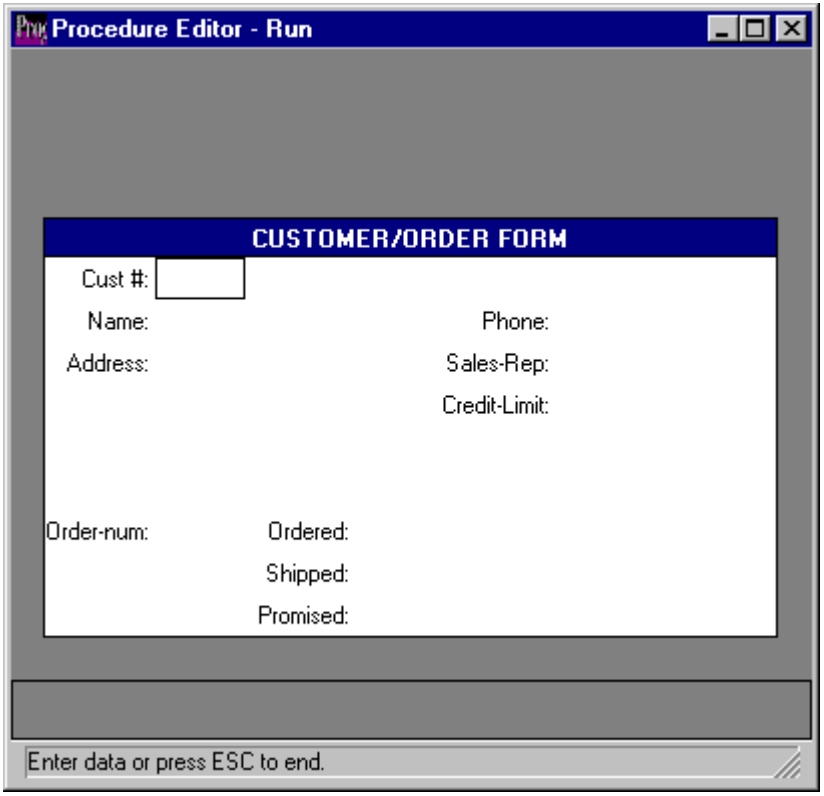
Figure 19–9: Shared Frames

The FORM statement in this example is in an include file, `p-dicust.i`. This file sets up and names the cust–frame:

p-dicust.i

```
FORM
     xcust.cust-num   COLON 10 LABEL "Cust #"
     xcust.name    COLON 10           xcust.phone        COLON 50
     xcust.address COLON 10           xcust.sales-rep    COLON 50
     csz     NO-LABEL COLON 12        xcust.credit-limit COLON 50
     SKIP (2)
     order.order-num    COLON 10      order.order-date   COLON 30
     order.ship-date    COLON 30
     order.promise-date COLON 30   WITH SIDE-LABELS 1 DOWN CENTERED ROW 5
        TITLE " CUSTOMER/ORDER FORM " FRAME cust-frame.
```

Although you cannot run the `p-dicust.i` file by itself because it is only a FORM statement, the file describes the following frame:



The p-dicust.p procedure displays customer information in this frame:

p-dicust.p

```
DEFINE NEW SHARED FRAME cust-frame.
DEFINE NEW SHARED VARIABLE csz AS CHARACTER FORMAT "x(22)".
DEFINE NEW SHARED BUFFER xcust FOR customer.

REPEAT:
   {p-dicust.i}     /* include file for layout of shared frame */

   PROMPT-FOR xcust.cust-num WITH 1 DOWN FRAME cust-frame.
   FIND xcust USING xcust.cust-num NO-ERROR.
   IF NOT AVAILABLE(xcust)
   THEN DO:
      MESSAGE "Customer not found".
      NEXT.
   END.

   DISPLAY
      name phone address sales-rep
      city + ", " + state + " " + STRING(postal-code) @ csz
      credit-limit WITH FRAME cust-frame.

   RUN p-updord.p.    /* External procedure to update customer's orders */

END.
```

The p-dicust.p procedure displays customer information in the shared frame cust–frame and lets the user update the order information for the customer on the screen. The procedure defines the new shared frame named cust–frame and a new shared buffer, xcust, to store customer information. (You can use only one DEFINE SHARED FRAME statement per frame in a procedure.) The name cust–frame corresponds to the frame in the FORM statement in the p-dicust.i include file.

The procedure prompts the user for a customer number and displays the customer information for that customer using the frame defined in the p-dicust.i file. Then the procedure calls the p-updord.p procedure, shown below:

p-updord.p

```
DEFINE SHARED FRAME cust-frame.
DEFINE SHARED VARIABLE csz AS CHARACTER FORMAT "x(22)".
DEFINE SHARED BUFFER xcust FOR customer.

FOR EACH order OF xcust:
    {p-dicust.i}      /* include file for layout of shared frame */

    DISPLAY order.order-num WITH FRAME cust-frame.
    UPDATE order.order-date order.ship-date
           order.promise-date WITH FRAME cust-frame.
END.
```

The p-updord.p procedure defines the shared frame cust–frame and the shared variable csz, which was first defined in the p-dicust.p procedure. It also defines the shared buffer to store customer information. You must use the DEFINE SHARED VARIABLE statement for the csz variable because variables named in a shared frame are not automatically shared variables. You must also use the DEFINE SHARED FRAME statement for the cust–frame so that Progress can recognize the frame.

The p-updord.p procedure finds the customer that the p-dicust.p procedure is using in the shared buffer, xcust. The procedure then displays the information for each order of the customer in the shared frame called cust–frame. Finally, the procedure allows the user to update the order information.

When you use shared frames, remember these important points:

- In the first procedure where you reference a shared frame, use the DEFINE NEW SHARED FRAME statement. In subsequent procedures where you reference the shared frame, use the DEFINE SHARED FRAME statement.

- You must completely describe the shared frame in the initial DEFINE NEW SHARED FRAME statement or an additional FORM statement. Procedures that share this frame only have to define fields that correspond to the fields in the initial definition plus any specified ACCUM option. Other Frame phrase options for the SHARED frames are allowed but are ignored, except for the ACCUM option. This allows you to make use of the same FORM statement in an include file for both the NEW SHARED and matching SHARED frames.

- A shared frame is scoped to the procedure or the block within the procedure that defines it as NEW. For example, in the `p-updord.p` procedure above, the `p-dicust.i` include file has no effect on scope.
- A shared frame is scoped only once, and the scope is in the procedure where the frame is defined as NEW.
- If you want to share a frame family, you must make all parent and child frames in the family shared frames.

See the *Progress Language Reference* for more information on the DEFINE FRAME statement.

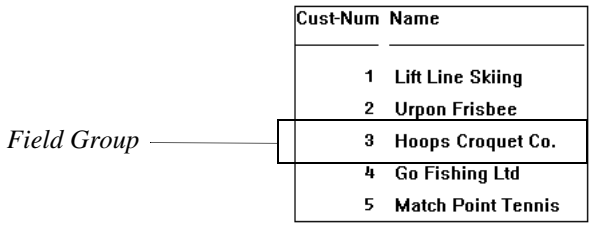
19.12 Field-group Widgets

When you display field-level widgets in a frame or assign a child frame to a parent frame, Progress creates one or more *field-group widgets* to hold the field-level widgets and child frames. (For information on assigning a child frame to a parent frame, see the "Frame Families" section.)

Progress uses field groups to collect field-level widgets and child frames. You might find it helpful to think of a field group as a kind of geometrical representation of a record (or an iteration). For example, this code fragment displays five records, each contained in a field group.

```
FOR EACH customer WITH 5 DOWN.
  DISPLAY cust-num name.
END.
```

The output of this fragment is shown below.



All field-level widgets, such as buttons, are placed in field groups. A field group is a collector of field-level widgets and child frames. However, Progress also allocates empty field groups. For example, Progress allocates a field group to the background of a frame, whether or not the field group contains any widgets.

Field groups are the children of frames and the real parents of field-level widgets and child frames. This makes frames the grandparents of field-level widgets and child frames. Thus, if you want an application to perform an operation on field-level widgets in a frame without knowing beforehand what widgets are in the frame, the application must access them by going through their respective field groups. For more information on accessing field groups, see the "Accessing Field Groups" section.

Field groups widget are unique in that Progress, not the application, is completely responsible for their creation and deletion. An application can access the field groups, but cannot create or delete them.

In most cases, it is not necessary to access field groups from within an application. Like child frames, you can move individual field-level widgets in and out of frames by setting their FRAME attribute.

Finally, field groups organize the tab order of the field-level widgets and child frames that they contain. For more information, see Chapter 25, "Interface Design."

19.12.1 Field-group Types

Progress allocates two types of field groups:

- Foreground
- Background

Progress can allocate multiple foreground field groups per frame but only one background field group. Figure 19–10 shows the different field groups.

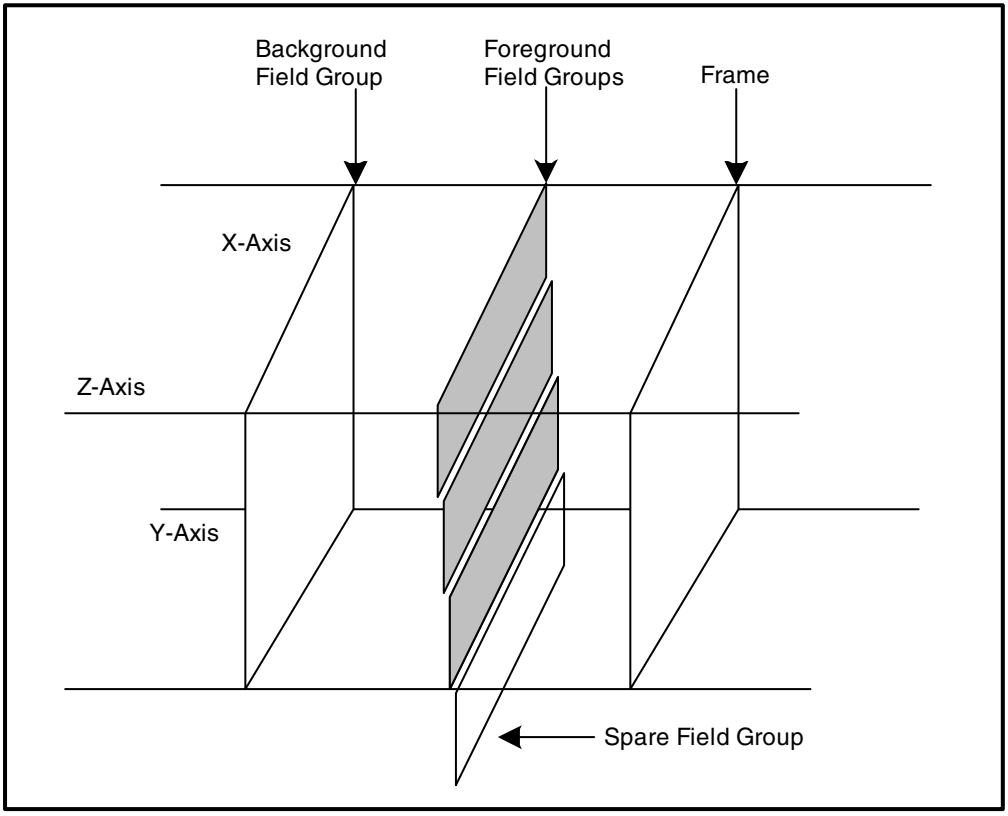


Figure 19–10: Field-Group Allocation

For a one-down frame, Progress allocates one foreground field group and one background field group. For a down frame, Progress allocates one foreground field group for each iteration (plus one extra if the number of visible iterations in the down frame is less than the number of records that need to be displayed) and one background field group.

19.12.2 Field-group Characteristics

Field groups have the following characteristics:

- They have geometric attributes (an x and y position, width, and height).
- They have no visible representation (color, bounding box, or font).
- They take no events.
- They are created implicitly by Progress (you cannot create or define a field group).

Once Progress has created a field group, you can access it from within an application. For more information, see the "Accessing Field Groups" section.

19.12.3 Field-group Size

Foreground field groups are as wide as the interior of the frame. The background field group is also as wide as the interior of the frame.

Foreground field groups vary in height. In a dynamic one-down frame (and a static one-down frame without column labels), the foreground field group is as high as the interior of the frame. In a static one-down frame with column labels, the foreground field group starts below the column headers; as a result, its height is less than the interior of the frame. In a down frame, the foreground field groups are high enough to contain the field-level widgets displayed in one iteration.

You can display dynamic field-level widgets within a field group. However, it is not recommended that you place dynamic widgets in the foreground field groups of a down frame. Not only is the height of these field groups determined by Progress (the application cannot control the height), but the field group may also scroll off the screen (either in a scrolling frame or if the frame retains a number of iterations). After it reappears, the field group will still contain the dynamic widget, but also may contain a new record. For more information, see the "Field-group Visibility and Scrolling" section.

You cannot mark, move, or resize field groups directly. However, if you resize a frame, Progress automatically resizes the field groups by default.

19.12.4 Field-group Position Relative to Frame

Frames contain field groups, which contain field-level widgets. Field groups are therefore positioned relative to frames, while field-level widgets are positioned relative to field groups. Consistent with this, the X, Y, ROW, and COLUMN attributes of a field group return values relative to the frame. The X, Y, ROW, and COLUMN attributes of a field-level widget (including radio buttons) return values relative to the parent field group.

To access the location of a field-level widget relative to its frame, query the field-level widget's FRAME–X and FRAME–Y attributes (for pixel values), or FRAME–COL and FRAME–ROW attributes (for character unit values).

For a list of field-group attributes, see the FIELD-GROUP Widget reference entry in the *Progress Language Reference*.

19.12.5 Z order and Field Groups

If you overlap widgets in an application, Progress must determine which widget to place on top. Within a frame, there are three layers:

- The foreground layer, with all the *nondrawable* field-level widgets in the foreground field group. This is the topmost layer. (A drawable field-level widget is a rectangle or image.)
- The background layer, with all the nondrawable field-level widgets in the background field group. This is the next topmost layer.
- The drawable layer, with all the drawable field-level widgets from all of the field groups. This is the bottommost layer.

Within each z layer, widgets also have a z order (that is, one foreground button may be on top of another foreground button). However, Progress places all nondrawable foreground widgets on top of all nondrawable background widgets, and places all drawable widgets on the bottom layer. For example, if you have a text widget in the background and an image in the foreground, the text widget appears on top of the image.

NOTE: Figure 19–10 shows the relative z-order positions of the foreground and background field groups (Z–Axis), but does not show the separate drawable layer derived from both of these field groups. The drawable layer is not a field group, itself, but a z-order layer **under** the background field group (extreme left of Figure 19–10) containing drawable widgets from both field group layers.

If you want an application to be portable to character displays, you should avoid using overlapping widgets. In a character display, if two field-level widgets overlap, one of the widgets may appear, both may appear, or some part of each may appear.

19.12.6 Field-group Visibility and Scrolling

Figure 19–10 shows a spare field group. Progress allocates a spare field group if the number of visible iterations is less than the number of records that need to be displayed. At any given time, one of the foreground field groups in a down frame can be the spare. That is, the foreground field group can scroll off the screen.

The frame's FIRST–CHILD/NEXT–SIBLING chain includes all of the field groups in the frame (including the spare). Its order is not affected by scrolling (see the "Accessing Field Groups" section).

However, the frame also has an GET–ITERATION( ) method, which along with the NUM–ITERATIONS attribute, simulates an array of all of the **visible** foreground field groups. Thus, if you want to find only the visible foreground field groups at any instant, you can use the GET–ITERATION( ) method with the NUM–ITERATIONS attribute.

The GET–ITERATION( ) method takes an integer argument *n* and, starting at the top of the frame, returns the widget handle of the field group that displays the *n*th visible iteration. You can access all of these field groups by using a loop that starts at one (GET–ITERATION(1)) and runs through the total number of visible iterations.

If the frame is named X, you can access the last visible iteration as follows:

```
GET-ITERATION(FRAME X:NUM-ITERATIONS)
```

If you have a widget handle variable named X, you can access the last visible iteration as follows:

```
GET-ITERATION(X:NUM-ITERATIONS)
```

As the frame scrolls, these attributes return different values. Thus, if an application wants to cycle through the visible iterations using the iteration loop, it should not do anything during the cycle that would scroll the frame.

This code fragment produces a frame with five visible iterations.

```
FOR EACH customer WITH 5 DOWN.
  DISPLAY cust-num name.
END.
```

Before Progress executes the FOR EACH loop, NUM–ITERATIONS equals 0 (because Progress hasn't created any field groups yet). The first time through the loop, after Progress displays the first cust–num and name, NUM–ITERATIONS equals 1. The second time through, NUM–ITERATIONS equals 2. Once Progress has displayed the fifth cust–num and name, NUM–ITERATIONS equals 5. The count will remain 5 until the frame is terminated. Note that after Progress starts scrolling, there will be 6 field groups—5 plus 1 spare (the spare will be different as the frame scrolls). The NUM–ITERATIONS attribute will stay at 5, but there will actually be 6 field groups.

Progress does not create new field groups (and all their underlying field-level widgets) as they are scrolled off the screen. Instead, it clears the existing field groups and then reuses them.

If an application uses the UNDERLINE statement, Progress uses one foreground field group to underline the fields in the field group directly above it. Also, if the application uses a DOWN statement to advance one or more iterations in the frame without displaying any data, Progress uses one of the foreground field groups to display a blank iteration. For example, this code fragment uses a DOWN statement to display a blank iterations.

```
REPEAT WITH FRAME a DOWN:
  FIND NEXT customer.
  DISPLAY cust-num name.
  DOWN 1 WITH FRAME a.
END.
```

If you insert dynamic field-level widgets in a foreground field group of a down frame, you should be aware of the following defaults:

- As a frame scrolls, Progress moves the foreground field groups around in the frame. Thus, if an application places a dynamic widget in a field group, the dynamic widget will jump around in the frame as the field group moves.
- Any foreground field group may become the spare field group as the field groups are moved around in the frame. When a field group becomes the spare, it will not be visible on screen.
- If Progress uses a field group to display a blank iteration or uses an underline field group, it hides all of the field group's children (including any dynamic widgets).
- When a field group scrolls off the top or bottom of the frame, Progress clears the field group. All fill-in widgets are reset to blanks, including any dynamic fill-in widgets.

NOTE: Because of the complex nature of these defaults, PSC recommends that you do not place dynamic widgets in a foreground field group of a down frame.

19.12.7 Accessing Field Groups

Different applications may want to access all of the field groups in a frame, the foreground field groups, or the visible foreground field groups. The following examples show how an application can access these different field groups.

Example 1

This example shows how an application might access all of the field groups (and their field-level widgets and child frames) in a frame.

```
DEFINE VARIABLE GRP AS WIDGET-HANDLE.  /* the field group */
DEFINE VARIABLE FLW AS WIDGET-HANDLE.  /* the field-level widget */

GRP = FRAME FRX:FIRST-CHILD.
DO WHILE (GRP <> ?).
  FLW = GRP:FIRST-CHILD.
  DO WHILE (FLW <> ?). /* loop through field-level widgets in
                          field-group */

    /* insert code here to access the widget */

    FLW = FLW:NEXT-SIBLING.
  END.
  GRP = GRP:NEXT-SIBLING.
END.
```

This example uses the frame's FIRST–CHILD/NEXT–SIBLING chain to access all of the field groups, field-level widgets, and child frames in frame FRX (including the spare field group, if one exists). FLW is set to each sibling field-level widget and child frame of the frame. Note that the order in which this loop accesses the field groups is not connected with the order in which the field groups are displayed on screen.

NOTE: This example does not access any widgets parented by child frames. To accomplish that requires a recursive version of this procedure.

Example 2

This example shows how an application might access all of the foreground field groups (and their field-level widgets and child frames) in a frame.

```
DEFINE VARIABLE GRP AS WIDGET-HANDLE.  /* the field group */
DEFINE VARIABLE FLW AS WIDGET-HANDLE.  /* the field-level widget */

GRP = FRAME FRX:FIRST-CHILD.
DO WHILE (GRP <> ?).
  IF (GRP:FOREGROUND) THEN DO:
    FLW = GRP:FIRST-CHILD.
    DO WHILE (FLW <> ?). /* loop through field-level widgets in
                            field-group */

      /* insert code here to access the widget */

      FLW = FLW:NEXT-SIBLING.
    END.
  END.
  GRP = GRP:NEXT-SIBLING.
END.
```

This example uses the frame's FIRST–CHILD/NEXT–SIBLING chain and the FOREGROUND attribute to access all of the foreground field groups, field-level widgets, and child frames in frame FRX (including the spare field group, if one exists).

NOTE: This example does not access any widgets parented by child frames. To accomplish that requires a recursive version of this procedure.

Example 3

This example shows how an application might access all of the visible foreground field groups (and their field-level widgets) in a down frame.

```
DEFINE VARIABLE IDX AS INTEGER.       /* index of visible iterations */
DEFINE VARIABLE GRP AS WIDGET-HANDLE. /* the field-group */
DEFINE VARIABLE FLW AS WIDGET-HANDLE  /* the field-level widget */

DO IDX = 1 TO FRAME FRX:NUM-ITERATIONS:
  GRP = FRAME FRX:GET-ITERATION(IDX).
  FLW = GRP:FIRST-CHILD.
  DO WHILE (FLW <> ?). /* loop through field-level widgets in
                          field-group */

    /* insert code to access the widget */

    FLW = FLW:NEXT-SIBLING.
  END.
END.
```

This example uses the frame's FIRST–CHILD/NEXT–SIBLING chain and the GET–ITERATION( ) method to access all of the foreground widgets in the visible field groups frame FRX.

19.13 Validation of User Input

Progress allows you to define *validation expressions* and user *validation messages* directly in the Data Dictionary or programmatically with the VALIDATE option of the Format phrase and the Browse Format phrase. The validation expression establishes the Boolean condition that defines valid data. Progress displays the validation message to the user (in the message area or in an alert box, depending on the configuration) if the new data fails the validation expression. Progress executes validation expressions when the user attempts to leave a field. Once the user begins to enter new data in, the user cannot leave the field without entering valid data or cancelling the input.

Validation of user input is fundamental to guaranteeing the integrity of your database. Progress provides all the language and system options you require to provide seamless validation of all user input. First, this section outlines the most efficient and straightforward way to provide complete validation. Then, it discusses the role of each language and system option that contribute to the validation story and scenarios that can lead to validation holes.

19.13.1 Recommended Validation Techniques

Follow the short list of recommendations below for complete validation of user input:

- Put as much of your validation expressions in the database schema with the Data Dictionary.
- For custom validation, add the VALIDATE option to the field reference in the DEFINE FRAME, FORM, or DEFINE BROWSE statements. While the VALIDATE option can legally be added in many places, placing it in one of these statements guarantees that it will be compiled into the frame or browse definition.
- Do not add fields to a frame after the initial definition (not possible for a browse). Or, if you do, use programmatic validation.
- Do not use the *field*:SENSITIVE = YES syntax to enable fields for input. Data Dictionary validation is not compiled into a frame when this syntax is used.

19.13.2 The Rules of Frame-based Validation

In Progress, validation of user input is a frame-based activity. The compiler adds validation expressions to a frame from the Data Dictionary the first time a field is referenced in that frame in an input statement (ENABLE, PROMPT–FOR, SET, UPDATE). The compiler adds programmatic validation (VALIDATE option) to the frame when it is encountered, as long as it has not already compiled in Data Dictionary validation. So, if you want programmatic validation to override existing Data Dictionary validation it is safest to place it in the DEFINE FRAME, FORM, or DEFINE BROWSE statement. After the compiler adds initial validation for a field in a frame, all other validation for that field in that frame is ignored.

To take advantage of this default behavior, add all of your default validation to database schema and use the VALIDATE option for special cases. Since the default behavior is frame-based, you can use different validation for the same field in different frames.

The ENABLE statement with the ALL option poses a special problem for the compiler, since it cannot know syntactically which fields in the frame require Data Dictionary validation. To avoid possible validation holes, the compiler adds Data Dictionary validation for all fields in the frame, even if they are defined as non-sensitive widgets. If you add a field to a frame after the compiler encounters the first ENABLE ALL statement, Data Dictionary validation will not be added to the frame for this field. To avoid creating a possible validation hole, do not add new fields to frames, or use the VALIDATE option to provide programmatic validation.

If you use the EXCEPT option with ENABLE ALL, the compiler will not add Data Dictionary validation to the frame for the fields listed with EXCEPT.

Finally, note that Progress does not have any default validation behavior for the *widget*:SENSITIVE = YES syntax. Using this back–door technique for enabling fields for input creates a validation hole. If you use this technique, then you will have to force frame-wide validation to close any possible validation holes. The next section discusses frame-wide validation techniques.

19.13.3 Forcing Frame-wide Validation

To make sure that your application does not have any validation holes, you have three options:

- Add the USE–DICT–EXPS option on the Frame phrase for each affected frame.
- Use the Dictionary Expressions startup parameter to apply the USE–DICT–EXPS option to every frame in the application.
- Use the frame VALIDATE( ) method to check all the fields in a frame before committing the changes.

Using the USE–DICT–EXPS Frame Phrase Option

The USE–DICT–EXPS option of the frame phrase forces all Data Dictionary validation and help strings to be compiled into the frame on the first reference of a field in the frame. Note that this is different from the normal behavior. Data dictionary validation is compiled in on the first reference of the field in an input statement. For this reason, programmatic VALIDATE or HELP options must be added to the DEFINE FRAME, FORM, or DEFINE BROWSE statements to override existing Data Dictionary expressions. (The first reference of a field in the frame is normally the reference in one of these statements, unless the field is added later.)

This option may add a lot of non-essential code to a frame, so its use should be restricted to where necessary. The option is intended to be used with frames that might have fields enabled with the *widget*:SENSITIVE = YES syntax. It is not necessary in any other case.

Using the Dictionary Expressions (–dictexps) Startup Parameter

The Dictionary Expressions (–dictexps) startup parameter is intended to be used as a temporary step to close possible validation holes in existing applications. It has the effect of adding a USE–DICT–EXPS option to every frame in the application, including those that are not used for input. While this parameter quickly closes all possible validation holes, it is a highly inefficient way of doing it.

For best long-term results, search for and replace *widget*:SENSITIVE = YES constructs, or use the USE–DICT–EXPS option on effected frames.

For more information on the –dictexps parameter, see the *Progress Startup Command and Parameter Reference*.

Using the Frame VALIDATE( ) Method

However, you can force Progress to perform all established variable and field validations at any time by executing the VALIDATE( ) method for the frame. The method returns TRUE if all validate checks in the frame succeed. It returns FALSE for the first validate check that fails in the order that data representation widgets are specified in the frame definition. A typical application of this method is in a trigger for an event that accepts the current updates in the frame. If the method fails, Progress gives input focus to the widget that failed the test after the trigger returns to the blocking WAIT–FOR statement.

NOTE: You can also apply the VALIDATE( ) method to a single data representation widget to perform any validate check for the underlying field or variable of that widget. If the test fails, focus returns to the specified widget.

The following procedure shows the VALIDATE( ) method applied to a frame that is used to update existing item records in the sports database. Enter an item number and press **RETURN**. The procedure displays and enables fields from the selected record, if it is available. To update the record in the database and enter a new item number, choose the bContinue button.

Note that once an item record is enabled for update, the procedure does not allow the record to be written back to the database unless all the relevant fields pass the specified validate checks. This is true even if no input fields have been entered and no data has changed from the original record.

p-frmval.p

```
DEFINE BUTTON bContinue LABEL "Continue".
DEFINE BUTTON bQuit LABEL "Quit".
DEFINE FRAME ItemFrame
    item.item-num
    item.item-name
        VALIDATE(item-name BEGINS "B",
                 "Must enter item-name starting with B")
    item.on-hand
        VALIDATE(on-hand > 0, "Must enter items on-hand > 0.")
    item.allocated
        VALIDATE(allocated <= on-hand,
                 "Allocated cannot be greater than on-hand items.")
    bContinue
    bQuit
WITH SIDE-LABELS.

ON RETURN OF item.item-num IN FRAME ItemFrame DO:
    FIND FIRST item
        WHERE item-num = INTEGER(item-num:SCREEN-VALUE) NO-ERROR.
    IF AVAILABLE(item) THEN DO:
        DISABLE item-num WITH FRAME ItemFrame.
        DISPLAY item-name on-hand allocated WITH FRAME ItemFrame.
        ENABLE ALL EXCEPT item-num WITH FRAME ItemFrame.
    END.
    ELSE DO:
        MESSAGE "Item not on file.  Enter another.".
        RETURN NO-APPLY.
    END.
END.
ON CHOOSE OF bContinue IN FRAME ItemFrame DO:
    IF NOT FRAME ItemFrame:VALIDATE() THEN
        RETURN NO-APPLY.
    ASSIGN item.item-name item.on-hand item.allocated.
    DISABLE ALL EXCEPT bQuit WITH FRAME ItemFrame.
    ENABLE item.item-num WITH FRAME ItemFrame.
END.

FIND FIRST item NO-ERROR.
DISPLAY
    item.item-num item.item-name item.on-hand item.allocated
    bContinue bQuit
WITH FRAME ItemFrame.
ENABLE item.item-num bQuit WITH FRAME ItemFrame.
WAIT-FOR CHOOSE OF bQuit IN FRAME ItemFrame.
```

19.13.4 Disabling Data Dictionary Validation on a Frame or Field

To disable all Data Dictionary validation for the fields in a frame, add the NO–VALIDATE option to the Frame phrase. NO–VALIDATE only disables Data Dictionary validation—it has no effect on programmatic validation.

To disable Data Dictionary validation for a field, use the VALIDATE option as shown in the code fragment below.

```
customer.credit-limit VALIDATE(TRUE, "")
```

20

Using Dynamic Widgets

Most applications do not require dynamic widgets other than windows, because widget definitions can be determined statically, at compile time. However, sometimes an application does not have all the information it needs to define a widget until run time. One example is an application that allows the user to add and remove buttons that duplicate menu functions. Another is an application that designs user interfaces for other applications, such as the Progress AppBuilder. All such applications require that you create and delete widget definitions at run time. For applications like these, you must use dynamic widgets.

This chapter describes the basic requirements and techniques for creating and managing dynamic widgets in a user interface, and how they differ from static widgets. These include techniques for managing dynamic widgets individually and as a group. For an introduction to defining static and dynamic widgets, see Chapter 16, "Widgets and Handles."

Some dynamic widgets require a more focused explanation than the general information described here. For information on dynamic menus and submenus, see Chapter 22, "Menus." For information on dynamic windows (all windows are dynamic except for the static default window), see Chapter 21, "Windows."

This chapter describes techniques for:

- Managing dynamic widgets
- Managing dynamic widget pools
- Creating and using dynamic queries
- Creating and using a dynamic browse

20.1 Managing Dynamic Widgets

Although you can create and delete a dynamic widget at run time, you must specify the attributes and environment of a dynamic widget much more explicitly than for a static widget. Progress provides attributes that allow you to specify for dynamic widgets at run time most of the options that you specify for static widgets at compile time. While Progress automatically sets read-only attributes for dynamic widgets just as it does for static widgets, Progress provides fewer defaults for the attributes that you can set. In addition, you must use different 4GL statements to work with dynamic widgets than you typically do with static widgets.

Note that dynamic widgets do not adhere to scoping rules. That is, they exist until you delete them or the session ends. Thus, if you create a dynamic button in the FOR EACH loop of a subprocedure, that button continues to exist after the subprocedure returns. However, note that any widget handle variables are locally or globally scoped as you define them. Also, while dynamic widgets are not scoped, trigger definitions are scoped. Thus, for dynamic widgets, define persistent triggers or put your trigger definitions in a persistent procedure.

20.1.1 Static Versus Dynamic Widget Management

Table 20–1 compares the major 4GL actions involving static and dynamic widgets and summarizes their differences.

Table 20–1: Static Versus Dynamic Widget Management *(1 of 2)*

4GL Widget Action	Accomplished for Static Widgets by . . .	Accomplished for Dynamic Widgets by . . .
Create	DEFINE *widget* statement, VIEW–AS phrase, FORM statement (for frames), and default scoping (for frames).	CREATE *widget* statement
Delete	N/A	DELETE WIDGET or DELETE WIDGET–POOL statement[1]
Reference	Definition name at compile time and run time; widget handle at run time	Widget handle at run time
View on Display	Frame I/O statements, including ENABLE, DISPLAY, INSERT, UPDATE, SET, or PROMPT–FOR; VIEW statement	VIEW statement or setting the VISIBLE attribute of each widget to TRUE[2,3]

Table 20–1: Static Versus Dynamic Widget Management *(2 of 2)*

4GL Widget Action	Accomplished for Static Widgets by . . .	Accomplished for Dynamic Widgets by . . .
Hide from Display	HIDE statement	HIDE statement or setting the VISIBLE attribute of each widget to FALSE[2]
Make Sensitive to Input	ENABLE, INSERT, UPDATE, SET, or PROMPT–FOR statement	Setting the SENSITIVE attribute of each widget to TRUE[2]
Make Insensitive to Input	DISABLE statement	Setting the SENSITIVE attribute of each widget to FALSE[2]
Block for Input	WAIT–FOR, INSERT, UPDATE, SET or PROMPT–FOR statement	WAIT–FOR statement
Move Data from Screen to Record Buffer	ASSIGN, SET, INSERT, or UPDATE statement applied to corresponding field or variable	Explicit assignment from the SCREEN–VALUE attribute of the widget to a specified field or variable[2]
Move Data from Record to Screen Buffer	DISPLAY, INSERT, or UPDATE statement applied to corresponding field or variable	Explicit assignment from a specified field, variable, or constant to the SCREEN–VALUE attribute of the widget[2]

[1] For more information on widget pools, see the "Managing Dynamic Widget Pools" section.

[2] You can also use these techniques with static widgets.

[3] The behavior of the VISIBLE attribute also depends on the setting of the HIDDEN attribute.

20.1.2 Setting Up Dynamic Field-level Widgets

When setting up dynamic field-level widgets, it helps to keep these points in mind:

- **Frame parenting** — You can place a dynamic field-level widget in either a static or dynamic frame (see the "Setting Up Dynamic Frames" section). Assign the frame handle to the FRAME attribute of the field-level widget. For a field-level widget, the PARENT attribute points to the field group, not the frame, of the widget. Progress automatically puts the widget in a field group when you set its frame, and also assigns the widget a default tab position if it is an input widget.

- **Widget positioning** — To space them out in a frame, you must explicitly position all dynamic widgets by setting the appropriate vertical (ROW or Y) and horizontal (COLUMN or X) placement attributes. However, Progress **does** assume the top-most and left-most position in the frame if you do not set a placement attribute for the widget.

- **Widget sizing** — You can size a widget, depending on its widget type and data type, using either the various height and width attributes or the FORMAT attribute. For example, a fill-in field with its FORMAT attribute set to "x(20)" is exactly the same size as one with its WIDTH–CHARS attribute set to 20.

- **Label handling** — Not every field-level widget gets its label by setting the LABEL attribute. You must provide separate text widgets as labels for some data representation widgets. For side labels, you set the SIDE–LABEL–HANDLE attribute of the specified data representation widget to the handle of the text widget containing the label in its SCREEN–VALUE attribute. For any other type of label, such as for vertical columns, you must create and manage the text widget completely separately. You must also position text widgets used as labels explicitly, even for side labels. Progress assigns no positioning information for dynamic side labels, as it does for button or toggle box labels.

- **Data handling** — Unlike static data representation widgets, dynamic widgets have no field or variable implicitly associated with them. You must explicitly assign data between a widget's SCREEN–VALUE attribute and the field or variable that you choose for data storage. Thus, you can use a single widget to represent several fields and variables at different times, depending on the widget and data type.

- **Data typing** — Some dynamic widgets support entry and display data types other than CHARACTER, and some, such as fill-in fields and combo boxes support the full range of Progress entry and display data types. Note that for dynamic widgets this support is for entry and display purposes only. The SCREEN–VALUE attribute always stores the data in character format, no matter what the widget data type. You must make all necessary data type conversions using the appropriate functions (STRING, INTEGER, etc.) when assigning data between the widget SCREEN–VALUE attribute and the field or variable that you choose for data storage.

The following procedure creates a dynamic fill-in with the INTEGER data type when you choose the Customer Number button. You can then enter a value according to the ">>>>9" format. The entered integer value is stored as a character string in the screen buffer. Pressing **RETURN** displays this value in the message area. Choosing the Delete Field button deletes the fill-in, removing it from the display:

p-dynfl1.p *(1 of 2)*

```
DEFINE BUTTON bCustNumber LABEL "Customer Number".
DEFINE BUTTON bDelete LABEL "Delete Field".
DEFINE VARIABLE fCustHandle AS WIDGET-HANDLE.
DEFINE VARIABLE lCustHandle AS WIDGET-HANDLE.
DEFINE FRAME CustFrame
  SKIP(3)
  SPACE (1) bCustNumber bDelete
  WITH SIZE 40 BY 5 SIDE-LABELS.

ON CHOOSE OF bCustNumber IN FRAME CustFrame
DO:
  IF fCustHandle <> ? THEN
  DO:
    MESSAGE bCustNumber:LABEL "field already exists.".
    RETURN.
  END.
  CREATE TEXT lCustHandle
    ASSIGN
    FRAME = FRAME CustFrame:HANDLE
    DATA-TYPE = "CHARACTER"
    FORMAT = "x(16)"
    SCREEN-VALUE = "Customer Number:"
    ROW = 2
    COLUMN = 2
  .
  CREATE FILL-IN fCustHandle
    ASSIGN
      FRAME = FRAME CustFrame:HANDLE
      DATA-TYPE = "INTEGER"
      FORMAT = ">>>>9"
      SIDE-LABEL-HANDLE = lCustHandle
      ROW = 2
      COLUMN = lCustHandle:COLUMN + lCustHandle:WIDTH-CHARS + 1
      SENSITIVE = TRUE
      VISIBLE = TRUE
    TRIGGERS:
      ON RETURN PERSISTENT RUN SetFieldTrig.
    END TRIGGERS
  .
END.
```

p-dynfl1.p *(2 of 2)*

```
ON CHOOSE OF bDelete IN FRAME CustFrame
DO:
  IF fCustHandle <> ? THEN
  DO:
    DELETE WIDGET fCustHandle.
    fCustHandle = ?.
    DELETE WIDGET lCustHandle.
  END.
END.

ENABLE ALL WITH FRAME CustFrame.
WAIT-FOR GO OF FRAME CustFrame.

PROCEDURE SetFieldTrig:
  MESSAGE "You entered" lCustHandle:SCREEN-VALUE
          fCustHandle:SCREEN-VALUE.
END PROCEDURE.
```

20.1.3 Setting Up Dynamic Frames

Progress allows you to create dynamic one-down frames. You cannot create dynamic down frames. You can populate a dynamic frame only with dynamic field-level widgets and you must explicitly specify their position within that frame. When Progress builds a static frame and populates it with static widgets, it creates an intelligent default size for the frame and intelligent default placement for the widgets. Progress does not do this with dynamic frames. You must explicitly define frame size and widget placement.

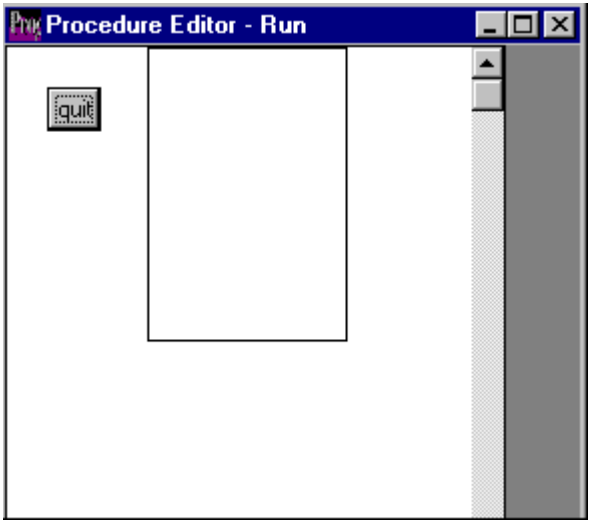
The following code creates a dynamic frame, then populates it with an editor widget and a quit button:

p-dynfrm.p

```
DEFINE VARIABLE ed AS WIDGET-HANDLE.
DEFINE VARIABLE frame1 AS WIDGET-HANDLE.
DEFINE VARIABLE button1 AS WIDGET-HANDLE.

CREATE FRAME frame1
ASSIGN
        WIDTH-CHARS = 50
        HEIGHT-CHARS = 28
        SENSITIVE = YES.
VIEW frame1.

CREATE BUTTON button1
ASSIGN
        X = 20
        Y = 20
        LABEL = "quit"
        FRAME = frame1
        SENSITIVE = YES
        TRIGGERS:
            ON CHOOSE STOP.
        END TRIGGERS.
VIEW button1.

CREATE EDITOR ed
ASSIGN
        WIDTH-CHARS = 20
        HEIGHT-CHARS = 7
        X = 70
        Y = 0
        FRAME = frame1
        SENSITIVE = YES.
VIEW ed.
WAIT-FOR GO OF frame1.
```

By default, Progress places the dynamic frame in the current window. To place the frame in a different window, set the PARENT attribute to the widget handle of that window.

This is the output of `p-dynfrm.p`:



20.2 Managing Dynamic Widget Pools

Every dynamic widget you create is assigned to a *widget pool*. A widget pool is a group of widgets that are scoped together and can be deleted as a group.

Progress creates a single unnamed widget pool for each client session. The session pool is initially the default pool for all dynamic widgets created during the session. It is automatically deleted when the session ends.

You can also create your own named or unnamed widget pools with the CREATE WIDGET–POOL statement:

SYNTAX

```
CREATE WIDGET-POOL [ pool-name [ PERSISTENT ] ]
  [ NO-ERROR ]
```

You can delete a widget pool with the DELETE WIDGET–POOL statement:

SYNTAX

```
DELETE WIDGET-POOL [ pool-name ] [ NO-ERROR ]
```

In general, when you create a dynamic widget, Progress assigns it to the most recently created unnamed widget pool by default. You can explicitly assign a dynamic widget only to a named widget pool. When you delete a widget pool, all dynamic widgets assigned to that pool are deleted as well.

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

20.2.1 Named Widget Pools

Progress supports two types of named widget pools: persistent and nonpersistent. A *persistent widget pool* remains allocated until it is explicitly deleted or the session ends. A *nonpersistent widget pool* remains allocated until it is explicitly deleted or the procedure block that creates it goes out of scope. When execution of a procedure or trigger ends or goes out of scope, Progress automatically deletes any nonpersistent widget pool defined in that routine.

NOTE: A persistent procedure goes out of scope only when it is explicitly deleted. Thus, non-persistent widget pools created in it can persist as long as the procedure persists.

When you create a dynamic widget, you can assign it to a named pool by using the IN WIDGET–POOL option.

The following procedure creates a named widget pool and creates a series of buttons in it:

p-dybut2.p

```
DEFINE VARIABLE num-buts  AS INTEGER.
DEFINE VARIABLE temp-hand AS WIDGET-HANDLE.

DEFINE FRAME butt-frame
   WITH WIDTH 60 CENTERED TITLE "Sales Representatives".

FORM
   salesrep
   WITH FRAME rep-frame.

CREATE WIDGET-POOL "rep-buttons".

num-buts = 0.
FOR EACH salesrep:
   CREATE BUTTON temp-hand IN WIDGET-POOL "rep-buttons"
         ASSIGN LABEL = salesrep.sales-rep
                FRAME = FRAME butt-frame:HANDLE
                ROW = TRUNC(num-buts / 3, 0) + 1
                COLUMN = ((num-buts MOD 3) * 20) + 1
                SENSITIVE = TRUE
         TRIGGERS:
            ON CHOOSE
               DO:
                 FIND salesrep WHERE salesrep.sales-rep = SELF:LABEL.
                 DISPLAY salesrep WITH FRAME rep-frame.
               END.
         END TRIGGERS.
   num-buts = num-buts + 1.
END.

FRAME butt-frame:HEIGHT-CHARS = (num-buts / 3) + 2.

VIEW FRAME butt-frame.

WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.

DELETE WIDGET-POOL "rep-buttons".
```

The widget pool in this example is nonpersistent. Therefore, if you omit the DELETE WIDGET–POOL statement at the end of the procedure, the widget pool is automatically deleted when the procedure ends.

Because all the widgets in p-dybut2.p are created in the main procedure and are deleted at the end of the procedure, the widget pool does not need to be persistent. However, the following procedure requires a persistent widget pool. The following procedure, p-dybut3.p, accepts a month and year and then creates a button for each order entered during that month:

p-dybut3.p *(1 of 2)*

```
DEFINE VARIABLE report-month AS INTEGER LABEL "Month" FORMAT ">9".
DEFINE VARIABLE report-year  AS INTEGER LABEL "Year" FORMAT "9999".
DEFINE VARIABLE temp-hand    AS WIDGET-HANDLE.

FORM
  report-month report-year
  WITH FRAME prompt-frame SIDE-LABELS.

DEFINE FRAME butt-frame WITH WIDTH 80.

ON GO OF FRAME prompt-frame
   DO:
       IF report-month:MODIFIED or report-year:MODIFIED
       THEN DO:
          ASSIGN report-month report-year.
          DELETE WIDGET-POOL "order-pool".
          RUN make-buttons.
       END.
   END.

ASSIGN report-month = MONTH(TODAY)
       report-year = YEAR(TODAY).

DISPLAY report-month report-year WITH FRAME prompt-frame.

RUN make-buttons.

ENABLE report-month report-year WITH FRAME prompt-frame.

DO ON ERROR UNDO, LEAVE ON ENDKEY UNDO, LEAVE ON STOP UNDO, LEAVE:
   WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
END.

DELETE WIDGET-POOL "order-pool".
```

p-dybut3.p *(2 of 2)*

```
PROCEDURE make-buttons:
DEFINE VARIABLE num-buts  AS INTEGER.

   CREATE WIDGET-POOL "order-pool" PERSISTENT.
   num-buts = 0.
   FRAME butt-frame:HIDDEN = TRUE.
   FOR EACH order WHERE MONTH(order-date) = report-month AND
                        YEAR(order-date) = report-year:
      FRAME butt-frame:HEIGHT-CHARS = TRUNC(num-buts / 5, 0) + 3.
      CREATE BUTTON temp-hand IN WIDGET-POOL "order-pool"
              ASSIGN LABEL = STRING(order-num) + "/" + STRING(cust-num)
                     FRAME = FRAME butt-frame:HANDLE
                     ROW = TRUNC(num-buts / 5, 0) + 1
                     COLUMN = ((num-buts MOD 5) * 15) + 1
                     SENSITIVE = TRUE
              TRIGGERS:
                 ON CHOOSE
                    PERSISTENT RUN disp-order.
              END TRIGGERS.
      num-buts = num-buts + 1.
   END.

   FRAME butt-frame:HIDDEN = FALSE.
   IF num-buts = 0
   THEN MESSAGE "No orders found for" STRING(report-month) + "/" +
            STRING(report-year) VIEW-AS ALERT-BOX INFORMATION BUTTONS OK.
END PROCEDURE.

PROCEDURE disp-order:

FORM
   order
   WITH FRAME order-frame SIDE-LABELS.

   FIND order WHERE order-num =
          INTEGER(SUBSTRING(SELF:LABEL, 1, INDEX(SELF:LABEL, "/") - 1)).
   CREATE WINDOW temp-hand IN WIDGET-POOL "order-pool"
         ASSIGN TITLE = "Order Detail"
                VIRTUAL-HEIGHT-CHARS = FRAME order-frame:HEIGHT-CHARS.
   CURRENT-WINDOW = temp-hand.
   DISPLAY order WITH FRAME order-frame.
   CURRENT-WINDOW = DEFAULT-WINDOW.
END PROCEDURE.
```

In this example, the widget pool is created within the internal procedure make–buttons. Because the buttons must be available outside the internal procedure, the widget pool must be persistent; otherwise the pool would be deleted at the end of the internal procedure. When you change the month or year value, the GO trigger deletes the widget pool (and therefore deletes all the buttons as well). It then runs make–buttons to create a new widget pool and a new set of buttons.

Note that the internal procedure, disp-order, creates a window in the persistent widget pool. When you change the month or year, any windows so created are deleted along with the buttons.

When you use a persistent widget pool you must make sure the pool is deleted. In `p-dybut3.p` the DO group surrounding the WAIT–FOR statement ensures that control will pass to the DELETE WIDGET–POOL statement no matter how you exit the procedure.

20.2.2 Unnamed Widget Pools

Any unnamed widget pool you create becomes the default pool until it is deleted or until you create another unnamed widget pool. Any unnamed pools you create are scoped to the routine in which they are created. Also, any subprocedure or trigger inherits, as its default pool, the most recent unnamed widget pool created in a calling procedure until (and unless) it creates an unnamed widget pool of its own. When execution of a routine ends, or it goes out of scope, any unnamed pools created in the routine (and therefore, all widgets in them) are automatically deleted.

NOTE: A persistent procedure goes out of scope only when it is explicitly deleted. Thus, unnamed widget pools created in it can persist as long as the procedure persists.

If you omit the IN WIDGET–POOL option in the statement that creates the widget, the widget is automatically assigned to the most recently created unnamed widget pool that has not been deleted.

You can delete the current default widget pool by using the DELETE WIDGET–POOL statement without a pool name. You can delete the default widget pool only within the routine that defined it. For example, if you define an unnamed widget pool in a procedure, you cannot delete it in a subprocedure or trigger called by that procedure. If you attempt to do so, Progress ignores the DELETE WIDGET–POOL statement.

You cannot specify the PERSISTENT option when you create an unnamed widget pool. All unnamed pools, except the session pool and pools created in persistent procedures, are nonpersistent.

You might use an unnamed widget pool, for example, to ensure that all widgets created in the default pool in a run procedure are deleted when that procedure returns or goes out of scope. For example, the following procedure calls `p-dybuts.p` (see Chapter 16, "Widgets and Handles"):

p-unname.p

```
CREATE WIDGET-POOL.
RUN p-dybuts.p.
DELETE WIDGET-POOL.
```

In this example, the CREATE WIDGET–POOL statement creates a new default widget pool. Any widgets created in the default pool within `p-dybuts.p` are placed in this pool. After `p-dybuts.p` completes, the DELETE WIDGET–POOL statement deletes that widget pool.

On the other hand, in a persistent procedure, you can use an unnamed widget pool to ensure that its dynamic widgets are **not** deleted after it returns from instantiating its persistent context. Otherwise, if the calling procedure deletes the widget pool that was current when it created the persistent procedure, any dynamic widgets created for the persistent context are deleted as well.

Thus, if you ran `p-dybuts.p` persistently in `p-unname.p`, creating an unnamed widget pool in `p-dybuts.p` prevents `p-unname.p` from deleting the dynamic widgets created in the persistent context of `p-dybuts.p`. For a working example, see the section on multiple-window applications with persistent procedures in Chapter 21, "Windows."

20.3 Creating and Using Dynamic Queries

The static query is one of the basic methods of fetching database records. For more information on static queries, see "Defining a Set of Records to Fetch" in Chapter 9, "Database Access." This section describes programming and using the dynamic query—a query whose elements are resolved at runtime.

Progress supports creating and using dynamic queries by providing query, buffer, and buffer-field objects.

20.3.1 Creating a Query Object

The query object corresponds to an underlying Progress query, which can be static or dynamic. A static query is one you define at compile time using the DEFINE QUERY statement. A dynamic query is one you create at run time using the QUERY option of the CREATE Widget statement.

The query object, like other Progress objects, is represented by a variable of type HANDLE. You can use a HANDLE variable to acquire a new query object for a dynamically-created query, as the following code fragment demonstrates:

```
DEFINE VARIABLE my-query-handle AS HANDLE.
CREATE QUERY my-query-handle.
...
```

You can also use a HANDLE variable to acquire the query object of an existing statically-created query, as the following code fragment demonstrates:

```
DEFINE VARIABLE my-query-handle AS HANDLE.
DEFINE QUERY q1 FOR customer.
my-query-handle = QUERY q1:HANDLE.
...
```

20.3.2 Using a Query Object

Using a dynamic query requires using several methods in addition to the CREATE QUERY statement. The CREATE QUERY statement does not define what database table or buffer you wish to access or what subset of records you want to fetch. The SET–BUFFERS( ) method and the QUERY–PREPARE( ) method specify this information and the QUERY–OPEN( ) method opens the query.

The following code fragment depicts one way of creating and using a dynamic query:

```
DEFINE VARIABLE qh8 AS WIDGET-HANDLE.

CREATE QUERY qh8.
qh8:SET-BUFFERS(BUFFER invoice:HANDLE).
qh8:QUERY-PREPARE("FOR EACH invoice BY invoice-date").
qh8:QUERY-OPEN().

REPEAT WITH FRAME fr1:
    qh8:GET-NEXT.
    IF qh8:QUERY-OFF-END THEN LEAVE.
    . . .
END.
.
.
.
DELETE OBJECT qh8.
```

For a list of the attributes and methods of the query object, see the Query Object Handle reference entry in the *Progress Language Reference*. For a complete description of the attributes and methods, see the “Attributes and Methods Reference” chapter of the same book.

20.3.3 The Buffer Object

The buffer object corresponds to an underlying Progress buffer, which can be static or dynamic. A static buffer is one you define at compile time by using the DEFINE BUFFER statement, or by implicitly referencing a table in a 4GL construct such as `customer.cust-num`. A dynamic buffer is one you create at run time using the BUFFER option of the CREATE Widget statement.

The buffer object, like the query object and other Progress objects, is represented by a variable of type HANDLE. You can use a HANDLE variable to acquire a new buffer object as the following code fragment demonstrates:

```
DEFINE VARIABLE my-buffer-handle AS HANDLE.
CREATE BUFFER my-buffer-handle FOR TABLE "mycusttab".
...
```

You can also use a HANDLE variable to acquire a buffer object for an existing statically-created buffer, as the following code fragment demonstrates:

```
DEFINE VARIABLE my-buffer-handle AS HANDLE.
my-buffer-handle = BUFFER customer:HANDLE.
...
```

With the buffer object you can:

- Create a dynamic buffer for a dynamic query
- Manipulate an existing static buffer
- Write generic 4GL code without knowing at compile time the precise names of the databases, buffers, and fields
- Determine schema properties, such as the names and values of individual buffer-fields, by using dynamic buffer-fields and their methods

The following program example demonstrates the use of buffer objects in a dynamic query. Note the use of the dynamic predicate (WHERE and BY clauses) in the query:

p-qryob1.p

```
/* p-qryob1.p - demonstrates buffer and query objects */

DEFINE VARIABLE qry1 AS HANDLE.
DEFINE VARIABLE wherev AS CHARACTER INITIAL "WHERE cust-num < 10".
DEFINE VARIABLE sortv AS CHARACTER INITIAL "BY sales-rep".
DEFINE VARIABLE bval AS LOGICAL.
DEFINE VARIABLE i AS INTEGER.
DEFINE VARIABLE bh AS WIDGET-HANDLE EXTENT 4.

CREATE QUERY qry1.

REPEAT:
    UPDATE wherev FORMAT "x(70)"
      LABEL "Enter WHERE and BY information."  SKIP
           sortv FORMAT "x(70)" NO-LABEL.

    qry1:SET-BUFFERS(BUFFER customer:HANDLE, BUFFER order:HANDLE).
    bval = qry1:QUERY-PREPARE("FOR EACH customer " + wherev +
           ", EACH order OF customer " + sortv).

    REPEAT i = 1 TO qry1:NUM-BUFFERS:
        bh[i] =  qry1:GET-BUFFER-HANDLE(i).
        DISPLAY bh[i]:NAME.  /* display the buffer names */
    END.

    IF (bval = FALSE) THEN NEXT.
    qry1:QUERY-OPEN.

    REPEAT WITH FRAME fr1 ROW 8:
        qry1:GET-NEXT.
        IF (qry1:QUERY-OFF-END) THEN LEAVE.
        DISPLAY cust.cust-num cust.name cust.sales-rep
                  cust.state order.order-num.
    END.

    qry1:QUERY-CLOSE.
END.

DELETE OBJECT qry1.
```

When the program pauses, press the GO key (usually F2) to continue.

Notes on Dynamic Buffers

- Since the dynamic buffer lets you manipulate database records at run time without the benefit of the compiler's security checking, Progress ensures that when you use a dynamic buffer, you have the necessary read, write, create, and delete permissions. In addition, if the table corresponding to a dynamic buffer specifies delete validation, Progress does not let you delete instances of the table dynamically.

- When you have finished using the buffer object, it is important to apply the BUFFER–RELEASE method, since Progress does not automatically release the object until the underlying buffer object is deleted.

- A dynamic buffer has the same scope as the widget–pool in which it was created. This means that Progress automatically deletes a dynamic buffer object only when it deletes its widget–pool. To delete a dynamic buffer object manually, you must use the DELETE OBJECT statement.

- If you place the phrase BUFFER *name* anywhere in a procedure file, where *name* represents the name of a table, not necessarily the name of a buffer you defined using the DEFINE statement, Progress scopes *name* as it would a free reference.

- For a list of the attributes and methods of the buffer object, see the Buffer Object Handle reference entry in the *Progress Language Reference*. For a complete description of the attributes and methods, see the "Attributes and Methods Reference" chapter of the same book.

20.3.4 The Buffer-field Object

The buffer-field object represents a field of a buffer. You do not create buffer-field objects. Progress creates them automatically when you reference any field of a buffer object.

Using buffer-field objects lets you examine and modify the fields of a buffer and examine the schema properties of the fields.

When your code accesses buffer-fields, Progress checks security permissions at run time.

You can use HANDLE variables to retrieve the handle of a buffer-field object, as the following code fragments demonstrate:

```
DEFINE VARIABLE my-buffer-handle as HANDLE.
DEFINE VARIABLE my-buffer-field-handle as HANDLE.
...
my-buffer-handle = BUFFER custx:HANDLE.
my-buffer-field-handle = my-buffer-handle:BUFFER-FIELD(3).
```

```
DEFINE VARIABLE my-buffer-field-handle as HANDLE.
...
my-buffer-field-handle = BUFFER customer:BUFFER-FIELD("cust-num").
```

The following program example demonstrates using buffer–field objects and their BUFFER–VALUE, NAME and EXTENT attributes. Note that buffer–field objects can be used without using query or buffer objects:

p-qryob2.p

```
/* p-qryob2.p - demonstrates buffer and buffer-field objects */

DEFINE VARIABLE i AS INTEGER.
DEFINE VARIABLE arrayi AS INTEGER.
DEFINE VARIABLE bh AS WIDGET-HANDLE.
DEFINE VARIABLE fh AS WIDGET-HANDLE EXTENT 100.
DEFINE VARIABLE fhc AS HANDLE.
DEFINE BUFFER custx FOR customer.

FIND FIRST custx.
bh = BUFFER custx:HANDLE.
MESSAGE "Value of City field is: " city:BUFFER-VALUE IN BUFFER custx.
fhc = bh:BUFFER-FIELD("city").
MESSAGE "Name of City field is: " fhc:NAME.
DISPLAY bh:NAME.

REPEAT i = 1 TO bh:NUM-FIELDS TRANSACTION:
    fh[i] = bh:BUFFER-FIELD(i).
    IF i = 5 THEN fh[i]:BUFFER-VALUE = "new addr2".
    DISPLAY fh[i]:NAME fh[i]:EXTENT.
    arrayi = 0.
    IF fh[i]:EXTENT > 0 THEN arrayi = 2.
    DISPLAY STRING(fh[i]:BUFFER-VALUE(arrayi)) FORMAT "x(20)".
END.
```

Notes on Dynamic Buffer–Fields

- If you assign the UNKNOWN value (?) to the BUFFER–VALUE attribute of a buffer–field object, the value of BUFFER–VALUE becomes the UNKNOWN value. Similarly, if you assign the empty string to the BUFFER–VALUE attribute of a buffer-field object, the value of BUFFER–VALUE becomes the UNKNOWN value—unless the buffer-field object is CHARACTER, in which case the value becomes the empty string. Conversely, if you assign the UNKNOWN value to the BUFFER–VALUE attribute of a buffer-field object, the value of its STRING–VALUE attribute becomes the empty string.

- For a list of the attributes and methods of the buffer-field object, see the Buffer-field Object Handle reference entry in the *Progress Language Reference*. For a complete description of the attributes and methods, see the "Attributes and Methods Reference" chapter of the same book.

20.3.5 Error Handling

Each method of the query object, buffer object, and buffer-field object that can have errors returns a value of type LOGICAL that you can test. If an error occurs, the method returns FALSE, but does not automatically raise the error condition. Progress Software Corporation (PSC) recommends that you test the return value of methods like QUERY–PREPARE, where an error causes subsequent OPENs, GET–NEXTs, etc. to fail.

The following code fragments demonstrates checking the return value after the QUERY–PREPARE method:

```
DEFINE VARIABLE retval AS LOGICAL.
retval = q:QUERY-PREPARE("FOR EACH customer...").
IF retval = FALSE THEN.../* error exit */
```

The following code fragment demonstrates checking the return value after the QUERY–OPEN method:

```
DEFINE VARIABLE retval AS LOGICAL.
retval = q:QUERY-OPEN().
IF retval = FALSE, THEN.../* error exit */
```

20.3.6 Dynamic Query Code Example

The following program example demonstrates how you can change the predicate of a dynamic query using radio-set input and populate a dynamic browse (covered in the next section) with varying results-lists:

p-fnlqry.p *(1 of 3)*

```
CURRENT-WINDOW:ROW = 1.
CURRENT-WINDOW:HEIGHT-CHARS = 18.
CURRENT-WINDOW:WIDTH-CHARS = 132.
CURRENT-WINDOW:TITLE = "Dynamic Browse With Dynamic Query".
CURRENT-WINDOW:KEEP-FRAME-Z-ORDER = TRUE.

/* Test controls */
DEFINE BUTTON Make       LABEL "CreateDynBrowse".
DEFINE BUTTON AddCols    LABEL "Add Columns".
DEFINE BUTTON btn-delete LABEL "DeleteDynBrowse".
DEFINE BUTTON btn-quit   LABEL "&Quit"  AUTO-ENDKEY.

DEFINE VARIABLE query-criteria AS CHARACTER
    VIEW-AS RADIO-SET VERTICAL
    RADIO-BUTTONS
      "All customers", "FOR EACH customer NO-LOCK", "USA customers",
     "FOR EACH customer WHERE customer.country EQ 'USA' NO-LOCK BY
customer.state ",
     "Non-USA customers",
     "FOR EACH customer WHERE customer.country NE 'USA' NO-LOCK BY
customer.country "
     SIZE 25 BY 3 TOOLTIP "Choose the customers you want to see"
      INITIAL "FOR EACH customer NO-LOCK" NO-UNDO.

/* Widget and other Handles */
DEFINE VARIABLE Browse-Hndl AS WIDGET-HANDLE.
DEFINE VARIABLE bufFieldHandle AS WIDGET-HANDLE.
DEFINE VARIABLE qh AS WIDGET-HANDLE.
DEFINE VARIABLE bh AS WIDGET-HANDLE.
bh = BUFFER customer:HANDLE.

/* Create Query  */
CREATE QUERY qh.
qh:SET-BUFFERS(bh).

DEFINE FRAME F1
    skip(13)
    make  AddCols  btn-delete btn-quit query-criteria
    WITH SIZE 132 BY 18 THREE-D NO-LABELS.
```

p-fnlqry.p *(2 of 3)*

```
ON CHOOSE OF make DO:        /*  LABEL "CreateDynBrowse". */
    CREATE BROWSE Browse-Hndl
      ASSIGN
        FRAME = FRAME F1:HANDLE
        QUERY = qh
        TITLE = " "
        X = 2
        Y = 2
        WIDTH = 130
        DOWN = 12
        VISIBLE = TRUE
        SENSITIVE = TRUE
        READ-ONLY = NO
        COLUMN-SCROLLING = TRUE
        SEPARATORS = YES.
END.
ON CHOOSE OF AddCols DO:   /*  LABEL "Add Columns"  */
    ASSIGN query-criteria.
    qh:QUERY-PREPARE(query-criteria).
    qh:QUERY-OPEN.
     Browse-Hndl:ADD-LIKE-COLUMN(bh:BUFFER-FIELD(1),1).
    Browse-Hndl:ADD-LIKE-COLUMN(bh:BUFFER-FIELD(3),2).
    Browse-Hndl:ADD-LIKE-COLUMN(bh:BUFFER-FIELD("city"),3).

    CASE query-criteria:
        WHEN  "for each customer NO-LOCK" THEN
        DO:
            Browse-Hndl:ADD-LIKE-COLUMN(bh:BUFFER-FIELD("state")).
            Browse-Hndl:ADD-LIKE-COLUMN(bh:BUFFER-FIELD("country")).
            Browse-Hndl:ADD-LIKE-COLUMN(bh:BUFFER-FIELD("postal-code")).
            Browse-Hndl:TITLE = "All Customers".
        END.
        WHEN "FOR EACH customer WHERE customer.country EQ 'USA' NO-LOCK BY
customer.state " THEN
        DO:
            Browse-Hndl:ADD-LIKE-COLUMN(bh:BUFFER-FIELD("state")).
            Browse-Hndl:ADD-LIKE-COLUMN(bh:BUFFER-FIELD("postal-code")).
            Browse-Hndl:TITLE = "All US Customers".
        END.
        WHEN "FOR EACH customer WHERE customer.country NE 'USA' NO-LOCK BY
customer.country " THEN
        DO:
            Browse-Hndl:ADD-LIKE-COLUMN(bh:BUFFER-FIELD("country")).
            Browse-Hndl:ADD-LIKE-COLUMN(bh:BUFFER-FIELD("postal-code")).
            Browse-Hndl:TITLE = "All non-US Customers".
        END.
    END CASE.
```

p-fnlqry.p *(3 of 3)*

```
    bufFieldHandle = bh:BUFFER-FIELD("balance").
    Browse-Hndl:ADD-LIKE-COLUMN(bufFieldHandle).
    AddCols:SENSITIVE = FALSE.
    query-criteria:SENSITIVE = FALSE.
    make:SENSITIVE = FALSE.
END.

ON CHOOSE OF btn-delete  /*  LABEL "DeleteDynBrowse". */
DO:
    DELETE WIDGET Browse-Hndl.
    AddCols:SENSITIVE = TRUE.
    query-criteria:SENSITIVE = TRUE.
    make:SENSITIVE = TRUE.
END.

ON CHOOSE OF btn-quit
DO:
    APPLY "window-close" TO CURRENT-WINDOW.
END.

ENABLE ALL WITH FRAME F1.
WAIT-FOR CLOSE OF CURRENT-WINDOW.
```

20.4 Creating and Using a Dynamic Browse

The design and programming of the static browse was described in Chapter 10, "Using the Browse Widget." This section describes programming and using a dynamic browse, that is, a browse whose elements are resolved at runtime.

The dynamic browse can be either a read-only or an updateable browse. If the dynamic browse is updateable, it is a NO–ASSIGN browse and you must do all database updating manually. The dynamic browse can be used with either a static or a dynamic query.

20.4.1 Creating the Dynamic Browse

A dynamic browse is one you create at runtime using the BROWSE option of the CREATE Widget statement. Within the CREATE BROWSE statement, you can assign attributes to the browse and specify its associated query.

The following code fragment depicts creating a dynamic browse:

```
DEFINE VARIABLE br1 AS HANDLE.
. . .
/* define query and define frame here */

CREATE BROWSE br1
    ASSIGN FRAME = FRAME f1:HANDLE
           X = 2
           Y = 2
           WIDTH = 80
           DOWN = 10
           QUERY = QUERY q1:HANDLE
           TITLE = "Dynamic Browse"
           SENSITIVE = TRUE
           VISIBLE = TRUE
           READ-ONLY = FALSE
           SEPARATORS = TRUE.
. . .
```

20.4.2 Creating Dynamic Browse Columns

The dynamic browse that you create with the CREATE BROWSE statement is an empty browse. You must add the browse columns using the following methods:

- ADD–COLUMNS–FROM( )
- ADD–LIKE–COLUMN( )
- ADD–CALC–COLUMN( )

ADD–COLUMNS–FROM( ) Method

The ADD–COLUMNS–FROM( ) method creates a browse column for every field in the specified table or buffer except for any fields specified in an except-list. The specified table or buffer must be in the associated query. Use this method when you want the browse to contain all or most of the fields in a table or buffer.

The following program creates a browse which displays all the fields in the customer table except for terms and comments:

p-dybrw1.p

```
DEFINE VARIABLE b1 AS HANDLE.
DEFINE QUERY q1 FOR customer EXCEPT (terms comments) SCROLLING.
OPEN QUERY q1 FOR EACH customer WHERE customer.cust-num < 10.

DEFINE FRAME f1
    WITH SIZE 82 BY 29.

CREATE BROWSE b1
    ASSIGN
      FRAME = FRAME f1:HANDLE
      X = 2
      Y = 2
      WIDTH = 80
      DOWN = 10
      QUERY = QUERY q1:HANDLE
      TITLE = "Dynamic Browse with static query"
      SENSITIVE = TRUE
      VISIBLE = TRUE
      READ-ONLY = FALSE
      SEPARATORS = TRUE.

b1:ADD-COLUMNS-FROM("customer","terms,comments").

ENABLE ALL WITH FRAME f1.
WAIT-FOR CLOSE OF CURRENT-WINDOW.
```

ADD–LIKE–COLUMN( ) Method

The ADD–LIKE–COLUMN method creates a single browse column from the specified field name or buffer field handle. You can also define the position of this column within the browse. The specified field must be a field in one of the buffers of the associated query. Use this method when you want the browse to contain only a few fields in a table or buffer.

The following program creates a browse containing only four fields from the customer table:

p-dybrw2.p

```
DEFINE VARIABLE b1 AS HANDLE.
DEFINE QUERY q1 FOR customer
        FIELDS (cust-num name phone contact) SCROLLING.
OPEN QUERY q1 FOR EACH customer WHERE customer.cust-num < 10.

DEFINE FRAME f1
    WITH SIZE 82 BY 29.

CREATE BROWSE b1
    ASSIGN
      FRAME = FRAME f1:HANDLE
      X = 2
      Y = 2
      WIDTH = 80
      DOWN = 10
      QUERY = QUERY q1:HANDLE
      TITLE = "Dynamic Browse with static query"
      SENSITIVE = TRUE
      VISIBLE = TRUE
      READ-ONLY = FALSE
      SEPARATORS = TRUE.

b1:ADD-LIKE-COLUMN("customer.cust-num").
b1:ADD-LIKE-COLUMN("customer.name").
b1:ADD-LIKE-COLUMN("customer.phone").
b1:ADD-LIKE-COLUMN("customer.contact").

ENABLE ALL WITH FRAME f1.
WAIT-FOR CLOSE OF CURRENT-WINDOW.
```

For other examples of the ADD–LIKE–COLUMN( ) method, see the program, p-fnlqry.p, in section “Dynamic Query Code Example.”

ADD–CALC–COLUMN( ) Method

The ADD–CALC–COLUMN( ) method creates a single browse column with the specified properties rather than from a table or buffer field. This is typically used as a placeholder column for a calculated value.

The following program creates a browse containing four fields from the customer table plus a fifth field containing the calculated current credit limit:

p-dybrw3.p

```
DEFINE VARIABLE b1 AS HANDLE.
DEFINE VARIABLE calch AS HANDLE.

DEFINE QUERY q1 FOR customer
     FIELDS (cust-num name phone contact credit-limit balance) SCROLLING.
OPEN QUERY q1 FOR EACH customer WHERE customer.cust-num < 30.

DEFINE FRAME f1
    WITH SIZE 78 BY 18.

CREATE BROWSE b1
    ASSIGN
      FRAME = FRAME f1:HANDLE
      X = 2
      Y = 2
      WIDTH = 76
      DOWN = 15
      QUERY = QUERY q1:HANDLE
      TITLE = "Dynamic Browse with static query"
      SENSITIVE = TRUE
      VISIBLE = FALSE
      READ-ONLY = FALSE
      SEPARATORS = TRUE.

ON ROW-DISPLAY OF b1 DO:
    IF VALID-HANDLE(calch) THEN
      calch:SCREEN-VALUE=STRING(customer.credit-limit - customer.balance).
END.

b1:ADD-LIKE-COLUMN("customer.cust-num").
b1:ADD-LIKE-COLUMN("customer.name").
b1:ADD-LIKE-COLUMN("customer.phone").
b1:ADD-LIKE-COLUMN("customer.contact").
calch=b1:ADD-CALC-COLUMN("DECIMAL","->,>>>,>>9.99","0","CurrentLimit").

b1:VISIBLE = TRUE.

ENABLE ALL WITH FRAME f1.
WAIT-FOR CLOSE OF CURRENT-WINDOW.
```

Notes on Dynamic Browse Columns

- You cannot specify the CAN–FIND function in the validation expression for a dynamic browse-column. Since Progress compiles validation expressions at runtime for dynamic browse-columns, CAN–FIND produces an error and the validation will not run for that column. You can still use the CAN–FIND function for a static-browse column.
- You can set the VISIBLE attribute for a browse-column as well as a browse.
- You can change the SCREEN–VALUE attribute for a browse-column within the browse's ROW–DISPLAY trigger as well as outside the trigger.
- You must define the ROW–DISPLAY trigger where the value of the calculated column is set prior to using the ADD–CALC–COLUMN( ) method so that the trigger is already set before the column is added. This is to ensure that the initial viewport of the calculated column is populated.

20.4.3 Dynamic Enhancements to the Static Browse

You can now dynamically create some or all of the columns for a static browse by using the ADD–COLUMNS–FROM( ), ADD–LIKE–COLUMN( ) and ADD–CALC–COLUMN( ) methods. If you use the ADD–COLUMNS–FROM( ) or ADD–LIKE–COLUMNS( ) methods on a static browse, the browse becomes a NO–ASSIGN browse and you will have to manage the database updates manually. See Chapter 10, "Using the Browse Widget," for more information on the static browse.

In addition, you can now change the query of a static browse even if the underlying fields are not the same as those of the original query. However, if the new underlying fields are not the same as the original, all browse columns will be removed and you will have to specify new columns using the new ADD–COLUMNS–FROM( ), ADD–LIKE–COLUMN( ) and ADD–CALC–COLUMN( ) methods. If the QUERY attribute is set to the UNKNOWN value (?), all browse–columns are removed.

The following code example depicts an updateable static browse with all dynamic columns:

p-stbrw1.p

```
DEFINE BUTTON btn-quit LABEL "&Quit" AUTO-ENDKEY.
DEFINE VARIABLE i AS INTEGER.
DEFINE VARIABLE ColHandle AS HANDLE.
DEFINE QUERY q1 FOR customer SCROLLING.
OPEN QUERY q1 FOR EACH customer NO-LOCK.

DEFINE BROWSE StaticBrowse
  QUERY q1
       WITH TITLE "A Static Browse with Dynamic Columns"
       MULTIPLE
       SEPARATORS
       SIZE 104 BY 10.

DEFINE FRAME f1
    StaticBrowse SKIP(2)
    btn-quit AT ROW 13 COLUMN 3
    WITH THREE-D SIZE 105 BY 15 .

CURRENT-WINDOW:ROW = 1.
CURRENT-WINDOW:HEIGHT-CHARS = 16.
CURRENT-WINDOW:WIDTH-CHARS = 107.
CURRENT-WINDOW:TITLE = "A Static Browse with All Dynamic Columns".
CURRENT-WINDOW:KEEP-FRAME-Z-ORDER = TRUE.

StaticBrowse:READ-ONLY = FALSE.
StaticBrowse:ROW-MARKERS = TRUE.

StaticBrowse:ADD-LIKE-COLUMN("customer.state").
StaticBrowse:ADD-LIKE-COLUMN("customer.postal-code").
StaticBrowse:ADD-LIKE-COLUMN("customer.address").
StaticBrowse:ADD-LIKE-COLUMN("customer.address2").
StaticBrowse:ADD-LIKE-COLUMN("customer.terms").

REPEAT i=2 TO StaticBrowse:NUM-COLUMNS:
    ColHandle = StaticBrowse:GET-BROWSE-COLUMN(i).
    ColHandle:READ-ONLY = FALSE.
END.

ON CHOOSE OF btn-quit DO:
    APPLY "window-close" TO CURRENT-WINDOW.
END.

ENABLE ALL WITH FRAME F1.
WAIT-FOR CLOSE OF CURRENT-WINDOW.
```

21

Windows

When you start a session, Progress automatically creates a static window for the session. You can access this static window by using the DEFAULT–WINDOW system handle. This static window is also your initial current window, which you can access and reset using the CURRENT–WINDOW system handle.

The current window is the default session window for parenting frames, dialog boxes, and messages. Depending on your user interface, you can create one or more dynamic windows, using each one in turn as the current window. You can also specify a default window for the current external procedure by assigning the widget handle of any window to the CURRENT–WINDOW attribute of the THIS–PROCEDURE handle. The setting of this attribute overrides the setting of the CURRENT–WINDOW handle for the context of the current external procedure only. It has no effect on other external procedures in the session.

In a character interface, you can use only the static window for the entire session. In a graphical interface, you can create multiple windows dynamically within an application and create parent and child relationships between them. You can also change the appearance of any window, including (with some restrictions) the static window.

This chapter describes:

- Multiple window pros and cons
- Window attributes and methods
- Window events
- Creating and managing windows
- Window-based applications

21.1 Multiple-window Pros and Cons

There are several reasons why you might want to use more than one window in an application:

- **Maximizing use of screen space** — The simplest reason for using an additional window is to get more screen space. In some cases, you can avoid using a new window by enlarging the existing window or overlaying frames.

- **Grouping functionality** — You can use multiple windows to group widgets that are logically related. Each window can have its own menu bar with submenus related to those specific widgets. You can also group related windows into window families, hierarchies of parent and child windows.

- **Providing greater user control** — By placing information in multiple windows, you can provide greater access to application functions. This also gives the user the ability to move the windows (functionality) around the screen. The user can choose to put one window in front of another or to minimize a window that is not currently needed.

- **Displaying multiple copies of the same frame** — You can only display a specific frame once in a window. This can be a limitation, for example, if you want to show all the order items associated with an order. You can define the frame once (in a subprocedure or trigger) and view it in several windows simultaneously, populating each copy with different data.

  NOTE: You cannot enable fields for update in more than one copy of the same frame—even if the copies are in different windows. If you want to be able to update multiple copies of the frame, you must create separate frames for each window.

There are several disadvantages to using multiple windows in an application:

- **Window management** — You must manage the windows yourself. Specifically, you must ensure that windows are deleted when appropriate, usually when the controlling procedure goes out of scope.

- **Widget access** — In a multi-window interface, one window may become concealed behind another or inadvertently minimized, losing immediate access to application widgets. This may cause confusion for the user.

- **Lack of interface portability** — Since Progress supports only a single window in character interfaces, you cannot directly port a multi-window application to a character interface.

- **Transaction management** — Even though you may have several windows on the screen, you can only have one transaction active at a time. Undoing work in one window might cause work in another window to be undone also.

21.1.1 Windows Versus Dialog Boxes

A dialog box is a type of frame that shares some of the visual characteristics of a separate window. For example, the user can move a dialog box around the display outside its parent window. However, a dialog box is always modal. That means that when the dialog box is displayed, the user must react to that dialog box and cannot work in any other window until the dialog box has been dismissed.

Windows are non-modal by default, and are not easily made modal. Thus, the user can move freely from one window to another to enter input. If you want to enforce modality, use a dialog box. For more information on dialog boxes, see Chapter 25, “Interface Design.”

21.1.2 Multiple Windows and Transactions

When an application uses multiple windows, the user may tend to assume that actions in one window are totally independent of actions in another window. However, regardless of how many windows are on the screen, only one transaction is active at any time. This means that you cannot undo or commit work in one window without undoing or committing the work pending in all other windows. You must design your application so that transaction scoping is clear and intuitive to the user.

In multi-window applications, it simplifies transaction management if you can ensure that a transaction started in one window is committed before the user gives focus to another window. Thus, multi-window applications work more naturally with atomic transactions (those opened and committed by a single user action or dialog box). However, your application must ultimately determine your transaction management requirements.

21.2 Window Attributes and Methods

You can set attributes for both the static window created by Progress and windows that you create. You can, for example:

- Specify the window title (TITLE attribute).
- Set the normal, minimum, and maximum sizes of the window. You can do this in either of two units:
  - In pixels (HEIGHT–PIXELS, WIDTH–PIXELS, MIN–HEIGHT–PIXELS, MIN–WIDTH–PIXELS, MAX–HEIGHT–PIXELS, MAX–WIDTH–PIXELS attributes).
  - In character units (HEIGHT–ROWS–CHARS, WIDTH–COLUMNS–CHARS, MIN–HEIGHT–CHARS, MIN–WIDTH–CHARS, MAX–HEIGHT–CHARS, MAX–WIDTH–CHARS attributes).
- Specify the maximum display area within the window (VIRTUAL–HEIGHT–CHARS, VIRTUAL–WIDTH–CHARS, VIRTUAL–HEIGHT–PIXELS, VIRTUAL–WIDTH–PIXELS).
- Allow or prevent the user resizing the window by setting the RESIZE attribute.
- Specify the presence or absence of a window component or its appearance, such as message area (MESSAGE–AREA, MESSAGE–AREA–FONT), status area (STATUS–AREA, STATUS–AREA–FONT), or scroll bars (SCROLL–BARS).
- Change the state (minimized, maximized, restored) of a window within a program (WINDOW–STATE attribute).
- Associate a menu bar or pop-up menu with the window (MENUBAR or POPUP–MENU attribute).
- Find all frames within the window (FIRST–CHILD or LAST–CHILD attribute of the window; NEXT–SIBLING or PREV–SIBLING attribute of frames).
- Set up parent and child relationships between windows (PARENT attribute).

There are also several methods available on a window widget. The most common of these methods that apply to windows are the LOAD–ICON() and LOAD–SMALL–ICON() methods. These methods allow you to associate icons with windows. The icon displays to reference a window in one of its states such as minimized or maximized. For example, using the LOAD–ICON() method allows you to specify an icon to display in the title bar of a window (maximized), in the task bar (minimized), and when selecting a program using **ALT–TAB**. The LOAD-SMALL–ICON() method allows you to specify an icon to display in the title bar of a window and in the task bar only. The value you assign with either the LOAD–ICON() or the LOAD–SMALL–ICON() methods must be the name of an icon (.ico) file. Both of these methods accommodate icons formatted as small size (16x16), regular size (32x32), or both.

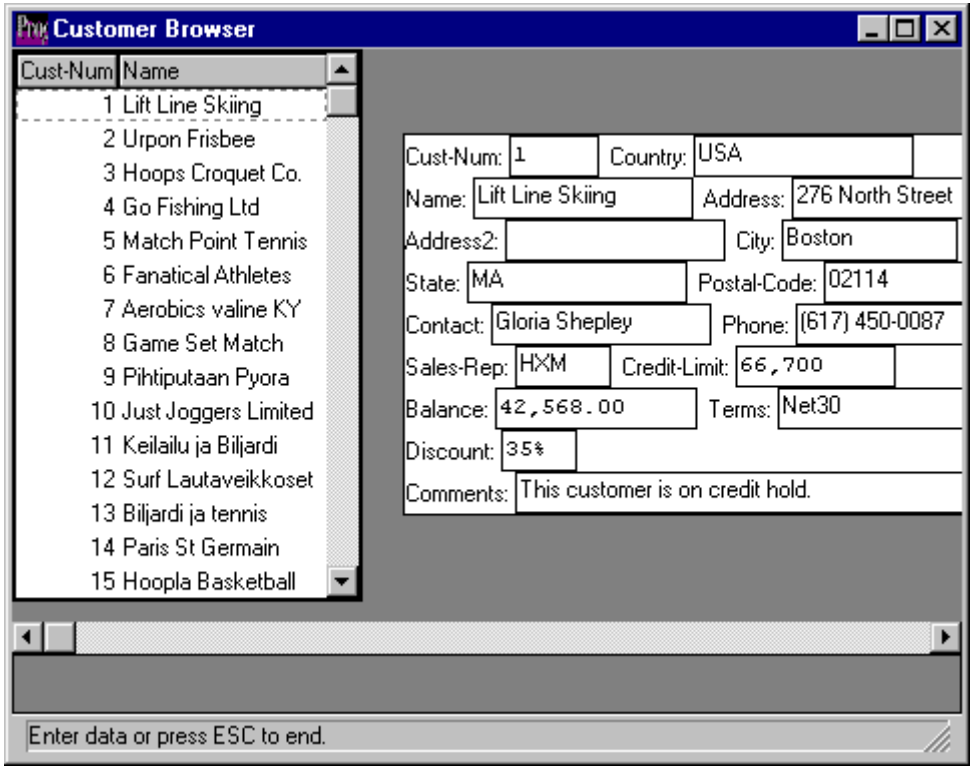
For more information on these and other window attributes and methods, see the *Progress Language Reference*.

The following procedure uses attributes of the static window to make the window wide and short and to change its title:

p-wina.p

```
OPEN QUERY custq FOR EACH customer.
DEFINE BROWSE custb QUERY custq DISPLAY cust-num name WITH 15 DOWN.

FORM
  custb
  WITH FRAME x.

FORM
   customer
   WITH FRAME y SIDE-LABELS COLUMN 40 ROW 3 WIDTH 75.

ASSIGN DEFAULT-WINDOW:VIRTUAL-WIDTH-CHARS = 120
       DEFAULT-WINDOW:VIRTUAL-HEIGHT-CHARS = 15
       DEFAULT-WINDOW:WIDTH-CHARS = 120
       DEFAULT-WINDOW:HEIGHT-CHARS = 15
       DEFAULT-WINDOW:TITLE = "Customer Browser".

ON ITERATION-CHANGED OF BROWSE custb
  DO:
     DISPLAY customer WITH FRAME y.
  END.

ON WINDOW-CLOSE OF CURRENT-WINDOW
     QUIT.

ENABLE custb WITH FRAME x.

APPLY "ITERATION-CHANGED" TO BROWSE custb.

ENABLE ALL WITH FRAME y.

WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

When you run this code, the following window appears:



21.3 Window Events

In addition to several standard events, windows respond to unique events of their own.

Several events indicate window state changes, including WINDOW–MINIMIZED, WINDOW–MAXIMIZED, and WINDOW–RESTORED.

The WINDOW–RESIZED event occurs any time a keyboard or mouse action starts to change the window's vertical or horizontal dimensions. Note that the window RESIZE attribute must be set to TRUE for this event to occur.

The WINDOW–CLOSE event occurs when the user tries to close a window. By default, Progress takes no action when the WINDOW–CLOSE event occurs—it does not close the window. If you want your application to react to the WINDOW–CLOSE event, you must code a trigger for it or reference it in a WAIT–FOR statement. In fact, this is a typical termination condition.

```
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

For more information on window events, see the *Progress Language Reference*.

21.4 Creating and Managing Windows

You can create a new window dynamically with the CREATE Widget statement.

This is the syntax for creating a window:

SYNTAX

```
CREATE WINDOW handle
  [ IN WIDGET-POOL pool-name ]
  [ ASSIGN attribute = value [ attribute = value ] ... ]
  { [ trigger-phrase ] }
```

NOTE: Character interfaces support only a single window. If you attempt to create additional windows, Progress raises the ERROR condition.

You must define *handle* as a WIDGET–HANDLE variable, field, or parameter before creating the window. Likewise, you must create the widget pool specified by *pool–name* using a CREATE WIDGET–POOL statement before you reference the widget pool in the CREATE WINDOW statement. In some applications, you might prefer to use unnamed widget pools to contain your windows. This is especially useful with windows created by persistent procedures, where a single unnamed widget pool can encompass the entire dynamic context of the procedure.

One of the advantages of creating your own window is the ability to specify all of its attributes. The ASSIGN option allows you to specify any window attribute you want, including those that must be specified before realization. These include sizing and component attributes, such as RESIZE and MESSAGE–AREA.

You can specify user interface triggers for a window using either the *trigger–phrase* or the ON statement. The *trigger–phrase* offers the convenience of a single window context, while the ON statement offers the flexibility to define your triggers conditionally, after the window is created.

For more information on handles, attributes, and user interface triggers, see Chapter 16, "Widgets and Handles." For more information on widget pools, see Chapter 20, "Using Dynamic Widgets."

21.4.1 Creating Window Families

By default, when you create a window, Progress parents that window transparently to the window system. In this way, windows are siblings of each other. You must manage these windows individually.

You can also parent a window (*child window*) to another window (*parent window*) by setting the child window's PARENT attribute to the widget handle of the parent window. Windows that are parented by a window, which in turn is parented by the window system, form a *window family*. The window parented by the window system is the *root window* of the window family. Windows parented by any child window, in turn, form a *child window family*. A child window can only be parented by one window at a time.

Window Families vs. Individual Windows

Each window in a window family, by itself, functions much the same as an individual window. That is, you can individually move, resize, and interact with a member of a window family like an individual window.

However, window families share a number of additional properties that make them convenient for both applications and users to manage:

- **Coordinated viewing and hiding** — When you view any member of a window family, the whole window family is viewed, unless the HIDDEN attribute is TRUE for at least one other member. If HIDDEN is TRUE for a parent or ancestor window, no windows in the window family are viewed, however any HIDDEN attribute setting for the explicitly viewed window is set to FALSE. If HIDDEN is TRUE for a descendant window, all windows in the family are viewed except the hidden descendant window and all of its descendants. When you hide a member of a window family (set VISIBLE to FALSE), that window and all of its descendant windows are hidden (VISIBLE set to FALSE), but none of their HIDDEN attributes are affected.

- **Coordinated minimizing and restoring** — When you minimize (iconify) a window, all of its descendants disappear from view, unless they are already minimized. Any minimized descendants remain minimized and can be restored individually. When you restore a parent window, any of its hidden descendants are redisplayed. Note that when you hide a window (set VISIBLE to FALSE), any minimized descendants are hidden also, and when you redisplay that window, its minimized descendants reappear minimized.

- **Coordinated close events** — If a parent window receives a WINDOW–CLOSE event, it propagates a PARENT–WINDOW–CLOSE event to all of its descendant windows. However, any action on these events is trigger-dependant. The WINDOW–CLOSE does not propagate any events upward to ancestor windows.

For more information on window family properties, see the WINDOW Widget, HIDDEN Attribute, and VISIBLE Attribute reference entries in the *Progress Language Reference*.

Window Family Example

The following procedure creates a window family with three windows—a parent, child, and grandchild window. By choosing the buttons in the default window (Control Panel), you can visualize most of the properties described in the previous list.

p-wow1.p *(1 of 2)*

```
DEFINE VARIABLE whandle1 AS HANDLE.
DEFINE VARIABLE whandle2 AS HANDLE.
DEFINE VARIABLE whandle3 AS HANDLE.
DEFINE BUTTON bviewp LABEL "VIEW".
DEFINE BUTTON bviewc LABEL "VIEW".
DEFINE BUTTON bviewgc LABEL "VIEW".
DEFINE BUTTON bhidep LABEL "HIDE".
DEFINE BUTTON bhidec LABEL "HIDE".
DEFINE BUTTON bhidegc LABEL "HIDE".
DEFINE BUTTON bhiddp LABEL "HIDDEN".
DEFINE BUTTON bhiddc LABEL "HIDDEN".
DEFINE BUTTON bhiddgc LABEL "HIDDEN".
DEFINE FRAME alpha SKIP
    "Parent" AT 11 "Child" AT 37 "Grand Child" AT 64 SKIP
    bviewp AT 11 bviewc AT 37 bviewgc AT 64 SKIP(.5)
    bhidep AT 11 bhidec AT 37 bhidegc AT 64 SKIP(.5)
    bhiddp AT 11 bhiddc AT 37 bhiddgc AT 64
WITH SIZE 80 BY 6.

CREATE WINDOW whandle1
    ASSIGN  TITLE = "Parent Window"
            HEIGHT-CHARS = 5
            WIDTH-CHARS = 27
            PARENT = CURRENT-WINDOW.
CREATE WINDOW whandle2
    ASSIGN  TITLE = "Child Window"
            HEIGHT-CHARS = 5
            WIDTH-CHARS = 27
            PARENT = whandle1.
CREATE WINDOW whandle3
    ASSIGN  TITLE = "Grand Child Window"
            HEIGHT-CHARS = 5
            WIDTH-CHARS = 27
            PARENT = whandle2.

ON CHOOSE OF bviewp DO: /* View Parent */
    VIEW whandle1.
    RUN win-status IN THIS-PROCEDURE.
END.
```

p-wow1.p *(2 of 2)*

```
ON CHOOSE OF bhidep DO: /* Hide Parent */
    HIDE whandle1.
    RUN win-status IN THIS-PROCEDURE.
END.
ON CHOOSE OF bhiddp DO: /* Hidden Parent */
    whandle1:HIDDEN = TRUE.
    RUN win-status IN THIS-PROCEDURE.
END.
ON CHOOSE OF bviewc DO: /* View Child */
    VIEW whandle2.
    RUN win-status IN THIS-PROCEDURE.
END.
ON CHOOSE OF bhidec DO: /* Hide Child */
    HIDE whandle2.
    RUN win-status IN THIS-PROCEDURE.
END.
ON CHOOSE OF bhiddc DO: /* Hidden Child */
    whandle2:HIDDEN = TRUE.
    RUN win-status IN THIS-PROCEDURE.
END.
ON CHOOSE OF bviewgc DO: /* View Grand Child */
    VIEW whandle3.
    RUN win-status IN THIS-PROCEDURE.
END.
ON CHOOSE OF bhidegc DO: /* Hide Grand Child */
    HIDE whandle3.
    RUN win-status IN THIS-PROCEDURE.
END.
ON CHOOSE OF bhiddgc DO: /* Hidden Grand Child */
    whandle3:HIDDEN = TRUE.
    RUN win-status IN THIS-PROCEDURE.
END.

CURRENT-WINDOW:TITLE = "Control Panel".
CURRENT-WINDOW:HEIGHT-CHARS = 6.
ENABLE ALL IN WINDOW CURRENT-WINDOW WITH FRAME alpha.
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.

PROCEDURE win-status:
    MESSAGE "Parent HIDDEN:" whandle1:HIDDEN
            "/ Parent VISIBLE:" whandle1:VISIBLE
            "/ Child HIDDEN:" whandle2:HIDDEN
            "/ Child VISIBLE:" whandle2:VISIBLE.
    MESSAGE "Grand Child HIDDEN:" whandle3:HIDDEN
            "/ Grand Child VISIBLE:" whandle3:VISIBLE.
END.
```

Figure 21–1 shows the Control Panel window and the example window family.

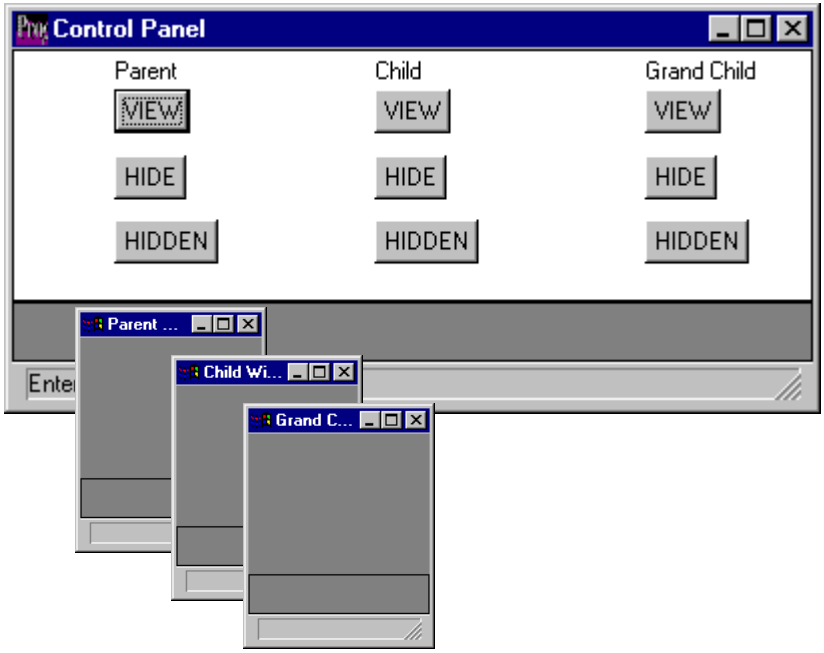


Figure 21–1: Window Family

Combine the VIEW, HIDE, and HIDDEN button events with the WINDOW–MINIMIZE and WINDOW–RESTORE events using the window controls to see how window hiding and viewing work together with window minimizing and restoring.

For an example that uses a window family in a database application, see the "Persistent Multi-window Management" section.

21.5 Window-based Applications

Progress supports three types of interactive application:

- **Single-window** — Every interactive Progress application uses at least a single window. Typically, you use the static window provided by Progress, although you can create your own, instead, if you want to change any attributes set prior to realization. You can change any attributes of the static window that can be set after realization.

- **Multi-window with non-persistent management** — You can create additional windows for viewing or updating specific information using non-persistent procedures. In non-persistent procedures, you can use multiple windows to group related information or just to provide greater flexibility and more screen space. In a non-persistent environment, you must manage each window explicitly throughout the application. Also, you cannot easily manage multiple windows as copies or iterations of a single window. No matter how similar the definition of each window is to another, each window requires separate management code that must be integrated with the rest at every point in the application.

- **Multi-window with persistent management** — You can also create additional windows using persistent procedures. This technique allows you to have more than one copy of a like-defined window on the screen simultaneously, managed by the same application code. Managing multiple windows with persistent procedures affords far greater flexibility than using only non-persistent procedures.

  Typically, a persistent procedure creates the window it controls when it executes. This window then remains on screen and available for input after the procedure returns from execution. Unlike windows created by non-persistent procedures, windows created in persistent procedures require little or no management by the main line of your application.

  The essential management code for each window is contained in the persistent procedure that created it. Also, each time you execute a persistent procedure, it can create a completely new window that is a copy of others created by the same procedure. Each copy is managed by the same application code in its own procedure context. If you so design, your main-line application can interact with the windows of these persistent procedures through standard procedure triggers or internal procedures provided to the application by the persistent procedures.

  NOTE: You can also create multiple windows using persistent triggers in a non-persistent procedure, but this is a less robust technique with more complicated window management requirements.

As the single-window application is the default application style used throughout this manual, the following sections describe the two basic techniques of managing multi-window applications: non-persistent and persistent.

21.5.1 Non-persistent Multi-window Management

The following procedure creates a concurrent window and stores its handle in the new–win variable. It then preforms I/O to both the new window and the static window:

p-win1.p

```
DEFINE VARIABLE new-win AS WIDGET-HANDLE.

FORM
   customer
   WITH FRAME cust-frame.

FORM
   salesrep
   WITH FRAME rep-frame.

CREATE WINDOW new-win
      ASSIGN TITLE = "Sales Representative".

ASSIGN DEFAULT-WINDOW:TITLE = "Customer".

VIEW new-win.
MESSAGE "This is the default window.".
MESSAGE "This is the newly created window." IN WINDOW new-win.

PAUSE 0 BEFORE-HIDE.
FOR EACH customer, salesrep OF customer ON ERROR UNDO, LEAVE
       ON ENDKEY UNDO, LEAVE ON STOP UNDO, LEAVE:
   DISPLAY customer WITH FRAME cust-frame.
   ENABLE ALL WITH FRAME cust-frame.

   CURRENT-WINDOW = new-win.
   DISPLAY salesrep WITH FRAME rep-frame.
   ENABLE ALL WITH FRAME rep-frame.
   CURRENT-WINDOW = DEFAULT-WINDOW.

   WAIT-FOR GO OF FRAME cust-frame, FRAME rep-frame.

   ASSIGN customer.
   ASSIGN salesrep.
END.

DELETE WIDGET new-win.
```

The p-win1.p procedure demonstrates two ways to direct I/O to a specific window: the IN WINDOW option and the CURRENT–WINDOW handle. Because the first MESSAGE statement does not reference a window, it writes to the current window, which is initially the static window for the session. The second message uses the IN WINDOW option to direct the output to the newly created window. Within the FOR EACH group, the first DISPLAY statement writes to the static window. Before the second DISPLAY statement, the CURRENT–WINDOW value is reset to the new window. Therefore, that DISPLAY statement writes to the new window. The value of CURRENT–WINDOW is then reset to the static window. Note that the DISPLAY statements must use separate frames, because a frame can only appear in one window at a time.

21.5.2 Persistent Multi-window Management

An application that uses persistent multi-window management generally provides its basic application options in the default application window. These options then open one or more windows under persistent procedure control to provide functionality in addition to the default application window. All of these windows, while visible, remain available to the user in non-modal fashion. You can also provide modal capabilities anywhere in the non-modal interface using alert boxes and dialog boxes.

Creating Windows in Persistent Procedures

In general, you create and manage each persistent window using a persistent procedure. Each time you run a procedure persistently, it instantiates (creates) a context for itself. This context can include any windows that you create during instantiation. However, each persistent procedure can maintain its own current window by setting the CURRENT–WINDOW attribute of the THIS–PROCEDURE handle. This attribute overrides (but does not change) the setting of the CURRENT–WINDOW handle. If set to the widget handle of a valid window, all frames and dialog boxes defined in the procedure parent, by default, to the window specified by this attribute. Thus, a persistent procedure instance usually manages a single window in the multi-window application. For more information on procedure handles and persistent procedures, see Chapter 3, “Block Properties.”

Managing Windows in Persistent Procedures

It is also often helpful to create a separate dynamic widget pool for the window you create in the persistent procedure. Usually, you want the window to exist for the life of the persistent procedure that manages it. The simplest way to guarantee this is by creating a single unnamed widget pool for the procedure. When you delete a persistent procedure that maintains its own widget pool, its persistent window is automatically deleted also. For more information on dynamic widget pools, see Chapter 20, “Using Dynamic Widgets.”

Example — Persistent Multi-window Management

The following three procedures, p-perwn1.p, p-perwn2.p, and p-perwn3.p implement a small application for updating customer orders in the sports database using multiple windows. The main procedure (p-perwn1.p) displays a browse of the customer table in the default application window. You can browse the order table for a selected customer by choosing the bOpenorders button. This creates a persistent procedure (p-perwn2.p) that opens a child window for the order browse (p-perwn1.p (1)):

p-perwn1.p *(1 of 2)*

```
    DEFINE QUERY custq FOR customer.
    DEFINE BROWSE custb QUERY custq
        DISPLAY name cust-num balance credit-limit phone sales-rep
    WITH 10 DOWN.

    DEFINE BUTTON bExit LABEL "Exit".
    DEFINE BUTTON bOpenorders LABEL "Open Orders".
    DEFINE VARIABLE whand AS WIDGET-HANDLE.
    DEFINE VARIABLE lcust-redundant AS LOGICAL.

    DEFINE FRAME CustFrame
        custb SKIP bExit bOpenorders
    WITH SIZE-CHARS 96.5 by 11.

    ON CHOOSE OF bExit IN FRAME CustFrame DO:
        RUN exit-proc.
    END.

    ON CHOOSE OF bOpenorders IN FRAME CustFrame DO:
        IF custb:NUM-SELECTED-ROWS >= 1 THEN DO:
            RUN check-redundant(OUTPUT lcust-redundant).
            IF NOT lcust-redundant THEN DO:
                MESSAGE "Opening orders for" customer.name + ".".
(1)             RUN p-perwn2.p PERSISTENT
                    (BUFFER customer, INPUT whand:TITLE,
                     INPUT whand) NO-ERROR.
            END.
            ELSE DO:
                BELL.
                MESSAGE "Orders already open for" customer.name + ".".
            END.
        END.
        ELSE DO:
            BELL.
            MESSAGE "Select a customer to open orders ...".
        END.
    END.
```

p-perwn1.p *(2 of 2)*

```
(2) CREATE WINDOW whand
        ASSIGN
            TITLE = "Customer Order Maintenance"
            SCROLL-BARS = FALSE
            RESIZE = FALSE
            HEIGHT-CHARS = FRAME CustFrame:HEIGHT-CHARS
            WIDTH-CHARS = FRAME CustFrame:WIDTH-CHARS.
    CURRENT-WINDOW = whand.

    OPEN QUERY custq PRESELECT EACH customer BY name.
    PAUSE 0 BEFORE-HIDE.
    ENABLE ALL WITH FRAME CustFrame.

    WAIT-FOR CHOOSE OF bExit IN FRAME CustFrame.

    PROCEDURE exit-proc:
        DEFINE VARIABLE phand AS HANDLE.
        DEFINE VARIABLE nhand AS HANDLE.

        MESSAGE "Exiting application ...".
(3)     phand = SESSION:FIRST-PROCEDURE.
        DO WHILE VALID-HANDLE(phand):
            nhand = phand:NEXT-SIBLING.
            IF LOOKUP(whand:TITLE, phand:PRIVATE-DATA) > 0 THEN
                RUN destroy-query IN phand NO-ERROR.
            phand = nhand.
        END.
        DELETE WIDGET whand.
    END PROCEDURE.

    PROCEDURE check-redundant:
        DEFINE OUTPUT PARAMETER lcust-redundant AS LOGICAL INITIAL FALSE.

        DEFINE VARIABLE phand AS HANDLE.
        DEFINE VARIABLE nhand AS HANDLE.

(4)     phand = SESSION:FIRST-PROCEDURE.
        DO WHILE VALID-HANDLE(phand):
            nhand = phand:NEXT-SIBLING.
            IF LOOKUP(whand:TITLE, phand:PRIVATE-DATA)   > 0 AND
               LOOKUP(customer.name, phand:PRIVATE-DATA) > 0 THEN DO:
                    lcust-redundant = TRUE.
                    RETURN.
            END.
            phand = nhand.
        END.
    END PROCEDURE.
```

You can repeatedly select another customer and choose the bOpenorders button to browse orders for as many customers as you want. The customer and order browses displayed for every customer all remain available to the user for input simultaneously.

This application makes ample use of procedure and SESSION handle attributes to help manage the persistent windows of the application. The main procedure uses them in two ways:

- To delete all persistent procedures (and their windows) when you terminate the application
- To identify whether an order window has already been created for a selected customer so that only one order browse is enabled for a customer at a time

When you terminate the application by choosing the bExit button, `p-perwn1.p` calls a local internal procedure, exit–proc. After locating the first persistent procedure instance, exit–proc loops through all persistent procedures in the session and deletes each one created by the application as specified by the PRIVATE–DATA procedure handle attribute (`p-perwn1.p` (3)). Note that exit–proc deletes each persistent procedure by calling the destroy–query procedure owned by the persistent procedure. This ensures that each persistent procedure manages its own resource clean-up and deletion.

When you attempt to open the order browse for a customer, `p-perwn1.p` also checks to see if there is already an order browse open for that customer. It does this by looking for a persistent procedure with the customer name as part of its private data (`p-perwn1.p` (4)).

Note also that `p-perwn1.p` creates its own default application window instead of using the static window (`p-perwn1.p` (2)). This allows the setting of the SCROLL–BARS and RESIZE window attributes.

The persistent procedure, `p-perwn2.p`, displays a browse of all orders for the selected customer. Like the customer browse, you can repeatedly select orders and choose the bOpenlines button (which runs `p-perwn3.p`) to browse order lines in a child window of a selected order for as many orders as you want (`p-perwn2.p` (1)). In `p-perwn2.p`, you can also update each order by choosing the bUpdate button and close all orders for a customer by choosing the bClose button.

Note that as persistent procedures, both `p-perwn2.p` and `p-perwn3.p` are written to be run non-persistently for testing or other application purposes. This is accomplished by making some of the code conditionally executable based on the value of the PERSISTENT attribute of the THIS–PROCEDURE system handle. For example, the bClose button in `p-perwn2.p` runs the destroy–query procedure or just exits `p-perwn2.p`, depending on how it is run (`p-perwn2.p` (4)). You can devise many variations on this approach. One possible variation includes using the preprocessor to conditionally compile the PERSISTENT attribute tests based on whether you are compiling for a development or production environment.

Each persistent procedure easily maintains the current window that it creates by setting the CURRENT–WINDOW attribute of the THIS–PROCEDURE handle (`p-perwn2.p` (3) and `p-perwn3.p` (2)). Thus, there is no need to set and reset the CURRENT–WINDOW handle:

p-perwn2.p *(1 of 5)*

```
    DEFINE PARAMETER BUFFER custbuf FOR customer.
    DEFINE INPUT PARAMETER appdata AS CHARACTER.
    DEFINE INPUT PARAMETER wparent AS WIDGET-HANDLE.

    DEFINE QUERY orderq FOR order.
    DEFINE BROWSE orderb QUERY orderq
        DISPLAY
            order.order-num order.order-date
            order.ship-date order.promise-date order.carrier
        WITH 5 DOWN.
    DEFINE BUTTON bClose LABEL "Close Orders".
    DEFINE BUTTON bOpenlines LABEL "Open Order Lines".
    DEFINE BUTTON bUpdate LABEL "Update Order".
    DEFINE VARIABLE whand AS WIDGET-HANDLE.
    DEFINE VARIABLE lorder-redundant AS LOGICAL.
    DEFINE FRAME OrderFrame SKIP(.5)
        custbuf.name COLON 11 VIEW-AS TEXT SKIP
        custbuf.cust-num COLON 11 VIEW-AS TEXT SKIP(.5)
        orderb SKIP
        bClose bOpenlines bUpdate
    WITH SIDE-LABELS SIZE-CHARS 65.8 by 9.

    ON CHOOSE OF bOpenlines IN FRAME OrderFrame DO:
        IF orderb:NUM-SELECTED-ROWS >= 1 THEN DO:
            RUN check-redundant(OUTPUT lorder-redundant).
            IF NOT lorder-redundant THEN DO:
                MESSAGE "Opening order lines for order"
                        STRING(order.order-num) + ".".
(1)             RUN p-perwn3.p PERSISTENT
                    (BUFFER order, INPUT THIS-PROCEDURE:PRIVATE-DATA,
                     INPUT whand) NO-ERROR.
            END.
            ELSE DO:
                BELL.
                MESSAGE "Order lines already open for order"
                        STRING(order.order-num) + ".".
            END.
        END.
        ELSE DO:
            BELL.
            MESSAGE "Select an order to open order lines ...".
        END.
    END.
```

p-perwn2.p *(2 of 5)*

```
    ON CHOOSE OF bUpdate IN FRAME OrderFrame DO:
        IF orderb:NUM-SELECTED-ROWS >= 1 THEN DO:
            RUN update-order.
        END.
        ELSE DO:
            BELL.
            MESSAGE "Select an order to update ...".
        END.
    END.

(2) IF THIS-PROCEDURE:PERSISTENT THEN DO:
        THIS-PROCEDURE:PRIVATE-DATA = appdata + "," + custbuf.name.
        CREATE WIDGET-POOL.
    END.

(3) CREATE WINDOW whand
        ASSIGN
            TITLE = "Orders for Customer ..."
            PARENT = wparent
            RESIZE = FALSE
            SCROLL-BARS = FALSE
            HEIGHT-CHARS = FRAME OrderFrame:HEIGHT-CHARS
            WIDTH-CHARS = FRAME OrderFrame:WIDTH-CHARS.

    THIS-PROCEDURE:CURRENT-WINDOW = whand.

    OPEN QUERY orderq PRESELECT EACH order
        WHERE order.cust-num = custbuf.cust-num BY order-num.
    DISPLAY custbuf.cust-num custbuf.name WITH FRAME OrderFrame.
    ENABLE ALL WITH FRAME OrderFrame.

(4) IF THIS-PROCEDURE:PERSISTENT THEN DO:
        ON CHOOSE OF bClose IN FRAME OrderFrame DO:
            RUN destroy-query.
        END.
    END.
    ELSE DO:
        WAIT-FOR CHOOSE OF bClose IN FRAME OrderFrame.
    END.
```

p-perwn2.p *(3 of 5)*

```
    PROCEDURE destroy-query:
        DEFINE VARIABLE phand AS HANDLE.
        DEFINE VARIABLE nhand AS HANDLE.

(5)     MESSAGE "Exiting orders for" custbuf.name "...".
        phand = SESSION:FIRST-PROCEDURE.
        DO WHILE VALID-HANDLE(phand):
            nhand = phand:NEXT-SIBLING.
            IF LOOKUP(custbuf.name, phand:PRIVATE-DATA) > 0 AND
               phand <> THIS-PROCEDURE THEN
                    RUN destroy-query IN phand NO-ERROR.
            phand = nhand.
        END.
        DELETE PROCEDURE THIS-PROCEDURE NO-ERROR.
        DELETE WIDGET-POOL.
    END.

    PROCEDURE check-redundant:
        DEFINE OUTPUT PARAMETER lorder-redundant AS LOGICAL INITIAL FALSE.

        DEFINE VARIABLE phand AS HANDLE.
        DEFINE VARIABLE nhand AS HANDLE.

(6)     phand = SESSION:FIRST-PROCEDURE.
        DO WHILE VALID-HANDLE(phand):
            nhand = phand:NEXT-SIBLING.
            IF LOOKUP(appdata, phand:PRIVATE-DATA) > 0 AND
               LOOKUP(STRING(order.order-num),
                             phand:PRIVATE-DATA) > 0 THEN DO:
                    lorder-redundant = TRUE.
                    RETURN.
            END.
            phand = nhand.
        END.
    END PROCEDURE.

    PROCEDURE update-order:
        DEFINE VARIABLE rid AS ROWID.
        DEFINE VARIABLE choice AS LOGICAL.
        DEFINE BUTTON bSave LABEL "Save Changes".
        DEFINE BUTTON bCancel LABEL "Cancel".
```

p-perwn2.p *(4 of 5)*

```
(7)     DEFINE FRAME UpdateFrame SKIP(.5)
            order.order-num COLON 12 VIEW-AS TEXT
                order.sales-rep VIEW-AS TEXT "For..." VIEW-AS TEXT
                custbuf.name VIEW-AS TEXT SKIP
            custbuf.cust-num COLON 48 VIEW-AS TEXT SKIP(1) SPACE(1)
            order.order-date order.ship-date order.promise-date
            SKIP SPACE(1)
            order.carrier order.instructions SKIP SPACE(1)
            order.PO order.terms SKIP(1)
            bSave bCancel
        WITH TITLE "Update Order" SIDE-LABELS VIEW-AS DIALOG-BOX.

        ON CHOOSE OF bSave IN FRAME UpdateFrame
           OR GO OF FRAME UpdateFrame DO:
            MESSAGE "Are you sure you want to save your changes?"
                VIEW-AS ALERT-BOX QUESTION BUTTONS YES-NO
                UPDATE choice AS LOGICAL.
            CASE choice:
(8)             WHEN TRUE THEN DO TRANSACTION ON ERROR UNDO, RETRY:
                    rid = ROWID(order).
                    FIND order WHERE ROWID(order) = rid EXCLUSIVE-LOCK.
                    ASSIGN
                        order.order-date
                        order.ship-date
                        order.promise-date
                        order.carrier
                        order.instructions
                        order.PO order.terms.
                    DISPLAY
                        order.order-num order.order-date
                        order.ship-date order.promise-date order.carrier
                    WITH BROWSE orderb.
                    APPLY "WINDOW-CLOSE" TO FRAME UpdateFrame.
                END.
                WHEN FALSE THEN DO:
                   MESSAGE "Changes not saved."
                       VIEW-AS ALERT-BOX INFORMATION BUTTONS OK.
                   RETURN NO-APPLY.
                END.
            END CASE.
        END.
```

p-perwn2.p *(5 of 5)*

```
        ON CHOOSE OF bCancel DO:
            MESSAGE
                "Are you sure you want to cancel your current updates?"
                VIEW-AS ALERT-BOX QUESTION BUTTONS YES-NO
                UPDATE choice AS LOGICAL.
            CASE choice:
                WHEN TRUE THEN DO:
                   MESSAGE "Current updates cancelled."
                       VIEW-AS ALERT-BOX INFORMATION BUTTONS OK.
                   APPLY "WINDOW-CLOSE" TO FRAME UpdateFrame.
                END.
                WHEN FALSE THEN DO:
                   MESSAGE "Current update is continuing ..."
                       VIEW-AS ALERT-BOX INFORMATION BUTTONS OK.
                END.
            END CASE.
        END.

        DISPLAY
            order.order-num order.sales-rep custbuf.name custbuf.cust-num
            order.order-date order.ship-date order.promise-date
            order.carrier order.instructions order.PO order.terms
        WITH FRAME UpdateFrame IN WINDOW ACTIVE-WINDOW.
        ENABLE
            order.order-date order.ship-date order.promise-date
            order.carrier order.instructions order.PO order.terms
            bSave bCancel
        WITH FRAME UpdateFrame IN WINDOW ACTIVE-WINDOW.

        WAIT-FOR WINDOW-CLOSE OF FRAME UpdateFrame.
    END PROCEDURE.
```

Two other important features of persistent management include setting the PRIVATE–DATA attribute and creating a widget pool so that the context of each persistent procedure can be more easily managed as a unit (`p-perwn2.p` (2) and `p-perwn3.p` (1)).

Like `p-perwn1.p`, `p-perwn2.p` uses procedure and SESSION handle attributes:

- To delete all persistent procedures (and their windows) created for customer orders when you exit a customer order browse (`p-perwn2.p` (5))
- To identify whether an order–line window has already been created for a selected order so that only one order–line browse is enabled for an order at a time (`p-perwn2.p` (6))

Note that each persistent procedure has a unique destroy–query procedure. This allows any **other** procedure to delete the persistent procedures under its control while allowing each persistent procedure to manage its own area of control appropriately.

The persistent procedure, `p-perwn3.p`, browses and updates order lines very much like `p-perwn2.p` browses and updates orders, except that it is at the bottom of the hierarchy of persistent management. When you choose the bClose button in `p-perwn3.p`, destroy–query only has to delete the current instance of `p-perwn3.p` (`p-perwn3.p` (3)). Depending on your application, you can also use a management model that is more or less hierarchical. Because Progress procedures are recursive, you can even create persistent procedures and windows recursively. However, there is no necessary connection between a persistent context and its recursive instance unless you establish it, for example, using the PRIVATE–DATA attribute or a shared temporary table.

p-perwn3.p *(1 of 4)*

```
    DEFINE PARAMETER BUFFER orderbuf FOR order.
    DEFINE INPUT PARAMETER appdata AS CHARACTER.
    DEFINE INPUT PARAMETER wparent AS WIDGET-HANDLE.

    DEFINE QUERY ordlineq FOR order-line, item.
    DEFINE BROWSE ordlineb QUERY ordlineq
        DISPLAY
            order-line.line-num order-line.item-num item.item-name
            order-line.price order-line.qty order-line.extended-price
            order-line.discount order-line.backorder
        WITH 5 DOWN.
    DEFINE BUTTON bClose LABEL "Close Order Lines".
    DEFINE BUTTON bUpdate LABEL "Update Order Line".
    DEFINE VARIABLE whand AS WIDGET-HANDLE.
    DEFINE FRAME OrdlineFrame SKIP(.5)
        orderbuf.order-num COLON 12 VIEW-AS TEXT SKIP(.5)
        ordlineb SKIP
        bClose bUpdate
    WITH SIDE-LABELS SIZE-CHARS 98.8 by 8.5.

    ON CHOOSE OF bUpdate IN FRAME OrdlineFrame DO:
        IF ordlineb:NUM-SELECTED-ROWS >= 1 THEN DO:
            RUN update-order-line.
        END.
        ELSE DO:
            BELL.
            MESSAGE "Select an order line to update ...".
        END.
    END.

(1) IF THIS-PROCEDURE:PERSISTENT THEN DO:
        THIS-PROCEDURE:PRIVATE-DATA = appdata + ","
                                    + STRING(orderbuf.order-num).
        CREATE WIDGET-POOL.
    END.
```

p-perwn3.p *(2 of 4)*

```
(2) CREATE WINDOW whand
        ASSIGN
            TITLE = "Order Lines for Order ..."
            PARENT = wparent
            RESIZE = FALSE
            SCROLL-BARS = FALSE
            HEIGHT-CHARS = FRAME OrdlineFrame:HEIGHT-CHARS
            WIDTH-CHARS = FRAME OrdlineFrame:WIDTH-CHARS.
    THIS-PROCEDURE:CURRENT-WINDOW = whand.

    OPEN QUERY ordlineq PRESELECT EACH order-line
        WHERE order-line.order-num = orderbuf.order-num, EACH item
        WHERE item.item-num = order-line.item-num
        BY order-line.line-num.
    DISPLAY orderbuf.order-num WITH FRAME OrdlineFrame.
    ENABLE ALL WITH FRAME OrdlineFrame.

    IF THIS-PROCEDURE:PERSISTENT THEN DO:
        ON CHOOSE OF bClose IN FRAME OrdlineFrame DO:
            RUN destroy-query.
        END.
    END.
    ELSE DO:
        WAIT-FOR CHOOSE OF bClose IN FRAME OrdlineFrame.
    END.

    PROCEDURE destroy-query:
        MESSAGE "Exiting order lines for order"
                STRING(orderbuf.order-num) "...".
(3)     DELETE PROCEDURE THIS-PROCEDURE NO-ERROR.
        DELETE WIDGET-POOL.
    END.

    PROCEDURE update-order-line:
        DEFINE VARIABLE rid AS ROWID.
        DEFINE VARIABLE choice AS LOGICAL.
        DEFINE BUTTON bSave LABEL "Save Changes".
        DEFINE BUTTON bCancel LABEL "Cancel".

(4)     DEFINE FRAME UpdateFrame SKIP(.5)
            orderbuf.order-num COLON 12 VIEW-AS TEXT
                order-line.line-num VIEW-AS TEXT SKIP(1) SPACE(1)
            order-line.qty order-line.discount order-line.backorder
            SKIP(1)
            bSave bCancel
        WITH TITLE "Update Order Line" SIDE-LABELS VIEW-AS DIALOG-BOX.
```

p-perwn3.p *(3 of 4)*

```
        ON CHOOSE OF bSave IN FRAME UpdateFrame
           OR GO OF FRAME UpdateFrame DO:
            MESSAGE "Are you sure you want to save your changes?"
                VIEW-AS ALERT-BOX QUESTION BUTTONS YES-NO
                UPDATE choice AS LOGICAL.
            CASE choice:
(5)             WHEN TRUE THEN DO TRANSACTION ON ERROR UNDO, RETRY:
                    rid = ROWID(order-line).
                    FIND order-line WHERE ROWID(order-line)
                        = rid EXCLUSIVE-LOCK.
                    ASSIGN
                        order-line.qty
                        order-line.discount
                        order-line.backorder
                        order-line.extended-price = INPUT order-line.qty
                            * order-line.price
                            * (1 - (INPUT order-line.discount * 0.01))
                    .
                    DISPLAY
                        order-line.qty order-line.discount
                        order-line.backorder order-line.extended-price
                    WITH BROWSE ordlineb.
                    APPLY "WINDOW-CLOSE" TO FRAME UpdateFrame.
                END.
                WHEN FALSE THEN DO:
                   MESSAGE "Changes not saved."
                       VIEW-AS ALERT-BOX INFORMATION BUTTONS OK.
                   RETURN NO-APPLY.
                END.
            END CASE.
        END.
```

p-perwn3.p *(4 of 4)*

```
        ON CHOOSE OF bCancel DO:
            MESSAGE
                "Are you sure you want to cancel your current updates?"
                VIEW-AS ALERT-BOX QUESTION BUTTONS YES-NO
                UPDATE choice AS LOGICAL.
            CASE choice:
                WHEN TRUE THEN DO:
                   MESSAGE "Current updates cancelled."
                       VIEW-AS ALERT-BOX INFORMATION BUTTONS OK.
                   APPLY "WINDOW-CLOSE" TO FRAME UpdateFrame.
                END.
                WHEN FALSE THEN DO:
                   MESSAGE "Current update is continuing ..."
                       VIEW-AS ALERT-BOX INFORMATION BUTTONS OK.
                END.
            END CASE.
        END.

(6)     DISPLAY
            orderbuf.order-num order-line.line-num order-line.qty
            order-line.discount order-line.backorder
        WITH FRAME UpdateFrame IN WINDOW ACTIVE-WINDOW.
        ENABLE
            order-line.qty order-line.discount order-line.backorder
            bSave bCancel
        WITH FRAME UpdateFrame IN WINDOW ACTIVE-WINDOW.

        WAIT-FOR WINDOW-CLOSE OF FRAME UpdateFrame.
    END PROCEDURE.
```

To help the user manage their screen, the application organizes the windows for orders and order lines into a window family, with the customer browse window as the root (`p-perwn1.p` (1), `p-perwn2.p` (3), `p-perwn2.p` (1), and `p-perwn3.p` (2)). Thus, the user can minimize the entire application or any order along with all of its child windows for order lines.

Another feature of this application is the use of dialog boxes to provide modal functionality in an otherwise non-modal, multi-window application (`p-perwn2.p` (7) and `p-perwn3.p` (4)). The `p-perwn2.p` and `p-perwn3.p` procedures each use a dialog box to update the order and order–lines table, respectively. This can help to control transaction size during an update. In these examples, the transaction size is limited to a single trigger block (`p-perwn2.p` (8) or `p-perwn3.p` (5)). So, the update windows in this case could also be persistently managed non-modal windows without affecting the scope of transactions.

Note also that these dialog boxes are parented to the window specified by the ACTIVE–WINDOW system handle (for example, `p-perwn3.p` (6)). This handle specifies the window that has received the most recent input focus in the application. Using this handle guarantees that the dialog box appears in the Progress window where the user is working, even if it is not the current window of the procedure that displays the dialog box. For more information on the ACTIVE–WINDOW handle, see Chapter 25, "Interface Design."

NOTE: This example updates the qty and extended–price field in the order–line table. However, it does **not** update the corresponding balance field in the customer table, or the on–hand and allocated fields in the item table. You might want to add the necessary update code to maintain your working sports database. For example, you could update the customer balance field by passing the procedure handle of `p-perwn1.p` down to `p-perwn3.p` and calling an internal procedure in `p-perwn1.p` that updates the balance field in the customer record specified by orderbuf.cust–num (in `p-perwn3.p`). Calling an internal procedure in `p-perwn1.p` keeps the customer table and browse management all together in `p-perwn1.p`.

22

Menus

Progress supports two kinds of menus—menu bars and pop-up menus. You can define either type of menu statically or create them dynamically.

A *menu bar* is a horizontal bar displayed at the top of a window. The menu bar contains menu titles (submenus) arranged horizontally. When you select a menu title, a pull-down menu containing a vertically arranged list of items is displayed.

A *pop-up menu* is a menu that contains items arranged vertically and it is associated with a widget. The pop-up menu is context sensitive and appears only when the user performs a particular mouse or keyboard action while the widget has focus.

22.1 Menu Types

A *menu* is a widget that contains a list of commands or functions that users frequently use. The two kinds of static and dynamic menu widgets available are menu bars and pop-up menus.

22.1.1 Menu Bar

A *menu bar* is associated with a window. Each Progress window can have only one menu bar, and that menu bar contains menu items know as *menu titles*. When the user selects a menu title, a *pull-down menu* with one or more items may appear directly below the menu bar. Depending on the application, *nested pull-down menus* may appear to the sides of menu items when you select them.

Figure 22–1 shows a menu bar with two menu titles. When you select Edit, a pull-down menu with three menu items appears below Edit. When you further choose the menu item Add, a nested submenu with four menu items appears to the side of Add. The Static Menu section describes how to select menu items using the mouse and the keyboard.

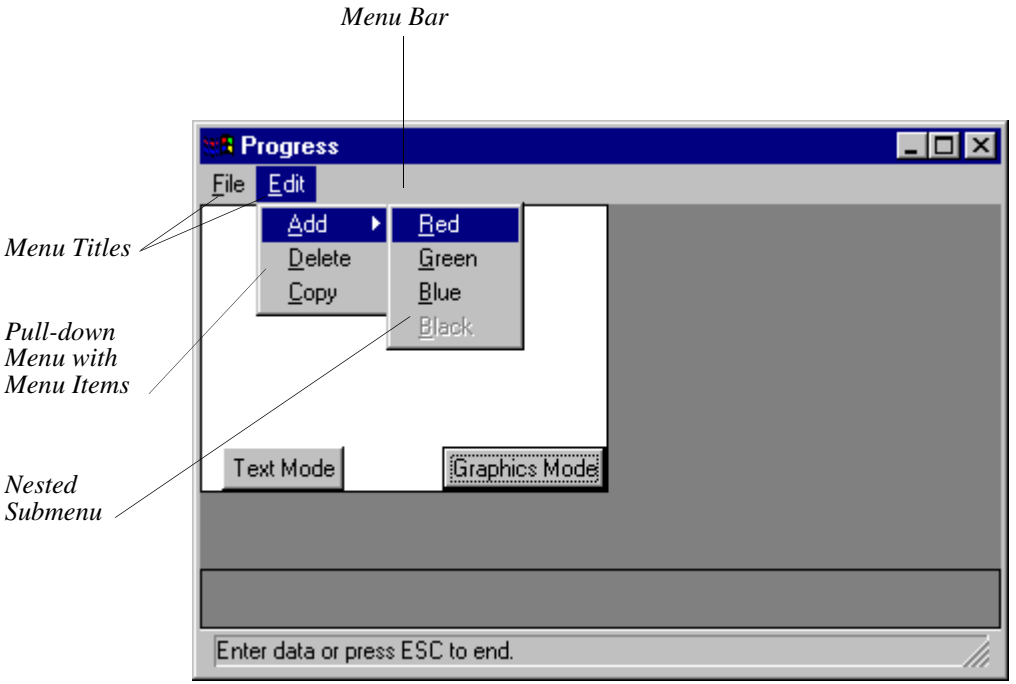


Figure 22–1: Window with a Menu Bar

In a graphical interface, you can use the mouse to pull down menus from the menu bar and choose menu items. In a character interface, use the ENTER–MENUBAR key function (usually mapped to **F3**) to give focus to the menu bar. You can then use the left and right arrow keys to position to a menu title. Press **RETURN** to pull down the menu and the up and down arrow keys to position to a menu item. Press **RETURN** again to choose the menu item.

22.1.2 Pop-up Menu

A *pop-up menu* is associated with a widget. The pop-up menu can contain one or more items as well as nested submenus. Figure 22–2 shows two button widgets. When you choose the button Hi, a pop-up menu that contains five choosable menu items appears.

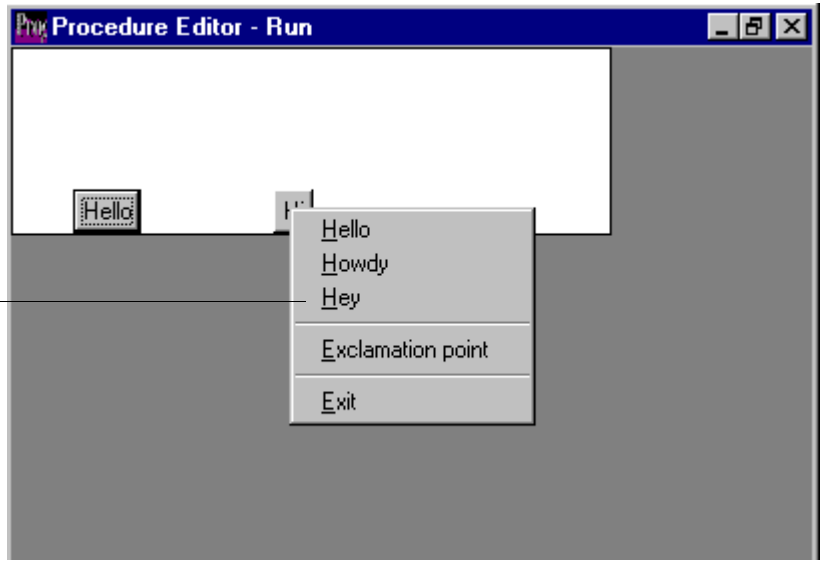


Figure 22–2: Button with a Pop-up Menu

In a graphical interface, you can press and hold the mouse MENU button to pop up a menu by default or use **SHIFT–F10**. To choose an item, move the mouse pointer to the item and click the SELECT button. You can define a different mouse button to use by setting the MENU–MOUSE attribute of the menu.

In a character interface, use the key associated with the DEFAULT–POP–UP key function (usually **ESC–U**; **F4** on Windows). Use the up and down arrow keys to move within the menu, and press **RETURN** to choose a menu item. You can use this keyboard method in both character and graphical interfaces.

22.2 Menu Hierarchy

Progress allows you to generate static and dynamic menus according to the requirements of your application. If you know up-front the kinds of menus you need and the exact number of menu items and submenus in your menus, you can generate static menus. Conversely, if you don't know how many menus you need or the number of items your menu will contain, or even what is in your menus, you can create dynamic menus.

Whether you generate static or dynamic menus, the relationships remain the same between menus, their owners, and their child widgets. Figure 22–3 shows the *menu hierarchy*.

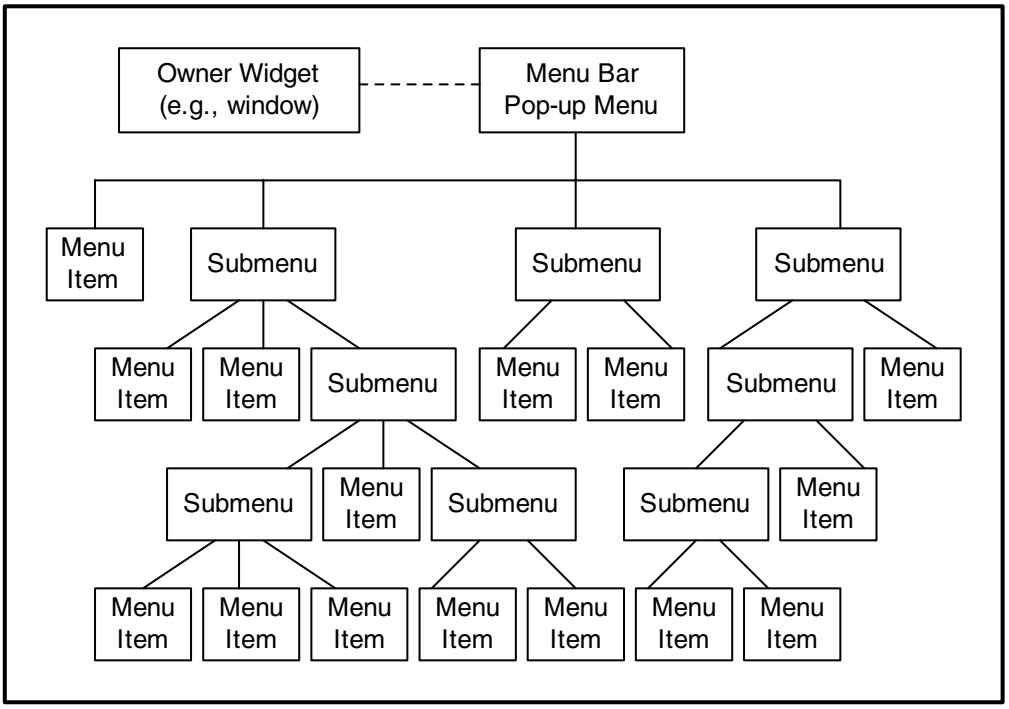


Figure 22–3: Menu Hierarchy

At the top of the menu hierarchy is the menu bar or pop-up menu. Menus are the *parents* of menu items and/or submenus. Each child submenu can, in turn, be the parent of menu items and submenus. Your application can have multiple levels of nested submenus.

Note that menus themselves do not have parents; instead, they have *owners*. All widgets, except images, rectangles, text, labels, menus, menu-items, and submenus, can be menu owners. Windows are the only widgets that can own a menu bar. All the other widgets can own only a pop-up menu.

22.3 Features for Menu Widgets

You can add the following features to your static and dynamic menus:

- Nested submenus
- Duplicate menus
- Accelerators
- Menu mnemonics
- Enabling and disabling characteristics
- Toggle-box items

Detailed descriptions of these features appear later in this chapter.

22.4 Defining Menus

This section describes static menus, how to define them, and how to add features (such as accelerators, mnemonics, etc.) to them.

22.4.1 Static Menu Bars

The following sections describe how to set up a menu bar. They contain sample code that generates the following menu bar.



Setting Up a Static Menu Bar

Setting up a static menu bar is a three-part process:

1. Define the pull-down submenus.
2. Define the menu bar itself.
3. Assign the menu bar to a window.

The following sections describe this process in greater detail.

Defining Pull-down Submenus

To generate a submenu, you use the DEFINE SUB–MENU statement. Each submenu must have a unique name. Use the MENU–ITEM phrase to specify menu items, and use the LABEL option to define the text for the menu item. If you omit LABEL, Progress displays the item handle name by default. Progress lays out the menu items consecutively in a top-to-bottom order. The following code fragment defines three submenus and their menu items.

```
DEFINE SUB-MENU topic
  MENU-ITEM numbr    LABEL "Cust-num"
  MENU-ITEM addr     LABEL "Address"
  MENU-ITEM othrinfo LABEL "Other".

DEFINE SUB-MENU move
  MENU-ITEM forward  LABEL "Next" ACCELERATOR "PAGE-DOWN"
  MENU-ITEM backward LABEL "Prev" ACCELERATOR "UP".

DEFINE SUB-MENU quitit
  MENU-ITEM quit LABEL "E&xit".
```

The ACCELERATOR option specifies a key or key combination that the user can use to select a menu item without having to pull down the menu. For more information on accelerators, see the "Menu Item Accelerators" section.

The ampersand (&) before the letter x in "E&xit" causes x to be underlined when the menu is displayed. The letter x is called a mnemonic. A mnemonic provides a way to access menu items from the keyboard. For more information on mnemonics, see the "Menu Mnemonics" section.

To generate a help submenu, add the SUB–MENU–HELP phrase to the DEFINE SUB–MENU statement, as shown below.

```
DEFINE SUB-MENU userhelp SUB-MENU-HELP
  MENU-ITEM message    LABEL "Recent Messages"
  MENU-ITEM keybd      LABEL "Keyboard"
  MENU-ITEM command    LABEL "Menu Commands".
```

For more information on the DEFINE SUB–MENU statement, see the *Progress Language Reference*.

Defining a Menu Bar

To generate a menu bar, you use the DEFINE MENU statement. You must explicitly specify the MENUBAR phrase. You can add menu items and submenus to the menu bar.

In the following code, you assign the three previously defined submenus to the menu bar with the SUB–MENU phrase. You use the LABEL option to define the text for the submenu. Progress lays out the submenus consecutively in left-to-right order on the menu bar.

```
DEFINE MENU mbar MENUBAR
  SUB-MENU topic    LABEL "Topic"
  SUB-MENU move     LABEL "Move"
  SUB-MENU quitit   LABEL "E&xit"
  SUB-MENU userhelp LABEL "Help".
```

NOTE: You can add submenus or menu items to existing dynamic menu structures at any time. Progress appends the newly added widgets to the end of the list. See the "Dynamic Menus" section.

For more information on the DEFINE MENU statement, see the *Progress Language Reference*.

Assigning a Menu Bar to a Window

Now that you have set up a menu bar, you need to assign it to a window. You do so by setting the window's MENUBAR attribute equal to the handle of the menu. For example, if you are using the current window or default window, you can refer to it by using the CURRENT–WINDOW or DEFAULT–WINDOW system handle.

```
ASSIGN DEFAULT-WINDOW:MENUBAR  = MENU mbar:HANDLE.
```

If you are using a window that your application has created, you can refer to the window's widget handle. See the following code fragment. At the end of the code, delete the window with the DELETE WIDGET statement.

```
DEFINE VARIABLE mwin AS WIDGET-HANDLE.
          .
          .
          .
CREATE WINDOW mwin
  ASSIGN MENUBAR = MENUmbar:HANDLE.
          .
          .
          .
DELETE WIDGET mwin.
```

Example of a Menu Bar with Pull-down Submenus

The p-bar.p procedure that follows contains most of the sample definition codes used in the previous sections. The procedure defines a menu bar, mbar, which contains three pull-down submenus. The handle of mbar is assigned to the current window. The ON statements define triggers that execute when you select the corresponding menu items.

p-bar.p *(1 of 2)*

```
DEFINE SUB-MENU topic
    MENU-ITEM numbr  LABEL "Cust. Number"
    MENU-ITEM addr LABEL "Address"
    MENU-ITEM othrinfo LABEL "Other".

DEFINE SUB-MENU move
    MENU-ITEM forward    LABEL "NextRec" ACCELERATOR "PAGE-DOWN"
    MENU-ITEM backward    LABEL "PrevRec" ACCELERATOR "PAGE-UP".

DEFINE SUB-MENU quitit
    MENU-ITEM quititem LABEL "E&xit".

DEFINE MENU mbar     MENUBAR
    SUB-MENU topic   LABEL "Topic"
    SUB-MENU move    LABEL "Move"
    SUB-MENU quitit  LABEL "E&xit".

FIND FIRST customer.
DISPLAY customer.name LABEL "Customer Name" WITH FRAME name-frame.

ON CHOOSE OF MENU-ITEM numbr
    DISPLAY customer.cust-num WITH FRAME num-frame ROW 6.

ON CHOOSE OF MENU-ITEM addr
    DISPLAY customer.address customer.address2 customer.city
       customer.state customer.country customer.postal-code
       WITH FRAME addr-frame NO-LABELS COLUMN 25 ROW 6.

ON CHOOSE OF MENU-ITEM othrinfo
    DISPLAY customer EXCEPT name cust-num address
     address2 city state country postal-code
     WITH FRAME oth-frame SIDE-LABELS ROW 11.
```

p-bar.p *(2 of 2)*

```
ON CHOOSE OF MENU-ITEM forward
  DO:
     HIDE ALL NO-PAUSE.
     CLEAR FRAME name-frame.
     FIND NEXT customer NO-ERROR.
     IF AVAILABLE(customer)
     THEN DISPLAY customer.name WITH FRAME name-frame.
  END.

ON CHOOSE OF MENU-ITEM backward
  DO:
     HIDE ALL NO-PAUSE.
     CLEAR FRAME name-frame.
     FIND PREV customer NO-ERROR.
     IF AVAILABLE(customer)
     THEN DISPLAY customer.name WITH FRAME name-frame.
  END.

ASSIGN CURRENT-WINDOW:MENUBAR = MENU mbar:HANDLE.

WAIT-FOR CHOOSE OF MENU-ITEM quititem.
```

When you run this code, you see a menu bar at the top of the window and a frame that displays the name of the first customer. The menu bar contains three titles: Topic, Move, and Exit. You can select each title, one at a time, and pull down its menu to select a menu item.

The Exit menu has only one item. When you choose that item, the procedure ends.

22.4.2 Static Pop-up Menus

You set up a static pop-up menu by using the DEFINE MENU statement. By default, if you don't specify the MENUBAR phrase, Progress treats this widget as a pop-up menu. To associate a pop-up menu with a widget, you assign the handle of the menu to the widget's POPUP–MENU attribute.

The p-popup.p procedure displays a button that has an associated pop-up menu:

p-popup.p

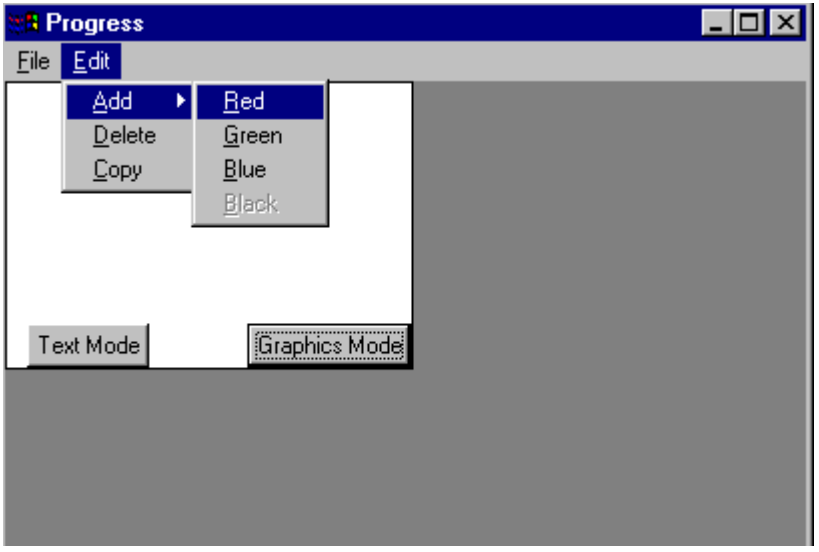
```
DEFINE BUTTON hi     LABEL "Hello".

DEFINE MENU popmenu TITLE "Button State"
   MENU-ITEM ve      LABEL "Hello"
   MENU-ITEM vd      LABEL "Howdy"
   MENU-ITEM iv      LABEL "Hey"
   RULE
   MENU-ITEM ep      LABEL "Exclamation point" TOGGLE-BOX
   RULE
   MENU-ITEM ex      LABEL "Exit".

FORM
   hi  AT ROW 4 COLUMN 5
     WITH FRAME button-frame.

/* Set popmenu to be the pop-up menu for hi. */
ASSIGN hi:POPUP-MENU = MENU popmenu:HANDLE.

/* Define action for menu selections. */
ON CHOOSE OF MENU-ITEM ve, MENU-ITEM vd, MENU-ITEM iv
   ASSIGN hi:LABEL IN FRAME button-frame = SELF:LABEL.

/* Define action for button selection. When the button is
   selected, display the current button label as a message.
   If Exclamation Point is checked, add an exclamation point
   to the message; otherwise, add a period.                  */
ON CHOOSE OF hi
   MESSAGE hi:LABEL IN FRAME button-frame +
           (IF MENU-ITEM ep:CHECKED IN MENU popmenu THEN "!" ELSE ".").

/* Enable input on the button and wait for the
   user to select Exit from menu.              */
ENABLE hi WITH FRAME button-frame.

WAIT-FOR CHOOSE OF MENU-ITEM ex.
```

When you run this code, a button with a label Hello appears on the screen. When you pop up the associated menu and choose any of the first three menu items—Hello, Howdy, or Hey—the label of the button changes accordingly to the value of the chosen item. Subsequently, when you choose the button itself, the current value of the button's label displays as a message at the bottom of the window. The message ends in either a period or an exclamation point, depending on the current state of the ep toggle-box menu item. See the "Menu Toggle Boxes" section for more information.

22.5 Nested Submenus

When the user selects a menu item assigned with *nested submenus*, a pull-down submenu appears next to it. Although your pull-down and pop-up menus can contain many levels of nested submenus, you should limit them to three or less. Multiple levels of nested submenus tend to clutter up the screen and can make it difficult for the user to access menu items.

The following screen shows a nested submenu. When you select the Add item (in the pull-down menu of Edit), you can, while holding down the mouse SELECT button, pull another submenu to the side. On Windows, submenus appear automatically as you drag down in the containing menu.



The following code fragment defines the menus shown in the previous screen. The code defines a menu bar, mybar, that contains three submenus: myfile, mycolors, and myedit. The submenu mycolors (1) is nested within the myedit submenu (2).

```
DEFINE SUB-MENU myfile
  MENU-ITEM m1 LABEL "Save"
  MENU-ITEM m2 LABEL "Save &As"
  MENU-ITEM m3 LABEL "E&xit".

DEFINE SUB-MENU mycolors
  MENU-ITEM m1 LABEL "Red"
  MENU-ITEM m2 LABEL "Green"
  MENU-ITEM m3 LABEL "Blue"
  MENU-ITEM m4 LABEL "Black".

DEFINE SUB-MENU myedit
  SUB-MENU mycolors LABEL "Add"
  MENU-ITEM e1      LABEL "Delete"
  MENU-ITEM e2      LABEL "Copy".

DEFINE MENU mybar MENUBAR
  SUB-MENU myfile LABEL "File"
  SUB-MENU myedit LABEL "Edit".
```

Submenu mycolors nested in submenu myedit

22.6 Enabling and Disabling Menu Items

You can selectively enable and disable menu items in your menus. When an item is enabled, it is a valid selection. However, when an item is disabled, it appears grayed out and it cannot be selected. For example, in the following code, when you select button b1 labeled “Text Mode,” the first three items (which correspond to graphic commands) are disabled; the only item enabled is the Text command. Conversely, when you select button b2 labeled “Graphics Mode,” only the items for the graphic commands are enabled.

NOTE: The `p-menu.p` example produces a run-time warning in character interfaces because it cannot create a dynamic window.

p-menu.p

```
DEFINE VARIABLE mywin AS WIDGET-HANDLE.

DEFINE SUB-MENU myfile
   MENU-ITEM m1  LABEL "Save"
   MENU-ITEM m2  LABEL "Save &As"
   MENU-ITEM m3  LABEL "E&xit".
DEFINE SUB-MENU mycolors
   MENU-ITEM m1  LABEL "Red"
   MENU-ITEM m2  LABEL "Green"
   MENU-ITEM m3  LABEL "Blue"
   MENU-ITEM m4  LABEL "Black".

DEFINE SUB-MENU myedit
   SUB-MENU mycolors  LABEL "Add"
   MENU-ITEM e1       LABEL "Delete"
   MENU-ITEM e2       LABEL "Copy".
DEFINE MENU mybar  MENUBAR
   SUB-MENU myfile LABEL "File"
   SUB-MENU myedit LABEL "Edit".
DEFINE BUTTON b1 LABEL "Text Mode".
DEFINE BUTTON b2 LABEL "Graphics Mode".
FORM
   b1 at X 10  Y 120
   b2 at x 120  Y 120
   WITH FRAME x.

ON CHOOSE OF b1 IN FRAME x DO:
   MENU-ITEM m1:SENSITIVE IN MENU mycolors = NO.
   MENU-ITEM m2:SENSITIVE IN MENU mycolors = NO.
   MENU-ITEM m3:SENSITIVE IN MENU mycolors = NO.
   MENU-ITEM m4:SENSITIVE IN MENU mycolors = YES.
END.

ON CHOOSE OF b2 IN FRAME x DO:
   MENU-ITEM m1:SENSITIVE IN MENU mycolors = YES.
   MENU-ITEM m2:SENSITIVE IN MENU mycolors = YES.
   MENU-ITEM m3:SENSITIVE IN MENU mycolors = YES.

   MENU-ITEM m4:SENSITIVE IN MENU mycolors = NO.
END.

CREATE WINDOW mywin
   ASSIGN MENUBAR = MENU mybar:HANDLE.
CURRENT-WINDOW = mywin.

ENABLE b1 b2 WITH FRAME x.
APPLY "CHOOSE" TO b1.

WAIT-FOR CHOOSE OF MENU-ITEM m3 IN MENU myfile.

DELETE WIDGET mywin.
```

Menu item names can be identical in different submenus

All menu items are disabled except the last one

If menu item names are non-unique, use IN MENU clause to identify the parent menu

Note that the items in myfile and mycolors have identical names. Progress allows you to use non-unique names for menu items as long as they belong to different submenus. To avoid ambiguity when you subsequently refer to these items, use the IN MENU clause to identify the parent menu of the item or the IN SUB–MENU clause to identify the parent submenu of the item. The IN MENU or IN SUB–MENU clause is not necessary if the name of the menu item is unique.

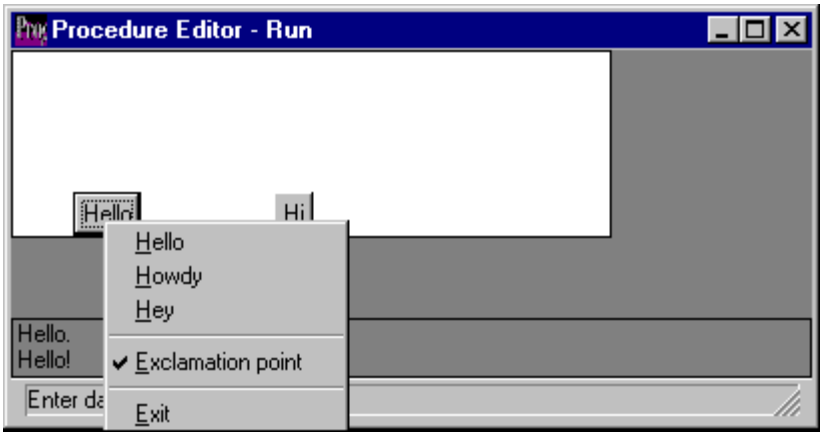
When you run this procedure, a window containing a menu bar and two buttons appears. If you choose the Text Mode button and subsequently pull down the Edit menu, you see that only the last item in the menu, Black, is enabled. The rest of the items (Red, Green, and Blue) associated with the Graphics Mode button are grayed out. They are enabled when you select the Graphics Mode button prior to accessing the Edit menu.

NOTE: There are situations where Progress cannot disambiguate a menu item. One is where the same menu item occurs in a submenu that itself is duplicated in two menus, and where the second menu also contains the same menu item. In that case Progress can only reference the menu item in the first copy of the submenu. This is because Progress cannot distinguish the three identical menu items contained in the first and second copies of the submenu and also in the second menu.

22.7 Menu Toggle Boxes

You can specify the TOGGLE–BOX option for selectable menu items in your pull-down menus and pop-up menus. When you use this option, the user can alternately check and uncheck the associated menu item by choosing it. The application can examine the CHECKED attribute for the item to determine whether the item is currently checked or unchecked. You can also initialize or change the condition of the menu item by setting the CHECKED attribute in the program.

Depending on the interface, the state of the menu item may be shown visually to the user. For example, on Windows, when you select the item Exclamation Point in `p-popup.p`, a check mark appears, as shown in the following screen. When you subsequently choose the button, the message ends with either an exclamation point or a period, depending on whether the Exclamation Point item is highlighted.



When the user toggles a toggle-box item, Progress sends the VALUE–CHANGED event to the menu item. Therefore, by defining a trigger on that event, you can take immediate action when the user toggles the value.

22.8 Duplicating Menus

Once a menu or submenu is defined, you can make a duplicate of it and use it in another submenu that you are defining with the LIKE phrase. This duplicate inherits all the properties of the original menu—such as name, menu items, and definitional triggers. To avoid ambiguity when you subsequently refer to menu items either in the original menu or in its duplicate, use the IN MENU clause.

The following code defines two buttons, each with a pop-up menu. The second pop-up menu is defined to be like the first one and thereby inherits the former's characteristics.

p-popup2.p

```
DEFINE BUTTON hi1    LABEL "Hello".
DEFINE BUTTON hi2    LABEL "Hi".
DEFINE MENU popmenu1 TITLE "Button1 State"
   MENU-ITEM ve      LABEL "Hello"
   MENU-ITEM vd      LABEL "Howdy"
   MENU-ITEM iv      LABEL "Hey"
   RULE
   MENU-ITEM ep      LABEL "Exclamation point" TOGGLE-BOX
   RULE
   MENU-ITEM ex      LABEL "Exit".

DEFINE MENU popmenu2 TITLE "Button2 State" LIKE popmenu1.
 FORM
   hi1          AT x 30   Y 70
   hi2          AT x 130  Y 70
   WITH FRAME button-frame WIDTH 30.

/*Set popmenu1 and popmenu2 to be the pop-up menus for hi1 and hi2.*/
ASSIGN hi1:POPUP-MENU  = MENU popmenu1:HANDLE
       hi2:POPUP-MENU  = MENU popmenu2:HANDLE.

/* Define action for menu selections. */
ON CHOOSE OF MENU-ITEM ve IN MENU popmenu1, MENU-ITEM vd IN MENU
popmenu1, MENU-ITEM iv IN MENU popmenu1
   ASSIGN hi1:LABEL IN FRAME button-frame = SELF:LABEL.

ON CHOOSE OF MENU-ITEM ve IN MENU popmenu2, MENU-ITEM vd IN MENU
popmenu2, MENU-ITEM iv IN MENU popmenu2
   ASSIGN hi2:LABEL IN FRAME button-frame = SELF:LABEL.

/* Define action for button selection. When the button is
   selected, display the current button label as a message.
   If Exclamation Point is checked, add an exclamation point
   to the message; otherwise, add a period.                    */
ON CHOOSE OF hi1
   MESSAGE hi1:LABEL IN FRAME button-frame +
       (IF MENU-ITEM ep:CHECKED IN MENU popmenu1 THEN "!" ELSE ".").

ON CHOOSE OF hi2
   MESSAGE hi2:LABEL IN FRAME button-frame +
       (IF MENU-ITEM ep:CHECKED IN MENU popmenu2 THEN "!" ELSE ".").

/* Enable input on the button and wait
   for the user to select Exit from menu. */
ENABLE hi1 hi2 WITH FRAME button-frame.

WAIT-FOR CHOOSE OF MENU-ITEM ex IN MENU popmenu1,
                   MENU-ITEM ex IN MENU popmenu2.
```

The LIKE phrase specifies the name of the submenu you want to duplicate

The IN MENU phrase identifies the parent menu of the item

22.9 The MENU–DROP Event

The MENU–DROP event occurs every time you bring a pull-down menu or pop-up menu into view. You can write a trigger on the MENU–DROP event to enable or disable specific items within the menu.

NOTE: To trigger this event for a menu widget, the menu widget must be a pop-up menu. Otherwise, you can only trigger this event for a submenu widget.

CAUTION: Do not interact with the window manager from within a MENU–DROP trigger. Doing so causes the window manager to lose control of the system, forcing you to reboot or restart the window manager. Actions to avoid include any window system input or output. These include actions that can generate a warning or error message, forcing window system output. Use the NO–ERROR option on supported statements to help avoid this situation. Otherwise, check valid values, especially for run-time resources like widget handles, to prevent Progress from displaying unexpected messages.

22.10 Menu Item Accelerators

In both graphical and character interfaces, you can define an accelerator key for menu item. An *accelerator* is a key or key combination that executes an item from a pull-down menu without the user having to pull down the menu. Progress recognizes menu accelerators for the active window, except during modal interactions—that is, when the user is prompted by any type of alert box or dialog box.

When the user presses an accelerator to invoke a menu item, Progress generates the MENU–DROP event for the menu, but does not display the menu. Also, for a standard menu item Progress generates a CHOOSE event, and for a toggle box menu item Progress generates a VALUE–CHANGED event.

22.10.1 Defining Accelerators

You can define an accelerator for a menu item by adding the ACCELERATOR option to the menu item description.

This is the syntax for a menu accelerator:

SYNTAX

```
ACCELERATOR keylabel
```

In this syntax, the value *keylabel* must be a character-string constant that evaluates to a valid Progress key label. You can modify the *keylabel* by specifying one or more of these

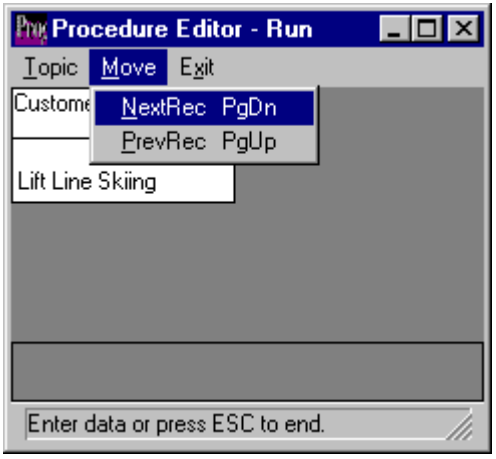
keys—**SHIFT**, **CTRL**, or **ALT**. For example, you can specify “ALT–F8", “PAGE–UP”, etc. When the user presses the specified key(s), the menu item is selected.

Note that Progress automatically adds the specified accelerator key to the menu item after the label. You should not put the accelerator into the label. For example, if you specify "CTRL–P," Progress adds CTRL+P to the menu item (CTRL–P in character interfaces).

The following code fragment specifies two accelerator keys, **CNTL–N** and **CNTL–P**.

```
DEFINE SUB-MENU move
  MENU-ITEM forward  LABEL "NextRec" ACCELERATOR "CNTL-N"
  MENU-ITEM backward LABEL "PrevRec" ACCELERATOR "CNTL-P".
```

Progress displays the menu items as follows:



NOTE: Progress does not support accelerators for pop-up menu items. If you define them, Progress ignores them.

22.10.2 Accelerators in Character Interfaces

For character interfaces, the choice of key labels is limited by those keys available on your keyboard. Thus, key labels are not necessarily portable between terminal types. For example, some terminals have no **F2** key, but might have a **PF2** key or require a **CTRL** key combination to emulate common **F2** key functionality.

Specifying Portable Accelerators

For portability, you can use the KBLABEL function to specify the accelerator key label.

```
MENU-ITEM miHelp LABEL "Help Menu" ACCELERATOR KBLABEL("HELP").
```

This allows you to use a portable Progress key function (such as HELP) to identify a valid key label for the current keyboard (terminal type). The key label is one that is associated with the key function for the terminal type in the current environment. On Windows, the current environment might reside in the registry or in an initialization file. On UNIX, the current environment resides in the PROTERMCAP file.

However, note that the current environment can (and often does) define more than one key label for a given Progress key function. The KBLABEL function returns the first such definition specified for the terminal type. For example, some terminal types define **F2** and also **ESC–?** as the HELP key function. In this case, using KBLABEL("HELP") changes **F2** to a menu item accelerator (losing its HELP key function), but leaves **ESC–?** as the one remaining HELP key function.

Handling Invalid Accelerators

There are two types of invalid menu accelerators:

- Those specified by using a key label that is unsupported by the terminal.
- Those specified with supported key labels, but when invoked, generate unrecognized keyboard codes.

If you specify an unsupported key label, say **ALT** in **ALT–F9**, Progress substitutes a supported key label in its place. Thus, Progress might substitute **ESC–F9** for **ALT–F9** as the accelerator definition.

However, an accelerator like **ESC–F9** might not work at run time. In this case, **ESC–F9** generates unrecognizable keyboard code sequences that fail to fire the appropriate Progress event. The terminal might also indicate the failure with a beep or other warning signal.

When in doubt, specify the menu accelerator using the terminal-portable KBLABEL function described earlier.

22.11 Menu Mnemonics

For quick access, the user can select an item from a pull-down menu or menu bar by pressing **ALT** and one mnemonic character. Progress indicates the mnemonic character by underlining it within the menu. For example, a menu bar might contain the entries File, Edit, and Exit. This means that you can access the File menu by pressing **ALT–F**, the Edit menu by pressing **ALT–E**, and the Exit menu by pressing **ALT–X**.

NOTE: If you define a menu mnemonic key combination as an accelerator, the accelerator takes precedence. For more information on key precedence, see Chapter 6, "Handling User Input."

In Progress, you can define mnemonics for items in a menu bar, pull-down menu, or pop-up menu. To specify a mnemonic, insert an ampersand (&) before that letter in the label. For example, to make x the mnemonic for Exit, you specify LABEL "E&xit". Note that in graphical and character interfaces, the first character in an item's label is the default mnemonic. You can only specify one mnemonic for each label; do not precede more than one letter with an ampersand.

To include a literal ampersand within a menu label, use two ampersands. For example, the label `"Undo && Restart"` is displayed as `Undo & Restart`.

22.12 Dynamic Menus

Dynamic menus are similar to static menus in several ways:

- They both share the same hierarchy of relationship to owners and descendents.
- You can also add features such as toggle boxes, accelerators, mnemonics—all of which were discussed in the Static Menu section—to dynamic menus.

Dynamic menus are different from static menus in the following ways:

- Dynamic menus are created during run time. Your application can create as many menu items as needed or can add menu items to existing static or dynamic menus. In contrast, static menus and menu items are fixed in quantity at compile time.
- The syntax you use to generate dynamic menus, as well as to assign characteristics, are different. Recall that to generate static menus, you use the DEFINE MENU and DEFINE SUB–MENU statements. For dynamic menus, you use the CREATE MENU, CREATE SUB–MENU, and CREATE MENU–ITEM statements.
- To delete a dynamic widget, you use the DELETE WIDGET statement. When the application deletes a dynamic widget, Progress automatically deletes all its children or descendents.

22.12.1 Properties of Dynamic Menus

As discussed in the first part of this chapter, a menu bar has a window for an owner, while a pop-up menu has an associated widget. The owner widget has a readable and setable MENUBAR or POPUP–MENU attribute that allows you to set or query the ownership status of a widget. The menu widget has a read-only OWNER attribute that returns the widget handle of the owner widget. You can reset the MENUBAR or POPUP–MENU attribute to a different menu widget handle at any point in your application. Progress automatically replaces the first menu with the second after you reset that attribute.

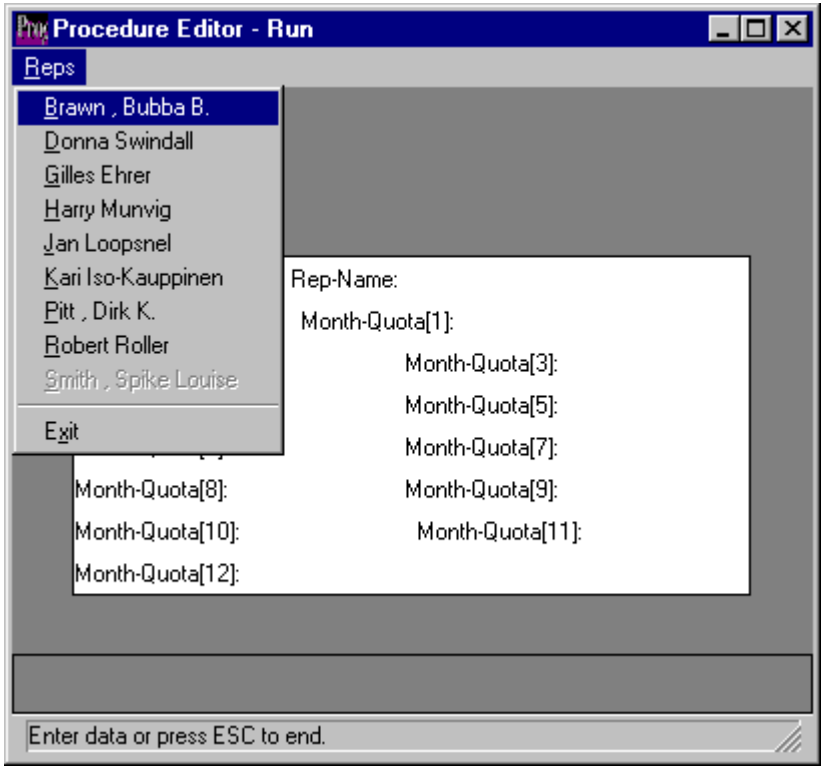
If your application deletes a menu widget, Progress finds the owner widget that refers to it and sets its ownership attribute to unknown (?). Progress then automatically deletes all the children or descendents of that menu widget.

Like all dynamic widgets, the menu widgets go into a widget pool. You can specify a widget pool with the IN WIDGET–POOL clause of the CREATE statement. If you do not specify a widget pool, the menu widget is created in the default unnamed widget pool. Should you, however, specify a widget pool that does not exist, Progress stops its execution of the procedure. The menu widget exists until you explicitly delete it with the DELETE WIDGET statement or until the widget pool is deleted.

You cannot set the geometry of menu, submenu, and menu item widgets; that is, you cannot set characteristics such as X, Y, ROW, COLUMN, WIDTH, HEIGHT, HEIGHT–ROWS, and WIDTH–COLUMNS. These characteristics are all fixed and set by Progress.

22.12.2 Dynamic Menu Bar

The following sections contain steps on how to set up a dynamic menu bar, as well as a sample code that generates the following menu bar and pull-down menu.



Setting Up a Dynamic Menu Bar

Setting up a dynamic menu bar is a three-part process:

1. Create the menu bar and assign it to a window.
2. Create submenus and attach them as children to the parent menu or to another submenu.
3. Create menu items and attach them to the submenu(s).

The following sections describe this process in detail.

Creating the Menu Bar

Recall that when you define a static menu widget (for example, a menu bar) you use the DEFINE MENU statement and you must specify the MENUBAR attribute. Without the MENUBAR attribute, Progress automatically treats the widget as a pop-up menu. For a dynamic menu, the POPUP–ONLY attribute determines whether it is a menu bar or a pop-up menu. To create a dynamic menu bar, you must do the following:

1 ♦ Define a variable for the menu as a widget–handle.

2 ♦ Use the CREATE MENU statement to create a menu widget. The POPUP–ONLY attribute defaults to FALSE, so the menu is treated as a menu bar.

3 ♦ To display the menu bar, attach it to a window. Set the MENUBAR attribute of the window equal to the widget handle of the menu.

The following code fragment defines a widget-handle variable, creates a dynamic menu, and sets the MENUBAR attribute of the current window to main–bar–ptr.

```
DEFINE VARIABLE main-bar-ptr  AS WIDGET-HANDLE.
        .
        .
        .
CREATE MENU main-bar-ptr.

CURRENT-WINDOW:MENUBAR = main-bar-ptr.
```

Creating Submenus

To create a submenu, you use the CREATE SUB–MENU statement. Each submenu must have a unique label. Use the ASSIGN clause of the CREATE SUB–MENU statement to set the PARENT attribute equal to the widget handle of the menu or another submenu. You cannot change the parent of submenus or menu items once you make the assignment. The only way you can change the parent/child relationship is by deleting the child. Use the LABEL option to define the text for the submenu.

Progress lays out dynamic menu widgets the same way it does static menu widgets. Submenus are laid out consecutively in a left-to-right order on the menu bar; menu items are laid out consecutively in a top-to-bottom order. If the application deletes a dynamic submenu or menu item widget, Progress automatically shifts remaining widgets up or to the left. For example, when a menu item is deleted, the remaining items are shifted up to fill the previously occupied space. You can add submenus or menu items to existing menu structures at any time. Progress appends the newly added widgets to the end of the list.

The following code fragment creates a submenu srep–men–ptr and sets the parent attribute to main–bar–ptr. The label of the submenu is Reps.

```
CREATE SUB-MENU srep-men-ptr
  ASSIGN PARENT = main-bar-ptr
  LABEL = "Reps".
```

Creating Menu Items

Recall that when you create a menu item for a static menu widget, you use the MENU–ITEM phrase to specify the menu item. However, for a dynamic menu, you must first define the widget-handle variables, then use the CREATE MENU–ITEM statement to create each individual menu item. Each menu item can be used in only one menu or submenu. Use the ASSIGN clause of the CREATE MENU–ITEM statement to set the PARENT attribute of the menu item equal to the widget handle of the menu or submenu. Use the LABEL option to define the text for the menu item.

The following code fragment creates a menu item temp–hand–ptr, which is set to the parent attribute of a previously defined submenu srep–menu–ptr. The label of the menu item is salesrep.rep–name.

```
CREATE MENU-ITEM  temp-hand-ptr
  ASSIGN PARENT = srep-men-ptr
    LABEL = salesrep.rep-name
```

Each MENU–ITEM widget can be one of four different subtypes:

- **NORMAL** — A choosable menu item or toggle box item
- **READ–ONLY** — A read-only menu item
- **SKIP** — A skip menu item
- **RULE** — A rule menu item

The default subtype is NORMAL, and only the first two subtypes—NORMAL and READ–ONLY—have labels.

You must set the subtype before Progress realizes the menu item. Enter the subtype in uppercase and in quotes, as shown in the following code fragment.

```
CREATE MENU-ITEM  temp-hand-ptr
    ASSIGN SUBTYPE = "RULE"
        PARENT = srep-men-ptr.
```

22.12.3 Querying Sibling Attributes

For both menus and submenus, your code can traverse all the menu items and nested submenus. You query the FIRST–CHILD attribute of a menu or submenu, which returns the widget handle of the first menu item or child submenu. The code then follows the NEXT–SIBLING attribute to the right, etc. You can achieve the same result by starting with the LAST–CHILD attribute and follow the PREV–SIBLING attribute to the left. All these four attributes are read-only, while the PARENT attribute is read-write. The `p-dymenu.p` procedure traverses the FIRST–CHILD and NEXT–SIBLING attributes of the Reps menu and disables the menu item for each sales rep in the West region.

22.12.4 Example of a Dynamic Menu

The following procedure creates a menu bar with a pull-down menu that contains dynamically created menu items of sales representatives' names in the salesrep table:

p-dymenu.p *(1 of 2)*

```
DEFINE VARIABLE exit-item-ptr    AS WIDGET-HANDLE.
DEFINE VARIABLE srep-menu-ptr    AS WIDGET-HANDLE.
DEFINE VARIABLE main-bar-ptr     AS WIDGET-HANDLE.
DEFINE VARIABLE temp-hand-ptr    AS WIDGET-HANDLE.

FORM
   salesrep.sales-rep rep-name salesrep.region month-quota
   WITH FRAME x WITH SIDE-LABELS ROW 5 CENTERED.

VIEW FRAME x.

/* Create the main menu bar. */
CREATE MENU main-bar-ptr.

/* Create a pull-down menu to list all sales reps. */
CREATE SUB-MENU srep-menu-ptr
     ASSIGN PARENT = main-bar-ptr
            LABEL = "Reps".

/* Create a menu item for each record in the Salesrep table. */
FOR EACH Salesrep BY rep-name:
   CREATE MENU-ITEM temp-hand-ptr
      ASSIGN PARENT = srep-menu-ptr
             LABEL = salesrep.rep-name
      TRIGGERS:
          ON CHOOSE
            DO:
               FIND FIRST salesrep WHERE rep-name = SELF:LABEL.
               DISPLAY salesrep WITH FRAME x.
            END.
      END TRIGGERS.
END.

/* Add a rule to the srep-menu-ptr. */
CREATE MENU-ITEM temp-hand-ptr
     ASSIGN SUBTYPE = "RULE"
            PARENT = srep-menu-ptr.
```

p-dymenu.p *(2 of 2)*

```
/* Add an exit item to the srep-menu-ptr. */
CREATE MENU-ITEM exit-item-ptr
    ASSIGN PARENT = srep-menu-ptr
           LABEL = "E&xit"
           SENSITIVE = TRUE.

/* Set up the menu bar. */
CURRENT-WINDOW:MENUBAR = main-bar-ptr.

/* Disable menu items for all west coast sales reps.
   To begin, find the first item in srep-men-ptr.   */
temp-hand-ptr = srep-menu-ptr:FIRST-CHILD.test-items:
DO WHILE temp-hand-ptr <> ?:
   /* Find the Salesrep record for this item (if any). */
   IF temp-hand-ptr:SUBTYPE = "NORMAL"
   THEN DO:
       FIND FIRST salesrep WHERE salesrep.rep-name =
                                 temp-hand-ptr:LABEL NO-ERROR.

      /* Check if this rep is in the West region.
         If so, disable the menu item.           */
      IF AVAILABLE(salesrep)
      THEN IF salesrep.region = "West"
      THEN temp-hand-ptr:SENSITIVE = FALSE.
   END.

   /* Find the next item in srep-men. */
   temp-hand-ptr = temp-hand-ptr:NEXT-SIBLING.
END.

/* Wait for the user to select Exit. */
WAIT-FOR CHOOSE OF exit-item-ptr.
```

When you run this procedure, a window containing a menu bar and a frame (to hold information on the sales rep) appears. When you pull down the Reps menu, it displays menu items that are created dynamically for each record in the salesrep table. The sales rep who services the West region is grayed out. When you select one of the enabled sales reps, information on the sales rep appears in the frame.

22.12.5 Dynamic Pop-up Menus

Dynamic pop-up menus are associated with widgets. You set up a pop-up menu by using the CREATE MENU statement. You must set the POPUP–ONLY attribute to TRUE. To associate a pop-up menu with a widget, you assign the menu to the widget's POPUP–MENU attribute.

The following procedure displays a button with a pop-up menu that is similar to the one seen in the "Static Pop-up Menus" section. You can compare the two procedures to see the different syntax used to create the dynamic pop-up menu:

p-dypop.p *(1 of 2)*

```
DEFINE BUTTON hi LABEL "Hello".

DEFINE VARIABLE pop-menu AS WIDGET-HANDLE.
  DEFINE VARIABLE ve AS WIDGET-HANDLE.
  DEFINE VARIABLE vd AS WIDGET-HANDLE.
  DEFINE VARIABLE iv AS WIDGET-HANDLE.
  DEFINE VARIABLE ep AS WIDGET-HANDLE.
  DEFINE VARIABLE ex AS WIDGET-HANDLE.
  DEFINE VARIABLE dummy-rule AS WIDGET-HANDLE.

FORM
  hi AT x 30 y 70
  WITH FRAME button-frame.

CREATE MENU pop-menu
  ASSIGN POPUP-ONLY = TRUE
         TITLE = "Button State".

CREATE MENU-ITEM ve
ASSIGN
  PARENT = pop-menu
  LABEL = "Hello".

CREATE MENU-ITEM vd
ASSIGN
  PARENT = pop-menu
  LABEL = "Howdy".

CREATE MENU-ITEM iv
ASSIGN
  PARENT = pop-menu
  LABEL = "Hey".
```

p-dypop.p *(2 of 2)*

```
CREATE MENU-ITEM dummy-rule
ASSIGN
  PARENT = pop-menu
  SUBTYPE = "RULE".

CREATE MENU-ITEM ep
ASSIGN
  PARENT = pop-menu
  LABEL = "Exclamation point"
  TOGGLE-BOX = yes.

/* Create another rule--okay to re-use same widget-handle,
   because we aren't going to ever refer to the first rule again. */
CREATE MENU-ITEM dummy-rule
ASSIGN
  PARENT = pop-menu
  SUBTYPE = "RULE".

CREATE MENU-ITEM ex
ASSIGN
  PARENT = pop-menu
  LABEL = "Exit".

/* Set pop-menu to be the popup menu for hi-but */
ASSIGN hi:POPUP-MENU = pop-menu.

/* Define action for menu selections */
ON CHOOSE OF ve, vd, iv
  ASSIGN hi:LABEL IN FRAME button-frame = SELF:LABEL.

/* Define action for button selection. When the button is
   selected, display the current button label as a message.
   If Exclamation Point is checked, add an exclamation point
    to the message; otherwise, add a period.                */
ON CHOOSE of hi
  MESSAGE hi:LABEL IN FRAME button-frame +
          (IF ep:CHECKED THEN "!" ELSE ".").

/* Enable input on the button and wait
   for the user to choose Exit from menu. */
ENABLE hi WITH FRAME button-frame.

WAIT-FOR CHOOSE of ex.
```

23

Colors and Fonts

Progress allows you to control the colors and fonts that are displayed in a widget. The extent of this control depends on your user interface and how you manage colors and fonts in your application.

This chapter describes:

- Making colors and fonts available to an application
- Assigning colors and fonts to a widget
- Color and font inheritance
- Color in character interfaces
- Color on Windows
- Managing colors and fonts in graphical applications
- Allowing the user to change colors and fonts
- Accessing the current color and font tables
- Retrieving and changing color and font definitions
- Managing application environments

23.1 Making Colors and Fonts Available to an Application

Progress applications typically access a subset of the system's colors and fonts. This subset is specified in the following places:

- On Windows, in the registry or in an initialization file
- On UNIX, in the PROTERMCAP file.

These environment settings establish a one-to-one mapping between a range of integers and the system's colors and fonts. These integers, in turn, correspond to entries in internal color and font tables. For example, on Windows, Progress, as installed, maps the color red to the integer 4. To assign red to a widget, an application assigns the value 4 to the appropriate widget attribute. Thus, the widget is assigned the fourth color from the internal color table.

For graphical interfaces, you can specify up to 256 colors and 256 fonts. However, it is also possible to define an arbitrary color, expanding beyond the 256 colors defined in the color table, using the RGB–VALUE function. The font for an ActiveX control is set through standard font properties and has no programmatic relationship to the font table. For information about using colors and fonts for ActiveX controls, see the chapter that discusses Active X controls in the *Progress External Program Interfaces* manual.

For character interfaces, you can specify up to 128 colors. When Progress starts up, it loads the color and font definitions from the specified environment into internal color and font tables. The numbers for both your color and font definitions must increase by 1, starting from 0. Progress stops loading its color or font table at the first skipped value.

For example, if colors 0, 1, 2, and 4 are defined, Progress loads only colors 0 through 2 because color 3 is missing. If colors 1, 2, and 3 are defined, Progress loads no colors because the definitions start after 0. The order of color and font definition does not matter; it is only important that a complete sequence of color and font numbers is defined.

For more information on specifying and editing environments for graphical and character interfaces, see the *Progress Client Deployment Guide*.

23.1.1 Progress Default Colors

The environment that comes with your Progress installation specifies 16 colors. Table 23–1 shows these colors and the integers you use to reference them in an application. Note that for backward compatibility, the 16 colors are the same as those available in Version 6 of Progress.

Table 23–1: Progress Default Colors

Color Number	Windows RGB Values	Actual Color Defined
0	0, 0, 0	Black
1	0, 0, 128	Dark blue
2	0, 128, 0	Dark green
3	0, 128, 128	Blue green
4	128, 0, 0	Red
5	128, 0, 128	Purple
6	128, 128, 0	Olive
7	128, 128, 128	Gray
8	192, 192, 192	Light gray
9	0, 0, 255	Blue
10	0, 255, 0	Green
11	0, 255, 255	Turquoise
12	255, 0, 0	Red
13	255, 0, 255	Pink
14	255, 255, 0	Yellow
15	255, 255, 255	White

CAUTION: The Progress Application Development Environment (ADE) reserves these 16 colors (0 through 15) defined in your environment. If you change the mappings of these colors, the ADE tools might not function properly. You can add your own application colors beginning with number 16.

23.1.2 Progress Default Fonts

The default Progress environment specifies a set of fonts to be used with the Progress Tools. On Windows, these fonts are MS Sans Serif (a proportional font) and Courier New (a fixed-space font). If you have not added custom fonts to your environment, Progress uses these default fonts for your application. Progress uses the default proportional font for alphanumeric data, and the default fixed font for integer fields and for fields with format strings containing fill characters (such as 9 or X).

CAUTION: The Progress ADE reserves the 8 fonts (0 through 7) defined in your environment. If you change the mappings of these fonts, the ADE tools might not function properly. You can add your own application fonts, beginning with number 8.

23.1.3 Application Colors and Fonts

For an example of application color and font definitions, see the environment installed with the Progress Test Drive.

23.2 Assigning Colors and Fonts to a Widget

You can assign colors and fonts to a widget either in the widget definition statement or at run time after the widget is displayed. Use the FGCOLOR, BGCOLOR, DCOLOR, PFCOLOR, and FONT options at definition time and the FGCOLOR, BGCOLOR, DCOLOR, PFCOLOR, and FONT attributes at run time.

Progress uses the foreground color you specify for any values that appear in the widget; Progress uses the background color for the area around the widget values.

NOTE: For rectangle widgets, Progress uses the foreground color for the edge and the background color to fill the interior.

Progress uses the font you specify for any text that appears within the widget.

Note that FGCOLOR, BGCOLOR, and FONT apply to graphical interfaces only, and DCOLOR and PFCOLOR apply to character interfaces only. For more information on specifying color in character interfaces, see the"Color in Character Interfaces" section.

The p-clrfnt.p procedure demonstrates how to initialize colors and fonts at widget definition and how to change them dynamically at run time:

p-clrfnt.p

```
DEFINE BUTTON quitbtn LABEL "QUIT" BGCOLOR 4 FGCOLOR 15.
DEFINE VARIABLE fgc_frm AS INTEGER BGCOLOR 15 FGCOLOR 0
   VIEW-AS SLIDER MAX-VALUE 15 MIN-VALUE 0.
DEFINE VARIABLE bgc_frm AS INTEGER BGCOLOR 15 FGCOLOR 0
   VIEW-AS SLIDER MAX-VALUE 15 MIN-VALUE 0.
DEFINE VARIABLE font_frm AS INTEGER FONT 1 VIEW-AS SLIDER
   MAX-VALUE 15 MIN-VALUE 0.

FORM
   SKIP (1) "Form Foreground" AT 7 SKIP
   fgc_frm AT 6 SKIP (1)
   "Form Background" AT 7 SKIP
   bgc_frm AT 6 SKIP (1)
   "Form Font" AT 10 skip
   font_frm AT 6 SKIP (2)
   quitbtn AT 12 SKIP (1)
   WITH FRAME x TITLE "Color and Font Test"
      NO-LABELS CENTERED ROW 2 WIDTH 50.

ASSIGN FRAME x:RULE-Y = 282
       fgc_frm:MAX-VALUE = COLOR-TABLE:NUM-ENTRIES - 1
       bgc_frm:MAX-VALUE = COLOR-TABLE:NUM-ENTRIES - 1
       font_frm:MAX-VALUE = FONT-TABLE:NUM-ENTRIES - 1.

ON VALUE-CHANGED OF fgc_frm
   FRAME x:FGCOLOR = INPUT FRAME x fgc_frm.

ON VALUE-CHANGED OF bgc_frm
   FRAME x:BGCOLOR = INPUT FRAME x bgc_frm.

ON VALUE-CHANGED OF font_frm
   FRAME x:FONT = INPUT FRAME x font_frm.

ENABLE ALL WITH FRAME x .

WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW OR CHOOSE OF quitbtn.
```

This procedure creates three sliders representing foreground color, background color, and font. The maximum value of each slider is determined by examining the NUM–ENTRIES attributes of the COLOR–TABLE and FONT–TABLE system handles. As you move the appropriate trackbar, the foreground color, background color, or font of the interface changes. Note that, because the procedure does not explicitly set the size of any of the sliders, Progress resizes them at run time as you change the font.

You can assign colors and fonts to all widgets, with the following exceptions:

- On Windows, to assign colors to buttons, menus, and window titles, you must use the Display Properties dialog box in the Control Panel. You cannot change these colors in your Progress application.
- In all environments, images and rectangles have no text and, therefore, cannot take a font assignment.

For more information on the COLOR–TABLE and FONT–TABLE handles, and on the FGCOLOR, BGCOLOR, DCOLOR, PFCOLOR, and FONT options and attributes, see the *Progress Language Reference*. For more information on using the COLOR–TABLE and FONT–TABLE handles for run-time color and font management, see the "Accessing the Current Color and Font Tables" section in this chapter.

23.3 Assigning Colors and Fonts to ActiveX Automation Objects and ActiveX Controls

Properties associated with ActiveX Automation objects and ActiveX controls allow you to define unique characteristics, such as color and font, for each object. For example, to set colors for an ActiveX control, you need to set one or more of the color properties of the control to some RGB value. There are three ways to obtain an RGB value. You can use the RGB–VALUE function, use the COLOR-TABLE:GET-RGB-VALUE( ) method, or you can get the value from some color property of an ActiveX control or an ActiveXAutomation object. For fonts, an ActiveX Automation Server or ActiveX control generally provide a Font object that allows you to specify font properties.

For further information about assigning colors and fonts to ActiveX controls and ActiveX Automation objects, see the *Progress External Program Interfaces* manual.

23.4 Color and Font Inheritance

If you do not specify colors or a font for a widget, Progress assigns colors and a font using the following rules of precedence:

1. Field-level widgets take the colors and font of the containing frame.

2. Frame-level widgets take the default colors and fonts specified in the environment.

   NOTE: In the environment, you can specify two default fonts but you cannot specify default widget colors.

3. Otherwise, frame-level widgets take the default colors and fonts specified for the operating system.

   NOTE: On Windows, the default foreground color is color Window Text. For two-dimensional widgets, the default background color is color Window. For three-dimensional widgets, the default background color is color Button Face. These colors are configurable using the Control Panel.

Note that frames do **not** inherit colors and fonts from the containing window. If you do not specify the font for a frame, Progress uses the default font, not the font of the window. This is because Progress determines the frame layout at compile time when the window's colors and fonts (determined at run time) are not yet available.

23.5 Color in Character Interfaces

Color in a character interface differs from color in a graphical interface in these ways:

- In character interfaces, the BGCOLOR and FGCOLOR attributes have the unknown value (?).
- Each location in the color table contains a foreground/background color pair.
- DCOLOR specifies a foreground/background color pair that a widget uses when it displays data.
- PFCOLOR specifies a foreground/background color pair that a widget uses when it prompts for data.

When designing your interface, make sure that you specify different color pairs for DCOLOR and PFCOLOR.

23.5.1 Widget States and Color

Frames, dialog boxes, and rectangles use only DCOLOR. For all other widgets, Progress considers the widget's category and its state to determine when to use the display colors (DCOLOR) and when to use the prompt-for colors (PFCOLOR). The categories are text input widgets (fill-ins and editors) and selectable widgets (all other widgets). The widget states are insensitive, sensitive, and focus.

Table 23–2 shows how Progress assigns colors based on the state and category of widget.

Table 23–2: Colors Used in Character Interfaces

Widget State	Text Input Widget	Selectable Widget
Insensitive	DCOLOR	DCOLOR
Sensitive	PFCOLOR	DCOLOR
Focus	PFCOLOR	PFCOLOR

Note that for repaint, look, and usability reasons, text input widgets have the same colors for the sensitive and focus states but different colors for the insensitive state. This is in contrast to selectable widgets, which must change colors when they receive focus.

23.5.2 Color Specification

In character interfaces, Progress reserves color table locations 0 through 4 for the following foreground/background color pairs:

- Color 0 holds colors used as the NORMAL color by Progress.
- Color 1 holds the colors used as the INPUT color by Progress. As installed, this is underline mode.
- Color 2 holds the colors used as the MESSAGE color by Progress. As installed, this is reverse video.
- Color 3 holds the colors used for high-intensity mode. This mode is not available for all terminals.
- Color 4 holds the colors used for blink mode. This mode is not available for all terminals.

By default, text-input widgets use the INPUT colors when in prompt-for mode. Note, however, that when displaying an editor widget with scroll bars to display read-only text, using NORMAL rather than INPUT for prompt-for mode might make more visual sense to the user.

In the PROTERMCAP file, you can specify additional application color pairs from color 5 to 127. For more information, see the *Progress Client Deployment Guide*.

23.6 Colors on Windows in Graphical Interfaces

On Windows in graphical interfaces, your system handles color differently depending on whether it supports the Palette Manager.

23.6.1 Systems with the Palette Manager

If your system supports the Palette Manager, you can define a palette with some number of simultaneous colors (usually 256). Of these colors, 20 are normally reserved for standard system colors. The others are custom colors that you can define and modify. The exact numbers of color definitions and reservations by the system depend on your display driver.

Managing Colors for Multiple Windows

Although you can define a separate palette for each window, sharing a single palette among several windows maximizes environment efficiency and eliminates any color contention between the sharing windows. For more information, see the "Managing Application Environments" section.

Managing Colors for Bitmap Images

If you use bitmap images saved in 256–color mode, Windows tries to create an individual color map for the image each time it is realized in Progress. This has the following effects:

- The background flashes as the color palette is loaded for each 256–color image, unless the color map is identical to the one previously loaded.
- Performance degrades.

To reduce or eliminate these effects, convert your images from 256 colors to 16 colors or recreate them with 16 colors at a time.

23.6.2 Systems without the Palette Manager

On systems that do not support the Palette Manager, Windows can only display 16 standard colors. However, if you request another color, Windows tries to create that color by *dithering*; that is, by filling a space with dots of different hues. This might work well, for example, in a frame background. However, if a field-level widget, such as a fill-in or radio set, inherits a dithered color, only one hue is used. The result might be undesirable. Therefore, avoid using dithered colors for field-level widgets.

23.7 Managing Colors and Fonts in Graphical Applications

Graphical interfaces provide a variety of techniques to help manage color and font definitions for an application. Using these techniques, Progress allows a graphical application to manage colors and fonts at four levels of run-time operation:

1. **User Access to the Color and Font Tables** — Allowing the user to change the definitions for current dynamic color and font table entries using the SYSTEM–DIALOG COLOR and SYSTEM–DIALOG FONT statements. For more information, see the"Allowing the User to Change Colors and Fonts" section.

2. **Procedure Access to the Color and Font Tables** — Examining and changing the definitions for color table entries in the current environment using the COLOR–TABLE handle, and examining height and width characteristics of font table entries in the current environment using the FONT–TABLE handle. Also, this section addresses setting the definition for arbitrary colors using the RGB–VALUE function. For more information, see the "Accessing the Current Color and Font Tables" section.

3. **Procedure Access to Color and Font Definitions** — Reading and writing color and font definitions for the current environment using the GET–KEY–VALUE and PUT–KEY–VALUE statements. For more information, see the "Retrieving and Changing Color and Font Definitions" section.

4. **Procedure Replacement of the Current Environment** — Replacing the application's current color and font tables with tables from a different environment using the LOAD, USE, and UNLOAD statements. For more information, see the "Managing Application Environments" section.

23.8 Allowing the User to Change Colors and Fonts

In graphical interfaces, Progress allows you to invoke native common dialogs for color and font selection in the 4GL using the SYSTEM–DIALOG COLOR and the SYSTEM–DIALOG FONT statements. Using these dialogs, you can allow the user to choose a new value for a color or font number. Once the user chooses a value, Progress stores the choice in the color or font table location specified by the number. All visible widgets defined with the specified color or font number immediately redisplay according to the user selection.

23.8.1 Establishing Dynamic Colors

You can use the SYSTEM–DIALOG FONT statement to change any font in the font table. However, you can use the SYSTEM–DIALOG COLOR statement to update only those colors that are dynamic. To make a color dynamic, use the COLOR–TABLE system handle.

```
DEFINE VARIABLE status-ok AS LOGICAL.
status-ok = COLOR-TABLE:SET-DYNAMIC(n, yes).
```

In this example, *n* is the color table entry you want to make dynamic. For more information on the COLOR–TABLE handle, see the *Progress Language Reference* and the"Accessing the Current Color and Font Tables" section in this chapter.

23.8.2 Color Dialog Box

Figure 23–1 shows the Windows system color dialog box.

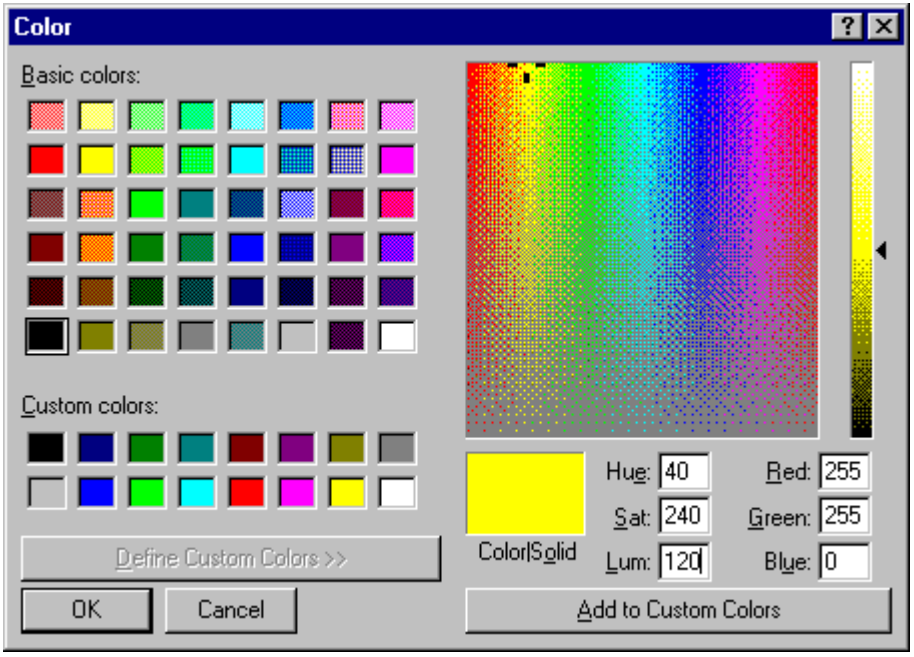


Figure 23–1: Windows Color Dialog Box

The Windows color dialog provides several ways to specify a new color:

- **Custom Colors** — You can create a palette of 16 custom colors. To specify a color, you select the color box from the 16 available custom colors. The specified color appears in the ColorSolid box. To change a custom color, select one of the Basic Colors, or specify a color by using the Numeric Colors or Visual Colors technique. Each change replaces an existing color in the Custom Colors list.

- **Numeric Colors** — You can enter either the Hue, Sat (Saturation), and Lum (Luminosity) values, or the Red–Green–Blue (RGB) values in the fields provided. The specified color appears in the Color|Solid box as you change each number.

- **Visual Colors** — You can visually choose the Hue and Sat values by pressing and holding the mouse SELECT button and moving the mouse pointer in the large rainbow square to the color you want. You can visually choose the Lum value by moving the luminosity slider (at right) up or down. For each choice, the specified color appears in the Color|Solid box as you move the mouse.

Clicking the OK button assigns the color that currently appears in the Color|Solid box of the Windows dialog to the dynamic color number specified in your SYSTEM–DIALOG COLOR statement.

23.8.3 Color Dialog Example

The `p-cdial1.p` procedure opens the dialog box that allows you to change its own foreground or background colors:

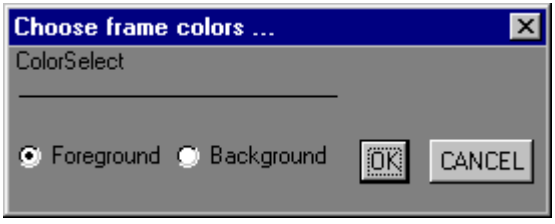
p-cdial1.p *(1 of 2)*

```
DEFINE VARIABLE FrontColor  AS INTEGER INITIAL 16.
DEFINE VARIABLE BackColor   AS INTEGER INITIAL 17.
DEFINE VARIABLE ColorSelect AS INTEGER INITIAL 16  VIEW-AS RADIO-SET
                      RADIO-BUTTONS "Foreground", 16, "Background", 17
                        HORIZONTAL.
DEFINE VARIABLE status-ok   AS LOGICAL.
DEFINE BUTTON bOK           LABEL "OK".
DEFINE BUTTON bCANCEL       LABEL "CANCEL".

IF COLOR-TABLE:NUM-ENTRIES < 18 THEN
   COLOR-TABLE:NUM-ENTRIES = 18.

status-ok = COLOR-TABLE:SET-DYNAMIC(16, TRUE).
IF NOT status-ok
THEN DO:
   MESSAGE "Cannot make color 16 dynamic.".
   RETURN.
END.

status-ok = COLOR-TABLE:SET-DYNAMIC(17, TRUE).
IF NOT status-ok
THEN DO:
   MESSAGE "Cannot make color 17 dynamic.".
   RETURN.
END.
```

p-cdial1.p *(2 of 2)*

```
COLOR-TABLE:SET-RGB-VALUE(16,RGB-VALUE(0,0,0)).
COLOR-TABLE:SET-RGB-VALUE(17,RGB-VALUE(128,128,128)).

FORM
    SKIP(0.5) SPACE(0.5)
    ColorSelect SPACE(2) bOK SPACE(2) bCANCEL
    SPACE(0.5) SKIP(0.5)
    WITH FRAME fColor TITLE "Choose frame colors ..." FGCOLOR FrontColor
         BGCOLOR BackColor VIEW-AS DIALOG-BOX.

ON CHOOSE OF bOK IN FRAME fColor
   DO:
      ASSIGN ColorSelect.
      SYSTEM-DIALOG COLOR ColorSelect.
   END.

ON CHOOSE OF bCANCEL IN FRAME fColor
   STOP.

ENABLE ColorSelect bOK bCANCEL WITH FRAME fColor.

WAIT-FOR WINDOW-CLOSE OF FRAME fColor.
```

When you run this procedure on Windows, the following frame appears:



If you choose or click the OK button, a color dialog box appears (see Figure 23–1) to assign a new system color to the specified color number. Choosing CANCEL terminates the procedure without any further color changes.

23.8.4 Saving a Modified Color

If you want to save any color definitions changed by the user, you can use the UPDATE option of the SYSTEM–DIALOG COLOR statement. This option sets a logical value to indicate whether the user has changed the specified color. If the value is TRUE, you can then save the new color definition using the PUT–KEY–VALUE statement.

23.8.5 Font Dialog

Figure 23–2 shows the native Windows system font dialog.

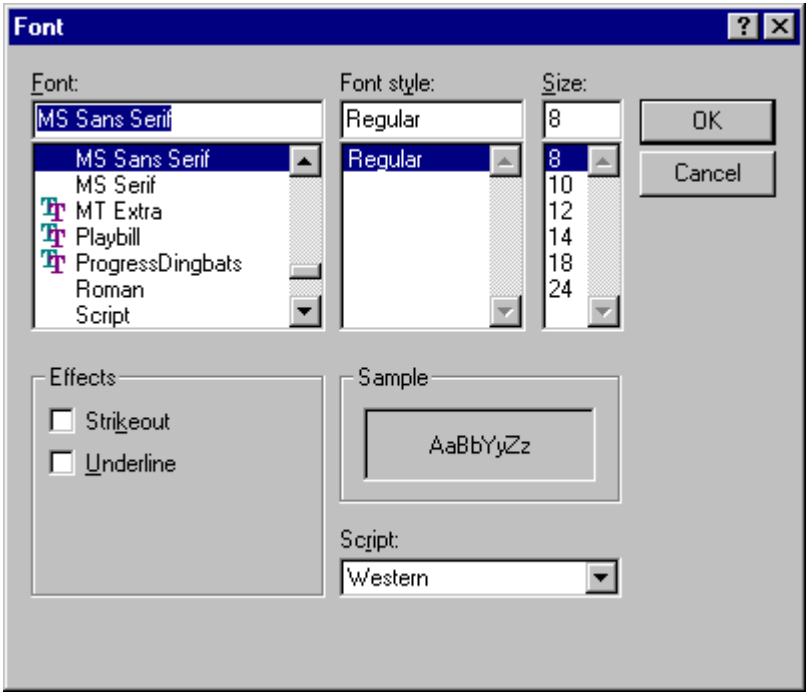


Figure 23–2: Windows Font Dialog Box

The Windows dialog box allows you to choose a single font by name and specify a style, size, script, and optional cosmetic effect. A sample of each choice appears in the Sample box.

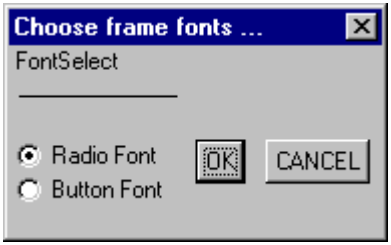
23.8.6 Font Dialog Example

The p-fdial1.p procedure opens the dialog box that allows you to separately change the font of either its radio set or buttons to a custom font:

p-fdial1.p

```
IF FONT-TABLE:NUM-ENTRIES < 13 THEN
     FONT-TABLE:NUM-ENTRIES = 13.

DEFINE VARIABLE RadioFont  AS INTEGER INITIAL 11.
DEFINE VARIABLE ButtonFont AS INTEGER INITIAL 12.
DEFINE VARIABLE FontSelect AS INTEGER INITIAL 11
    VIEW-AS RADIO-SET
        RADIO-BUTTONS "Radio Font", 11, "Button Font", 12
        FONT RadioFont.
DEFINE BUTTON bOK          LABEL "OK" FONT ButtonFont.
DEFINE BUTTON bCANCEL  LABEL "CANCEL" FONT ButtonFont.

FORM
    SKIP(0.5) SPACE(0.5)
    FontSelect SPACE(2) bOK SPACE(2) bCANCEL
    SPACE(0.5) SKIP(0.5)
    WITH FRAME fFont TITLE "Choose frame fonts ..."
         VIEW-AS DIALOG-BOX.

ON CHOOSE OF bOK IN FRAME fFont
DO:
    ASSIGN FontSelect.
    SYSTEM-DIALOG FONT FontSelect.
END.

ON CHOOSE OF bCANCEL IN FRAME fFont STOP.

ENABLE FontSelect bOK bCANCEL WITH FRAME fFont.

WAIT-FOR WINDOW-CLOSE OF FRAME fFont.
```

When you run this procedure on Windows, the following dialog box appears.



When you run the procedure, the radio set displays in the default MS Sans Serif font and buttons display in the default Courier New font. To change a font, choose the Radio Font (font 11) or Button Font (font 12) radio item, then choose or click the OK button to open the font dialog box shown in Figure 23–2. Choosing (or clicking) CANCEL terminates the procedure without any further font changes.

To change the radio-set font selected in Figure 23–2 to bold 8-point Arial, set up the font dialog as shown in Figure 23–3.

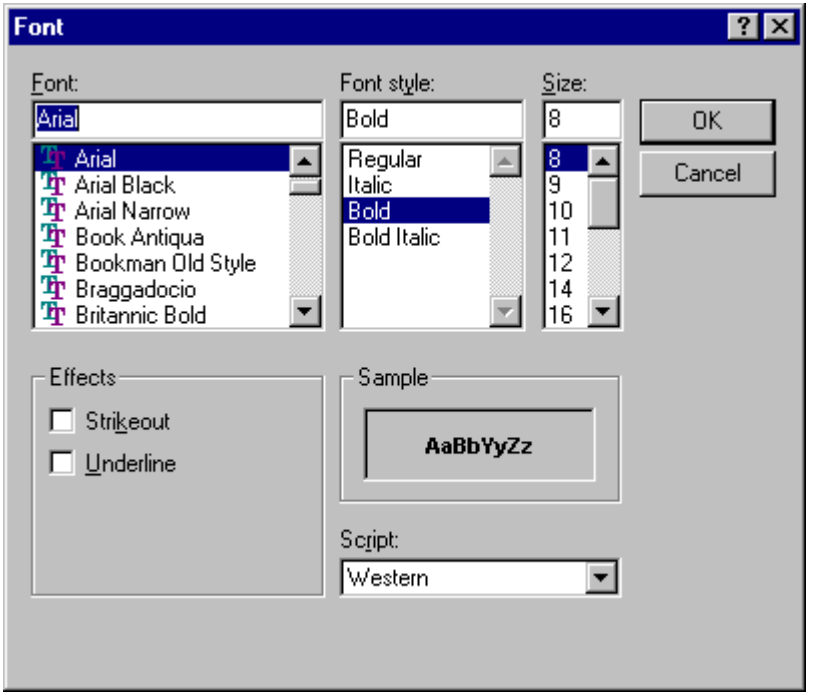
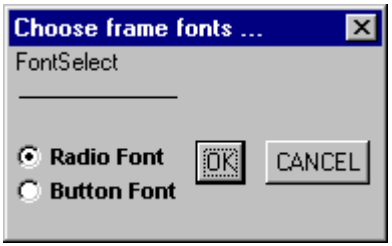


Figure 23–3: Changing Font 11 to Bold 8-Point Arial

After you choose OK, the Choose frame fonts dialog box reappears with the radio set in the new font:



23.8.7 Saving a Modified Font

If you want to save any font definitions changed by the user, you can use the UPDATE option of the SYSTEM–DIALOG FONT statement. This option sets a logical value to indicate whether the user has changed the specified font. If the value is TRUE, you can then save the new font definition using the PUT–KEY–VALUE statement.

23.9 Accessing the Current Color and Font Tables

In graphical interfaces, you can get information about colors and fonts defined for the current environment by querying attributes and methods of the COLOR–TABLE and FONT–TABLE system handles. You can also change the color configuration of the current environment using additional methods of the COLOR–TABLE handle.

23.9.1 COLOR–TABLE Handle

The NUM–ENTRIES attribute sets and returns the number of colors available in the color table. The following methods allow you to read existing color definitions and to set new definitions for dynamic colors:

- The GET–RED–VALUE(*n*), GET–GREEN–VALUE(*n*), and GET–BLUE–VALUE(*n*) methods let you read the red, blue, and green values for a specified color number. This example gets the red value of the 16th color in the color table.

```
DEFINE VARIABLE red-val AS INTEGER.
red-val = COLOR-TABLE:GET-RED-VALUE(16)
```

- You can determine whether a color is dynamic by reading the GET–DYNAMIC(*n*) method. You can make a color dynamic or nondynamic by using the SET–DYNAMIC(*n*, *logical*) method. This example makes color 16 dynamic if it is not already set.

```
IF NOT COLOR-TABLE:GET-DYNAMIC(16) THEN
COLOR-TABLE:SET-DYNAMIC(16, TRUE).
```

- You can change the red, blue, and green values of a dynamic color by using the SET–RED–VALUE(*n*, *integer*), SET–GREEN–VALUE(*n*, *integer*), and SET–BLUE–VALUE(*n*, *integer*) methods. Note that these methods change the effective color values of a color number in the current color table. Thus, every visible widget set to the specified color number changes to the corresponding colors immediately. However, this does **not** change the color definitions originally specified in the current environment. If you restart the application with the current environment, the original colors reappear. This example sets color 16 to a blue value of 192.

```
COLOR-TABLE:SET-BLUE-VALUE(16, 192).
```

- There is an efficient alternative to calling the SET–RED–VALUE(*n*, *integer*), SET–GREEN–VALUE(*n*, *integer*), and SET–BLUE–VALUE(*n*, *integer*) methods to change the RGB values of a dynamic color. You can specify the SET–RGB–VALUE() method to substitute for all three of these methods. You can determine a combined RGB value using the RGB–VALUE function or by accessing a color property from an ActiveX control.

```
COLOR-TABLE:SET-RGB-VALUE (16, RGB-VALUE (128, 0, 128)).
```

  The GET–RGB–VALUE() method returns an integer that represents a combination of the red, green, and blue values associated with the nth entry. This combined RGB value is most useful for setting colors in an ActiveX control.

For more information on the COLOR–TABLE handle, see the *Progress Language Reference*.

23.9.2 FONT–TABLE Handle

The NUM–ENTRIES attribute sets and returns the number of fonts available in the font table. The FONT–TABLE methods return information about font sizes. Each method takes an optional font number as an argument:

- GET–TEXT–HEIGHT–CHARS( ) and GET–TEXT–HEIGHT–PIXELS( ) return the height of the default font in either character units or pixels. You can pass a font number to these methods to get the height of a specific font. This example looks for the first font from the start of the font table with a height of at least two character units.

```
DEFINE VARIABLE font-number AS INTEGER.
DEFINE VARIABLE max-count AS INTEGER.
max-count = FONT-TABLE:NUM-ENTRIES + 1.
REPEAT font-number = 1 to max-count:
    IF font-number = max-count THEN LEAVE.
    IF FONT-TABLE:GET-TEXT-HEIGHT-CHARS(font-number) >= 2 THEN LEAVE.
END.
```

- GET–TEXT–WIDTH–CHARS(*string*) and GET–TEXT–WIDTH–PIXELS(*string*) return the width of the passed *string* expression in the default font in either character units or pixels. You can optionally pass a font number to these methods to get the width of the string in a specific font. This example tests whether a title for a new window will fit based on the font of the current window.

```
DEFINE VARIABLE font-number AS INTEGER.
DEFINE VARIABLE in-title AS CHARACTER.
font-number = CURRENT-WINDOW:FONT.
SET in-title.
IF FONT-TABLE:GET-TEXT-WIDTH-CHARS(font-number, in-title) > 80 THEN
    MESSAGE "Title" in-title "cannot fit in new window.".
```

For more information on the FONT–TABLE handle, see the *Progress Language Reference*.

23.10 Retrieving and Changing Color and Font Definitions

In graphical interfaces, you can retrieve and change the color and font definitions for your current environment using the GET–KEY–VALUE and PUT–KEY–VALUE statements. These statements read and write to the current environment.

23.10.1 Changing Resource Definitions

The GET–KEY–VALUE statement can read and the PUT–KEY–VALUE statement can change the definition of any environment resource, including the definitions of colors and fonts stored in the current environment. However, these statements by themselves do not affect the current Progress environment and its color and font tables. To have any definitions that are created using the PUT–KEY–VALUE statement take effect, you must replace the current environment by reloading the current environment (see the "Managing Application Environments" section).

Portable Color and Font Definitions

The portable and most effective way to change color and font definitions is to first change the definitions in the color and font tables, then use the PUT–KEY–VALUE statement to write the new definitions from the modified tables. You can affect the color and font tables in the current environment using the COLOR–TABLE and FONT–TABLE handles (see the "Accessing the Current Color and Font Tables" section), and you can allow the user to affect the current color and font tables using the SYSTEM–DIALOG COLOR and SYSTEM–DIALOG FONT statements (see the "Allowing the User to Change Colors and Fonts" section).

23.10.2 Using GET–KEY–VALUE and PUT–KEY–VALUE

The GET–KEY–VALUE and PUT–KEY–VALUE statements allow you to read or write a specified value for any resource by accessing the registry or an initialization file. The registry consists of sections called keys and subkeys arranged in a hierarchy. Keys and subkeys contain value entries, each of which consists of a value name and value data. Initialization files, by contrast, consist of a single level of sections. Sections contain entries, each of which consists of a name, an equals sign (=), and a value.

For example, to retrieve the Windows definition for font 8 from the current environment, which might be the registry or an initialization file, you might enter this statement. It returns the initial environment definition for font 8 in the FontString variable.

```
DEFINE VARIABLE FontString AS CHARACTER FORMAT "x(128)".
GET-KEY-VALUE SECTION "Fonts" KEY "Font8" VALUE FontString.
```

To specify “Times New Roman” as the new definition for font 8 on Windows, you might enter this statement. It sets the `font8` parameter in the current environment.

```
PUT-KEY-VALUE SECTION "fonts" KEY "font8" VALUE "Times New Roman".
```

Writing Portable Color and Font Definitions

To write portable color and font definitions directly from the current color and font tables, use the PUT–KEY–VALUE statement with the COLOR or FONT option. For example, if you allow the user to change color 8 during a session through the color common dialog (SYSTEM–DIALOG COLOR statement), you can save the new color definition in the current environment using this statement.

```
PUT-KEY-VALUE COLOR 8.
```

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

```
PUT-KEY-VALUE COLOR ALL.
```

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

23.11 Managing Application Environments

For some graphical applications, you might want to replace whole Progress environments. This applies especially to applications that build other applications, where you want to provide the ability to test run each application in a separate environment. The Progress AppBuilder is an example of such an application. You might also want to write a single application that has greater flexibility to change its own environment (for example, changing the number of color definitions available to the application).

23.11.1 Understanding Environment Management

When Progress starts up, it loads a default environment from the registry (Windows only) or from an initialization file. This default environment is the initial current environment. The initial current environment provides resources to the default window and any windows subsequently created in that environment. At any point, you can replace the current environment by loading and making a specified environment current.

23.11.2 Using Environment Management Statements

To replace the current environment with another application environment, Progress provides these statements:

- **LOAD** — Loads a previously defined application environment into Progress memory, or creates and loads an empty application environment to be defined. Generally, an application environment resides in the Registry or in an initialization file with a meaningful filename. You can load as many application environments as memory allows, but only one environment can be current at a time.

- **USE** — Makes a previously loaded environment available as the current environment. The new current environment has no effect on the default window or any previously created windows. The resources of the new current environment are available only for windows created after the new environment is made current.

- **UNLOAD** — Removes a loaded environment from Progress memory. Environments take up very little memory and generally do not have to be unloaded. However, if you are writing an application builder, this statement provides good housekeeping when you save and remove an application from memory. This allows you to remove all memory associated with a purged application.

  Note that you must delete all windows created in an environment before you can unload that environment, whether or not it is the current environment. If the unloaded environment is the current environment, you must set a new current environment with the USE statement before continuing operation.

For more information on these statements, see their reference entries in the *Progress Language Reference*.

23.11.3 Managing Multiple Environments

If you are writing an application builder, you might want to maintain separate environments for each application.

Typical Scenario

You can follow these steps to separate application environments:

1 ♦ Have the user select an existing application or create a new one.

2 ♦ Load the existing application environment, or create and load a new one using the LOAD statement.

3 ♦ Execute the USE statement to make the selected application environment current.

4 ♦ If the application exists, run it according to user input.

If the application is new:

a) Allocate the number of available color and font table entries for the new application by setting the NUM–ENTRIES attribute of the COLOR–TABLE and FONT–TABLE handles.

b) Set internal colors and fonts from user input using the SYSTEM–DIALOG COLOR and FONT statements, or set internal colors using the COLOR–TABLE handle.

c) Create application widgets and code from user input using the current color and font definitions.

5 ♦ Test and revise application resources and widgets according to user input.

6 ♦ Repeat Steps 1 through 5 for as many applications as you need to work on simultaneously.

7 ♦ When the user terminates all work on an application and wants to save it:

a) Save the color and font table entries to the current environment using the PUT–KEY–VALUE statement with the COLOR and FONT options.

b) Save the application code to procedure files.

c) Delete all widgets for the application from Progress memory using the DELETE WIDGET statement.

d) Remove the current application environment from Progress memory using the UNLOAD statement.

8 ♦ Execute the USE statement according to user input to set a new current environment, then repeat Steps 4 through 8 until all work is done.

Helpful Hints for Environment Management

Many variations in the typical scenario are possible, but this is a summary of important tasks to consider in any application that uses multiple environments:

- Before defining colors and fonts for immediate use at run time, set or adjust the size of the current color and font tables as required using the NUM–ENTRIES attribute of the COLOR–TABLE and FONT–TABLE handles.
- Use the PUT–KEY–VALUE statement with the COLOR and FONT options to save newly created and newly modified color and font definitions from the current color and font tables to the current environment.
- Always set a current environment with the USE statement before or after executing the UNLOAD statement for the current environment, even if the current environment you set is the default environment. Otherwise, Progress returns an error when you attempt to display or enable a widget.

23.12 Managing Color Display Limitations

Although Progress allows up to 256 colors to be defined in an environment, whether and how the widgets in each environment display these colors is system dependent. For example, if you have 256-color widgets displayed from three separate environments, and each environment defines an entirely different set of 256 colors, some of the widgets might not be displayed with the correct colors. If your display (driver and hardware) supports a maximum of 256 colors at one time, the 256-color widgets from only one of your environments can be displayed exactly as specified. All other widgets must be displayed incorrectly to allow the widgets from one environment to display correctly.

In general, no matter what your display limitations, Progress tries to ensure that the current window (the window that has focus) displays correct colors at the expense of one or more other windows that are displayed, but do not have focus. You can also do the following to ensure that all of your widgets display with correct colors:

- Do not define more colors for all loaded and used environments than your display can simultaneously support.
- Hide any widgets, especially images, that you no longer need to display and whose color content exceeds the color capacity of your system.

24

Direct Manipulation

Progress lets you write applications that allow the user to directly manipulate (move and resize) frames and field-level widgets using a mouse. For each widget that is to be manipulated, you must set the appropriate attributes. For example, to allow the user to move a frame, you must set the frame's MOVABLE attribute to TRUE. The user can then use a mouse to move the frame when running the application.

Progress also supports grids. A grid is a framework of crisscrossed bars appearing within a frame that helps the user align widgets within the frame.

24.1 Attributes

The following attributes are the main attributes associated with the direct manipulation of widgets. Other attributes are also associated with direct manipulation but apply solely to grids. For more information on these other attributes, see the "Grids" section.

Table 24–1: Direct-manipulation Attributes

Attribute	Description
SELECTABLE	Set this attribute to TRUE if you want to allow the user to highlight a widget before moving or resizing it. This attribute applies to frames and to all field-level widgets.
MOVABLE	Set this attribute to TRUE to allow the user to move a widget. This attribute applies to frames and to all field-level widgets.
RESIZABLE	Set this attribute to TRUE to allow the user to resize a widget. This attribute applies to frames and to all field-level widgets.
BOX–SELECTABLE	Set this attribute to TRUE if you want to allow the user to use *selection boxes* to select and deselect widgets within a frame. (For more information on selection boxes, see the "Box Selecting" section.) This attribute applies only to frames.
SELECTED	When the user marks a widget with a mouse, Progress sets this attribute to TRUE. You can also explicitly set this attribute within your code. Progress sets this attribute to FALSE when the user unmarks the widget. This attribute applies to frames and to all field-level widgets.
MANUAL–HIGHLIGHT	Set this attribute to TRUE if you want to use a custom highlight graphic design. When the user selects a widget, Progress highlights the widget by placing a box around that widget by default. By setting this attribute to TRUE, you can override the Progress default and highlighting the widget yourself.
NUM–SELECTED–WIDGETS	Use this read-only attribute to determine how many frames and dialog boxes are selected in a window, or how many field-level widgets are selected in a frame or dialog box.

By default, all of these attributes are set to FALSE, except for NUM–SELECTED–WIDGETS, which is an integer and read-only. In addition to the attributes listed in Table 24–1, the GET–SELECTED–WIDGET( ) method allows you to access the widget handle of all selected widgets in a window, dialog box, or frame. For more information on this method, see the *Progress Language Reference*.

The following procedure illustrates how you might use some of these attributes.

p-dirman.p

```
DEFINE VARIABLE tmp AS INTEGER.
DEFINE RECTANGLE rect1 size-pixels 39 by 39 edge-pixels 3 no-fill.
DEFINE RECTANGLE rect2 size-pixels 40 by 40 edge-pixels 3 no-fill.
DEFINE RECTANGLE rect3 size-pixels 40 by 39 edge-pixels 3 no-fill.
DEFINE RECTANGLE rect4 size-pixels 39 by 40 edge-pixels 3 no-fill.
DEFINE BUTTON manipulable LABEL "Manipulable".

FORM SKIP (1) SPACE (2)  rect1 SPACE(9) manipulable
                         rect2 SPACE(2)  SKIP (1)
              SPACE(9)   rect3 SPACE(10)
                         rect4 SPACE(7)  SKIP(3)
                         "Cust-Num: " tmp SKIP(1)
WITH FRAME a ROW 3 CENTERED NO-LABELS TITLE "Manipulable Widgets".
FRAME a:BOX-SELECTABLE = YES.

rect1:SENSITIVE        IN FRAME A = YES.   /* All Properties sensitive */
rect1:SELECTABLE       IN FRAME A = YES.
rect1:MOVABLE          IN FRAME A = YES.
rect1:RESIZABLE        IN FRAME A = YES.

manipulable:SENSITIVE  IN FRAME A = YES.   /* All Properties sensitive */
manipulable:SELECTABLE IN FRAME A = YES.
manipulable:MOVABLE    IN FRAME A = YES.
manipulable:RESIZABLE  IN FRAME A = YES.

rect2:SENSITIVE        IN FRAME A = YES.   /* Movable, but not resizable */
rect2:SELECTABLE       IN FRAME A = YES.
rect2:MOVABLE          IN FRAME A = YES.
rect2:RESIZABLE        IN FRAME A = NO.

rect3:SENSITIVE        IN FRAME A = YES.   /* Resizable but not Movable */
rect3:SELECTABLE       IN FRAME A = YES.
rect3:MOVABLE          IN FRAME A = NO.
rect3:RESIZABLE        IN FRAME A = YES.

rect4:SENSITIVE        IN FRAME A = YES.   /* Not resizable or Movable */
rect4:SELECTABLE       IN FRAME A = YES.   /* but selectable           */
rect4:MOVABLE          IN FRAME A = NO.
rect4:RESIZABLE        IN FRAME A = NO.

ENABLE ALL WITH FRAME a.
WAIT-FOR GO OF FRAME a.
```

If you set any of the writable attributes in Table 24–1 to TRUE, Progress assumes that you want the user to be able to perform direct manipulation on the widget. As a result, Progress interprets user mouse actions differently and gives higher priority to those mouse events that are associated with direct manipulation (see Table 24–2). For example, if you set a button's SELECTABLE attribute to TRUE, the user cannot choose the button with a mouse. To choose the widget, the user must use the keyboard instead.

Table 24–2: Direct-manipulation Mouse Events

SELECTION	START–RESIZE
DESELECTION	END–RESIZE
EMPTY–SELECTION	START–BOX–SELECTION
START–MOVE	END–BOX–SELECTION
END–MOVE	–

24.2 Mouse Buttons on Windows

On Windows, Progress maps the two logical buttons, MOVE and SELECT, to the left mouse button. Every time the user presses the left mouse button, Progress must decide whether to interpret it as either a MOVE or a SELECT. If the button is clicked (while pointing to a widget), Progress interprets it as a SELECT. If the button is held down and dragged, Progress interprets it according to the following rules:

- If the mouse was pointing to an empty space in a frame, or to a nonmovable object, Progress interprets it as a SELECT. Progress creates a selection box that the user can use to select widgets (given that the frame's BOX–SELECTABLE attribute is set to TRUE). All widgets contained within the selection box are selected; the widgets outside of the box are deselected. For more information on selection boxes, see the "Selecting Widgets" section.

- If the mouse was pointing to a widget, Progress interprets it as a MOVE (given that the widget's MOVABLE attribute is set to TRUE). If the widget is selected, Progress moves the widget and all other widgets that were selected. If the widget is not selected, Progress deselects all other selected widgets, selects the remaining widget, then moves the widget.

For more information on Progress logical mouse buttons, see Chapter 6, "Handling User Input."

24.3 Selecting Widgets

When the user selects a widget, Progress, by default, surrounds the widget with a highlight box. If the widget's RESIZABLE attribute is set to TRUE, the widget's highlight box also has small boxes, called resize handles, along its edges. The user can use these handles to resize the widget. Figure 24–1 shows the different highlight boxes.

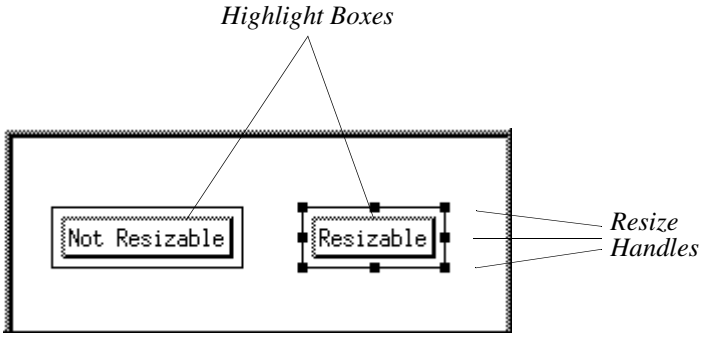


Figure 24–1: Highlight Boxes

When the user selects a widget, Progress sets its SELECTED attribute to TRUE. You can override this behavior by writing a trigger for the SELECTION event and having that trigger return a NO–APPLY. You can prevent Progress from changing a widget's SELECTED attribute to FALSE by writing a trigger for the DESELECTION event. You can also set the SELECTED attribute to TRUE or FALSE from within your application. When a widget is deselected, Progress sets its SELECTED attribute to FALSE by default.

A user must select a widget before resizing it. The user has two ways to select a widget:

- Pointing and clicking
- Box selecting

24.3.1 Selecting by Pointing and Clicking

When the user points to a field-level widget and clicks the SELECT button, Progress selects the widget by default (given that the widget's SELECTABLE attribute is set to TRUE). Progress also draws a highlight box around the widget and deselects all previously selected widgets. If the user points at an empty space within a frame and clicks the SELECT button, Progress deselects all previously selected object within the frame, and selects the frame if it is selectable.

To select a frame, the user must point to an empty space in the frame, the frame's border, or the frame's title, and click the SELECT button.

When the user points to a field-level widget and clicks the EXTEND button (CTRL plus the SELECT button), Progress toggles the selected state of the widget. If the widget was selected, Progress deselects it. If the widget was not selected, Progress selects it. Progress draws or erases highlight boxes according to the selected state of the widget. If the user points at an empty space within a frame and clicks the EXTEND button, Progress toggles the frame selection.

You can override the Progress default behavior by writing triggers for the SELECTION and DESELECTION events and having the triggers return a NO–APPLY.

24.3.2 Box Selecting

If the user holds down the SELECT or EXTEND button and drags the mouse, Progress draws a selection box if the frame's BOX–SELECTABLE attribute is set to TRUE. Figure 24–2 shows a selection box.

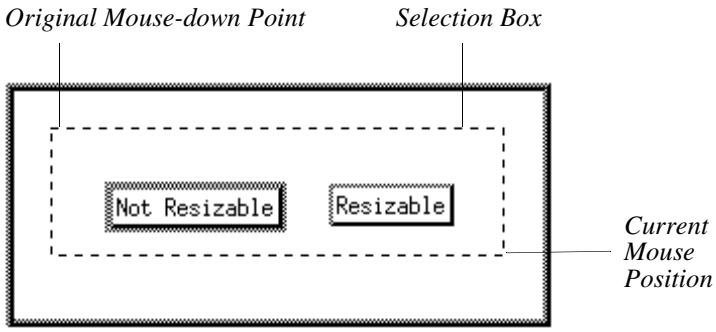


Figure 24–2: Selection Box

The appearance of the selection box is the same for both the SELECT and EXTEND buttons. But the effect of the selection boxes is different depending on which button the user presses. If the SELECT button is pressed, Progress selects all widgets completely contained within the box. Progress deselects all other previously selected widgets outside of the box. Deselection occurs on mouse down. If the EXTEND button is pressed, Progress toggles the selected state of the widgets within the box. All widgets outside of the box remain unaffected.

The user can create and use selection boxes only on field-level widgets within a frame. If the user moves the mouse pointer outside of the frame where the selection box was started, the selection box is confined to the frame. When the user reenters the frame, Progress redraws the selection box to the current mouse position.

24.3.3 Sets of Selected Widgets

Within a frame, the user can simultaneously select multiple field-level widgets. The group of selected widgets within a frame is called the *set of selected widgets*. Move operations that the user performs on one selected widget also apply to the set of selected widgets. It is impossible, however, for the user to perform a move operation against a widget whose attribute settings don't allow for it. If a widget's MOVABLE attribute is set to FALSE, then the user cannot move the widget, regardless of whether it is in the set of selected widgets.

24.4 Moving and Resizing Field-level Widgets

Follow these steps to move a widget or set of selected widgets:

1 ♦ To move more than one widget, select all widgets to move.

To move a single widget, put the mouse pointer inside the widget, press the MOVE button, and drag the mouse.

2 ♦ Place the mouse pointer inside one of the selected widgets.

3 ♦ Press down the MOVE button and drag the mouse.

Progress draws a *drag box* (Figure 24–3) around each of the selected widgets; these drag boxes move along with the mouse across the frame. A drag box is identical in appearance to a selection box. When the user moves widgets, drag boxes hold their relative position as the user moves them. A move operation only works for widgets whose MOVABLE attribute is set to TRUE.

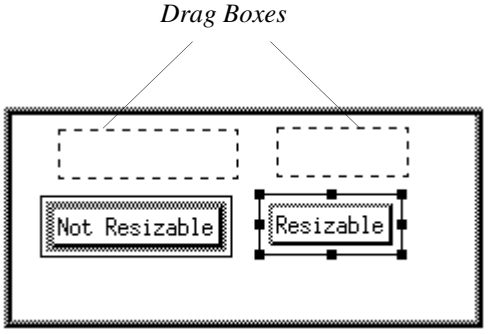


Figure 24–3: Drag Boxes

To resize a widget, the user must perform the following steps:

1 ♦ Select the widget to resize.

2 ♦ Place the mouse pointer on a resize handle of the selected widget.

3 ♦ Press down the SELECT button and drag the mouse.

Progress draws a drag box around the selected widget. The drag box changes in size as the user moves the mouse across the frame. A resize operation only works for widgets whose RESIZABLE attribute is set to TRUE.

Each resize handle allows the user to size a widget in a different direction. For example, if the user chooses the top-middle resize handle, Progress resizes the widget vertically. If the user chooses a corner resize handle, Progress resizes the widget diagonally in the direction of that corner. See Figure 24–4.

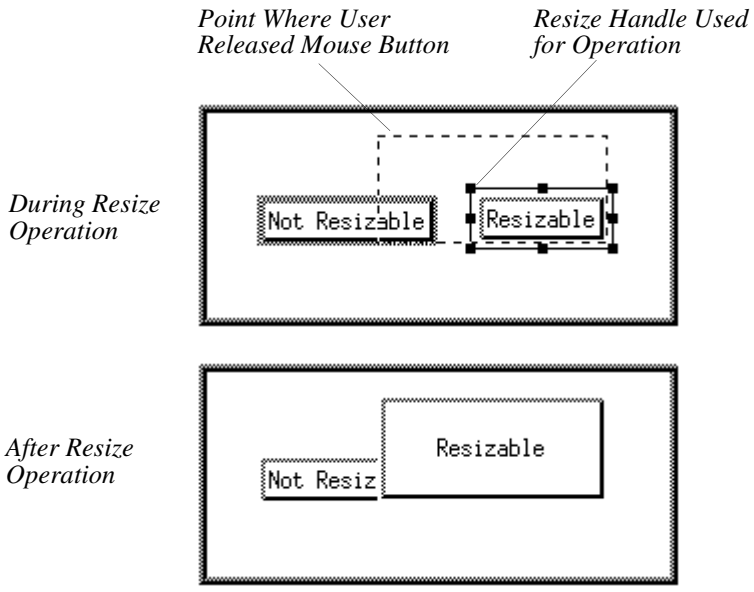


Figure 24–4: Resize Operation

If the user resizes a fill-in field, Progress does not change the format of the field to accommodate the greater amount of available space. You can, however, provide triggers for END–RESIZE events to modify the formats. You can use this feature to provide scrolling fill-ins. Set the format to a large value, then set the widget's size to a small value. Progress will scroll the text when the screen limit is reached.

24.5 Moving and Resizing Frames

To move a frame, the user must point the mouse pointer to the frame's border (the title is included in the border) when pressing down the MOVE button. Or the user can press **SHIFT** plus the MOVE button. To resize a frame, the user must first select the frame.

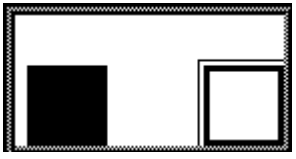
24.6 Custom Highlights

To customize a design for widget highlighting, you can set the MANUAL–HIGHLIGHT attribute to TRUE. When you set this attribute to TRUE, you override Progress's default highlight style. For an example of how you might design your own highlighting style, look at the following procedure.

p-manual.p

```
DEFINE RECTANGLE rect1 SIZE-PIXELS 40 BY 40 EDGE-PIXELS 3 NO-FILL.
DEFINE RECTANGLE rect2 SIZE-PIXELS 40 BY 40 EDGE-PIXELS 3 NO-FILL.

FORM SKIP(1) SPACE (1) rect1 SPACE(8) rect2 WITH FRAME a NO-LABELS.

rect1:SELECTABLE IN FRAME a = yes.
rect1:SENSITIVE IN FRAME a = yes.
rect1:MANUAL-HIGHLIGHT IN FRAME a = yes.
rect1:BGCOLOR = 1.
rect2:SELECTABLE IN FRAME a = yes.
rect2:SENSITIVE IN FRAME a = yes.

ON SELECTION OF rect1 IN FRAME a SELF:FILLED = yes.
ON DESELECTION OF rect1 IN FRAME a SELF:FILLED = no.

ENABLE rect1 rect2 WITH FRAME a.
WAIT-FOR GO OF FRAME a.
```

This procedure displays two rectangles that the user can highlight. One of the rectangles uses a customized style; the other rectangle uses the default highlight style. If the user selects both rectangles, the output of the procedure appears as follows.



The rectangle on the left, rect1, uses the customized highlight style of filling in the rectangle; the rectangle on the right, rect2, uses the default Progress highlight style.

24.7 Interaction Modes

There are three modes that Progress can enter during a direct-manipulation session:

- Move mode
- Resize mode
- Box-selecting mode

The user directs Progress to enter one of these modes by pressing the SELECT, EXTEND, or MOVE button and holding it down. The mode entered is determined by where the user's mouse pointer is pointing and by what button the user pressed. For example, if the user presses down the SELECT button while pointing to an empty space and drags the mouse, Progress enters box-selecting mode (given that the frame's BOX–SELECTABLE attribute is set to TRUE). But if the user presses the SELECT button when pointing at a resize handle, Progress enters resize mode. If the user presses the MOVE button, Progress always enter move mode. You can direct Progress not to enter a mode by writing a trigger for the event that causes Progress to enter the mode; you can have that trigger return a NO–APPLY.

The mode Progress is in affects how Progress interprets mouse-move events and mouse-button-release events. When the user releases the mouse button, Progress exits the mode it is in. If the user is in move mode, Progress generates an END–MOVE event. If the user is in resize mode, Progress generates an END–RESIZE event.

24.8 Direct-manipulation Events

Direct-manipulation events have general characteristics that you must know to use them effectively. For example, some events only apply to frames while others apply to both frames and field-level widgets. In addition, you need to know the order in which Progress generates these events during mouse operations. The following sections describe both aspects of direct-manipulation events.

24.8.1 General Characteristics

Before you can write a program that takes advantage of direct-manipulation events, you need to know:

- Which widgets Progress can send the event to in response to user actions
- Which user actions initiate the event
- How Progress responds to the event by default

This information is described for each event in the following list. Additional comments are added where necessary. Note that you can override the Progress default actions by writing a trigger for the event and having that trigger return a NO–APPLY.

SELECT

Widgets: Frames and field-level widgets.

User actions: (1) The user clicks the SELECT button on a widget; (2) the user clicks the EXTEND button on an widget that is not already selected; (3) the user uses the SELECT button to draw a selection box around a widget; (4) the user uses the EXTEND button to draw a selection box around a deselected widget.

Progress default actions: Progress draws a highlight box around the widget and sets its SELECTED attribute to TRUE.

DESELECT

Widgets: Frames and field-level widgets.

User actions: (1) The user clicks the SELECT button on another unselected widget; (2) the user clicks the EXTEND button on a widget that is already selected; (3) the user clicks the SELECT button on empty space in the frame; (4) the user starts a box-selecting operation with the SELECT button.

Progress default actions: Progress erases the highlight box around the widget and sets its SELECTED attribute to FALSE.

EMPTY–SELECTION

Widgets: Frames and windows.

User actions: The user clicks the SELECT button on empty space in the frame or window (or performs a box-selecting operation that does not include any widgets).

Progress default actions: None.

Comment: If the user clicks on an empty space, Progress deselects all selected widgets before it sends the EMPTY–SELECTION event to the frame or window.

START–MOVE

Widgets: Frames and field-level widgets.

User actions: The user presses down the MOVE button to start a move operation. For frames, the user presses **SHIFT** plus the MOVE button.

Progress default actions: Progress selects the widget if its SELECTABLE attribute is set to TRUE; if it is not already selected, Progress draws a drag box around the widget and enters move mode. If there are selected widgets outside the frame being moved or outside of the frame containing the field-level widgets being moved, Progress also sends DESELECTION events to those widgets outside the frame. Deselection occurs on mouse down.

Comments: When the user clicks the MOVE button, Progress sends a START–MOVE event to every widget that is selected. If your application puts a trigger on START–MOVE, and the trigger returns NO–APPLY, then Progress does not enter move mode, and does not generate subsequent END–MOVE events.

END–MOVE

Widgets: Frames and field-level widgets.

User actions: The user releases the MOVE mouse button while Progress is in move mode (that is, during a move operation).

Progress default actions: Progress moves the widget to its new location, redraws it there, and exits move mode.

Comment: Progress generates an END–MOVE event for each widget that the user moves.

START–RESIZE

Widgets: Frames and field-level widgets.

User actions: The user clicks the SELECT button on a resize handle of a widget (the widget must be selected).

Progress default actions: Progress draws a drag box around the widget and enters resize mode.

END–RESIZE

Widgets: Frames and field-level widgets.

User actions: The user releases the SELECT mouse button while Progress is in resize mode (that is, during a resize operation).

Progress default actions: Progress resizes the widget to correspond to the new size and location of its drag box and exits resize mode.

START–BOX–SELECTION

Widgets: Frames only.

User actions: The user starts to move the mouse button after having pressed the SELECT mouse button.

Progress default actions: Progress draws a selection box, which initially appears quite small but expands into a recognizable box when the user moves the mouse and enters box-selecting mode.

Comment: If your application puts a trigger on START–BOX–SELECTION, and the trigger returns NO–APPLY, then Progress does not enter box-selecting mode and does not generate subsequent and END–BOX–SELECTION events.

END–BOX–SELECTION

Widgets: Frames only

User actions: The user releases the SELECT mouse button while Progress is in box-selecting mode (that is, during a box-selecting operation).

Progress default actions: Progress normally erases the selection box and then sends a SELECTION or DESELECTION event to each widget that is completely contained inside the selection box. If the user starts the box-selecting operation by pressing the SELECT mouse button, then Progress generates SELECTION events for all the widgets. If the user pressed EXTEND instead, then Progress generates SELECTION events for the unselected widgets and DESELECTION events for the selected widgets. Progress also exits box-selecting mode.

24.8.2 Order of Direct-manipulation Events

A few mouse actions may initiate a several events. The order in which Progress generates these event depends on the type of operation the user performs.

The Box-selecting Operation

The following steps describe the order of events that Progress generates during a box-selecting operation:

1 ♦ The user presses down and holds the SELECT or EXTEND button.

When the user presses SELECT, Progress generates an EMPTY–SELECTION event. Progress then generates a DESELECT event for all widgets that were selected. If the user presses EXTEND, Progress generates a MOUSE–EXTEND–DOWN event.

2 ♦ The user begins to move the mouse.

Progress generates a START–BOX–SELECTION event for the frame.

3 ♦ The user releases the mouse button.

When the user releases SELECT, Progress generates DESELECTION events for all selected widgets and SELECTION events for all of the widgets the user included in the box-selecting operation. Then Progress generates an END–BOX–SELECTION event for the frame. The key code that the mouse sends to Progress is MOUSE–SELECT–UP.

When the user releases EXTEND, Progress generates SELECTION and DESELECTION events for the widgets included in the box-selecting operation. Then Progress generates an END–BOX–SELECTION event for the frame. The key code that the mouse sends to Progress is MOUSE–EXTEND–UP.

The Move Operation

The following steps describe the order of events that Progress generates during a move operation:

1 ♦ The user presses down the MOVE button while the graphics pointer is pointed at a widget.

Progress generates a MOUSE–SELECT–DOWN event.

2 ♦ The user begins to move the mouse.

Progress generates a START–MOVE event for the widgets the user moves.

3 ♦ The user releases the mouse.

Progress generates an END–MOVE event for each widget moved.

The Resize Operation

The following steps describe the order of events that Progress generates during a move operation:

1 ♦ The user presses down the SELECT button while the graphics pointer is pointed at a widget's resize handle.

Progress generates a MOUSE–SELECT–DOWN event and then a START–RESIZE event.

2 ♦ The user begins to move the mouse.

Progress generates a START–RESIZE event for the widget the user resizes.

3 ♦ The user releases the mouse.

Progress generates an END–RESIZE event for the widget the user resized.

24.9 Grids

Progress allows the user to use grids to align widgets within frames. A grid is a framework of crisscrossed bars whose width and height you can control by setting the appropriate attributes. Figure 24–5 shows how a grid appears on screen.

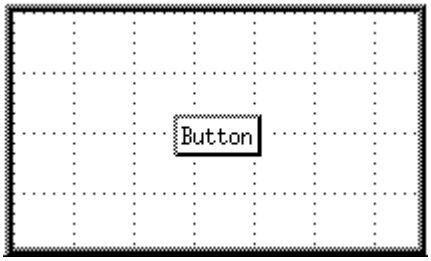


Figure 24–5: Grid

The dots in Figure 24–5 are called *visible grid points*. The distance between two of these small dots is called a *minor grid unit*. The distance between one grid line and another grid line is called a *major grid unit*. Inside each of the squares on the grid there are invisible *grid points* (see Figure 24–6).

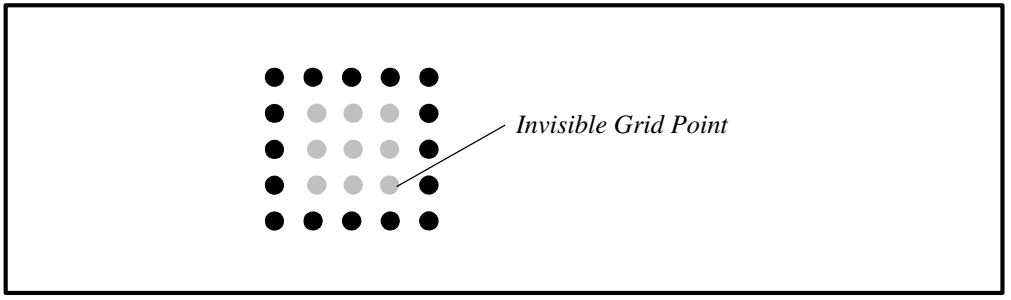


Figure 24–6: Grid Enlargement

You can control the appearance of grids with the following attributes (these attributes apply to frames only):

- **GRID–VISIBLE** — Determines whether the grid is visible or not.
- **GRID–FACTOR–HORIZONTAL** — Determines the number of horizontal minor grid units per major unit.
- **GRID–FACTOR–VERTICAL** — Determines the number of vertical minor grid units per major unit.

- **GRID–SNAP** — Determines whether objects align to (snap to) the grid when the user moves or resizes them.
- **GRID–UNIT–WIDTH–CHARS** — Determines the width in character units of a minor grid unit (that is, the distance in character units between adjacent horizontal grid points).
- **GRID–UNIT–WIDTH–PIXELS** — Determines the width in pixels of a minor grid unit (that is, the distance in pixels between adjacent horizontal grid points).
- **GRID–UNIT–HEIGHT–CHARS** — Determines the height in characters of a minor grid unit (that is, the distance in character units between adjacent vertical grid points).
- **GRID–UNIT–HEIGHT–PIXELS** — Determines the height in pixels of a minor grid unit (that is, the distance in pixels between adjacent vertical grid points).

When the user moves a widget and the frame's GRID–SNAP attribute is set to TRUE, the widget's upper left corner snaps to the nearest grid point. This is true even if the GRID–VISIBLE attribute is set to FALSE. Note that widgets that are already placed in the frame are not affected when you set GRID–SNAP to TRUE. However, if you move or resize a widget in the frame, it then snaps to the nearest grid point.

25

Interface Design

This chapter describes issues that relate to the layout of your application interface. The topics discussed include:

- Progress windows
- Frame characteristics
- Frame design issues
- Tab order
- Active window display
- Three-dimensional effects (Windows only)
- Windows interface design options

25.1 Progress Windows

The user interface of a Progress application consists of one or more windows whose size and number affect the space available for frames. Depending on your environment (character or graphical), there are different factors to consider when you lay out frames. The following sections describe these factors.

25.1.1 Character-based Environment

A character-based application can use only one window (the default window), which maps directly to the terminal screen. Progress can accommodate character-based terminal screens of different sizes. The space within the terminal screen, except for the bottom three lines, is available to frames. Progress reserves two of these lines for error messages and messages produced with the MESSAGE statement. Progress reserves the remaining line for status and help messages.

Figure 25–1 shows the basic layout of a character-based Progress window.

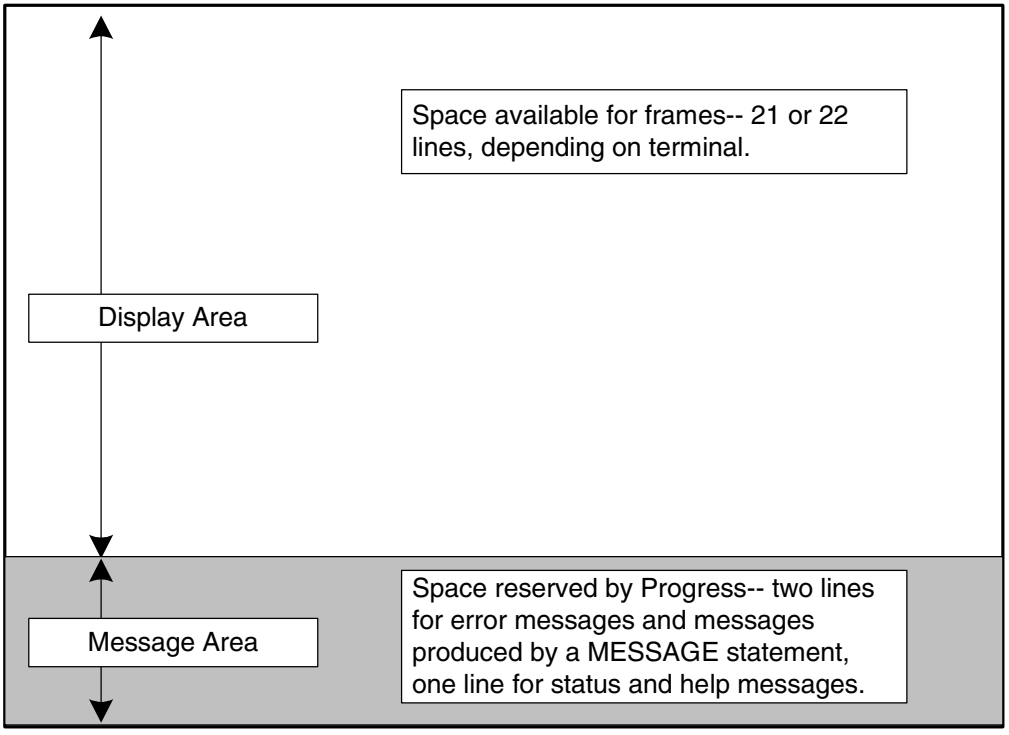


Figure 25–1: Character-based Window

Screen Variations

Most character-based terminal screens have 24 display lines, leaving 21 lines for frames. Some have 25 display lines, leaving 22 lines for frames. Other systems might have more display lines. To see how many lines are available, use the SCREEN–LINES function.

Character Display Height

You must size your frames to fit the display area. If you think some users might run your application on terminals that have 24 lines while others might run on terminals with more lines, design your frames to fit in the smaller display area. This way, you can be sure that your frames display in either situation.

Character Display Width

You must also size your frames to fit the width of the terminal screen. Most terminal screens are 80 characters wide, although some are wider. You can take advantage of the entire width of your terminal screen (up to 255 characters) when developing an application. However, if you take advantage of screen width greater than 80 characters, your frames do not fit or look the same on a screen with a smaller width.

Multi-window Simulation

Although character-based applications can only work with the terminal screen as a single window, you can simulate multiple windows by providing a menu bar and changing that menu bar for each simulated window context.

For example, suppose you want to simulate two windows with the following menu bars:

p-wbars.p

```
DEFINE MENU BasicWindow MENUBAR
  MENU-ITEM mfile      LABEL "File"
  MENU-ITEM medit      LABEL "Edit"
  MENU-ITEM moptions    LABEL "Options"
  MENU-ITEM madvance    LABEL "Advanced..."
  MENU-ITEM mexit      LABEL "E&xit".

DEFINE MENU AdvancedWindow MENUBAR
  MENU-ITEM mspec      LABEL "Special"
  MENU-ITEM medit      LABEL "Edit"
  MENU-ITEM moptions    LABEL "Calculations"
  MENU-ITEM madvance    LABEL "Basic...".
```

You can change between “windows” with code like this, used to complete `p-wbars.p`.

```
ON CHOOSE OF MENU-ITEM madvance
  CURRENT-WINDOW:MENUBAR = MENU AdvancedWindow:HANDLE.

ON CHOOSE OF MENU-ITEM mbasic
  CURRENT-WINDOW:MENUBAR = MENU BasicWindow:HANDLE.

CURRENT-WINDOW:MENUBAR = MENU BasicWindow:HANDLE.

WAIT-FOR CHOOSE OF MENU-ITEM mexit.
```

Of course, a real application likely has submenus in place of most of the menu–item entries shown in this example.

For more information on creating menu bars and submenus, see Chapter 22, “Menus.”

25.1.2 Graphical Environment

A graphical application can create and use multiple windows, including the default static window. Unlike character-based environments, where the window size is identical to the terminal screen, applications in graphical environments can create windows with any dimensions smaller, equal to, or larger than the terminal screen (with scroll bars, if necessary). Also, whereas characters in character-based windows typically have a fixed size and font (which is sometimes selectable), a graphical window can support many character sizes and styles, depending on the application and user interface. Graphical windows can also support a wide variety of frame sizes both smaller and larger than the window dimensions (with scroll bars, if necessary).

Character Units and Pixels

In a graphical environment, window layouts are generally measured in pixels, not characters. However, Progress lets you plan your window layout either in pixels or in character units. A character unit is equal in height to a fill-in widget using the default system font; it is equal in width to the average width of the characters in the default system font. Since the default system font determines the size of a character unit, you must consider what system font will be used when your application is running.

If you want your application to be portable across different platforms (or across different display resolutions within a single platform), use character units to lay out your screen displays. This way, you do not have to calculate your application layout in pixels, adjusting your calculations for different screen resolutions or different platforms.

Character Unit and Pixel Conversion

Note that character units are decimal values and pixels are integer values in Progress. If you set a pixel attribute to a decimal value, it is rounded to the nearest whole integer. This also means that if you set a character unit attribute (such as HEIGHT–CHARS) and then read it back, Progress can return a different value than the one you set based on the actual corresponding pixel value. This is a necessary rounding error because all graphic dimensions are ultimately stored as pixels, and the nearest whole pixel dimension might not exactly match the character units you specify.

For example, depending on the resolution and default system font of your interface, if you set the HEIGHT–CHARS attribute to 2.5, Progress might store and return its value as 2.51. This is because 2.51 most closely matches the number of pixels corresponding to 2.5 character units.

Thus, if your application uses character units to track widget size and location, be sure to reset your initial values to the values that Progress actually stores and returns.

Window Size

The window attributes in Table 25–1 let you query the interior size of a *maximized* window. When a window is maximized, it fills the entire terminal screen. The values returned by these attributes are the same whether or not the window is maximized.

Table 25–1: Maximized Window Attributes

Attribute	Description
FULL–HEIGHT–CHARS	The interior height of the window in character units.
FULL–HEIGHT–PIXELS	The interior height of the window in pixels.
FULL–WIDTH–CHARS	The interior width of the window in character units.
FULL–WIDTH–PIXELS	The interior width of the window in pixels.

The values returned by these attributes exclude the title bar, menu bar, message area, and status area. The interior of a window cannot exceed the area returned by these attributes. For more information on using graphical windows, Chapter 21, “Windows.”

Frame Borders

The frame attributes in Table 25–2 let you query the border widths of your frames.

Table 25–2: Frame Border Attributes

Attribute	Description
BORDER–BOTTOM–CHARS	The height of the bottom frame border in character units.
BORDER–BOTTOM–PIXELS	The height of the bottom frame border in pixels.
BORDER–TOP–CHARS	The height of the top frame border in character units.
BORDER–TOP–PIXELS	The height of the top frame border in pixels.
BORDER–LEFT–CHARS	The width of the left frame border in character units.
BORDER–LEFT–PIXELS	The width of the left frame border in pixels.
BORDER–RIGHT–CHARS	The width of the right frame border in character units.
BORDER–RIGHT–PIXELS	The width of the right frame border in pixels.

The values returned by these attributes change depending on whether the frame is selectable or movable. The BORDER–TOP and BORDER–TOP–CHARS attributes also change if the frame has a title.

25.1.3 Alert Boxes

An alert box is a special dialog box for displaying Progress messages instead of displaying them in the window message area. Alert boxes have no handles, attributes, or methods that you can access, and they do not respond to trigger events. They exist only to display messages. For information on defining and using dialog boxes for other purposes, see the "Dialog Boxes" section.

You can have Progress display both application and system messages in an alert box using these techniques:

- To put application messages in an alert box, invoke the MESSAGE statement with the VIEW–AS ALERT–BOX option.
- To have all Progress system messages displayed in an alert box, set the SYSTEM–ALERT–BOXES attribute of the SESSION system handle to TRUE.

25.2 Frame Characteristics

You can change the overall characteristics of a frame by using Frame phrase options or by setting frame attributes. The following sections describe these options and attributes and the rules for setting them.

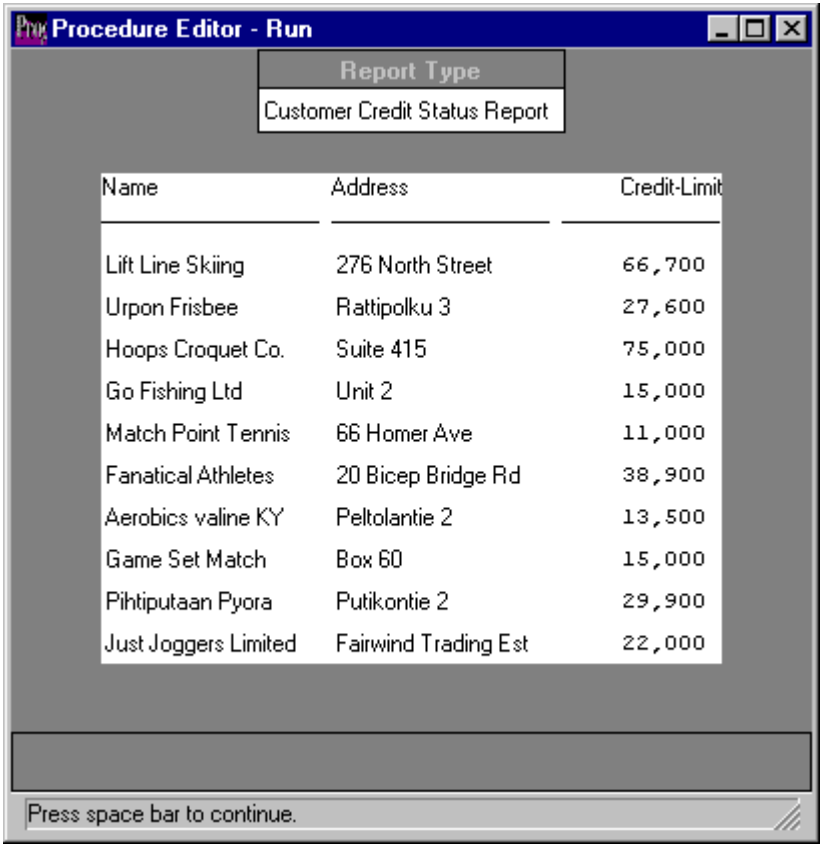
25.2.1 Using Frame Phrase Options

You can use a Frame phrase to describe the overall characteristics of a frame. A Frame phrase always begins with the word WITH and is followed by one or more options. The following procedure shows how you can use Frame phrases to control the appearance of your frames:

p-frm15.p

```
DISPLAY "Customer Credit Status Report"
        WITH TITLE "Report Type" CENTERED.

FOR EACH customer:
        DISPLAY name address credit-limit
                WITH NO-BOX 10 DOWN CENTERED RETAIN 2.
END.
```

This procedure produces the the following output:



In this example, the first DISPLAY statement uses the Frame phrase WITH TITLE “Report Type” CENTERED:

- The CENTERED option centers the frame on the display screen.
- The TITLE option names a title to use with the DISPLAY. Progress always places the title in the center of the top line of the box that surrounds the frame being displayed.

The second DISPLAY statement displays customer information:

- The 10 DOWN option displays only 10 customers on the screen at a time.
- The RETAIN 2 option redisplays the last two customers at the top of the screen after clearing the screen to make room for the next set of customers.
- The CENTERED option centers the display of customer records.

For more information on Frame phrase options, see the *Progress Language Reference*.

25.2.2 Setting Frame Attributes

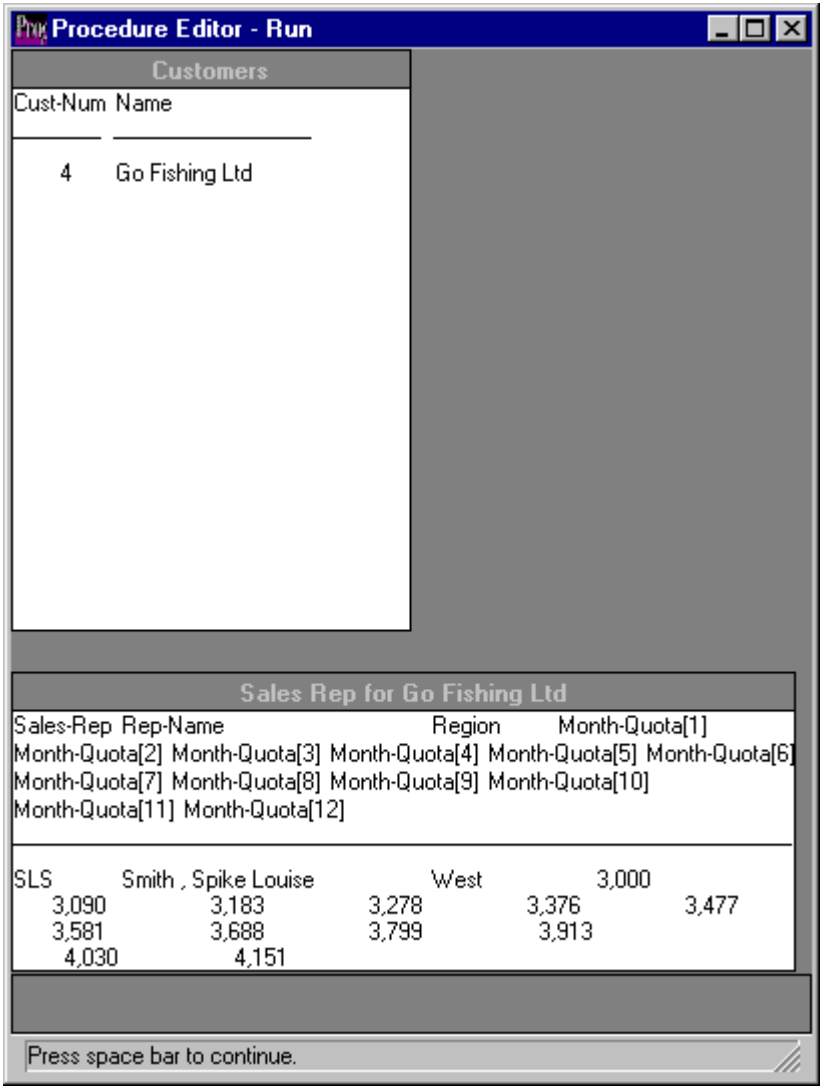
The following procedure shows how you can set frame attributes to control the appearance of frames:

p-frm16.p

```
DEFINE FRAME a
   customer.cust-num customer.name
   WITH DOWN USE-TEXT WIDTH 40 TITLE "Customers".

DEFINE FRAME b
    salesrep
    WITH USE-TEXT TITLE "Sales Rep".

FRAME a:HEIGHT-CHARS = SCREEN-LINES - (FRAME b:HEIGHT-CHARS + 1).

FOR EACH customer, salesrep OF customer WITH FRAME a:
   DISPLAY cust-num name.
   FRAME b:TITLE = "Sales Rep for " + customer.name.
   DISPLAY salesrep WITH FRAME b.
END.
```

This procedure adjusts the height of frame using the HEIGHT–CHARS attribute so that both a and b fit on the screen. In each iteration of the FOR EACH loop, the procedure resets the title of frame b to include the name of the current customer using the TITLE attribute.

This procedure produces the following output:



Many frame attributes correspond in name and function to the options of the Frame phrase. For more information on frame attributes, see the reference entries for the FRAME Widget and for the frame attributes in the *Progress Language Reference*.

25.2.3 Rules for Frame Phrases and Attributes

When you use a Frame phrase or frame attribute, the phrase or attribute applies to the entire frame regardless of the statements that use that frame. Different Frame phrases that apply to the same frame cannot conflict. For example:

```
DISPLAY name WITH FRAME a COLUMN 15.
```

conflicts with:

```
DISPLAY address WITH FRAME a COLUMN 30.
```

You cannot simultaneously display the same frame at column 15 and column 30.

25.3 Frame Design Issues

There are several issues that you must consider when you design frames. For example, you might want to use a certain background image for a frame. This section describes this and other frame design issues.

25.3.1 Frame-level Design

The following procedure shows how Progress designs a frame:

p-frm17.p

```
DEFINE BUTTON del-button LABEL "Delete Customer"
   TRIGGERS:
      ON CHOOSE
          DELETE customer.
   END.
DEFINE BUTTON next-button LABEL "Find Another Customer" AUTO-GO.
DEFINE BUTTON quit-button LABEL "Quit" AUTO-ENDKEY.

REPEAT:
        PROMPT-FOR customer.cust-num quit-button WITH FRAME a.
        FIND customer USING cust-num.
        DISPLAY name.
        UPDATE del-button next-button quit-button WITH FRAME a.
END.
```

Every data-handling statement in a procedure serves two purposes:

- To specify frame contents and layout at compile time
- To cause frame activity at run time

When Progress compiles this procedure, it designs frames as follows:

- The REPEAT block, since it displays data, automatically receives a frame. This frame is an unnamed frame and is the default frame for the block.
- The PROMPT–FOR statement uses frame a, not the default frame for the REPEAT block (because frame a is named explicitly). Progress sets up a frame that is large enough to hold the cust–num field and the quit–button.
- The DISPLAY statement does not name a specific frame, so it uses the default frame for the REPEAT block. Progress allocates enough room in that frame for the name field.
- The UPDATE statement also names frame a. Progress allocates more room in that frame to hold the buttons del–button and next–button (the frame initially was only big enough to hold the cust–num field).

To summarize, there are two frames used in this procedure. One frame, the default frame for the REPEAT block, displays the customer name. The second frame, frame a, holds the cust–num field and three buttons.

These are the defaults Progress uses when designing a frame:

- Start the frame in column 1, at the first free row on the screen.
- Enclose the frame in a box.
- Display field labels above fields (that is, use column labels).
- Underline the column labels.
- Make the frame as wide as necessary to accommodate all the items in the frame, adding more lines if everything does not fit on one line.

25.3.2 Field- and Variable-level Design

Progress designs frames at compile time. In a top-to-bottom pass of your procedure, the Progress Compiler encounters fields, variables, and expressions and their related format specifications. The Compiler adds these fields and expressions to the layout of the appropriate frame. Progress always designs for all possible cases. That is, if there are several fields that might be displayed in a frame, Progress makes room for all of them.

As Progress designs each frame, it makes some decisions regarding the individual fields and variables in the frame:

- All references to the same field or variable map to the same frame position.
- If labels are above the data (column labels), each field or variable is allocated a column. The width of the column is either the width of the format or the width of the label, whichever is larger.
- Array frame field labels are followed by a subscript number in square brackets. Progress determines the labels at compile time. It omits subscripts if the array subscript is variable or if you specify a label in the frame definition by using the LABEL keyword in a format phrase.
- Constants used in DISPLAY are treated as expressions, each occupying a separate frame field.

For more information on field and variable display formats, see the DISPLAY Statement reference entry in the *Progress Language Reference*. Also see Chapter 17, "Representing Data." in this manual.

25.3.3 FORM and DEFINE FRAME Statements

Often, you want to describe characteristics of a frame but do not want to include that description in the Frame phrase of a data-handling statement. This is especially true if the frame characteristics are very complex. You can use the FORM or DEFINE FRAME statements to describe the layout and processing properties of a certain frame. Specifically, you use these statements to:

- Lay out frame fields in one order when you are going to process them in another order.
- Describe frame headers and backgrounds.

The FORM and DEFINE FRAME statements only describe the layout of a frame. They do not actually bring the frame into view. To see the frame, you must use either a data-handling statement that uses that frame or the VIEW statement, or set the frame VISIBLE attribute to TRUE.

The location of the FORM statement affects the scope of your frame, so you must place any FORM statements you use within the appropriate block or blocks of a procedure.

Normally, the first reference to a frame scopes the frame to the current block. However, the DEFINE FRAME statement does not scope the frame. The FORM statement, like data handling statements, does scope the frame. For more information on these statements, see the *Progress Language Reference*.

Laying Out Frame Fields

When you are using a frame that has a very complex layout or when you want to use the same frame layout many times in a single procedure, you can use the FORM or DEFINE FRAME statements to describe the frame. That way, the frame description is in only one place, not scattered throughout the procedure.

In the following procedure, the DEFINE FRAME statement describes the layout of frame a. By default, Progress displays the fields in the order they appear in the DEFINE FRAME statement. You can use the AT phrase to explicitly position a field. When you run the procedure and update the fields, you move through them in the order they appear in the UPDATE statement. That is, the order of the fields in the DEFINE FRAME (or FORM) statement affects the positions of the fields, not their tab (or processing) order.

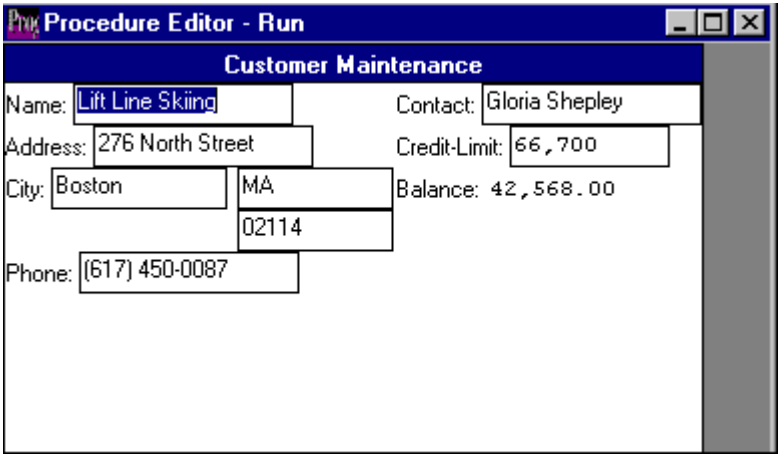
When you run p-form4.p, press **TAB** to move through the fields. Notice that the cursor moves among the fields in the order they are listed in the UPDATE statement:

p-form4.p

```
DEFINE FRAME a
     name contact AT 40 SKIP
     address credit-limit AT 40 SKIP
     city customer.state NO-LABEL SKIP
     postal-code AT COLUMN-OF customer.state ROW-OF city + 1  NO-LABEL
     SKIP
     phone balance AT COLUMN 40 ROW-OF city SKIP(1)
     WITH SIDE-LABELS TITLE "Customer Maintenance".

ASSIGN customer.state:WIDTH-CHARS IN FRAME a = postal-code:WIDTH-CHARS.

FOR EACH customer WITH FRAME a:
        DISPLAY balance.
        UPDATE name address city state postal-code phone
               contact credit-limit.
END.
```

NOTE: The AT *n* option, where *n* is an integer, specifies the column position of the field. Progress chooses the row based on the previous widget and form item layout of the frame. Form items include SKIP and SPACE. For more information on form items, see the reference entries for the DEFINE FRAME statement and the FORM statement in the *Progress Language Reference*.

This procedure produces the following output:



Note that because the DEFINE FRAME statement is used, the frame is scoped to the FOR EACH block. It, therefore, becomes a down frame. A FORM statement would have scoped the frame to the procedure block, making it a one-down frame. For more information on frame scoping, see Chapter 19, "Frames."

The syntax of the AT phrase of the FORM and DEFINE FRAME statements can specify either the absolute column and row, or the X and Y coordinates of the display field. You can also position a field relative to a previously positioned field.

SYNTAX

```
AT
  {      n
       | { COLUMN column | COLUMN-OF relative position }
           {ROW row | ROW-OF relative-position }
       |
         { X x | X-OF relative-position}
           {Y y | Y-OF relative-position }
  }
  [ COLON-ALIGNED | LEFT-ALIGNED | RIGHT-ALIGNED ]
```

For more information on this syntax, see the AT Phrase reference entry in the *Progress Language Reference*.

The following sample fragment specifies X and Y coordinates for two fields in a frame:

p-frm19.p

```
FORM
   customer.cust-num AT X 50 Y 14
   customer.name AT X 200 Y 50
   WITH SIDE-LABELS FRAME xcust.

FIND FIRST customer.
DISPLAY cust-num name WITH FRAME xcust.
```

If you position a field with X and Y coordinates, the position of the field depends on the resolution of the terminal on which the procedure is compiled and run. For example, if the resolution is 640 by 480 and you specify X 320, the field is displayed at the midpoint of the display. If you run the same procedure on a display with a resolution of 1280 by 960, the field is displayed at the one-quarter point of the screen.

Most interfaces provide a font that is compatible with the screen resolution. An 80-column display width is common across most platforms. Therefore, if you use character (ROW and COLUMN) coordinates rather than pixel (X and Y) coordinates, your code is more portable. Because Progress allows you to specify fractional character units, you can still specify very precise locations for a graphical environment. In a character environment, the ROW and COLUMN values are truncated to integer values.

Describing Frame Headers

You can use the FORM or DEFINE FRAME statement to describe a header that appears at the top of a frame. For example, the FORM statement in this procedure defines a frame that consists of only a header. The VIEW statement brings that frame into view. The DISPLAY statement uses a separate frame (the default frame for the FOR EACH block).

p-form3.p

```
FORM HEADER "Customer Credit Status Report" WITH CENTERED.
VIEW.

FOR EACH customer WITH CENTERED:
    DISPLAY name credit-limit.
END.
```

This procedure produces the following output:



Form Backgrounds

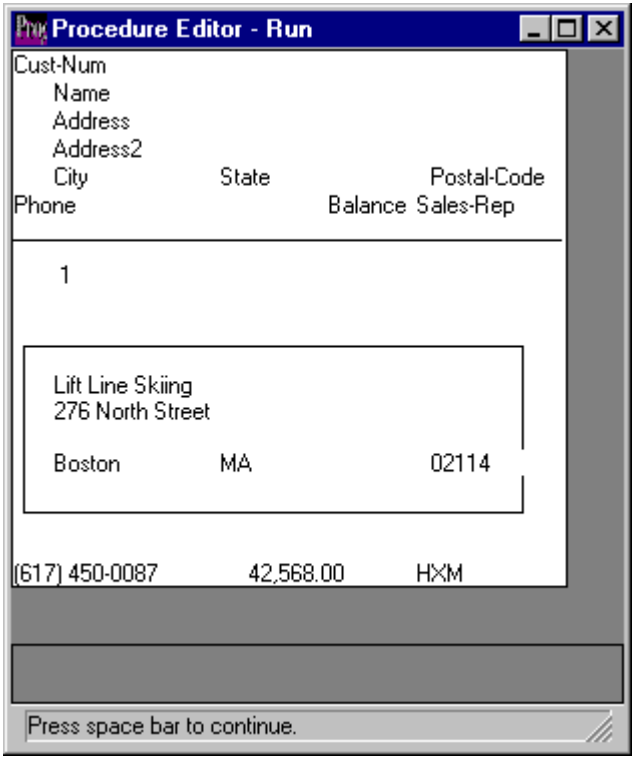
The BACKGROUND option of the FORM and DEFINE FRAME statements allow you to specify a rectangle or image to display in the background for a frame. The contents of the frame (both header and data) are displayed on top of the background.

For example, the following procedure displays a rectangle in the background of a frame:

p-back.p

```
DEFINE RECTANGLE back-rect  SIZE 50 BY 4 NO-FILL EDGE-PIXELS 1.

DEFINE FRAME x
    customer.cust-num  SKIP(2)
    customer.name AT 5 SKIP
    customer.address AT 5 SKIP
    customer.address2 AT 5 SKIP
    customer.city AT 5 customer.state customer.postal-code SKIP(2)
    customer.phone customer.balance customer.sales-rep
    BACKGROUND back-rect AT COLUMN 2 ROW 8
    WITH USE-TEXT.

FOR EACH customer:
    DISPLAY cust-num name address address2 city state postal-code
            phone balance sales-rep WITH FRAME x.
END.
```

This procedure produces the following output:



Note that you must define a rectangle or image before referencing it in the DEFINE FRAME or FORM statement.

25.3.4 FRAME–ROW and FRAME–COL Options

When you design an application, you can control where a frame appears on the screen and where one frame overlays another. Progress has several functions that let you control the position of frames in relation to the screen and in relation to other frames.

In the following procedure, the FRAME–ROW and FRAME–COL functions indicate the location of the order-information frame on the customer-information frame.

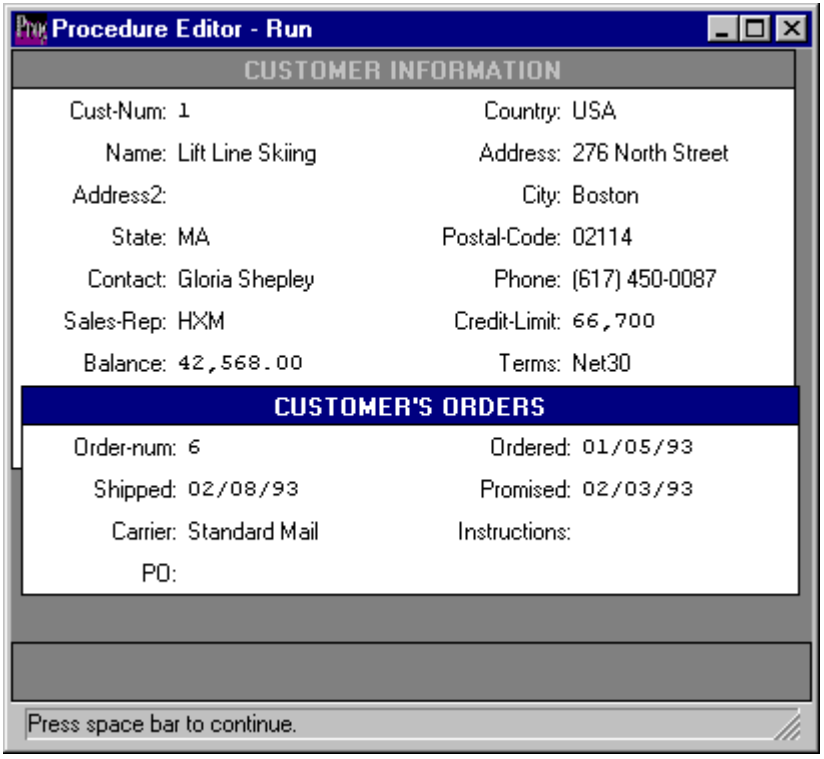
NOTE: The FRAME–ROW and FRAME–COL attributes have no relation to the FRAME–ROW and FRAME–COL functions. These attributes return the position of a child field-level widget relative to the upper left-hand corner of the parent frame. However, for child frames, the ROW and COLUMN attributes return the position relative to the upper left-hand corner of the parent frame. For more information on child frames, see Chapter 19, “Frames.”

p-frrow.p

```
cust-loop:
FOR EACH customer:
        DISPLAY customer WITH FRAME cust-frame
                2 COLUMNS TITLE "CUSTOMER INFORMATION".
        FOR EACH order OF customer
                    ON ENDKEY UNDO cust-loop, LEAVE cust-loop:
          DISPLAY order-num order-date ship-date promise-date
                  carrier instructions po
                  WITH 2 COLUMNS 1 DOWN OVERLAY TITLE "CUSTOMER'S ORDERS"
                       ROW FRAME-ROW(cust-frame) + 8
                       COLUMN FRAME-COL(cust-frame) + 1.
        END.
END.
```

The FRAME–ROW and FRAME–COL functions each return an integer value that represents the row or column position of the upper-left corner of the named frame. In the example, the FRAME–ROW function returns the value of the row position of the uppermost corner of the cust–frame. Then the procedure adds 8 to that value and displays the overlay frame at that row position. The column position is calculated to be 1 to the right of the right edge of cust–frame.

This procedure produces the following output:



If you move cust–frame to the third row on the screen, the FRAME–ROW and FRAME–COL functions place the order information frame in the same position relative to the customer information frame–below the eighth row and next to the first column of cust–frame. The following procedure shows this adjustment.

p-frrow1.p

```
cust-loop:
FOR EACH customer:
        DISPLAY customer WITH FRAME cust-frame ROW 3
                2 COLUMNS TITLE "CUSTOMER INFORMATION".
        FOR EACH order OF customer
                  ON ENDKEY UNDO cust-loop, LEAVE cust-loop:
          DISPLAY order-num order-date ship-date promise-date
                  carrier instructions po
                  WITH 2 COLUMNS 1 DOWN OVERLAY TITLE "CUSTOMER'S ORDERS"
                       ROW FRAME-ROW(cust-frame) + 8
                       COLUMN FRAME-COL(cust-frame) + 1.
        END.
END.
```

See the *Progress Language Reference* for more information on the FRAME–ROW and FRAME–COL functions.

In most cases, a dialog box is better than an overlay frame. For example, `p-frrow2.p` is similar to `p-frrow1.p`, but uses a simple dialog box instead of the overlay frame:

p-frrow2.p

```
DEFINE BUTTON ok-button LABEL "OK" AUTO-GO.
DEFINE BUTTON cancel-button LABEL "CANCEL" AUTO-ENDKEY.

cust-loop:
FOR EACH customer:
        DISPLAY customer WITH FRAME cust-frame ROW 3
                2 COLUMNS TITLE "CUSTOMER INFORMATION".
        FOR EACH order OF customer
                    ON ENDKEY UNDO cust-loop, LEAVE cust-loop:
          DISPLAY order-num order-date ship-date promise-date
                  carrier instructions po SKIP
                  ok-button AT 25 cancel-button AT 50
                  WITH 2 COLUMNS 1 DOWN OVERLAY TITLE "CUSTOMER'S ORDERS"
                       ROW FRAME-ROW(cust-frame) + 8
                       COLUMN FRAME-COL(cust-frame) + 1
                       VIEW-AS DIALOG-BOX.
          SET ok-button cancel-button.
        END.
END.
```

When you run this code in a graphical interface, you can move the dialog box to see all the data in both frames. For more information, see the discussion of dialog boxes later in this chapter.

25.3.5 Positioning Frames with FRAME–LINE

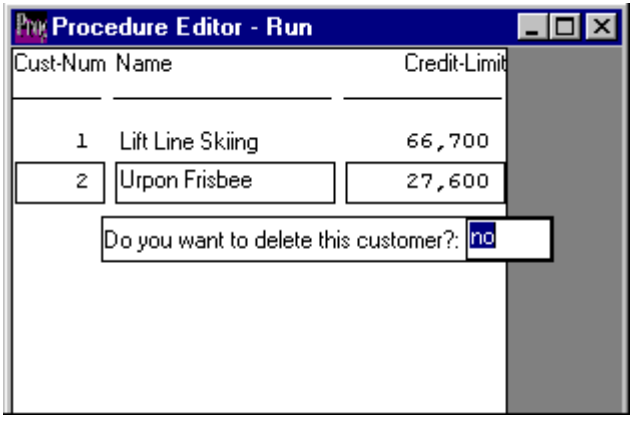
The FRAME–LINE function gives you information on the position of the current display line in a frame. The following procedure uses this information to determine how to display other information in the frame:

p-frline.p

```
DEFINE VARIABLE ans AS LOGICAL LABEL
        "Do you want to delete this customer?".

DEFINE FRAME a customer.cust-num customer.name customer.credit-limit.

STATUS INPUT "Enter data, or press CTRL-G to delete the customer".

ON CTRL-G OF customer.cust-num, customer.name, credit-limit
   DO:
      UPDATE ans WITH ROW FRAME-ROW(a) + 2 + FRAME-LINE(a) COLUMN 10
         SIDE-LABELS OVERLAY FRAME del-frame.
      IF ans THEN DELETE customer.
      HIDE FRAME del-frame.
   END.

REPEAT WITH FRAME a 10 DOWN:
    FIND NEXT cust.
    UPDATE cust-num name credit-limit.
END.
```

The `p-frline.p` procedure lets the user update a customer's number, name, and credit limit. The procedure displays information in the frame for one customer at a time. You can press **CTRL–G** at any time to see a prompt to delete the customer. The prompt always appears in its own frame, below the last customer displayed.

This procedure produces the following output:



The FRAME–LINE function in this procedure controls where the ans variable is displayed. The position of the prompt is calculated from the upper-right corner of frame a and the current line within the frame. That is, FRAME–ROW + 1 + FRAME–LINE gives the position of the current line in the frame, taking into account the lines for the frame box and the labels. The prompt is placed below the current line.

See the *Progress Language Reference* for more information on the FRAME–LINE function.

25.3.6 Positioning Frames with FRAME–DOWN

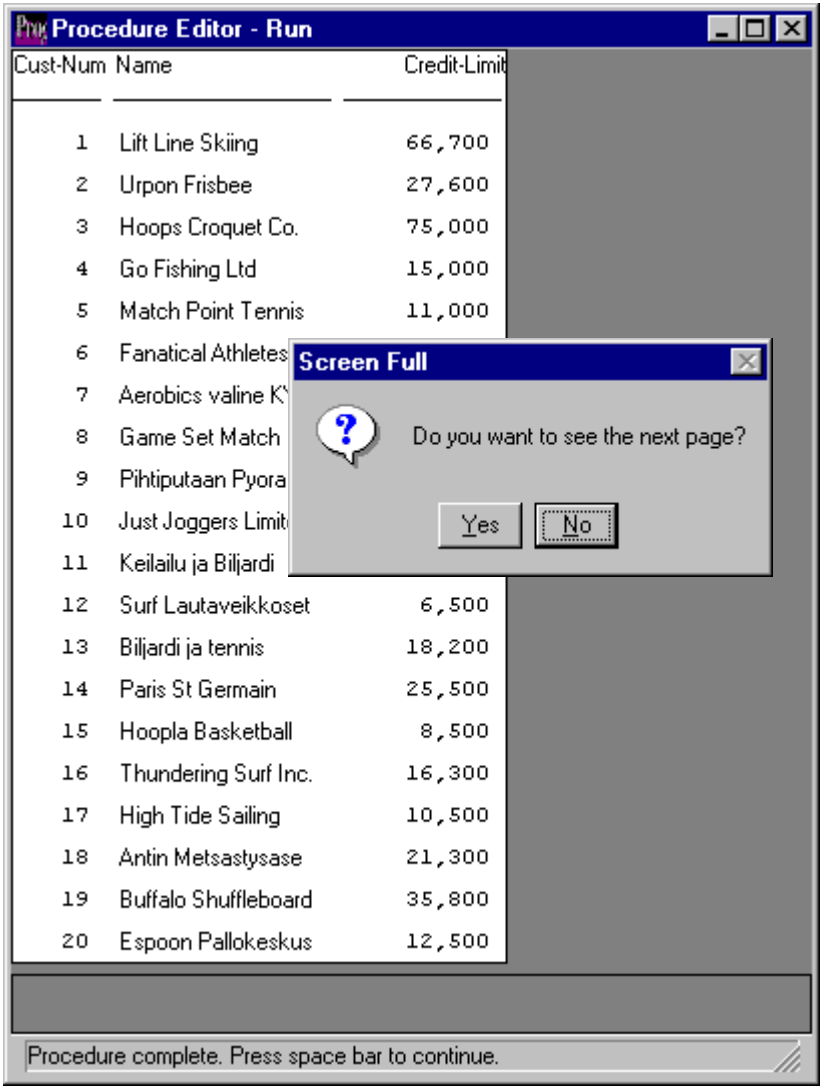
The FRAME–DOWN function returns the number of iterations that can fit in a frame. You can use this function to determine if a down frame is full. Then Progress can perform an action based on the value of FRAME–DOWN.

For example, you can display all the customers in the database, which takes several screens. If the customer you want to see is on the first screen, it is time-consuming to press **RETURN** until every customer in the database is listed and the procedure ends. The following procedure uses the FRAME–DOWN function to solve this problem:

p-frdown.p

```
DEFINE VARIABLE ans AS LOGICAL.

FOR EACH customer:
  DISPLAY cust-num name credit-limit.
  IF FRAME-LINE = FRAME-DOWN
  THEN DO:
    MESSAGE "Do you want to see the next page?" VIEW-AS ALERT-BOX
         QUESTION BUTTONS YES-NO TITLE "Screen Full"
         UPDATE ans.
    IF NOT ans
    THEN LEAVE.
  END.
END.
```

This procedure produces the following output:



Each time the frame becomes full, the question “Do you want to see the next page?” appears in a dialog box. If you answer No, the procedure ends.

In this procedure, the FRAME–DOWN function returns the number of iterations that can fit in a frame, and the FRAME–LINE function returns the current logical line in a frame. When the current logical line equals the number of lines in the frame, the procedure displays the message “Do you want to see the next page?” and gives the user the option to continue or to end the procedure.

Note that the DOWN and LINE attributes are analogous to the FRAME–DOWN and FRAME–LINE functions.

See the *Progress Language Reference* for more information on these functions and attributes.

25.3.7 Scrolling Frames

Scrolling frames are specialized down or one-down frames that allow the user to move a highlight bar through multiple iterations of data. Most often, scrolling frames are down frames. You use the SCROLL Frame phrase option together with the CHOOSE statement to create this type of frame.

NOTE: This feature is supported for backward compatibility. You can usually use either a browse widget or selection list instead of CHOOSE and SCROLL. For more information on browse widgets, see Chapter 9, "Database Access." For information on selection lists, see Chapter 17, "Representing Data."

The SCROLL statement opens a row and moves data in a frame with multiple rows. You can use the SCROLL statement to scroll data up or down to display another line in a frame. The following procedure shows how you might use this statement:

p-scroll.p *(1 of 3)*

```
DEFINE VARIABLE current_line AS INTEGER.

FORM customer.cust-num customer.name customer.address
     customer.city customer.postal-code
     WITH FRAME cust-frame SCROLL 1 5 DOWN WIDTH 90.

CURRENT-WINDOW:WIDTH-CHARS = 90.

FIND FIRST customer.
```

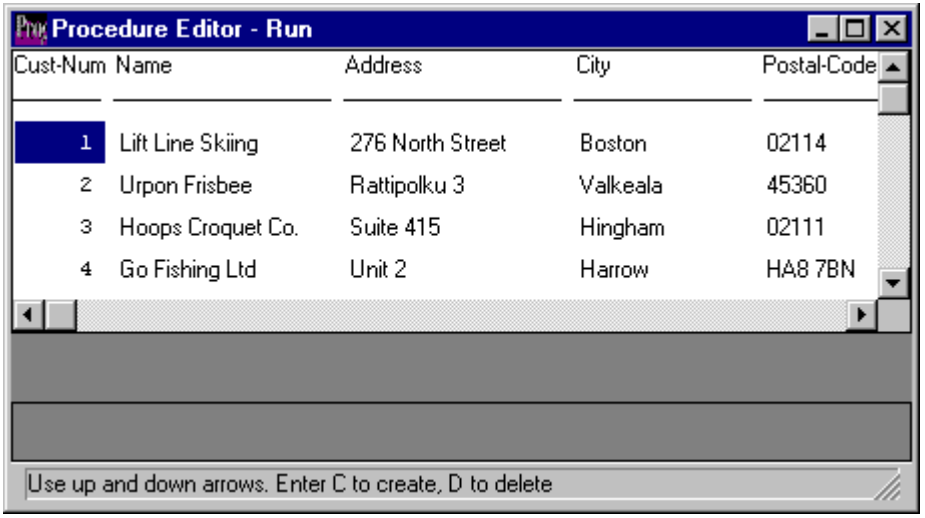
p-scroll.p *(2 of 3)*

```
REPEAT current_line = 1 TO 5:
    DISPLAY cust-num name address city postal-code
       WITH FRAME cust-frame.
    DOWN WITH FRAME cust-frame.
    FIND NEXT customer NO-ERROR.
    IF NOT AVAILABLE customer
    THEN LEAVE.
END.

UP 5 WITH FRAME cust-frame.

REPEAT:
   STATUS DEFAULT
      "Use up and down arrows. Enter C to create, D to delete".
    CHOOSE ROW customer.cust-num NO-ERROR GO-ON(CURSOR-RIGHT)
        WITH FRAME cust-frame.
    FIND customer WHERE cust-num = INTEGER(INPUT cust-num).

    /* React to moving cursor off the screen */
    IF LASTKEY = KEYCODE("CURSOR-DOWN")
    THEN DO:
        FIND NEXT customer NO-ERROR.
        IF NOT AVAILABLE customer
        THEN FIND FIRST customer.
        DOWN WITH FRAME cust-frame.
        DISPLAY cust-num name address city postal-code
          WITH FRAME cust-frame.
        NEXT.
    END.

    IF LASTKEY = KEYCODE("CURSOR-UP")
       THEN DO:
          FIND PREV customer NO-ERROR.
          IF NOT AVAILABLE customer
          THEN FIND LAST customer.
          UP WITH FRAME cust-frame.
          DISPLAY cust-num name address city postal-code
            WITH FRAME cust-frame.
          NEXT.
    END.
```

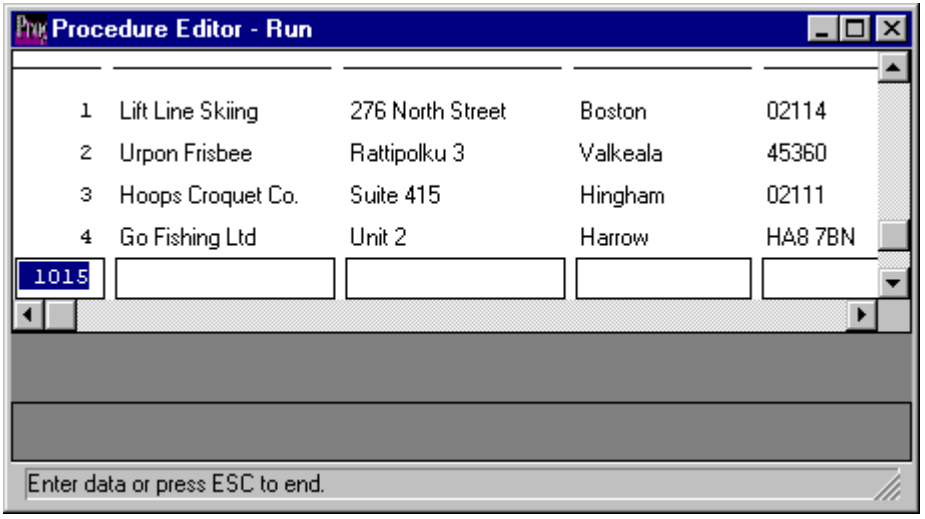
p-scroll.p *(3 of 3)*

```
     /* CHOOSE selected a valid key.  Check which key. */
    IF KEYLABEL(LASTKEY) = "c"
       THEN DO:
          /* Open a space in the frame */
          SCROLL FROM-CURRENT DOWN WITH FRAME cust-frame.
          CREATE customer.
          UPDATE cust-num name address city postal-code
            WITH FRAME cust-frame.
          NEXT.
    END.

    IF KEYLABEL(LASTKEY) = "d"
      THEN DO:
        /* Delete a customer from the database */
        DELETE customer.
        SCROLL FROM-CURRENT WITH FRAME cust-frame.
        current_line = FRAME-LINE(cust-frame).
        DOWN FRAME-DOWN(cust-frame) - FRAME-LINE(cust-frame) - 1
          WITH FRAME cust-frame.
          /* Place cursor on last active line in frame */
        FIND customer WHERE cust-num = INTEGER(INPUT cust-num).
        FIND NEXT customer NO-ERROR.
        IF NOT AVAILABLE customer
        THEN FIND FIRST customer.
        DOWN WITH FRAME cust-frame.
        DISPLAY cust-num name address city postal-code
           WITH FRAME cust-frame.
        UP FRAME-LINE(cust-frame) - current_line
           WITH FRAME cust-frame.
           /* Move cursor back to where it was at start of block */
    END.
END.
STATUS DEFAULT.
```

This procedure produces the following output:



Use the arrow keys to move the highlighted bar to the fifth customer. Type **C** for create.



At this point, you can add a new customer to the database by typing the customer information on the open line. The `p-scroll.p` procedure uses SCROLL and CHOOSE to allow the user to browse through information, then perform actions with the information.

The `p-scroll.p` procedure creates a scrolling frame of five fields. The frame displays the cust–num, name, address, city, and postal–code for each customer. The status default message displays "Enter C to create, D to delete" as long as the procedure runs. You use arrow keys to move the highlighted cursor bar through lines of the scrolling frame, and you can add or delete customers from the database. The CHOOSE statement allows you to move the highlight bar.

The SCROLL statement controls the scrolling action in the frame when you create and delete customers. You add a customer to the database by typing **C**. Create opens a line in the frame and the SCROLL statement moves data below the line down. Then you type the new customer information into the frame. You type **D** to delete a customer from the database. When you delete a customer, SCROLL moves the rows below the deleted customer row up into the empty line.

The `p-scroll.p` procedure works as follows:

- The SCROLL option on the Frame phrase creates a scrolling frame for the customer information.
- The CHOOSE statement allows the user to scroll through the list of customer numbers with a highlighted bar, and it also allows the user to select a location to insert or delete an item from the list.
- When the user types **C** for create, the SCROLL FROM–CURRENT DOWN statement opens a space for another customer above the highlighted customer.
- When the user types **D** for delete, the SCROLL FROM–CURRENT statement closes the space left by the deleted customer. However, when you use a SCROLL FROM–CURRENT statement, Progress opens a line at the bottom of the scrolling frame. The remaining statements in this block fill in the opened frame line with record information.

The `p-scroll.p` procedure uses UP and DOWN statements to control the current cursor position. These statements behave differently with scrolling frames than with regular down frames. If the cursor is on the top line of a frame, the UP statement opens a line at the top of the screen and moves the remaining frames lines down one line. This is also what happens if you use the SCROLL DOWN statement. Similarly, if the cursor is on the bottom line of a frame, the DOWN statement opens a line at the bottom of the screen and moves the remaining frame lines down one line.

25.3.8 Scrollable Frames

You can define a scrollable frame that is bigger than the display space allotted for it. The user can then use scroll bars to view the entire frame.

You can explicitly set the maximum dimensions of a frame with the VIRTUAL–WIDTH–PIXELS, VIRTUAL–HEIGHT–PIXELS, VIRTUAL–HEIGHT–CHARS, and VIRTUAL–WIDTH–CHARS attributes. You can specify the display size of the frame with the SIZE option of the Frame phrase, as shown below.

SYNTAX

```
{ SIZE | SIZE-CHARS | SIZE-PIXELS } width BY height
```

If the SCROLLABLE attribute is FALSE, the frame is forced to fit into the display space. If SCROLLABLE is TRUE, then if the size as specified by the SIZE phrase does not fit within the display area, only a portion of the frame is displayed and scroll bars appear.

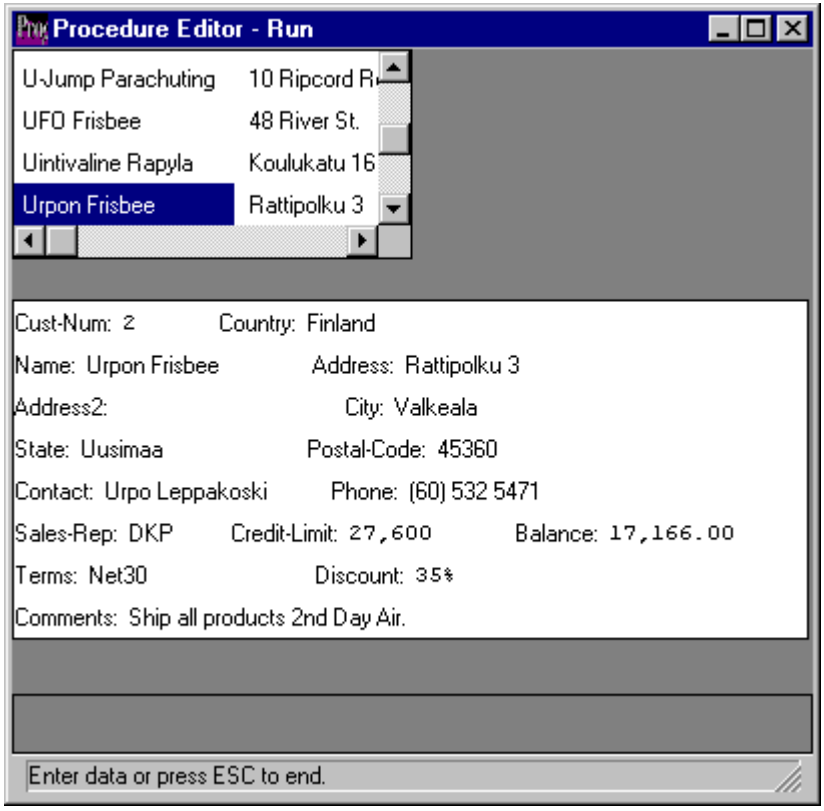
The SCROLLABLE option defaults to TRUE if you use the SIZE option; otherwise, it defaults to FALSE.

For example, the following code defines a frame, choose–frame, that is bigger than its display size:

p-scrlab.p

```
FORM
    Customer.name Customer.address Customer.address2
    Customer.city Customer.st
    WITH FRAME choose-frame 33 DOWN SIZE 40 BY 5.

PAUSE 0 BEFORE-HIDE.
FOR EACH customer BREAK BY Customer.name:
    DISPLAY name address address2 city st
            WITH FRAME choose-frame.
    IF NOT LAST(customer.name)
    THEN DOWN WITH FRAME choose-frame.
END.

REPEAT:
    CHOOSE ROW Customer.name WITH FRAME choose-frame.
    FIND customer WHERE Customer.name = FRAME-VALUE.
    DISPLAY customer WITH SIDE-LABELS.
END.
```

If you run this code and select Urpon Frisbee, the following screen appears:



You can use scroll bars to move within choose–frame (in the upper-left corner of the display) to see basic information on each customer. You can then choose to see more information on an individual customer.

In character interfaces, you can scroll a frame by pressing **SCROLL–MODE** (**ESC–T** on most terminals). This enters a separate scroll mode in which you can scroll the frame one row or column at a time with the cursor keys. You can page up and down with the **PAGE–UP, PAGE–DOWN, END, HOME, RIGHT–END**, and **LEFT–END** keys. No other frame input is possible in scroll mode. To exit scroll mode, press **SCROLL–MODE** or **END–ERROR**.

25.3.9 Strip Menus

A strip menu is a one-down frame that allows you to move a highlight bar between fields in an array or among fields in a table.

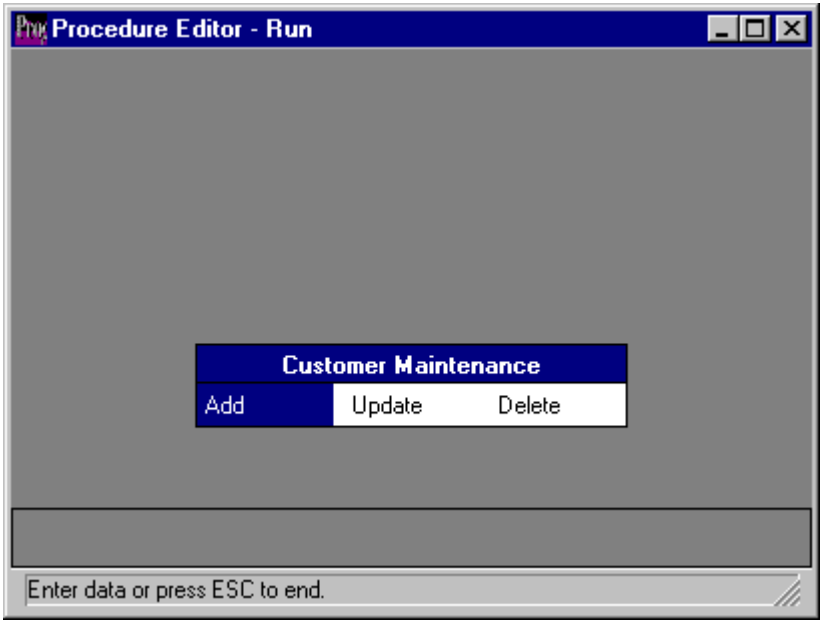
NOTE: Strip menus are supported for backwards compatibility. You can use buttons or menu widgets instead.

The following procedure sets up a simple strip menu:

p-strip.p

```
DEFINE VARIABLE abc AS character EXTENT 3
 INITIAL ["Add", "Update", "Delete"].

DISPLAY abc NO-LABELS WITH ROW 8 CENTERED
  TITLE "Customer Maintenance".

CHOOSE FIELD abc AUTO-RETURN.

IF FRAME-VALUE = "add" THEN MESSAGE "Add customer".
IF FRAME-VALUE = "update" THEN MESSAGE "Update customer".
IF FRAME-VALUE = "delete" THEN MESSAGE "Delete customer".
```

This procedure produces the following output:



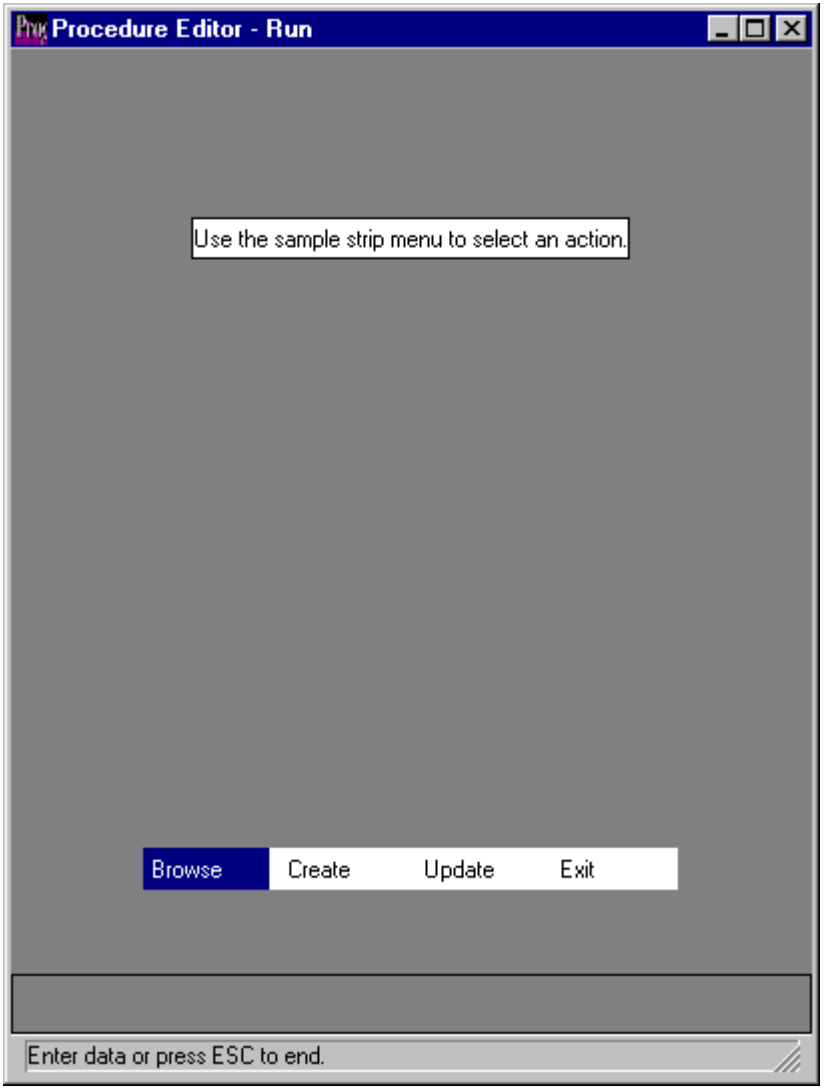
The `p-strip.p` procedure defines the array abc with an extent of 3 to display the selections on the menu. The CHOOSE statement allows you to move the highlight bar among the three fields in the array and to select a highlighted item by pressing **RETURN**. Progress displays a message based on the item CHOOSE holds in the frame. In your own application you can have Progress perform an action based on the item the user selects.

You can also use the CHOOSE statement to create a strip menu that appears at the bottom of the screen. The following procedure shows how you might do this:

p-chsmnu.p

```
DEFINE VARIABLE menu AS CHARACTER EXTENT 4 FORMAT "x(7)"
   INITIAL [ "Browse", "Create" , "Update", "Exit" ].

DEFINE VARIABLE proglist AS CHARACTER EXTENT 4
   INITIAL [ "brws.p", "cre.p", "upd.p", "exit.p"].

FORM "Use the sample strip menu to select an action."
  WITH FRAME instruc CENTERED ROW 5.

REPEAT:
   VIEW FRAME instruc.
   DISPLAY menu
      WITH NO-LABELS ROW SCREEN-LINES - 2 NO-BOX FRAME f-menu CENTERED.
   HIDE MESSAGE.
   CHOOSE FIELD menu AUTO-RETURN WITH FRAME f-menu.
   IF SEARCH(proglist[FRAME-INDEX]) = ?
   THEN DO:
       MESSAGE "The program" proglist[FRAME-INDEX] "does not exist.".
       MESSAGE "Please make another choice.".
   END.
   ELSE RUN VALUE(proglist[FRAME-INDEX]).
END.
```

This procedure produces the following output:



Use the arrow keys to move the highlight bar through the available selections and press **RETURN** when the bar highlights the selection you want. Because this is a sample procedure, none of the items perform actions other than returning messages.

Now look back at the `p-chsmnu.p` procedure. The procedure defines two arrays with an extent of four. The menu array holds the items for selection on the menu, and the proglist array holds the names of the programs associated with the menu selections. The CHOOSE statement allows the user to select an item from the strip menu. Progress finds the number in the menu array associated with the item, and then finds the program associated with the number in the proglist array. Progress runs the program if it exists. If it does not exist, Progress displays a message and allows the user to select another item from the strip menu.

In most cases, you use the SCROLL statement with the CHOOSE ROW statement to perform the actual work of selecting an item from a scrolling menu and taking an action based on that item.

25.3.10 Dialog Boxes

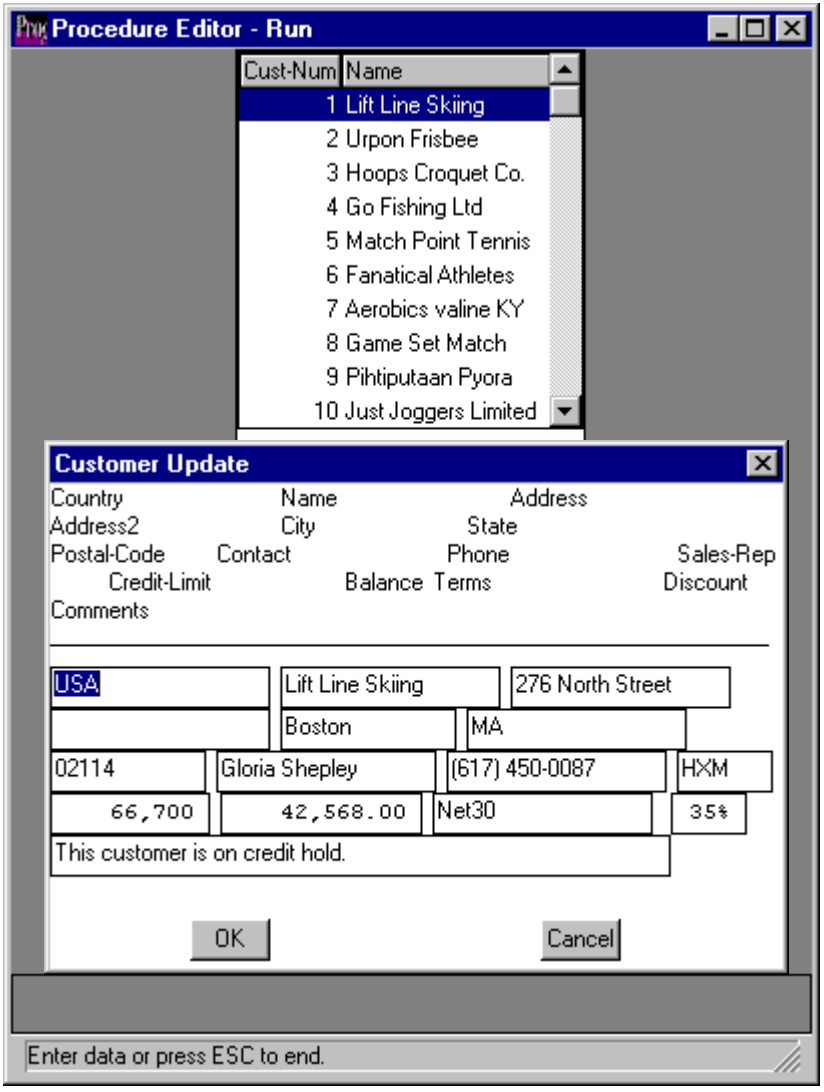
You can choose to display a frame as a dialog box. A dialog box appears in its own window overlaying the current window. The window manager allows the user to move the dialog box around the screen.

To display a frame as a dialog box, specify the VIEW–AS DIALOG–BOX option within the Frame phrase. For example, the following procedure defines a dialog box.

p-diagbx.p

```
DEFINE QUERY custq FOR customer.
DEFINE BROWSE custb QUERY custq DISPLAY cust-num name WITH 10 DOWN.
DEFINE BUTTON update-cust LABEL "Update Customer".
DEFINE BUTTON ok-button LABEL "OK" AUTO-GO SIZE 8 BY 1.
DEFINE BUTTON cancel-button LABEL "Cancel" AUTO-ENDKEY SIZE 8 BY 1.
DEFINE VARIABLE curr-rec AS ROWID.

FORM
  custb SKIP(1)
  update-cust AT 3 cancel-button AT 25
  SKIP(1)
  WITH FRAME main-frame CENTERED.

FORM
  country name address address2 city state postal-code contact
  phone sales-rep credit-limit balance terms discount comments SKIP(1)
  ok-button AT 15 cancel-button AT 50
  WITH FRAME upd-frame TITLE "Customer Update" VIEW-AS DIALOG-BOX.

FRAME upd-frame:HIDDEN = TRUE.
ENABLE ok-button cancel-button WITH FRAME upd-frame.

ON CHOOSE OF update-cust OR MOUSE-SELECT-DBLCLICK OF custb
   DO: /* Transaction */
      curr-rec = ROWID(customer).
      FIND customer WHERE ROWID(customer) = curr-rec EXCLUSIVE-LOCK.
      UPDATE customer EXCEPT cust-num WITH FRAME upd-frame.
   END.

OPEN QUERY custq FOR EACH customer.

ENABLE ALL WITH FRAME main-frame.

WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

When you run `p-diagbx.p`, you get the following output:



25.3.11 Frames for Nonterminal Devices

Progress designs frames independently of the output destination of the frame, which is not determined when the procedure runs, but when it is compiled. However, there are special considerations when you design frames for display on something other than a terminal screen. If you display frames to nonterminal devices, the following rules apply:

- Unless you use the NO–BOX Frame phrase option, Progress omits the bottom line of the box and converts the top line to blanks. In addition, Progress does not print or reserve space for the sides of the box.
- Progress does not output a frame until it executes a data-handling statement that uses another frame or until it executes a PUT statement. It collects all the changes to the frame and sends only one copy of the frame to the output destination.
- Progress ignores the ROW Frame phrase option. See the PUT statement in the *Progress Language Reference* for more information on controlling the format of output to nonterminal devices.

25.3.12 Multiple Active Frames

You can enable input for several frames simultaneously. In a graphical environment, the user can move among the active frames with the mouse—selecting any enabled field in a frame makes that the current frame.

In a character or graphical interface, the user can move to the next or previous frame with the **NEXT–FRAME** and **PREV–FRAME** keys. However, this moves focus between root frames parented by the same window, not between child frames in the same frame family.

You can adjust the spacing between frames in a window by setting the FRAME–SPACING attribute of the SESSION system handle. Its value specifies the number of display units between frames (pixels in graphical interfaces and character cells in character interfaces). By default, the value of FRAME–SPACING is the height of one row in the default system font.

The following procedure uses two frames to update the customer and salesrep records. They are spaced according to the default value for FRAME–SPACING:

p-frm18.p

```
FORM
  customer
  WITH FRAME cust-frame TITLE "Customer".

FORM
  salesrep WITH FRAME rep-frame TITLE "Salesrep".

ON LEAVE, GO OF customer.cust-num
   DO:
      FIND customer USING customer.cust-num.
      FIND salesrep OF customer.
      DISPLAY customer WITH FRAME cust-frame.
      DISPLAY salesrep WITH FRAME rep-frame.

      ENABLE ALL WITH FRAME rep-frame.
      DISABLE customer.cust-num WITH FRAME cust-frame.
   END.
ON GO OF FRAME cust-frame, FRAME rep-frame
   DO:
      IF FOCUS <> customer.cust-num:HANDLE IN FRAME cust-frame
      THEN DO:
         ASSIGN customer.
         ASSIGN salesrep.
         RUN reset-frames.
      END.
   END.
ON END-ERROR OF FRAME cust-frame, FRAME rep-frame
   DO:
      IF FOCUS <> customer.cust-num:HANDLE IN FRAME cust-frame
      THEN DO:
         RUN reset-frames.
         RETURN NO-APPLY.
      END.
   END.

ENABLE ALL WITH FRAME cust-frame.

WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.

PROCEDURE reset-frames:
  CLEAR FRAME cust-frame.
  CLEAR FRAME rep-frame.
  DISABLE ALL WITH FRAME rep-frame.
  ENABLE ALL WITH FRAME cust-frame.
  APPLY "ENTRY" TO customer.cust-num IN FRAME cust-frame.
END PROCEDURE.
```

The procedure initially prompts for a customer number. It then displays the customer record in frame cust–frame and the associated salesrep record in frame rep–frame. Fields in both frames are enabled for input. The cursor is initially placed in the cust–frame frame, but you can move to rep–frame to modify the salesrep information. Note that the ASSIGN statements within the GO trigger are necessary to preserve any changes you make to either record.

25.4 Tab Order

Tab order is the order in which field-level widgets receive TAB events in a frame family. Each frame in a frame family has a separate tab order for the widgets that it owns, and each child frame participates in the tab order of the field-level widgets owned by its parent frame. That is, child frames are tab-order siblings of the field-level widgets owned by the same parent frame. Thus, the tab order of field-level widgets in a frame family depends on how the individual tab orders of the member child frames are organized.

25.4.1 Tab Order for a Frame

The tab order of widgets in a frame is affected by four factors:

1. The order in which you place widgets within the frame definition.

   When Progress compiles a procedure, it lays out all the field-level widgets in static frames used by the procedure, and assigns the default tab order in the order the widgets appear in the frame definition. For a dynamic frame, Progress makes the default tab order the order in which you assign the widgets to the frame. When parenting a child frame or dynamic field-level widget to a static or dynamic frame, the parented widget assumes the last position in the current tab order of the frame, by default.

2. The statements you use to enable the widgets for input in a static frame.

   The PROMPT–FOR, SET, and UPDATE statements explicitly enable widgets and change their tab order. The ENABLE statement also establishes tab order if you explicitly enable widgets by name. However, if you ENABLE ALL widgets for a frame, the tab order is not affected.

3. Whether a static frame is defined with, or its definition is modified by, the KEEP–TAB–ORDER option.

   This option tells Progress that all statements that enable widgets for input, including the ENABLE, PROMPT–FOR, SET, and UPDATE statements will have no effect on the frame's tab order.

4. Whether you use methods or attributes to change the tab order.

   The MOVE–AFTER–TAB–ITEM( ) and MOVE–BEFORE–TAB–ITEM( ) methods allow you to change the tab order of field-level widgets and child frames, overriding the above considerations. At the field–group level, you can also change the tab order of field-level widgets and child frames using the FIRST–TAB–ITEM and LAST–TAB–ITEM attributes.

You can return the current tab order (relative to 1) of any field-level widget or child frame by reading its TAB–POSITION attribute. The NUM–TABS attribute for the field group returns the number of field-level widgets and child frames that have tab positions, and the GET–TAB–ITEM( ) method for the field group returns the widget handle of the widget with the specified tab position. You can also obtain the widget handle of the previous or next widget in the tab order by reading its PREV–TAB–ITEM or NEXT–TAB–ITEM attribute, respectively.

25.4.2 Tab Order for a Frame Family

Once you have established the tab orders for the widgets of each frame in a frame family, the tab order proceeds according to these definitions:

- **Tab-order widget** — A tab-order widget is a widget in a frame's tab order, and can be either a field-level widget or a child frame.

- **Tab item** — A tab item can only be a field-level widget that receives focus. If a child frame occupies a tab item position, the actual first tab item in the child frame is the first field-level widget descending in the tab order of the widgets owned by the child frame; the actual last tab item in the child frame is the last field-level widget descending in the tab order of the widgets owned by the child frame.

- **Next tab item** — If the current tab item is the last field-level widget in the frame family tab order, the next tab item is the first field-level widget in the frame family tab order. If the current tab item is the last tab-order widget of a child frame, the next tab item is the next widget in the tab order of the parent frame. If the next widget in the tab order of the parent frame is a field-level widget, the next tab item is that field-level widget. If the next widget in the tab order of the parent frame is a child frame, the next tab item is the first descendant tab item of the child frame. If the current tab item is not the last tab-order widget of a frame, the next tab item is determined from the next tab-order widget of the frame.

- **Previous tab item** — If the current tab item is the first field-level widget in the frame family tab order, the previous tab item is the last field-level widget in the frame family tab order. If the current tab item is the first tab-order widget of a child frame, the previous tab item is the previous widget in the tab order of the parent frame. If the previous widget in the tab order of the parent frame is a field-level widget, the previous tab item is that field-level widget. If the previous widget in the tab order of the parent frame is a child frame, the previous tab item is the last descendant tab item of the child frame. If the current tab item is not the first tab-order widget of a frame, the previous tab item is determined from the previous tab-order widget of the frame.

Note that you cannot tab **between** frame families, only **within** a single frame family.

Figure 25–2 shows a frame family with four frames. Thus, the tab-order widgets in frame FRAME 1 include:

- Buttons RED, GREEN, and BLUE
- Fill-in Field1
- Frame FRAME 2
- Frame FRAME 3

The tab-order widgets in frame FRAME 2 include:

- Buttons RED, GREEN, and BLUE
- Fill-in Field2
- Frame Frame 4

The tab-order widgets in frame Frame 4 include only fill-in Field4, and in frame FRAME 3 include buttons RED, GREEN, and BLUE, and fill-in Field3.

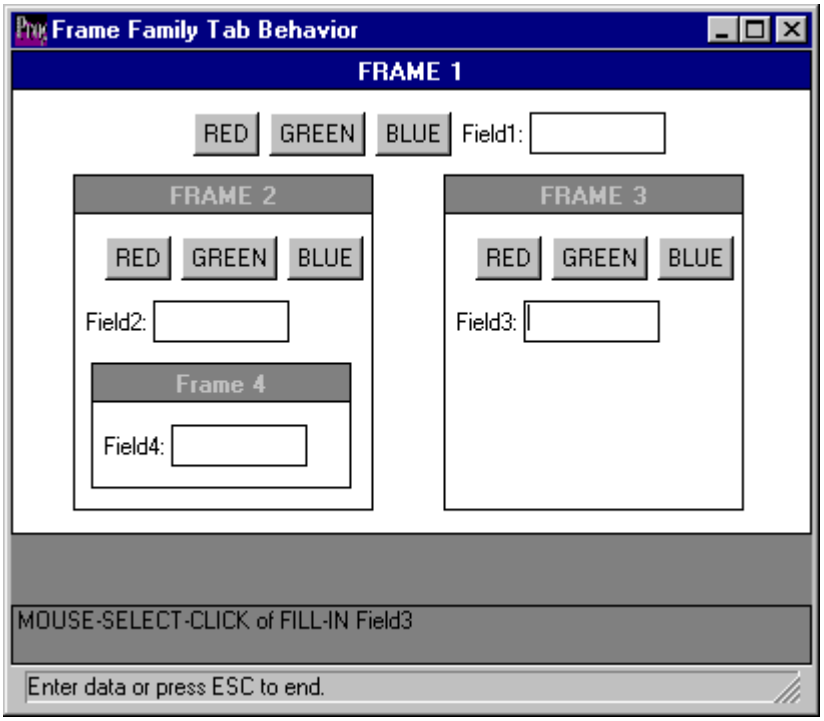


Figure 25–2: Frame Family Tab Order

However, the actual tab items include the respective RED, GREEN, and BLUE buttons and the Field1, Field2, Field3, and Field4 fill-ins.

The following procedure (`p-foftab.p`) displays the frames in Figure 25–2. Note that because FRAME f4 is assigned to the frame family before FRAME f3, f4 comes before f3 in the default tab order:

p-foftab.p *(1 of 2)*

```
DEFINE BUTTON bred1 LABEL "RED".
DEFINE BUTTON bred2 LABEL "RED".
DEFINE BUTTON bred3 LABEL "RED".
DEFINE BUTTON bgreen1 LABEL "GREEN".
DEFINE BUTTON bgreen2 LABEL "GREEN".
DEFINE BUTTON bgreen3 LABEL "GREEN".
DEFINE BUTTON bblue1 LABEL "BLUE".
DEFINE BUTTON bblue2 LABEL "BLUE".
DEFINE BUTTON bblue3 LABEL "BLUE".
DEFINE VARIABLE field1 AS CHARACTER LABEL "Field1".
DEFINE VARIABLE field2 AS CHARACTER LABEL "Field2".
DEFINE VARIABLE field3 AS CHARACTER LABEL "Field3".
DEFINE VARIABLE field4 AS CHARACTER LABEL "Field4".
DEFINE FRAME f1 SKIP(.5)
    SPACE(18) bred1 bgreen1 bblue1 field1
WITH SIDE-LABELS TITLE "FRAME 1" KEEP-TAB-ORDER
     SIZE 80 BY 11.5.
DEFINE FRAME f2 SKIP(.5)
    SPACE(3) bred2 bgreen2 bblue2 SKIP(.5) SPACE(1) field2
WITH SIDE-LABELS TITLE "FRAME 2" KEEP-TAB-ORDER
     SIZE 30 BY 8 AT COLUMN 7 ROW 3.
DEFINE FRAME f3 SKIP(.5)
    SPACE(3) bred3 bgreen3 bblue3 SKIP(.5) SPACE(1) field3
WITH SIDE-LABELS TITLE "FRAME 3" KEEP-TAB-ORDER
     SIZE 30 BY 8 AT COLUMN 44 ROW 3.
DEFINE FRAME f4 SKIP(.5)
    SPACE(1) field4
WITH SIDE-LABELS TITLE "Frame 4" KEEP-TAB-ORDER
     SIZE 26 BY 3 AT COLUMN 2.5 ROW 4.5.
```

p-foftab.p *(2 of 2)*

```
FRAME f2:FRAME = FRAME f1:HANDLE.
FRAME f4:FRAME = FRAME f2:HANDLE.
FRAME f3:FRAME = FRAME f1:HANDLE.
CURRENT-WINDOW:TITLE = "Frame Family Tab Behavior".

ON MOUSE-SELECT-CLICK, GO, TAB, RETURN ANYWHERE DO:
    MESSAGE LAST-EVENT:FUNCTION "of" SELF:TYPE
        IF SELF:TYPE = "FRAME" OR SELF:TYPE = "WINDOW"
        THEN SELF:TITLE ELSE SELF:LABEL.
END.

ENABLE ALL WITH FRAME f1.
ENABLE ALL WITH FRAME f2.
ENABLE ALL WITH FRAME f4.
ENABLE ALL WITH FRAME f3.
SESSION:DATA-ENTRY-RETURN = TRUE.
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

25.4.3 Tabbing and DATA–ENTRY–RETURN

When you set the DATA–ENTRY–RETURN attribute of the SESSION handle to TRUE, focus changes to the next tab item for any fill-in that receives a RETURN event. However, if the fill-in is the last tab item in the frame family, instead of tabbing back to the first tab item, focus remains in the fill-in and a GO event is applied to **all** frames of the frame family. Thus, in `p-foftab.p`, pressing the **RETURN** key in field3 causes a GO event to be applied to each of frames f1, f2, f3, and f4. The GO event is also applied in order from the innermost child frame to the root. As indicated in Figure 25–2, the event fires in order of frame f4, f2, f3, and then f1.

25.5 Active Window Display

When you display a message, alert box, or dialog box, you generally want the message or widget displayed in or near the window where the user is working. In multi-window applications, especially those managed by persistent procedures, it is possible for the user to be working in one window and have a persistent procedure display a dialog box in another. This can occur because the current window of the persistent procedure that displays the dialog box might not be the window where the user is working. The procedure controlling the user's window might be calling an internal procedure whose parent persistent procedure has a different current window.

To ensure that your messages, alert boxes, and dialog boxes get displayed in the window the user is working in, you can specify the ACTIVE–WINDOW system handle for the IN WINDOW option of the appropriate I/O statement. The ACTIVE–WINDOW handle always references the session window that has most recently received input focus. For an example using the ACTIVE–WINDOW handle and more information on managing multi-window applications, see Chapter 21, "Windows."

25.6 Three-dimensional Effects (Windows only; Graphical interfaces only)

On Windows, in graphical interfaces, you can specify that all widgets in a particular frame or dialog box have a three-dimensional look and feel. (In character interfaces, this specification is ignored.) By default, buttons, sliders, and browses already display with a three-dimensional appearance. By specifying the THREE–D Frame phrase option or by setting the THREE–D frame or dialog box option to TRUE, most other widgets in the specified frame or dialog box display with a three-dimensional appearance, including the toggle box, radio set, editor, selection list, and combo box widgets. This also means that the frame background color becomes color Button Face rather than color Window. Note that you can only set the THREE–D attribute before the frame or dialog box is realized.

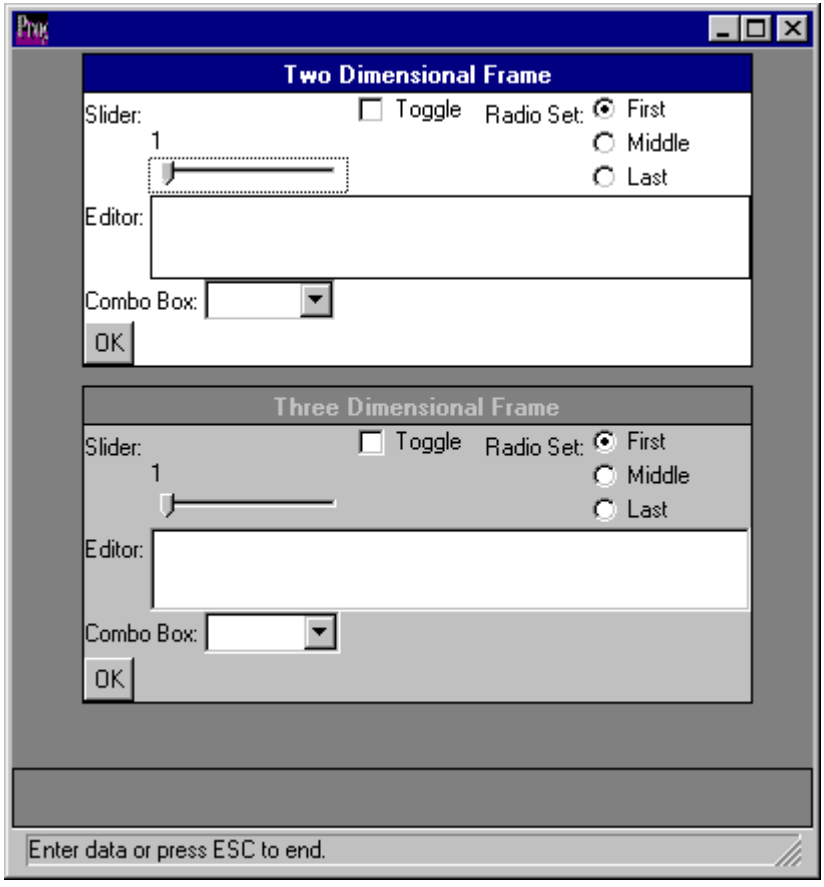
NOTE: The THREE–D look and feel is the preferred style on Windows 95 and Windows NT 4.0 platforms. The THREE–D style provides an overall consistent look to the user interface, thereby accommodating such widgets as the combo box, radio set, toggle box, and LARGE editor, which cannot be displayed in two dimensions on these platforms.

The following procedure, p-threed.p, displays two frames that are identical, except that one is two-dimensional and the other is three-dimensional.

p-threed.p

```
DEFINE BUTTON bOK LABEL "OK".
DEFINE VARIABLE vSlide AS INTEGER LABEL "Slider"
    VIEW-AS SLIDER MAX-VALUE 10 MIN-VALUE 1.
DEFINE VARIABLE vToggle AS LOGICAL LABEL "Toggle"
    VIEW-AS TOGGLE-BOX.
DEFINE VARIABLE vRadio AS INTEGER LABEL "Radio Set"
    VIEW-AS RADIO-SET RADIO-BUTTONS "First", 1, "Middle", 2, "Last", 3.
DEFINE VARIABLE vEdit AS CHARACTER LABEL "Editor"
    VIEW-AS EDITOR SIZE 60 by 2.
DEFINE VARIABLE vCombo AS CHARACTER LABEL "Combo Box"
    VIEW-AS COMBO-BOX LIST-ITEMS "Red", "White", "Blue", "Purple".
DEFINE FRAME D2-Frame
    vSlide vToggle vRadio SKIP vEdit SKIP vCombo SKIP bOK
WITH AT COLUMN 8 ROW 1.1 SIDE-LABELS
     TITLE "Two Dimensional Frame".
DEFINE FRAME D3-Frame
    vSlide vToggle vRadio SKIP vEdit SKIP vCombo SKIP bOK
WITH AT COLUMN 8 ROW 9 THREE-D SIDE-LABELS
     TITLE "Three Dimensional Frame".

CURRENT-WINDOW:TITLE = "".
ENABLE ALL WITH FRAME D2-Frame.
ENABLE ALL WITH FRAME D3-Frame.

WAIT-FOR CHOOSE OF bOK IN FRAME D2-Frame OR
         CHOOSE OF bOK IN FRAME D3-Frame.
```

When you run `p-threed.p`, the following window appears:



25.6.1 THREE–D, Rectangles, and Images

Rectangles and images do not automatically acquire three-dimensional characteristics when you specify THREE–D for a frame. Rectangles must have border widths of two or more pixels to display in a three-dimensional style. Images are completely unaffected by THREE–D. This avoids any indication that you can interact with them. If you want, you can achieve three-dimensional effects for an image by displaying it on an appropriate rectangle or by including the effects within the image itself.

25.6.2 THREE–D and Window Widgets

You can also set the THREE–D attribute of the window to change the window background to color Button Face, which matches three-dimensional frames. However, this is the only effect of changing the THREE–D attribute for windows. Frame and dialog boxes do not inherit this attribute value from windows. You must set this attribute before the window is realized.

25.6.3 2D and 3D Widget Size Compatibility

Three-dimensional widgets are different widgets and have different sizes than corresponding two-dimensional widgets. This means that you must adjust your layouts to accommodate them.

Note that the height of a three-dimensional fill-in is larger than that of a character unit. However, the height of a two-dimensional fill-in is, by default, the same as that of a three-dimensional fill-in to provide layout compatibility between them.

If you want two-dimensional fill-ins to equal the height of a character unit, you must set the Use–3D–Size key to No in the current environment, which might be the registry or an initialization file. However, where two standard fill-in fields fit neatly on ROW 1 COLUMN 1 and ROW 2 COLUMN 1, two three-dimensional fill-in fields do not. For more information on the Use–3D–Size key, see the section on specifying three-dimensional size in the chapter on user interface environments in the *Progress Client Deployment Guide*. For more information on setting keys in the current environment, see the reference entry for the PUT–KEY–VALUE statement in the *Progress Language Reference*.

25.7 Windows Interface Design Options

When developing Progress applications in the Windows environment, you can incorporate various elements that Progress supports to update the look and feel of your user interface. These elements include:

- ToolTip information
- Windows system help
- Small icon size

25.7.1 Incorporating Tooltip Details

ToolTips allow you to define a help text message for a text field or text variable. A ToolTip is a runtime attribute that can be defined for the following widgets: browse, button, combo box, editor, fill-in, image, radio-set, rectangle, selection-list, slider, text, and toggle box. However, they are typically used with a button widget. Progress automatically displays this text when the user pauses the mouse pointer over a widget for which a ToolTip value is defined.

You can define ToolTip values using the TOOLTIP option available for the DEFINE family of statements, the VIEW-AS phrase, or setting the TOOLTIP attribute of a widget. You can add or change the TOOLTIP value at any time. If a ToolTip value is set to either “” or ? (the unknown value), then the ToolTip is removed. The default is no ToolTip.

For more information on setting ToolTip values, see the specific DEFINE statement, VIEW-AS phrase, or widget reference entry in the *Progress Language Reference*.

25.7.2 Accessing the Windows Help Engine

On Windows, Progress supports the SYSTEM–HELP statement, which calls the Windows Help engine, `winhlp32.exe`. The Windows Help engine displays a Help viewer, which displays Help topics and lets the user navigate them. The SYSTEM–HELP statement lets you, the developer, integrate Windows help into your Progress application.

For more information on the SYSTEM-HELP statement, see the SYSTEM–HELP Statement reference entry in the *Progress Language Reference*. For a complete discussion of how to provide on-line help for Progress applications, see the *Progress Help Development Guide*.

25.7.3 Displaying the Small Icon Size

The LOAD-ICON() and the LOAD-SMALL-ICON() methods allow you to associate icons with windows. The LOAD–ICON() method allows you to specify an icon to display in the title bar of a window (maximized), in the task bar (minimized), and when selecting a program using **ALT–TAB** . The LOAD–SMALL–ICON() method allows you to specify an icon to display in the title bar of a window and in the task bar only.

The value you assign with either the LOAD–ICON() or the LOAD–SMALL–ICON() methods must be the name of an icon (`.ico`) file. Both of these methods accommodate icons formatted as small size (16x16), regular size (32x32), or both. However, their treatment of the small and regular size icons differ.

The LOAD–ICON() method looks for an icon defined as 32x32 pixels by default and will shrink it down to 16x16 pixels if necessary. The resolution of an icon that is changed in this manner may not be suitable for all icon images. Therefore, the alternative LOAD–SMALL–ICON() method that looks for icons defined as 16x16 pixels by default when there are multiple icons in an icon file may be more suitable for your icon style and presentation.

A

R-code Features and Functions

R-code is the intermediate binary code that Progress generates when it compiles 4GL source files. This is the code that Progress actually runs when it executes a procedure, whether from session compiles or from permanently generated r-code (.r) files.

Progress provides a dynamic r-code execution environment, as well as integrity and security mechanisms to ensure that your r-code is running in a compatible environment.

This appendix provides information about the following topics:

- R-code structure
- R-code libraries
- R-code libraries and PROPATH
- R-code execution
- R-code portability
- Code page compatibility
- Database CRCs and time stamps
- R-code CRCs and procedure integrity

A.1 R-code Structure

R-code is divided into multiple segments of varying length. Each r-code file contains an object header and segment location table followed by the actual r-code segments. The object header is a fixed-length descriptor that identifies the file as an r-code file and contains information about the version and size of the r-code file. The segment location table is a variable-length descriptor that contains the size and location of each r-code segment in the file. The maximum size for most segment types is 62K.

Table A–1 describes and lists the types, maximum size, and maximum number of segments in an r-code file.

Table A–1: R-code Segments *(1 of 2)*

Segment Type	Max. Size	Max. Number	Description
Action code	62K	1 per procedure	Holds the actual executable code in the form of action cells. Action cells drive the Progress interpreter and contain the executable code for 4GL verbs. There is a separate action code segment for the main procedure and each internal procedure.
Expression code	62K	4	Holds the executable expressions in Reverse Polish Notation (RPN) for the main procedure, all internal procedures, and all trigger blocks. An r-code file can have up to 248K of executable expressions.
Text	62K	1 per language	Holds all literal character strings for the r-code file. There is one text segment for the default language and one for each language specified in the COMPILE statement. Duplicate literals are removed. Only one text segment is loaded into memory at run time.
Initial value	62K	1	Contains information required to initialize an r-code file for execution, including database and table references, variable definitions, TEMP–TABLE and WORK–TABLE definitions, a list of defined frames, time stamps, and CRCs. There is one initial value segment per r-code file.

Table A–1: R-code Segments *(2 of 2)*

Segment Type	Max. Size	Max. Number	Description
Frame	32K	1 per frame	Contains layout information for a frame, including frame fields, attributes, and all RPN expression code from the frame phrase. There is one frame segment for each named and unnamed frame in the r-code file. Each frame segment has a 32K limit.
Debugger	62K	1	Used by the Application Debugger to maintain the debugger context for the procedure. There is one debugger segment per r-code file. This segment is loaded into memory on demand only when the Debugger is active. For information about the Debugger, see the *Progress Debugger Guide*.

A.1.1 Factors that Affect R-code Size

You can affect r-code size by how you write your 4GL, depending on the r-code segment. The action code and initial value segments are among the most tunable.

Action Code Segment

You can reduce the size of the action code segment by consolidating multiple 4GL statements into one. This also can increase the speed of execution, because the interpreter executes only one action instead of several.

For example, you can reduce action code size by combining several consecutive assignment statements into one ASSIGN statement.

```
a = 1.
b = 2.
c = 3.
```

```
ASSIGN a = 1
       b = 2
       c = 3.
```

NOTE: Both of these examples require the same amount of expression code.

Initial Value Segment

You can reduce the size of the initial value segment by limiting the number of SHARED variables that are accessed in a procedure. The r-code required to support a SHARED variable is larger than the r-code to support a NEW SHARED variable. Progress uses approximately 36 additional bytes in the initial value segment to resolve each SHARED variable at run time. This value can change depending on the environment.

A.1.2 R-code File Segment Layout

Figure A–1 shows the segment layout in an r-code file. A compiled procedure requires one initial value segment, one action code segment, one expression code segment, one text segment, and one debugger segment. There can be more action and expression code segments, up to the limit, as required by your procedure. There can be multiple frame segments, one segment for each frame in your procedure. The maximum number of frame segments is virtually unlimited.

There can also be multiple text segments, one segment for the default language and each language that you choose for translation. The default language segment contains the literal character strings defined in the original 4GL source file. You can create additional text segments by using the Translation Manager. Although multiple text segments are possible, only one text segment is available per language. Thus, the literal character strings for a given language cannot exceed 62K. For more information about natural language translation for 4GL procedures, see the *Progress Translation Manager Guide*.

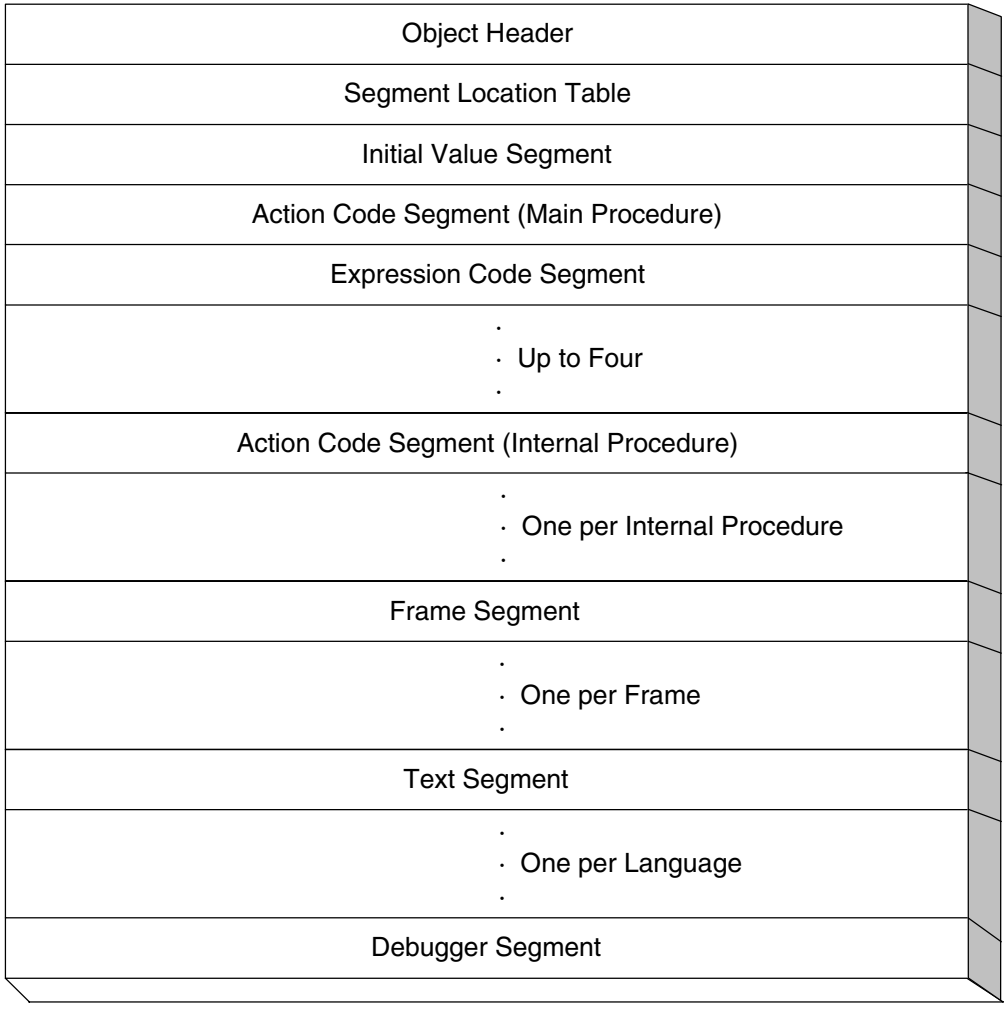


Figure A–1: R-code File Segment Layout

A.2 R-code Libraries

You can organize and store r-code files in a Progress r-code library. A Progress r-code library is a collection of r-code procedures combined in a single file. R-code libraries allow you to manage and execute r-code procedures more efficiently. You create r-code libraries by using the PROLIB utility.

Progress provides two types of r-code libraries: standard and memory-mapped. A standard library contains r-code procedures that execute in local memory. A memory-mapped library contains r-code procedures that execute in shared memory.

For information about using the PROLIB utility to create r-code libraries, see the *Progress Client Deployment Guide*.

A.3 R-code Libraries and PROPATH

The following rules govern how standard and memory-mapped libraries interact with PROPATH during a Progress session:

- The first time you run a member procedure from a standard or memory-mapped library that is specified in the PROPATH, Progress locates the procedure by searching through each directory and library in the PROPATH until it finds the procedure. To search a library for a member procedure, Progress must open the library. When Progress opens a standard library, it loads the procedure into local memory. When Progress opens a memory-mapped library, it maps the library in shared memory.

- When searching through directories and libraries in the PROPATH, Progress starts at the beginning of the PROPATH and searches each directory and library, in defined order, until it finds the procedure. Thus, placing the library at the beginning of the PROPATH improves performance.

- A library remains open until the end of your Progress session or until you remove the library from the PROPATH. If you remove a library from the PROPATH while a member procedure is still active (either running, or waiting for a subprocedure to return), Progress displays an error message and the library remains open until you end your Progress session. Otherwise, Progress closes a standard library and unmaps a memory-mapped library.

- If you use a SEARCH or RUN statement to open a library that is not in the PROPATH, the library remains open until you end your Progress session.

- If you use libraries, Progress accesses r-code files as if you had specified the Quick Request (–q) startup parameter. That is, Progress searches the PROPATH only on the first reference to a procedure. Thereafter, if the procedure still resides in memory, in the local session compile file, or in an r-code library, Progress reuses the procedure instead of searching the PROPATH again. For more information about the Quick Request (–q) startup parameter, see the *Progress Startup Command and Parameter Reference*.

A.4 R-code Execution

Progress executes r-code procedures in different ways, depending on whether you store the r-code files in an r-code library and the type of library.

When executing an r-code procedure from an operating system file in a directory, or from a standard r-code library, Progress accesses and executes the r-code file segments in an execution buffer in local memory. For more information about the standard r-code execution environment, see the "Standard R-code Execution Environment" section later in this appendix.

When executing an r-code procedure from a memory-mapped r-code library, Progress accesses and executes the r-code file segments in shared memory. For more information about the memory-mapped r-code execution environment, see the "Memory-mapped R-code Execution Environment" section later in this appendix.

For information about monitoring execution environment activity during a Progress client session, see the "R-code Execution Environment Statistics" section later in this appendix.

For information about monitoring and optimizing r-code performance, see the *Progress Client Deployment Guide*.

A.4.1 Standard R-code Execution Environment

At run-time, Progress manages the execution of a standard r-code procedure from either an operating system file or a standard procedure library using the following components:

- **Execution buffer** — The portion of local memory that Progress allocates and uses to store the chain of loaded r-code segments for all procedures executing in local memory.
- **Session sort file (.srt)** — A file that Progress uses to dynamically swap r-code segments in and out of the execution buffer.
- **Segment descriptor table** — An in-memory table that references the r-code segments required by all executing procedures, including the location of each r-code segment in the execution buffer and usage count information.
- **R-code directory** — An in-memory table that contains information about each executing r-code procedure, including r-code size, usage count, segment descriptions, and a reference to the segment descriptor table for each segment.

Progress uses the segment descriptor table and the r-code directory to manage r-code procedures from operating system files, standard libraries, and memory-mapped libraries.

Figure A–2 shows the layout for the standard r-code execution environment.

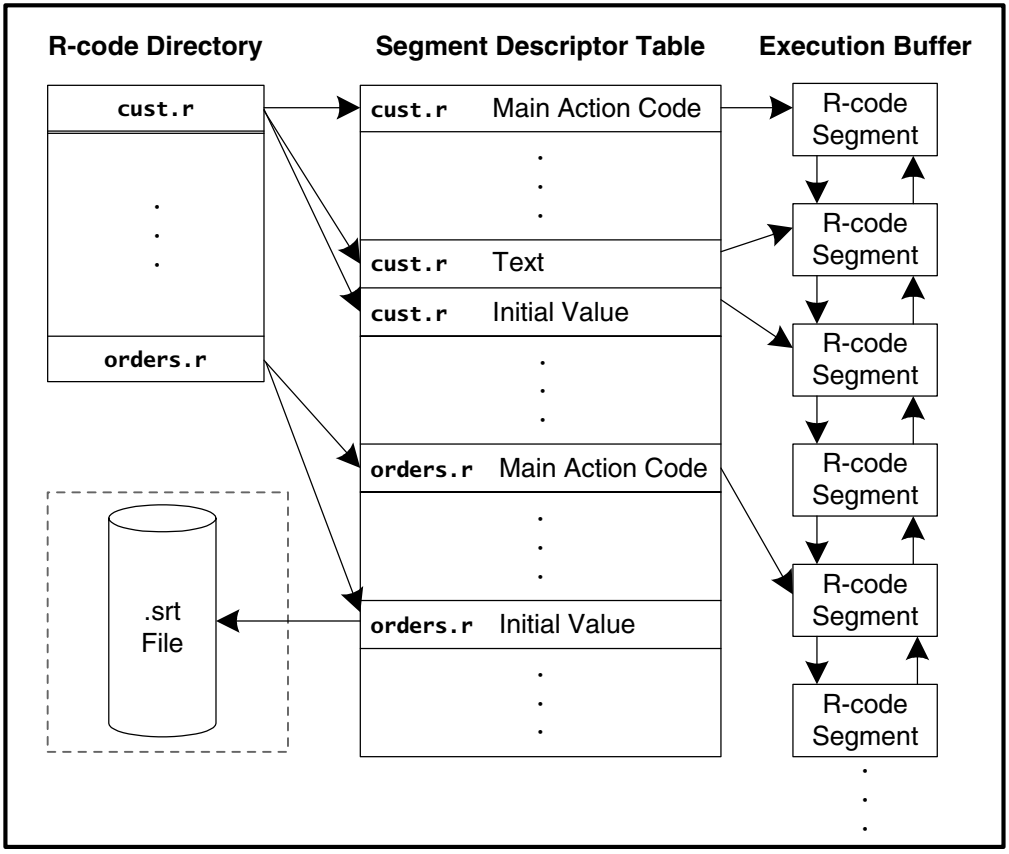


Figure A–2: Standard R-code Execution Environment

In Figure A–2, Progress located and loaded the `cust.r` and `orders.r` files into local memory from either operating system files or standard libraries. The execution buffer is shown with three `cust.r` segments and two `orders.r` segments. Note that one `orders.r` segment is located in the execution buffer, while the other segment is swapped to the session sort file. When space in the execution buffer is needed for new r-code segments, Progress uses the session sort file to swap out the least-recently used segments. When Progress needs a segment that has been swapped to the session sort file, it reloads the segment from the session sort file into the execution buffer.

Standard Execution Sequence

When you run a standard r-code procedure for the first time, from either an operating system file or a standard library, Progress loads and executes the procedure as follows:

1. Opens the procedure file (.p or .r) or procedure library (.pl), if the library is not already open.

2. Reads the r-code procedure into memory and creates an r-code directory entry for the procedure. Progress first compiles a procedure into r-code, if necessary.

3. Registers each required r-code segment in the execution environment as follows:

   a. Loads the r-code segment at the head of the segment chain in the execution buffer

   b. Adds an r-code segment entry to the segment descriptor table that references the segment in the execution buffer

   c. Inserts a segment descriptor reference in the r-code directory entry for the procedure

   If all the required r-code segments do not fit in the execution buffer, Progress attempts to free space by swapping r-code segments already in the buffer to the session sort file. If Progress cannot free enough space by swapping segments, it increases the execution buffer ceiling and allocates more space for the execution buffer.

   NOTE: When accessing r-code procedures stored in a standard library, Progress does not swap r-code segments to the session sort file unless you specify the PROLIB Swap (–pls) startup parameter. By default, if Progress needs an r-code segment in a standard library, it reloads the segment into the execution buffer from the library in local memory.

4. Once the required r-code segments in the procedure are registered in the execution environment, the interpreter begins executing the r-code procedure at the start of the main action code segment and accesses the remaining segments directly from local memory as required.

Standard Execution Environment Limits

The number of standard r-code procedures that you can run at one time and the memory used are determined by the following factors:

- **R-code directory size** — The default is 100 entries, the minimum is 5 entries, and the maximum is 500. You can set the initial number of entries using the Directory Size (–D) startup parameter. Progress dynamically increases the directory size up to the maximum, as required. Use the Hardlimit (–hardlimit) startup parameter to force Progress to adhere to the limit specified by the Directory Size (–D) startup parameter. .

- **Execution buffer ceiling** — The default is 3096K. You can set the initial ceiling for this buffer up to 65,534K using the Maximum Memory (–mmax) startup parameter. Progress dynamically increases the execution buffer size up to the maximum, as required. Use the Hardlimit (–hardlimit) startup parameter to force Progress to adhere to the limit specified by the Maximum Memory (–mmax) startup parameter.

- **Available memory** — Available memory is a factor only if it is smaller than the execution buffer ceiling or Progress needs to allocate memory beyond that ceiling.

Standard R-code Segment Management

While Progress loads an r-code file, all of its segments are locked in memory. After all required segments are loaded for the procedure, Progress unlocks all segments except its main action code and text segment. These two segments stay locked in memory until execution of the r-code file terminates. Internal procedure action code segments stay locked only until they return to the invoking procedure and are relocked each time they execute.

When a standard r-code segment does not fit in the execution buffer, Progress attempts to free space by swapping r-code segments already in the buffer to the session sort file. Progress can swap out any unlocked segments. Progress removes these segments from the tail end of the execution buffer chain, in least recently used (LRU) order.

If Progress cannot free enough memory for a newly loaded segment by swapping out older segments, it dynamically increases the maximum execution buffer size (execution buffer ceiling) and allocates the required memory up to the memory available.

When Progress needs a segment that has been swapped to the sort file, it reloads the segment from the sort file into the execution buffer. However, Progress keeps the reloaded segments in the sort file.

NOTE: When accessing r-code procedures stored in a standard library, Progress does not swap r-code segments to the session sort file unless you specify the PROLIB Swap (–pls) startup parameter. By default, if Progress needs an r-code segment in a

standard library, it reloads the segment into the execution buffer from the library in local memory.

A.4.2 Memory-mapped R-code Execution Environment

At run time, Progress manages the execution of memory-mapped r-code procedures using the following components:

- **Shared memory buffer** — The portion of shared memory that the operating system allocates and uses to store r-code segments for procedures in a memory-mapped procedure library.
- **Segment descriptor table** — An in-memory table that references the r-code segments required by all executing procedures, including the location of each r-code segment in memory and usage count information.
- **R-code directory** — An in-memory table that contains information about each executing r-code procedure, including r-code size, usage count, segment descriptions, and a reference to the segment descriptor table for each segment.

Progress uses the segment descriptor table and the r-code directory to manage r-code procedures from operating system files, standard libraries, and memory-mapped libraries.

Figure A–3 shows the layout for the memory-mapped r-code execution environment.

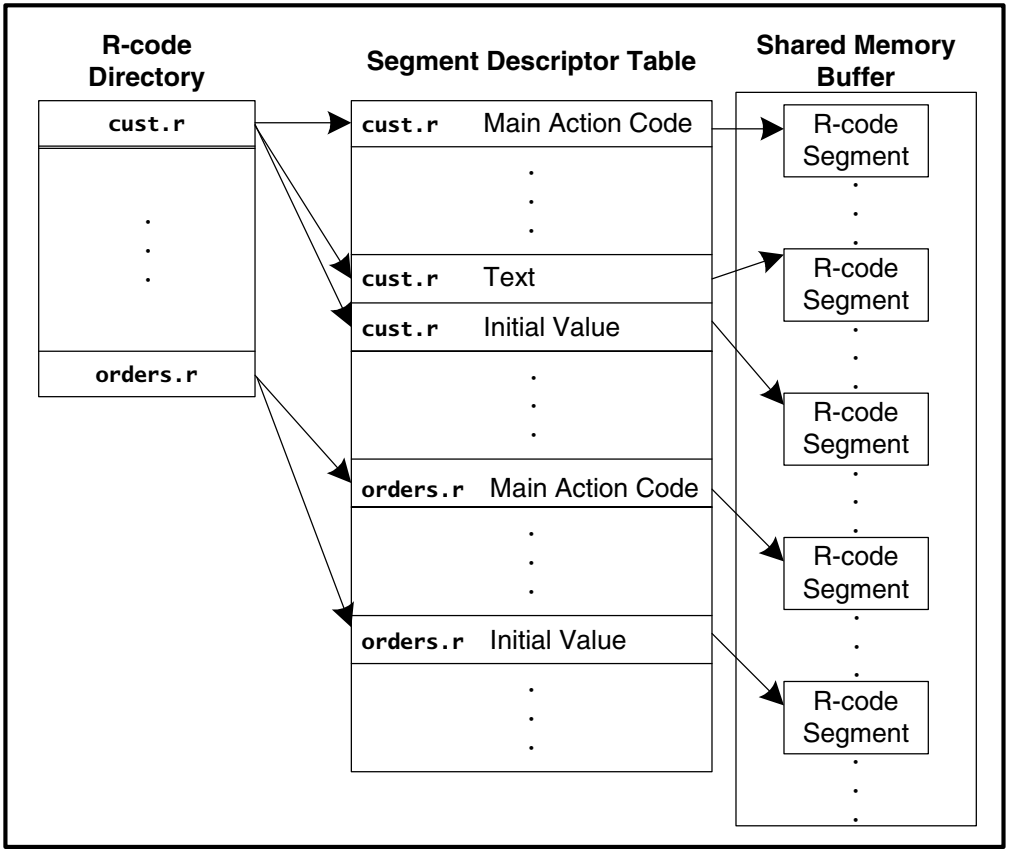


Figure A–3: Memory-mapped R-code Execution Environment

In Figure A–3, Progress located the `cust.r` and `orders.r` files in a memory-mapped library. Note that all r-code segments in a memory-mapped library are located in shared memory. When Progress needs a segment, it executes the segment directly from shared memory.

Memory-Mapped Execution Sequence

When you run a memory-mapped r-code procedure for the first time, Progress loads and executes the procedure as follows:

1. Opens the procedure library, if not already open, and memory-maps the library in shared memory.

2. Reads the r-code procedure and creates an r-code directory entry for the procedure.

3. Registers each required r-code segment in the execution environment as follows:

   a. Adds an r-code segment entry to the segment descriptor table that references the segment in shared memory

   b. Inserts a segment descriptor reference in the r-code directory entry for the procedure

4. Once the required r-code segments in the procedure are registered in the execution environment, the interpreter begins executing the r-code procedure at the start of the main action code segment and accesses the remaining segments directly from shared memory as required.

Memory-Mapped Execution Environment Limits

The number of memory-mapped r-code procedures that you can run at one time and the shared memory used are determined by the following factors:

- **R-code directory size** — The default is 100 entries, the minimum is 5 entries, and the maximum is 500. You can set the initial number of entries using the Directory Size (–D) startup parameter. Progress dynamically increases the directory size up to the maximum, as required. Use the Hardlimit (–hardlimit) startup parameter to force Progress to adhere to the limit specified by the Directory Size (–D) startup parameter.

- **Available memory** — The number of memory-mapped r-code libraries you can have open is limited by the amount of shared memory available on the system.

Memory-Mapped R-code Segment Management

When Progress needs a memory-mapped r-code segment, it executes the segment directly from shared memory. Progress does not store active memory-mapped r-code segments in the execution buffer. Nor does it swap non-active segments to the session sort file. Progress relies on the operating system to manage the swapping of r-code segments in and out of shared memory.

A.4.3 R-code Directory Management

If the r-code directory is full when loading a new standard or memory-mapped r-code file, Progress can reuse existing directory entries for any r-code files no longer in use.

To reuse an r-code directory entry, Progress performs the following steps:

1. Identifies the least recently used (LRU) r-code file.

2. Frees the r-code directory entry for the LRU r-code file.

3. Frees all of the segment descriptor entries in the segment descriptor table for the LRU r-code file.

4. When reusing a standard r-code directory entry, Progress removes all r-code segments for the LRU r-code file from both the execution buffer and the session sort file. When reusing a memory-mapped r-code directory entry, the memory-mapped r-code file remains mapped in shared memory.

When Progress needs to reload a standard r-code file, it follows the standard execution sequence. When Progress needs to reload a memory-mapped r-code file, it follows the memory-mapped execution sequence.

A.4.4 R-code Execution Environment Statistics

You can monitor execution environment activity using the Statistics (–y) and Segment Statistics (–yd) startup parameters. These parameters cause Progress to write memory statistics to the `client.mon` file. The –yd parameter provides all of the information about memory usage available with –y plus additional information about r-code segments loaded during a client session. For more information about the –yd and –y startup parameters, see the *Progress Startup Command and Parameter Reference*.

For information about monitoring and optimizing r-code performance, see the *Progress Client Deployment Guide*.

Figure A–4 through Figure A–8 show sections of `client.mon` output generated with the –yd startup parameter.

In Figure A–4, the "Progress client startup options" section shows the r-code directory size (–D), in this case, set to the default of 100 entries. The –y startup parameter also generates this information.

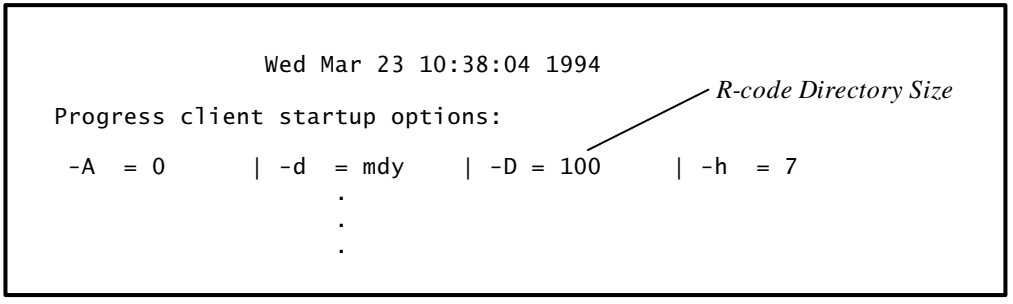


Figure A–4: Progress Client Startup Options for –yd

In Figure A–5, the "Execution buffer map" section shows the order and size of procedures loaded into the execution buffer.

```
Execution buffer map:                                  11:27:28
       Size                  Name
       ----                  ----
       3813                  _edit.r
     182871                  adeedit/_proedit.r
       1321      .           adecomm/_setcurs.r
                 .
                 .
```

R-code files listed in order of execution with sizes in bytes

Figure A–5: Execution Buffer Map for –yd

In Figure A–6, the "Per procedure temp file access statistics" section lists each r-code segment that has been read or written to the session sort file (.srt). Each segment is listed under its r-code file by segment type ("Int–Proc Action"), number (5), and size (1364 bytes).

```
Per procedure temp file access statistics:
        Segment        Read/Write    Times    Bytes
        -------        ----------    -----    -----
_edit.r
        E-code seg:  1     Read        1      508
        E-code seg:  1     Write       1      508
adeedit/_proedit.r
        E-code seg:  1     Read        1    44476
        E-code seg:  1     Write       1    44476
        Int-Proc Action:  5   Read          1    1364
        Int-Proc Action:  5   Write         1    1364
                      .
                      .
                      .
```

R-code files listed in order of execution showing segments read and written to the session sort file

Figure A–6: Accessing the Session Sort File for –yd

In this example, all segments shown have been read and written once. A large number of segments read or written a large number of times indicates a likely need for more memory.

In Figure A–7, the "Per procedure segment information" section lists all the r-code segments loaded during the session. Each segment is listed under its r-code file by segment type ("Int–Proc"), number (2), and size (816 bytes).

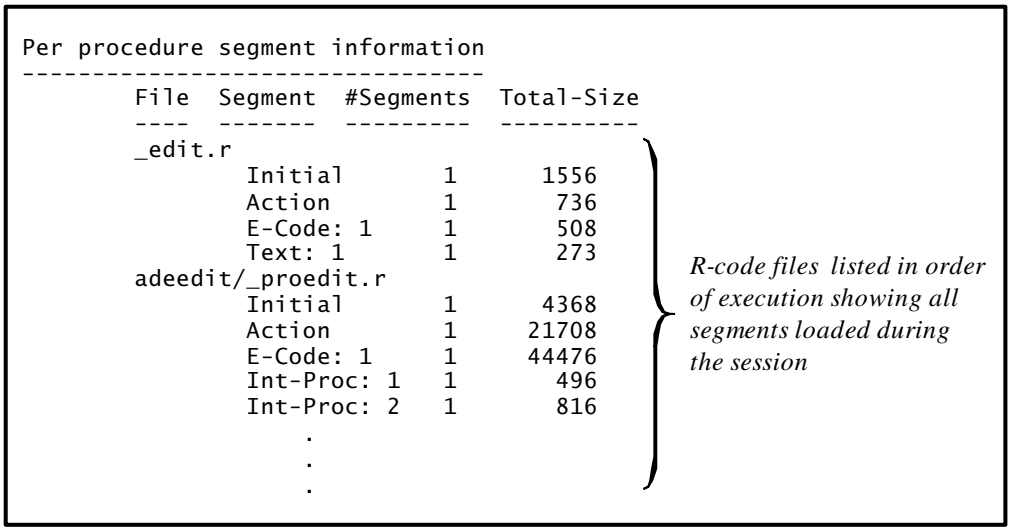


Figure A–7: Procedure Segment Information for –yd (Part 1)

Thus, if there are five internal procedure action segments in `adeedit/_prodedit.r`, they are listed in order (1 through 5). The number of segments for each entry is always 1. Note that the "Initial" and main "Action" segments have no segment number, because there is never more than one of these segment types per r-code file.

In Figure A–8, the listed sections provide memory and segment usage summaries. The –y startup parameter also generates this information.

```
----------------------------------------
Program access statistics:    Times    Bytes
                    .
                    .
                    .

Memory usage summary:       Current    Max Used       Limit (Bytes)
Stack usage (-s):                60        7248       40960
Local buffer usage:             864       14704
R-code Execution Buffer:     530025      710609      710656

Segment Descriptors Usage: (numbers)
Max Used:      508      Limit:      720

-----------------------------------------------
```

Figure A–8: Procedure Segment Information for –yd (Part 2)

The "R-code Execution Buffer" statistics show how far your application pushes the execution buffer ceiling. The "Segment Descriptors Usage" statistics shows how close your application is to running out of segment descriptors, and thus, whether you need to optimize the number of r-code segments in your application.

A.5 R-code Portability

R-code is portable between any two environments if the following are true:

- The user interface is the same — graphics mode or character mode.

  NOTE: If a procedure contains no user interface code, its r-code can run across user interfaces without change. This is typical, for example, of database schema trigger procedures.

- The DataServer or combination of DataServers connected during compilation is the same.

- The machine classes are compatible.

All machines that require four-byte alignment or less are compatible. Thus, r-code compiled on one of these machines (for example, a Sun SPARCstation) can run on most other machines. One of the few machines not compatible with this class is the AS/400, which requires greater than four-byte alignment.

Thus, you can use the same r-code compiled for Windows on any DataServer-compatible Windows system, and you can use the same r-code compiled in character mode on any DataServer-compatible character-mode system.

A.6 Code Page Compatibility

The Internal Code Page (-cpinternal) startup parameter determines the internal code page that Progress uses in memory. The code page in which r-code is compiled and the internal code page of the session in which the r-code runs must be compatible. That is, the r-code code page must be either the internal code page, undefined, or convertible to the internal code page.

If the code page for a standard r-code procedure, from either an operating system file or a standard library, is not the same as the session's internal code page, Progress performs an in-memory conversion of the r-code text segments to the session's internal code page.

NOTE: If you use the R-code In Code Page (–cprcodein) startup parameter to specify a code page for reading r-code text segments, Progress reads the text segments as if they were written in that code page, even if the text segments were written in a different code page.

Since r-code procedures in memory-mapped procedure libraries are read-only, Progress cannot perform an in-memory code page conversion of r-code text segments. If the code page of a memory-mapped r-code procedure is not compatible, Progress will not execute the r-code. It will report an error and stop execution.

For more information about the Internal Code Page (–cpinternal) and R-code In Code Page (–cprcodein) startup parameters, see the *Progress Startup Command and Parameter Reference*.

A.7 Database CRCs and Time Stamps

At run time, Progress must ensure that r-code files and the database(s) that they access are compatible. Otherwise, if the schema has changed since the code was compiled, running the code might damage the database. Progress supports two techniques for checking this compatibility—time stamp validation and cyclic redundancy check (CRC) validation. Each technique prevents you from running unauthorized or data-incompatible procedures against a particular database environment.

A.7.1 Time Stamp Validation

For each table, Progress maintains a time stamp indicating when the schema was last changed. When compiling a procedure, Progress inserts the time stamp into the r-code for each database table accessed by the procedure. This time stamp is the value of the _Last–change field in the metaschema _File record for each table. Note that sequence changes have no effect on time stamp validation.

R-code Execution with Time Stamps

When executing a procedure using time stamp validation, Progress checks to make sure that the time stamps for all accessed tables match those contained in the r-code. If a time stamp does not match, the procedure does not execute.

To use time stamp validation, compile the procedure with each database connected using the Time Stamp (–tstamp) connection parameter. Otherwise, Progress uses CRC validation.

NOTE: The Time Stamp (–tstamp) parameter is obsolete, and is supported only for backward compatibility.

A.7.2 CRC Validation

For each table, Progress computes a CRC value (*database* CRC) from selected elements of the table metaschema. When compiling a procedure, Progress inserts the database CRC for each database table the procedure accesses. The database CRC for each table is stored in the metaschema _File record _CRC field for the table. A matching CRC ensures that the schema referenced by the r-code is compatible with the table schema, regardless of its time stamp.

Time Stamps and CRCs

Time stamps and CRCs both help maintain database integrity, but they do it differently. Time stamps change whenever a table schema is updated or recreated at another time or place; CRCs change only when certain schema elements critical to field or record definitions change. As long as a table is structurally compatible with the r-code that references it, the CRCs match and the r-code can execute. It does not matter when or where the table's schema was created or updated.

R-code Execution with CRCs

When executing a procedure using CRC validation, Progress follows these steps:

1. For each table accessed by the r-code, check the time stamp. If it matches, execute the procedure. Otherwise, go to Step 2.

2. If the table is in a database that was not connected using the Time Stamp (–tstamp) parameter when the procedure was compiled, check the CRC. If it matches, execute the procedure. Otherwise, reject the procedure.

Thus, if the time stamps match, Progress assumes that the database schema and r-code must be compatible. Otherwise, the CRCs determine the outcome.

NOTE: The Time Stamp (–tstamp) parameter is obsolete, and is supported only for backward compatibility.

A.7.3 CRC Calculation

Table A–2 lists the metaschema fields that are involved in each CRC calculation.

Table A–2: Metaschema Fields in CRC Calculation

Metaschema Table			
_File	**_Field**	**_Index**	**_Index–Field**
_File–Name _DB–lang	_Field–Name _Data–Type _dtype _sys–field _field–rpos _Decimals _Order _Extent _Fld–stdtype _Fld–stlen _Fld–stoff _Fld–case	_Index–Name _Unique _num–comp	(_Field–Name)[1] _Ascending _Abbreviate _Unsorted

[1] The _Field–Name for index fields is referenced in the _Field table using _Field–recid in _Index–Field.

CRC Calculation for Tables

All of the listed metaschema fields, participate in the CRC calculation for each database table. The resulting CRC is stored in the _CRC field of the _File record for each table. A change to any of these fields automatically changes the value of _CRC.

A.7.4 CRC Versus Time Stamp Validation

The main advantage of using CRC validation is that you can run the same r-code against different databases as long as the databases have sufficiently similar schemas. This allows you to deploy code and database updates while avoiding either of the following effects:

- Forcing the user to dump and reload their data into a new database that you supply with new r-code
- Forcing the user to compile new source code (possibly encrypted) that you supply with new data definitions to update their database schema

Depending on the schema changes, the user still might have to dump and reload some data, but you can always supply r-code for their updated database.

Database Changes that Affect Time Stamps

You must recompile procedures that reference a database table in which you have made any of the following changes:

- Deleted the table
- Changed the table name
- Changed the database language or authorization key
- Added or deleted a field in the table
- Changed the name or order of a field
- Added or deleted an index
- Modified an index by changing its name, uniqueness, field components, collation order, or the abbreviation option

Although you do not have to recompile procedures when you make other schema changes, such changes do not apply to procedures that you do not recompile. For example, changes to the following features appear only when you recompile:

- Field appearance — format, label, column label, and decimal
- Data entry — initial value, mandatory value, case sensitivity, validation expression, delete validation, and security
- User information — help messages, validation messages, and description

Database Changes that Affect CRCs

You must recompile procedures using CRC-based deployment for the same schema changes that change a time stamp. In addition, you must recompile procedures that reference a database table in which you have made these changes to a field:

- The data type or extent (array size)
- The number of decimals
- The case sensitivity

Table A–2 lists all the metaschema fields involved in database CRC calculation.

NOTE: The order in which indexes are defined has no effect on database CRCs.

Time Stamp-based Deployment

Using time stamp validation, if you deploy r-code, you must compile the procedures against a schema-updated version of the user's database and send both the r-code and the new database to the user. Otherwise, the r-code cannot run, and you must deploy source code for the user to compile along with new database definitions to update their schema.

CRC-based Deployment

Using CRC validation, if you deploy r-code, you only have to compile the procedures against a database with the same basic structure as the end-user database. It does not matter when or where the database was created. If you make schema changes to an existing database, you can distribute the newly compiled r-code to end users, along with a data definition files to install the schema changes.

The flexibility of CRC-based deployment makes it easier to distribute a new version of an existing application. To deploy an application update based on CRC validation, follow these steps:

1 ♦ Install a copy of the production database from the target site in your development environment.

2 ♦ Create an incremental data definition file that captures the difference between your development database and the production copy. For information about creating an incremental data definition file, see the *Progress Database Administration Guide and Reference*.

3 ♦ Apply the incremental data definition file to your copy of the production database.

4 ♦ Recompile all procedures affected by the schema changes against your copy of the production database.

5 ♦ Apply the incremental data definition file to the original production database (on site). The CRC for the original production database should now be identical to the production copy, but probably not identical to your development database.

6 ♦ Distribute the new r-code files (from Step 4) to the updated production site.

The steps for CRC-based deployment are essentially the same as those for time stamp-based deployment, but you gain these advantages:

- You are not required to encrypt and give end users source code.
- End users do not have to compile your source code.
- You do not have to send a copy of a new database to end users.
- End users do not have to dump and reload all their data into your new database.

Differences in R-code Access Authorization

One possible drawback of using CRC validation is that anyone having a Progress development environment can create a counterfeit database (a database with the same structure as your database). The user can then write a program, compile it, and run it against your database. Using time stamp validation prevents this because the time stamps in the counterfeit code do not match those in your database.

However, you can also combine CRC validation with additional security to prevent unauthorized r-code access. The DBAUTHKEY option of the PROUTIL utility allows you to set an authorization key for your database. When you compile your source code, Progress includes the value of this authorization key in your r-code. You can also use the RCODEKEY option of PROUTIL to insert the authorization key in existing r-code. Any r-code that does not include the correct authorization key cannot run against your database. For more information about using PROUTIL to create and set database authorization keys, see the *Progress Database Administration Guide and Reference*.

CRC and Time Stamp Availability

During compilation, Progress always inserts both database time stamps and CRCs in the r-code. If a database is not connected with the Time Stamp (–tstamp) parameter during procedure compilation, Progress turns on a bit in the r-code. This bit tells the run-time interpreter to check the CRC if the time stamp check fails.

NOTE: The Time Stamp (–tstamp) parameter is obsolete, and is supported only for backward compatibility.

A.8 R-code CRCs and Procedure Integrity

When Progress compiles a procedure, it calculates a special CRC based on the procedure name and code content, and stores it in the r-code. This *r-code CRC* (as distinguished from database CRC) is useful to ensure the integrity and security of Progress procedures, especially for schema triggers.

If you choose to have CRCs checked for a schema trigger, the r-code CRC of the trigger procedure definition is stored in the trigger schema. The trigger procedure cannot run unless it has a matching r-code CRC. This prevents a trigger procedure with the same name, but different 4GL code, from being improperly substituted for the original trigger procedure. Such a substitution can cause damage to your database or override your security.

For other application r-code files, you can use the RCODE–INFO handle to build a procedure security table that contains the name and r-code CRC of each r-code file in your application (except the startup procedure). Before running a procedure, your application can use the RCODE–INFO handle to validate the r-code CRC of its r-code file against the entry established for it in the procedure security table.

NOTE: If there is no r-code for the procedure, any source (.p) procedure returns an unknown (?) CRC value.

A.8.1 Assigning CRCs to Schema Triggers

When you create a schema trigger in the Data Dictionary, you can specify CRC checking. Then, when you save the trigger procedure, the Data Dictionary calculates its r-code CRC and stores it in the trigger metaschema record (_File–Trig for table triggers and _Field–Trig for field triggers).

If you want CRC checking for all your schema triggers, you must specify it for each schema trigger in your database. For more information, see the *Progress Basic Development Tools* manual (Character only) and, in graphical interfaces, the on-line help for the Data Dictionary.

A.8.2 Validating CRCs for Schema Triggers

When a database event occurs in your application (such as creating a record or assigning a field), Progress checks whether the CRC of the corresponding trigger procedure matches the CRC in the trigger metaschema record. If it matches, Progress executes the procedure. Otherwise, Progress returns a run-time error.

Trigger CRC validation works for both r-code and source versions of a trigger procedure. For source versions, Progress calculates the CRC during the session compile.

A.8.3 RCODE–INFO Handle

The RCODE–INFO handle allows you to read the r-code CRC from an r-code file. You can use this value to build a procedure security table, and then to verify the integrity of your application r-code against it. You can also use this value in a deployment procedure that you run in a secure context to automatically install database schema trigger definitions.

Getting the R-code CRC

To read the r-code CRC from an r-code file, first set the FILE–NAME attribute of the RCODE–INFO handle to the pathname of your r-code file. Then read the value of the CRC–VALUE attribute.

```
DEFINE VARIABLE rCRC AS INTEGER.
RCODE-INFO:FILE-NAME = "sports/crcust.r".
rCRC = RCODE-INFO:CRC-VALUE.
```

NOTE: The RCODE–INFO handle cannot read the r-code CRC from a session compile. For source (.p) procedures, the CRC–VALUE attribute returns the unknown value (?).

Example — Verifying Application R-code Integrity

This example shows how you might provide r-code integrity and security in your application.

1 ♦ Build the procedure security table, RcodeSecurity, with these fields:

- Filename — CHARACTER field for the pathname of each r-code file.
- CRC — INTEGER field for the r-code CRC of the specified r-code file.

2 ♦ Construct a text file, `crctable.dat`, that contains a list of the relative pathnames for all the secure procedures called by your application.

3 ♦ Compile and save the r-code for all the secure procedures in your application.

4 ♦ Run a procedure that contains code like this to build the RcodeSecurity table.

```
DEFINE VARIABLE i AS INTEGER.
DEFINE VARIABLE proc-name AS CHARACTER FORMAT "x(32)".
DEFINE VARIABLE rCRC AS INTEGER.

INPUT FROM "crctable.dat". /* List of r-code file pathnames */

i = 0.
REPEAT:
    SET proc-name.
    FIND RcodeSecurity WHERE Filename = proc-name NO-ERROR.
    IF NOT AVAILABLE(RcodeSecurity) THEN
        CREATE RcodeSecurity.
    RCODE-INFO:FILE-NAME = proc-name.
    SET RcodeSecurity.Filename = proc-name
        RcodeSecurity.Crc = RCODE-INFO:CRC-VALUE.
    i = i + 1.
END.

INPUT CLOSE.
MESSAGE i "procedure security records created".
```

5 ♦ At each point where you call a secure procedure in your application, insert this code.

```
FIND RcodeSecurity WHERE Filename = "secret.r".
RCODE-INFO:FILE-NAME = "secret.r".

IF RcodeSecurity.Crc = RCODE-INFO:CRC-VALUE THEN
    RUN secret.
ELSE DO:
    MESSAGE "Procedure secret.r is invalid.".
    QUIT.
END.
```

Example — Deploying Schema Triggers

The following procedure installs new schema trigger definitions in an existing database. It reads dump files that contain the new data for table and field metaschema trigger records and updates these records with the new trigger procedure names and CRCs.

p-trload.p *(1 of 2)*

```
/* p-trload.p */
DEFINE VARIABLE i AS INTEGER.
DEFINE VARIABLE event AS CHARACTER FORMAT "x(6)".
DEFINE VARIABLE proc-name AS CHARACTER FORMAT "x(32)".
DEFINE VARIABLE table-name AS CHARACTER FORMAT "x(32)".
DEFINE VARIABLE field-name AS CHARACTER FORMAT "x(32)".
DEFINE VARIABLE rCRC AS INTEGER.
INPUT FROM "_file-tr.dat". /* Table trigger data */
i = 0.
_fl-loop:
REPEAT:
  SET table-name event proc-name.
  RCODE-INFO:FILE-NAME = proc-name.
  rCRC = RCODE-INFO:CRC-VALUE.
  FIND _file WHERE _file-name = table-name.
  FIND _file-trig WHERE _file-recid = RECID(_file) AND
                        _event = event NO-ERROR.
  IF AVAILABLE _file-trig THEN DO:
    IF  _File-trig._Proc-name = proc-name
    AND _File-trig._Trig-CRC  = rCRC THEN
        NEXT _fl-loop.
    ELSE DO:
    /* Progress doesn't let you modify a trigger record, so delete and
       recreate. */
      DELETE _File-trig.
      CREATE _File-trig.
      ASSIGN _File-trig._File-recid = RECID(_File)
             _File-trig._Event      = event
             _File-trig._Override = TRUE
             _File-trig._Proc-Name  = proc-Name
             _File-trig._Trig-CRC   = rCRC
             i = i + 1.
    END.
  END.
END.
```

p-trload.p *(2 of 2)*

```
INPUT CLOSE.
MESSAGE i "_file-trig records updated.".
INPUT FROM "_field-t.dat". /* Field trigger data */
i = 0.
_fld-loop:
REPEAT:
  SET table-name field-name event proc-name.
  RCODE-INFO:FILE-NAME = proc-name.
  rCRC = RCODE-INFO:CRC-VALUE.
  FIND _file WHERE _file-name = table-name.
  FIND _field WHERE _file-recid = RECID(_file) AND
                    _field-name = field-name.
  FIND _field-trig WHERE _field-trig._file-recid = RECID(_file) AND
                         _field-trig._field-recid = RECID(_field) AND
                         _event = event NO-ERROR.
  IF AVAILABLE _Field-trig
      AND _Field-trig._Proc-name = proc-name
      AND _Field-trig._Trig-CRC  = rCRC THEN NEXT _fld-loop.
  ELSE DO:
    DELETE _Field-trig.
CREATE _Field-trig.
    ASSIGN
      _Field-trig._File-recid  = RECID(_File)
      _Field-trig._Field-recid = RECID(_Field)
      _Field-trig._Event       = event
      _Field-trig._Override    = TRUE
      _Field-trig._Proc-Name   = proc-Name
      _Field-trig._Trig-CRC    = rCRC
      i = i + 1.
  END.
END.
INPUT CLOSE.
MESSAGE i "_field-trig records updated.".
```

NOTE: As this example shows, you must use the RECID function rather than the ROWID function to reference metschema table record locations.

Index

Symbols

! (exclamation point)
  format character 17–52
  user ID lists 14–9

& (ampersand), menu mnemonics 22–6

( ) (parentheses) 17–53

- (minus sign) 17–54

* (asterisk) 17–54
  user ID lists 14–9

+ (plus sign) 17–54

, (comma) 17–54

. (period) 17–54

< (less-than sign) 17–55

<> (angle brackets) 18–2

> (greater-than sign) 17–54

^ (caret), I/O statements 7–27

~ (tilde), input files 7–42

Numbers

9, format character 17–52, 17–54

A

A, format character 17–52

ACCELERATOR option, menu items 22–6

Accelerators
  character interface 22–20
  menu items 22–18
  portable 22–20

Action code segment, r-code A–2, A–3

ACTIVE-WINDOW handle 25–49
  example 21–30

ActiveX 16–5
  Automation objects 23–7
  controls 23–7

ActiveX components, handles, purpose 16–16

ActiveX controls
  assigning colors 23–7
  assigning fonts 23–7

ADD-FIRST method 17–35, 17–45

ADD-LAST method 17–35, 17–38, 17–44, 17–45, 17–48

Adding records 9–19

Advancing frames 3–8

Alert boxes
  creating 25–7
  defined 16–4
  displaying in active window 25–49

Aliases
  compiling procedures 9–14
  creating in applications 9–13
  DICTDB 14–5
  logical database name 9–12
  shared record buffers 9–16

All-or-nothing processing 12–3

Allocating frames 3–7

ALLOW REPLICATION phrase 11–10, 11–15

Ampersand (&), menu mnemonic 22–6

Angle brackets (<>) 18–2

ANYWHERE option 6–35, 6–41, 16–40

Application deployment
  CRC validation A–21
  run-time execution 1–26
  time stamp validation A–21

Application environment management 23–23

Application security 14–1
  activities-based checking 14–10
  CAN-DO function 14–14, 14–15
  checking user IDs at run time 14–7
  creating, permissions table 14–10
  using the CAN-DO function 14–9

Application-driven programming. *See* Procedure-driven programming

Applications
  compile-time execution 1–19, 1–26
  multi-user 13–2
  run-time execution 1–19

APPLY statement 10–38, 16–42
  compared to RUN statement 16–43
  procedure triggers 3–29

Applying events 16–42
  procedure triggers 3–29

AppServers, handles, purpose 16–16

Arguments, include files 8–18

Arrays, efficient processing 12–42

ASC function, code page conversion 7–28

ASSIGN statement 9–17

ASSIGN trigger 11–3, 11–8

Asterisk (*) 17–54
  user ID lists 14–9

AT phrase 25–16

Attributes
  AUTO-RESIZE 16–30, 17–4
  AUTO-RETURN 17–5
  AUTO-ZAP 17–5
  BLANK 17–5
  CHECKED 22–15
  COLUMN 20–4
  combo box widget 17–45
  CURRENT-WINDOW 21–1
    persistent procedures 3–40
  DATA-ENTRY-RETURN 17–5,

18–9
DATA-TYPE 17–45
DBNAME 17–5
DELIMITER 17–35, 17–45, 17–48
direct manipulation 24–2
DRAG-ENABLED 17–35
editor widget 17–27
FILE-NAME, persistent procedures 3–40
fill-in widget 17–5, 17–8
FIRST-CHILD 22–26
FIRST-PROCEDURE 3–39
FIRST-TAB-ITEM 25–44
FORMAT 17–5, 17–14, 17–45, 20–4
FRAME 19–22, 20–4
FRAME-COL 25–20
FRAME-NAME 17–5
FRAME-ROW 25–20
FRAME-SPACING 25–41
GET-SIGNATURE, persistent procedures 3–40
GRAPHIC-EDGE 18–16
GRID-FACTOR-HORIZONTAL 24–17
GRID-FACTOR-VERTICAL 24–17
GRID-SNAP 24–18
GRID-UNIT-HEIGHT-CHARS 24–18
GRID-UNIT-HEIGHT-PIXELS 24–18
GRID-UNIT-WIDTH-CHARS 24–18
GRID-UNIT-WIDTH-PIXELS 24–18
GRID-VISIBLE 24–17
HANDLE 16–6
HEIGHT 17–4
HEIGHT-CHARS 17–11
HEIGHT-PIXELS 17–11
INDEX 17–5
INNER-CHARS 17–35
INNER-LINES 17–35, 17–45
INTERNAL-ENTRIES, persistent procedures 3–40
KEEP-FRAME-Z-ORDER 19–32
LARGE 17–26
LAST-CHILD 22–26
LAST-PROCEDURE 3–39
LAST-TAB-ITEM 25–44
LIST-ITEMS 17–35, 17–36, 17–38, 17–44, 17–45, 17–46, 17–48
MAX-CHARS 17–24
MAX-VALUE 17–10
MIN-VALUE 17–10
MULTIPLE 17–35
NEXT-SIBLING 22–26
persistent procedures 3–39
NEXT-TAB-ITEM 25–44
NUM-ITEMS 17–35, 17–45
NUM-TABS 25–44
PARENT 22–26
PERSISTENT, persistent procedures 3–40
PREV-SIBLING, persistent procedures 3–39
PREV-TAB-ITEM 25–44
PRIVATE-DATA 16–35
persistent procedures 3–40
procedures 3–39
ROW 20–4
SCREEN-VALUE 17–35, 17–45, 20–4, 20–5
selection list widget 17–35
SENSITIVE 17–6
SIDE-LABEL-HANDLE 20–4
slider widget 17–10
SORT 17–35, 17–45
SUBTYPE 17–3
TAB-POSITION 25–44
TABLE 17–5
THREE-D 25–49
TYPE 3–29, 16–19
USE-TEXT 17–8
WIDTH 17–4
WIDTH-CHARS 17–11, 20–4
WIDTH-PIXELS 17–11
X 20–4
Y 20–4

Attributes, widget
example 16–23
expressions 16–22
handles 16–20

reading 16–21
referencing 16–21
setting 16–22

Audit trails. *See* Transactions

Authentication
defined 14–3
examples 14–6, 14–7
procedure in Progress 14–5

Auto-connect, connecting databases 9–5

AUTO-RESIZE attribute 16–30, 17–4

AUTO-RETURN attribute 17–5

AUTO-ZAP attribute 17–5

Automation, ActiveX Automation objects 23–7

AVAILABLE function 13–19

B

Batch processing 7–25

{&BATCH-MODE} name reference 8–14

Before-image files 12–41

BGCOLOR, widget assignment 23–4

BLANK attribute 17–5

Blank user IDs 14–5

Bleeding record locks 13–15

Block properties 3–2
See also Frames, designing

Blocks 3–1
4GL 1–14
context 1–15
default frames 25–11
default processing 3–3
defined 1–14
DO 3–2
DO ON ENDKEY 12–18
DO ON ERROR 12–18
DO TRANSACTION 12–18
dynamic resources 1–17
EDITING 3–3
editing 6–36
error processing 3–18
FOR 3–2
FOR EACH 3–6
frame scopes 3–8
LEAVE processing 5–4
looping 3–5
NEXT processing 5–4
Procedure blocks 3–2
properties 3–2, 3–3
reading records 3–6
record scoping 3–10
REPEAT 3–2
retrying 3–8, 5–4
RETURN processing 5–4
scope 1–16
scoping properties 3–5
services for frames 3–8, 19–26
TRANSACTION option 12–8, 12–18
transactions 3–17, 12–8
trigger 3–3
undoing 5–3
unscoped resources 1–16

BORDER-TOP attribute 25–6

BORDER-TOP-CHARS attribute 25–6

Borrowing frames 3–9

Box selecting 24–7, 24–11, 24–15

BREAK keyword. *See* Control-break reports

Browse
moving 10–41
resizable 10–40
resizing 10–40

row height, changing 10–43

BROWSE statement 9–17

Browse widgets 10–5 to 10–43, 13–9
calculated fields 10–12
character interfaces 10–46
colors 10–39
character interfaces 10–52
column events 10–23
creating rows 10–27
data set size 10–4
defining 10–5
deleting rows 10–30
described 10–2
events 10–20
field lists 10–4
fonts 10–23
character interfaces 10–52
justified labels 10–38
locking columns 10–31
overlaying widgets 10–34
programming techniques 10–20
queries 10–3, 10–11
read-only 10–5
refreshing rows 10–24
related tables 10–37
repositioning focus 10–24
row events 10–23
searching columns 10–15
sizing 10–13
stacked labels 10–38
style options 10–38
table joins 10–37
ToolTips 10–39
updating editable rows 10–25

Browse-column 20–27
moving 10–42
resizing 10–41

Browsing records 13–9

-Bt startup parameter, temporary table buffer size 15–15

BTN-DOWN-ARROW image 18–4

BTN-LEFT-ARROW image 18–4

BTN-RIGHT-ARROW image 18–4

BTN-UP-ARROW image 18–4

Buffer object 20–15, 20–18

BUFFER-COMPARE statement 11–18

Buffer-field object 20–15, 20–20

Buffers
free record scope 3–10
free references 3–10
internal procedures 3–33
locks and multiple 13–15
record 3–10, 9–16
screen 9–16
strong record scope 3–10
weak record scope 3–10

Button widget 18–2
calculating the default size 18–6
choosing 18–2
DEFAULT option 18–9
defaults in dialog boxes 18–9
images 18–3
NO-CONVERT-3D COLORS option 18–8
NO-FOCUS option 18–4
predefined images 18–4
states 18–3
TOOLTIP attribute 18–2

Buttons, mouse 6–27, 24–5

C

Calculated fields, browse widgets 10–12

Call stack, procedure 3–23

CAN-DO function
application security 14–14
using to check user IDs 14–9

Cascading replication. *See* Data replication

CENTERED option, DISPLAY statement 25–8

Character browse 10–46
  color management 10–52
  control keys
    edit mode 10–50
    row mode 10–48
  differences from Windows 10–52
  modes of operation 10–47
  tabbing 10–53

CHARACTER data type
  combo boxes 17–39
  editor widget 17–24
  radio set 17–19
  selection lists 17–30

Character display formats 17–51

Character interface accelerators 22–20

Character interfaces 2–12, 25–2
  colors 23–8
    reserved values 23–9
    widget states 23–8
  scroll mode 25–33
  simulating windows 25–3
  specifying colors 23–2

Character set management. *See* Code page conversion

Character units 25–4, 25–17
  conversion with pixels 25–5
  rounding error 25–5

CHECKED attribute 22–15

Checking data entry for keys 6–35

Child frames 19–19
  defining 19–22
  field-level widgets 19–21
  restrictions 19–23

Child windows 21–9

CHOOSE event 16–33

CHOOSE statement 17–2, 25–31, 25–35
  See also SCROLL statement

CHR function
  code page conversion 7–28

Clearing frames 3–8

Clipboard output 7–6

Code example, resizable browse 10–45

Code generation. *See* Dynamic code generation

Code pages
  conversion map file 7–29
  conversions 7–28
  example of conversion 7–29

CODEPAGE-CONVERT function 7–28

Collision log. *See* Data collision

Color
  adding for startup 23–3
  availability 23–2
  character interfaces 23–8
    See also Character interfaces
    management 23–8
    reserved values 23–9
    widget states 23–8
  common dialog 23–11
    example 23–12
  custom, Windows color dialog 23–12
  definitions 23–21
    changing 23–21
  dynamic 23–11
  environment files 23–2
    See also Environment files
  environments 23–23
  graphical interfaces
    definition access 23–21
    environment management 23–23
    management 23–10

table access 23–18
user changes 23–11
inheritance 23–7
managing display limitations 23–26
numeric, Windows color dialog 23–12
Progress defaults 23–3
RGB-VALUE function 23–2
sample application 23–4
table access 23–18
the registry 23–2
visual, Windows color dialog 23–12
widget assignment 23–4
example 23–5
Windows 23–9
Palette Manager 23–9, 23–10

Color dialogs 23–12
4GL statements 23–11, 23–13
Windows example 23–12

COLOR-TABLE handle 23–18
environment management 23–24
example 23–6, 23–11
using 23–18

Colors
assigning 23–4
assigning to ActiveX controls 23–7
modifying 23–18

COLUMN attribute 20–4

COLUMN frame phrase option 19–31

Combo box widget 17–39 to 17–48
attributes 17–45
example 17–43, 17–46
item selection 17–39 to 17–46
methods 17–45
options 17–40
value settings 17–42

Comma (,) 17–54

Comments, 4GL 1–13

Commit
two-phase 12–39

Commit unit. *See* Transactions

Common dialogs
4GL statements 23–11, 23–13, 23–16
color 23–12
examples 23–12, 23–15
font 23–15

COMPILE statement, compared to RUN statement 1–26, 1–27

Compiler
listing 3–15
messages 5–25
STREAM-IO option 7–10

Compile-time, security 14–1

Compile-time code 1–19, 1–24
execution 1–26
interaction with run-time 1–23
procedure execution 1–26
run-time expressions 1–26
syntax elements 1–20
uses 1–26

COMPLETE option, SAVE CACHE statement 9–79

Component handles, compared to other handles 16–16

Condition handling
internal procedures 3–33
persistent procedures 3–46

Conditions
ENDKEY 5–13
ERROR 5–6
QUIT 5–19
STOP 5–18
database disconnection 5–24

CONNECT statement 9–3
authentication 14–5
NO-ERROR option 9–3, 9–7

CONNECTED function 9–7, 12–39

Connection failures 9–6
  See also Multi-database connection(s)

Connection overhead 9–9

Connection parameters. *See* Startup parameters

Connection security 14–1

Connections, lost 5–24

CONTAINS operator 9–47

Context
  blocks 1–15
  compared to scope 3–24
  defined 1–14, 3–19

Control frame 16–5

Control tables, sequences 9–63

Control-break reports, using work tables to produce 15–3

Controlling frame allocation 19–9

Controls 16–1
  See also Widgets
  ActiveX controls 23–7

Conversion map files 7–29

CONVERT-3D-COLORS attribute 18–14

CONVERT-3D-COLORS option 18–14

Converting
  code pages 7–28
    See also Code page conversion
  data 7–30
    IMPORT statement 7–40
    QUOTER utility 7–31

CR, minus sign 17–55

CRC. *See* Cyclic redundancy check

CRC calculation
  database
    metaschema fields A–20
    tables A–20
  r-code A–24

CRC compared to time stamps, r-code availability A–24

CRC validation A–19

CREATE ALIAS statement 9–12

CREATE statement 9–17

CREATE trigger 11–2, 11–7

CREATE Widget statement 16–12

CREATE-RESULT-LIST-ENTRY method 10–27

Creating persistent procedures 3–38

Cross-tab reports, using work tables to produce 15–13

CURRENT option, SAVE CACHE statement 9–80

CURRENT-LANGUAGE function 3–46

CURRENT-LANGUAGE variable changes, persistent procedures 3–46

CURRENT-VALUE function 9–69
  syntax 9–68

CURRENT-VALUE statement 9–70
  syntax 9–68

CURRENT-WINDOW attribute 21–1
  persistent procedures 3–40, 21–20

CURRENT-WINDOW handle
21–1, 21–16

Cursor repositioning
field lists 9–36
FIND statement 9–31
after FIND fetches 9–31

Custom colors, Windows color dialog 23–12

Cyclic redundancy check
compared to time stamps A–21
database A–18
advantages A–23
calculation A–20
changes affecting A–22
code validation A–19
compared to time stamps A–21
deployment A–22
disadvantages A–23
R-code
RCODE-INFO handle A–25
schema triggers A–25
r-code A–24
See also R-code CRCs
application procedures A–24
calculation A–24
schema triggers A–25

D

-D startup parameter, Directory Size A–10, A–13

-d startup parameter, Date Format 17–58

Data
formats 17–49 to 17–58
handling statements 9–16
movement 9–16
read-only 17–6
representing 17–1

Data collision. *See* Data replication

Data Dictionary
creating, permissions table 14–10
validation 19–49

Data entry, monitoring 6–3, 6–35

Data replication
4GL support for 11–10
data collision 11–17
collision log 11–17
Data Dictionary support for 11–11
process configuration 11–20
replication changes log 11–12
replication models, cascading replication 11–16
trigger-based replication 11–11

Data types
RAW 9–73
WIDGET-HANDLE 16–6

DATA-ENTRY-RETURN attribute 17–5, 18–9

DATA-TYPE attribute 17–45

Database connection, lost connections 5–24

Database CRCs A–18
database changes affecting A–22

Database disconnection 5–24
persistent procedures 3–46

Database triggers
ASSIGN 11–3, 11–8
CREATE 11–2, 11–7
default error phrase 11–26
DELETE 11–2, 11–7
disabling for dump/load 11–27
error status 11–26
ESQL considerations 11–27
FIND 11–2, 11–24
FIND NEXT statement 11–25
FIND PREV statement 11–25
interaction 11–4
overriding 11–23
procedure library 11–27
reverting 11–23

schema 11–4
session 11–21
WRITE 11–3, 11–5

Databases
connect parameters, limits 9–2
CRC A–18
code validation A–19
disconnecting 12–39
integrity 12–2
multiple connections. *See* Multi-database connections
Number of Databases (-h) startup parameter 9–2
records, sharing 13–3
security 14–1
time stamps A–18
triggers 11–1

DATE data type
combo boxes 17–39
radio set 17–19

Date display formats 17–58

Date Format (-d) startup parameter 17–58

DB, minus sign 17–55

DBNAME attribute 17–5

DBTASKID function 11–10, 11–13

DCOLOR
using 23–8
widget assignment 23–4
widget states 23–8

Deadly embrace 13–21

Debugger segment, r-code A–3

DECIMAL data type
combo boxes 17–39
formats 17–53
radio set 17–19

Default action, selection lists 17–30

Default buttons 18–9

Default frames 3–7
DO blocks 3–7
specifying 19–7

DEFAULT option 18–9

DEFAULT-ACTION event 16–33, 17–30, 17–38
browse widget 10–20

DEFAULT-BUTTON option 18–9

DEFAULT-WINDOW handle 21–1

DEFINE BUTTON statement 18–2

DEFINE FRAME statement 25–13, 25–14

DEFINE IMAGE statement 18–9

DEFINE QUERY statement 9–28
SCROLLING option 9–29

DEFINE RECTANGLE statement 18–16

DEFINE SHARED FRAME statement 19–35

DEFINE SHARED STREAM statement 7–22
See also Streams, shared

DEFINE SHARED VARIABLE statement 19–38

DEFINE STREAM statement 7–19

DEFINE TEMP-TABLE statement 15–16
See also Temporary tables

DEFINE VARIABLE statement
use of FORMAT option 17–49
using VIEW-AS phrase 17–2

DEFINE widget statement 16–12

DEFINE WORK-TABLE statement 15–5, 15–8

See also Work tables

DEFINED function 8–6, 8–10

DELETE ALIAS statement 9–13

DELETE method 17–35, 17–45

DELETE PROCEDURE statement 3–41

DELETE statement 9–17, 9–19

DELETE trigger 11–2, 11–7

DELETE WIDGET statement 16–13
  environment management 23–24

DELETE-SELECTED-ROWS method 10–30

Deleting
  dynamic widgets 16–13
  field lists 9–35
  persistent procedures 3–41
  records 9–19

DELIMITER attribute 17–35, 17–45, 17–48

DELIMITER option
  EXPORT statement 7–38
  IMPORT statement 7–39, 7–40, 7–43

Derealizing widgets 16–29

Descendant frames 19–22

DESELECT event 24–12

Determining frame type 19–12

Dialog boxes
  compared to windows 21–3
  default buttons 18–9
  displaying in active window 25–49
  persistent procedures 21–29
  viewing frames 25–38

DICTDB alias 14–5

-dictexps startup parameter 19–51

Dictionary Expressions (-dictexps) startup parameter 19–51

Direct manipulation 24–1
  event ordering 24–15
  events 24–11
  modes 24–11

Directory Size (-D) startup parameter A–10, A–13

DISABLE TRIGGERS FOR LOAD statement 11–10, 11–15

DISCONNECT statement
  affect on aliases 9–13
  transactions 12–39

Display formats 17–49 to 17–58
  character 17–51
  date 17–58
  defaults 17–51
  logical 17–57
  numeric 17–53

DISPLAY statement 9–17
  CENTERED option 25–8
  OVERLAY option 19–31
  RETAIN option 25–8
  TITLE option 25–8
  use of FORMAT option 17–49
  using VIEW-AS phrase 17–2

DO blocks
  default frame 3–7
  described 3–2
  error property 5–3
  frame scoping 3–8
  infinite loop protection 5–4
  ON ERROR phrase 3–17

DO statement, FOR option 3–10

Down frames 25–25

DR, minus sign 17–55

Drag box 24–8

DRAG-ENABLED attribute 17–35

Dragging, selection lists 17–31, 17–32

Duplicating menus 22–16

Dynamic browse, Browse 20–25

Dynamic code generation
  4GL 1–27
  optimized 1–27

Dynamic frames 19–4

Dynamic menus 22–21
  creating menu bars 22–24
  example 22–27
  pop-up menus 22–29
  properties 22–22
  querying sibling attributes 22–26
  setting up menu bars 22–23

Dynamic resources 1–17

Dynamic scoping, internal procedures and user-defined functions 3–52

Dynamic submenus, defining submenus 22–24

Dynamic temporary tables. *See* Temporary tables

Dynamic widgets
  4GL actions 20–2
  creating 16–12
  defined 16–5
  deleting 16–13
  describing 16–12
  field-level widget setup 20–4, 20–6
    example 20–6
  frame setup 20–7
    example 20–8
  instantiating 16–12
  managing 20–2
    compared to static widgets 20–2
  reasons for using 20–1
  triggers 16–36
  widget pools 20–9

E

EDGE-CHARS option
  rectangle widget 18–15
  with RECTANGLE widget 18–16

EDGE-PIXELS option
  rectangle widget 18–15
  with RECTANGLE widget 18–16

Edit mode, character browse 10–46

EDITING blocks 6–35
  described 3–3

Editing blocks 6–36

Editor widgets 17–24
  attributes 17–27
  example 17–28
  interactions 17–26
  LARGE option 17–26
  MAX-CHARS attribute 17–24
  methods 17–27
  options 17–24
  save file portability 17–30
  scrolling 17–26
  word wrap 17–26

&ELSE directive 8–6

&ELSEIF directive 8–6

EMPTY-SELECTION event 24–12

ENABLE statement 9–17
  ALL option 19–50
  END-ERROR key 5–23
  EXCEPT option 19–50

Encapsulation
  defined 3–32

example 3–39, 3–42
internal procedures 3–32
persistent procedures 3–38

ENCODE function 14–2

END event, browse widget 10–20

End of file 5–16

END-BOX-SELECTION event 24–14

END-ERROR key 6–33
ENABLE/WAIT-FOR 5–23
handling by Progress 5–19

END-MOVE event 24–13

END-RESIZE event 24–13

END-SEARCH event 10–19

&ENDIF directive 8–6

ENDKEY condition 5–2, 5–13
default handling 5–14
END-ERROR key 5–21
handling 5–17

ENDKEY, processing 6–33
See also UNDO statement

ENTRY event 16–33

ENTRY method 17–35, 17–45

Environment files
changing resources 23–21
colors and fonts 23–2
hints for managing 23–25
managing 23–23
managing multiple 23–24
managing statements 23–23
scenario for managing 23–24

ERROR condition 5–2, 5–6
default handling 5–7
END-ERROR key 5–21
overriding default handling 5–10

ERROR key 5–6, 5–8

ERROR option 5–4
RETURN statement 5–6

Error processing, blocks 3–18

ERROR-STATUS handle 5–6

Errors
handling by Progress 5–8
handling in procedures 5–10
processing 12–2

European Numeric Format (-E) startup parameter 17–56

Event-driven programming 2–2
definition 2–2
example 2–10
flow of execution 2–5
interrupt-drive handling 2–7
real-time 2–7
WAIT-FOR statement 2–10

Events
applying 16–42
CHOOSE 16–33
database 11–2, 11–27
DEFAULT-ACTION 16–33, 17–30, 17–38
defined 2–5
defining 16–34
DESELECT 24–12
direct manipulation 24–5, 24–11, 24–15
EMPTY-SELECTION 24–12
END-BOX-SELECTION 24–14
END-MOVE 24–13
END-RESIZE 24–13
ENTRY 16–33
interrupt signal 2–8
ITERATION-CHANGED 16–33
keyboard 6–35
keystroke 16–31
LEAVE 16–33
MENU-DROP 16–33, 22–18
mouse 16–31, 24–5
named 4–1
resizable browse, user manipulation 10–44
RETURN 18–9

SELECT 24–12
START-BOX-SELECTION 24–14
START-MOVE 24–13
START-RESIZE 24–13
triggers 2–5
user-interface 16–31
VALUE-CHANGED 16–33, 17–17, 17–22, 17–30, 17–32, 17–40, 17–44, 17–48, 22–16
WINDOW-CLOSE 16–33, 21–8
WINDOW-MAXIMIZED 16–33, 21–8
WINDOW-MINIMIZED 16–33, 21–8
WINDOW-RESTORED 16–33, 21–8

Example code, resizable browse 10–45

Examples
external procedures, non-persistent 3–21
internal procedures 3–35
persistent procedures 3–42
shared objects 3–27

Exclamation point (!)
format character 17–52
user ID lists 14–9

EXCLUSIVE-LOCK option, record phrase 12–8, 13–5, 13–13, 13–16

Executable code. *See* Run-time code

Execution buffer
available memory A–10
ceiling
described A–10
size A–10
described A–7, A–11

Execution environment
limits A–10, A–13
r-code A–7, A–11
typical layout A–8, A–12
statistics A–14
.srt file access A–15
execution map A–15
segment information A–16
startup parameters A–14
summary information A–17

EXPAND option, radio sets 17–20

EXPORT statement 7–38
DELIMITER option 7–38

Exporting data 7–38

Expression code segment, r-code A–2

EXTEND mouse button 6–27

External procedures
defining 3–20
described 3–2
examples, non-persistent 3–21
naming 3–20
running non-persistently 3–20
options 3–21
running persistently 3–38
shared objects 3–25

External user-defined functions. *See* Persistent procedures

F

Fetching records 9–21
defining record set 9–19
order 9–21
preselect list 9–23
results list 9–23
ROWID and RECID 9–23
using field lists. *See* Field lists

FGCOLOR, widget assignment 23–4

Field List Disable (-fldisable) startup parameter 9–37

Field lists 9–32
benefits 9–32
browse widget 10–4

cursor repositioning 9–36
DataServers 9–32, 9–35, 9–37
degenerate cases 9–37
deletes 9–35
deployment 9–37
implied entries 9–35
multiple queries 9–37
server versions 9–37
shared queries 9–34
specifying 9–33
updates 9–35

Field references 3–17

Field(s), FORM statement 25–14

Field-level widgets
child frames 19–21
defined 16–3
dynamic 20–4

Field-group widgets 19–39
accessing 19–46
characteristics 19–42
position relative to frames 19–43
size 19–42
types 19–41
visibility 19–44
z-order 19–43

Fields
order of update 25–14
validation 7–30

FILE option, image widget 18–9

File types 7–26

FILE-INFO system handle 7–27

FILE-NAME attribute, persistent procedures 3–40

{&FILE-NAME} name reference 8–14

Files
attributes 7–26
before-image 12–41
conversion map 7–29
convmap.cp 7–29
information 7–27
input 7–30
local-before-image (.lbi) 12–41
redirecting input 7–14
redirecting output 7–6
schema cache 9–78
See also Schema cache files
sort (.srt) A–7

Fill characters, display formats 17–52

FILL-IN widget, width 17–4

Fill-in widgets 17–2, 17–6
attributes 17–5
AUTO-RESIZE attribute 17–3
font used in Windows 17–4
NATIVE option 17–2

FIND CURRENT statement 13–20

FIND FIRST statement, using to sort work tables 15–10

FIND statement 9–17
repositioning 9–31
after FIND fetches 9–31
temporary tables 15–16

FIND trigger 11–2, 11–24

FIRST-CHILD attribute 22–26

FIRST-PROCEDURE attribute 3–39

FIRST-TAB-ITEM attribute 25–44

Flow of execution 2–2
event-driven programs 2–5
procedure-driven programs 2–2

Focus 10–9

Font dialogs 23–15
4GL statements 23–11, 23–16
Windows example 23–15

FONT, widget assignment 23–4

FONT-TABLE handle 23–18
environment management 23–24
example 23–6
using 23–20

Fonts
adding for startup 23–4
assigning 23–4
assigning to ActiveX controls 23–7
availability 23–2
character interfaces, management 23–8
common dialog 23–11
example 23–15
definitions 23–21
changing 23–21
environment files 23–2
See also Environment files
environments 23–23
graphical interfaces
definition access 23–21
environment management 23–23
management 23–10
table access 23–18
user changes 23–11
inheritance 23–7
Progress defaults on Windows 23–4
proportional width 17–4
sample application 23–4
table access 23–18
widget assignment 23–4
example 23–5

FOR blocks
described 3–2
error property 5–3

FOR EACH blocks 3–17
frame scoping 3–8
infinite loop protection 5–4

FOR EACH statement 9–17

FOR option
DO statement 3–10
REPEAT statement 3–10

FOR statement
EACH option 3–10

Form backgrounds 25–19

FORM statement 25–13, 25–14
frame header 25–17
in shared frames 19–38
shared frames 19–36

FORMAT attribute 17–5, 17–14, 17–45, 20–4

FORMAT option 17–49
fill-ins 17–3

Formats, display 17–49 to 17–58
character 17–51
date 17–58
defaults 17–51
logical 17–57
numeric 17–53

FRAME attribute 20–4
child frames 19–22
field groups 19–40

Frame families 19–19
defining 19–22
fill-in tabbing 25–48
shared 19–39
tab order 25–44
using 19–22
viewing and hiding 19–32

FRAME option, frame phrase 3–7

Frame phrase, FRAME option 3–7

Frame segment, r-code A–3

Frame services
advancing 3–8
clearing 3–8
hiding 3–8
viewing 3–8

FRAME-COL attribute 25–20

FRAME-COL function 25–20

FRAME-DOWN function 25–25

FRAME-FIELD function 6–41

FRAME-LINE function 25–24

FRAME-NAME attribute 17–5

FRAME-NAME function 17–5

FRAME-ROW attribute 25–20

FRAME-ROW function 25–20

FRAME-SPACING attribute 25–41

Frames 25–8
  advancing 19–34
  allocating 3–7, 19–5
  behavior 19–27, 19–31, 25–20, 25–22, 25–24, 25–25
    FRAME-ROW and FRAME-COL 25–20
  borders 25–6
  borrowed 3–9
  carry-over display 25–8
  CENTERED option 25–8
  characteristics 25–7
  child 19–19
  clearing 19–34
  default 3–7
  default button 18–9
  default services 19–26
  defined 19–3
  design issues 25–11
  determining type 19–12
  down 25–25
  dynamic 20–7
  families 19–19
  field-level design 25–13
  flashing 19–18
  hiding 19–27, 19–28
  internal procedures 3–9
  layout 25–14, 25–17
  moving 24–10
  multiple active 25–41
  non-terminal devices 25–41
  options 25–7
  overlaying 19–31
  parent 19–19
  resizing 24–10
  resolving references 3–7
  root 19–19
  scope 19–15
  scope of shared frames 19–39
  scoping properties 3–8
  scrollable 25–32
    character interfaces 25–33
  scrolling 25–27
    See also SCROLL statement 25–27
  services for blocks 3–8
  setting attributes 25–9
  shared 19–35
  tab order 25–43
  triggers 3–9, 19–17
  types 19–11
  unnamed 3–7
  usage rules 25–11
  validation 19–50
  variable-level design 25–13
  viewing 19–27
    See also TOP-ONLY, OVERLAY, NO-HIDE frame phrase options

Free references, buffers 3–10

FREQUENCY attribute 17–10

FREQUENCY option 17–10

Function keys
  changing definitions 6–25
  monitoring usage 6–35

Functions
  ASC 7–28
  AVAILABLE 13–19
  CAN-DO 14–7, 14–12, 14–14
  CHR 7–28
  CODEPAGE-CONVERT 7–28
  CONNECTED 12–39
  CURRENT-LANGUAGE 3–46
  ENCODE 14–2
  FRAME-COL 25–20
  FRAME-DOWN 25–25
  FRAME-FIELD 6–41
  FRAME-LINE 25–24
  FRAME-NAME 17–5

FRAME-ROW 25–20
KBLABEL 6–4
KEYCODE 6–4
KEYFUNCTION 6–4
KEYLABEL 6–4
LASTKEY 6–30
LOCKED 13–19
OS-GETENV 7–27
RETRY 5–4
RETURN-VALUE 5–4, 5–26
SEARCH 7–8
SETUSERID 14–5
SUBSTRING 7–35
TRANSACTION 12–40
user-defined
See also Persistent procedures
external. *See* Persistent procedures
local. *See* Persistent procedures
remote. *See* Persistent procedures
USERID 14–9
VALID-HANDLE 16–19
persistent procedures 3–39
procedure handles 3–29
WIDGET-HANDLE 16–19

G

GET CURRENT statement 13–20

GET statement, browse widget 10–11

GET-BYTE function 9–74

GET-KEY-VALUE statement 23–21
example 23–21

GET-SIGNATURE attribute, persistent procedures 3–40

GET-TAB-ITEM method 25–44

Global shared objects 3–25

&GLOBAL-DEFINE directive 8–2

GO event, DATA-ENTRY-RETURN 25–48

GO-ON phrase 6–30

GRAPHIC-EDGE attribute 18–16

GRAPHIC-EDGE option 18–16

Graphical user interfaces 2–1, 2–12

Greater-than sign (>) 17–54

GRID-FACTOR-HORIZONTAL attribute 24–17

GRID-FACTOR-VERTICAL attribute 24–17

GRID-SNAP attribute 24–18

GRID-UNIT-HEIGHT-CHARS attribute 24–18

GRID-UNIT-HEIGHT-PIXELS attribute 24–18

GRID-UNIT-WIDTH-CHARS attribute 24–18

GRID-UNIT-WIDTH-PIXELS attribute 24–18

GRID-VISIBLE attribute 24–17

Grids 24–17

GUI. *See* Graphical user interface

H

HANDLE attribute 16–6

Handle management 16–19

Handles
ACTIVE-WINDOW 25–49
attributes 16–20
COLOR-TABLE 23–18
component, purpose 16–16

CURRENT-WINDOW 21–1, 21–16
DEFAULT-WINDOW 21–1
ERROR-STATUS 5–6
FILE-INFO 7–27
FONT-TABLE 23–20
managing 16–19
procedures 3–28
query, buffer, and buffer-field object, purpose 16–16
server, purpose 16–16
SESSION 3–39
frame spacing 25–41
system 16–17
THIS-PROCEDURE 3–28
See also THIS-PROCEDURE handle
transaction, purpose 16–16
types 16–15
widgets 16–6
purpose 16–15

Header statements 19–7

Headers, frame 25–17

HEIGHT attribute 17–4

HEIGHT-CHARS attribute 17–11

HEIGHT-PIXELS attribute 17–11

Help
Progress 5–25
providing context-sensitive 17–5

Help facilities 1–10

Help submenu 22–7

HIDE statement 19–27

Hiding frames 3–8, 19–28

Highlighting 24–6
custom 24–10

HOME event, browse widget 10–20

I

&IF directive 8–6

Image widgets 18–9 to 18–14
built-in 18–4
buttons 18–3
CONVERT-3D-COLORS option 18–14
determining image file size 18–10
example 18–12
ignoring file at compile time 18–10
options 18–9
TOOLTIP option 18–14
TRANSPARENT attribute 18–14

IMAGE-SIZE option, image widget 18–9

IMAGE-SIZE phrase 18–10
button border thickness 18–6
buttons 18–6
NO-FOCUS option 18–6

IMAGE-SIZE-CHARS option, image widget 18–9

IMAGE-SIZE-PIXELS option, image widget 18–10

IMPORT statement 7–40, 7–41
DELIMITER option 7–39, 7–40, 7–43
UNFORMATTED option 7–41

Importing, data 7–40

IN FRAME phrase
attribute references 16–21
method references 16–23

IN MENU phrase 22–15

IN option, RUN statement 3–32
condition handling 3–33

IN WINDOW option 21–16

Include files, arguments 8–18

INDEX attribute 17–5

Index cursors 9–31

Index-sorted queries 9–26

INDEXED-REPOSITION option,
browse widgets 10–4

Indexes
defining, for temporary tables
15–22
positioning 3–17

Indexing, word 9–47

Infinite loop protections 5–4

Initial value segment, r-code A–2,
A–4

Initialization files, colors and fonts
23–2

Inner joins
defined 9–40
example 9–38, 9–43
mixing with left outer 9–47
specifying 9–40

INNER-CHARS attribute 17–35

INNER-CHARS option
editor widget 17–24
selection lists 17–31

INNER-LINES attribute 17–35,
17–45

INNER-LINES option
combo boxes 17–40
editor widget 17–24
selection lists 17–31

Input
changing the source 7–14
control character 7–18
devices 7–15
directory contents 7–15
end of file 5–16
file preparation 7–30
IMPORT statement 7–40
QUOTER utility 7–31
files 7–14
multiple sources 7–16, 7–19
streams 7–2
terminal 7–15
working within procedures 7–1

INPUT CLOSE statement 7–18
QUOTER utility 7–35

INPUT FROM statement 7–14
BINARY option 7–18
code page conversion 7–28
OS-DIR option 7–26
QUOTER utility 7–34
redirecting input 7–18

INPUT THROUGH statement 7–25
code page conversion 7–28

INPUT-OUTPUT THROUGH
statement
code page conversion 7–28

INSERT method 17–35, 17–45

INSERT statement 9–17, 9–19

Instantiating, RUN statement 3–38

INTEGER data type
combo boxes 17–39
formats 17–53
radio set 17–19
slider 17–9

Interaction modes 24–11

Interfaces
character 2–12
graphical 2–12

Internal procedures
buffers 3–33
call stack 3–37
compared to triggers 16–43
context 3–33
defining 3–30

described 3–2
encapsulation 3–32
See also Encapsulation
examples 3–35
execution options 3–31
frames 3–9
IN execution option 3–32
naming 3–31
prototyping 3–55
running 3–31
running in any context 3–32
streams 3–33
temporary tables 3–33
work tables 3–33

INTERNAL-ENTRIES attribute, persistent procedures 3–40

internationalization, word break tables 9–47

Interrupt-drive handling, event-driven programming 2–7

IS-SELECTED method 17–35

Iterating blocks 19–18

Iteration 3–5, 12–8

ITERATION-CHANGED event 16–33

J

Joins
See also Inner join, Left outer join
conditions 9–41
defined 9–38
full outer 9–46
inner 9–38, 9–40
left outer 9–40
mixing inner and left outer 9–47
nested reads 9–43
right outer 9–46
specifying 9–40

Journaling. *See* Transactions

K

KBLABEL function 6–4

KEEP-FRAME-Z-ORDER attribute 19–32

KEEP-TAB-ORDER option 25–44

Keyboard events 6–35

KEYCODE function 6–4

KEYFUNCTION function 6–4

KEYLABEL function 6–4

Keys
changing the function 6–25
codes 6–5, 6–6, 6–17
END-ERROR 5–19, 6–33
ENDKEY 5–13
ERROR 5–8
functions 6–19, 6–36
input 6–2
input precedence 6–3
labels 6–5, 6–6, 6–17, 6–36
STOP 5–18

Keystroke events 16–31

Keystrokes
monitoring 6–3, 6–35

L

LARGE attribute 17–26

LARGE option, editor widget 17–26

LARGE-TO-SMALL attribute 17–9

LARGE-TO-SMALL option 17–9

LAST-CHILD attribute 22–26

LAST-PROCEDURE attribute 3–39

LAST-TAB-ITEM attribute 25–44

LASTKEY function 6–30

LEAVE event 16–33

LEAVE processing 5–4, 5–17
  avoiding infinite loops 5–4

Left outer joins
  defined 9–40
  example 9–41, 9–45
  mixing with inner 9–47
  specifying 9–40

Less-than sign (<) 17–55

Libraries A–5
  PROPATH A–6

Licensed user count, exceeding 9–2

LIKE option 16–35
  connected database 9–78

{&LINE-NUMBER} name
  reference 8–14

LIST-ITEMS attribute 17–35, 17–36, 17–38, 17–44, 17–45, 17–46, 17–48

LIST-ITEMS option
  combo boxes 17–40
  selection lists 17–31

LISTING option, COMPILE statement 3–15

LOAD statement
  scenario 23–24
  using 23–23

LOAD-ICON method 21–5

LOAD-IMAGE method 18–11

LOAD-IMAGE-DOWN method 18–6

LOAD-IMAGE-INSENSITIVE method 18–6

LOAD-IMAGE-UP method 18–6

LOAD-SMALL-ICON 21–5

Local before-image files 12–41

Local user-defined functions. *See* Persistent procedures

LOCKED function 13–19

Locking 13–3

Locking records
  browse 12–33
  handling 12–9
  update 12–33

Locks
  See also Record locks
  browse widget 10–3
  deadly embrace 13–21

Logging. *See* Transactions

LOGICAL data type
  combo boxes 17–39
  radio set 17–19
  toggle box 17–14

Logical display formats 17–57

_login.p procedure 14–3, 14–5

LOOKUP method 17–35, 17–45

Looping 3–5, 12–8

Loops, infinite 5–4

Lost database connections 5–24

M

MANUAL-HIGHLIGHT attribute 24–10

MAX-CHARS attribute, editor widget 17–24

MAX-VALUE attribute 17–10

MAX-VALUE option, sliders 17–10

Maximized window, attributes 25–5

Maximum Memory (-mmax) startup parameter A–10

Menu bars 22–1, 22–2
  assigning a static menu bar to a window 22–8
  defining a static menu bar 22–7
  dynamic 22–23
  static 22–5

Menu hierarchy 22–4

Menu items
  creating menu items 22–25
  enabling and disabling 22–13
  referencing 22–15
  subtypes 22–25

Menu mnemonics 22–21

MENU mouse button 6–27

MENU-DROP event 16–33, 22–18

Menus 22–2
  accelerator keys 22–6, 22–18
  accelerators, character interface 22–20
  dynamic menu bars 22–23
  dynamic menus 22–21
    creating a menu bar 22–24
    creating menu items 22–25
    creating submenus 22–24
    example 22–27
    pop-up menus 22–29
    properties 22–22
    querying sibling attributes 22–26
    setting up menu bars 22–23
  help 22–7
  menu bars 22–1, 22–2
  menu hierarchy 22–4
  mnemonics 22–6
  nested pull-down menus 22–12
  pop-up menus 22–11, 22–29
  pull-down menus 22–2
  scrolling 25–27
  static menu bars 22–5
  static menu, defining a menu bar 22–7
  static menus 22–5
    assigning a menu bar to a window 22–8
    defining submenus 22–6
    duplicating menus 22–16
    enabling and disabling menu items 22–13
    menu mnemonics 22–21
    nested submenus 22–12
    pop-up menus 22–11
    setting up menu bars 22–6
    toggle boxes 22–15
  strip 25–34

&MESSAGE directive 8–10

MESSAGE statement 25–2
  alert boxes 25–7

Messages
  application 5–26
  compiler 5–25
  displaying in active window 25–49
  during execution 5–25
  startup 5–25

Metaschema tables, CRC calculations A–20

Methods
  ADD-FIRST 17–35, 17–45
  ADD-LAST 17–35, 17–38, 17–44, 17–45, 17–48
  combo box widget 17–45
  DELETE 17–35, 17–45
  editor widget 17–27
  ENTRY 17–35, 17–45
  GET-TAB-ITEM 25–44
  INSERT 17–35, 17–45
  IS-SELECTED 17–35
  LOAD-ICON 21–5
  LOAD-IMAGE 18–11

LOAD-IMAGE-DOWN 18–6
LOAD-IMAGE-INSENSITIVE 18–6
LOAD-IMAGE-UP 18–6
LOAD-SMALL-ICON 21–5
LOOKUP 17–35, 17–45
MOVE-AFTER-TAB-ITEM 25–44
MOVE-BEFORE-TAB-ITEM 25–44
MOVE-COLUMN 10–32
MOVE-TO-BOTTOM 19–32
MOVE-TO-TOP 19–32
REPLACE 17–35, 17–45
SCROLL-TO-ITEM 17–35
selection list 17–35
VALIDATE 19–49

Methods, widget 16–23
example 16–25
expressions 16–24
invoking 16–24
referencing 16–23

MIN-VALUE attribute 17–10

MIN-VALUE option, sliders 17–10

Minus sign (-) 17–55, 17–57

-mmax startup parameter, execution buffer ceiling A–10

Mnemonics, menu items 22–6

Models, programming 2–1

Modes
direct manipulation 24–11
transactions 12–28

Monitoring keystrokes 6–3, 6–35
EDITING blocks 6–35
user interface triggers 6–35

Mouse
buttons 6–27, 24–5
EXTEND 6–27
MENU 6–27
MOVE 6–27
events 16–31, 24–5
specifying 6–28
input 6–2

Mouse buttons, SELECT 6–27

Move mode 24–11

MOVE mouse button 6–27

MOVE-AFTER-TAB-ITEM method 25–44

MOVE-BEFORE-TAB-ITEM method 25–44

MOVE-COLUMN method, browse widget 10–32

MOVE-TO-BOTTOM method 19–32

MOVE-TO-TOP method 19–32

Moving
browse 10–41
browse-column 10–42

Moving widgets 24–8, 24–16

MS-Windows, three-dimensional effects 25–49

Multi-database applications
referencing files and field names 9–75
transaction behavior 12–39

Multi-database connections
CONNECTED function 9–7
connection overhead 9–9
connection parameters 9–2
conserving 9–9
considerations at development 9–5
disconnecting 9–10
failures and disruptions 9–6
logical database name 9–11
methods list 9–2
Physical Database Name (-db) startup parameter 9–3

positioning references in applications 9–77
Progress functions 9–8
run-time considerations 9–5

Multi-databases
disconnecting 9–10
logical database name 9–10

Multi-user
applications 13–2
environment 13–2
record locks 13–3
sharing records 13–3
starting client 13–17
starting server 13–17

Multiple
input sources 7–16
output destinations 7–7

MULTIPLE attribute 17–35

MULTIPLE option, selection lists 17–31

Multiple windows
non-persistent management 21–14
persistent management 21–16
creating 21–16
example 21–17
widget pools 21–16
pros and cons 21–2

Multiple-select browse, defined 10–9

Multi-window applications, described 21–14

N

N, format character 17–52

Named events 4–1

Native fill-ins 17–2

Nested Blocks (-nb) startup parameter 3–19, 3–24

Nested pull-down menus 22–2

Nested submenus 22–12

NEXT processing 5–4
avoiding infinite loops 5–4

Next tab item 25–45

NEXT-SIBLING attribute 22–26
persistent procedures 3–39

NEXT-TAB-ITEM attribute 25–44

NEXT-VALUE function 9–69
syntax 9–68

NO-APPLY option 5–4

NO-BOX option, frame phrase 25–41

NO-CONVERT-3D-COLORS attribute 18–8

NO-CONVERT-3D-COLORS option 18–8

NO-CURRENT-VALUE attribute 17–10

NO-CURRENT-VALUE option 17–10

NO-DRAG option, selection lists 17–31

NO-ERROR option 5–6
CONNECT statement 9–3

NO-FILL option, rectangle widget 18–15

NO-FOCUS attribute 18–4

NO-FOCUS option 18–4

NO-LOCK option, record phrase 13–8, 13–13, 13–16

risks 13–10

NO-ROW-MARKERS option 10–8

NO-UNDO option
temporary tables 15–19
variables defined 12–36, 12–42

Non-persistent procedures, described 3–37

Nonexecutable code. *See* Compile-time code

Non-Progress databases 9–2

NORMAL menu items 22–25

NUM-ITEMS attribute 17–35, 17–45

NUM-TABS attribute 25–44

Number of Databases (-h) startup parameter 9–2

Number of Users (-n) startup parameter 9–6

Numeric colors, Windows color dialog 23–12

Numeric display formats 17–53

O

ON ENDKEY phrase 5–17

ON ERROR phrase 5–10

ON QUIT phrase 5–19

ON statement 11–22, 16–34
ANYWHERE option 6–35, 6–41, 16–40
changing key function 5–9, 5–13, 6–25
REVERT option 16–39

ON STOP phrase 5–18
database disconnection 5–24

OPEN QUERY statement, browse widget 10–11

{&OPSYS} name reference 8–15

OS-DIR option, INPUT FROM statement 7–26

OS-GETENV function 7–27

_osprint.p 7–11

Outer joins
see also Left outer join
full outer 9–46
right outer 9–46

Output
changing the destination 7–2
devices 7–6
files 7–6
multiple destinations 7–7, 7–19
printers 7–3
PUT statement 7–39
streams 7–2
terminal 7–6
Windows clipboard 7–6
working within procedures 7–1

OUTPUT CLOSE statement 7–8, 7–9, 7–22

OUTPUT STREAM statement 7–21

OUTPUT THROUGH statement 7–25
code page conversion 7–28

OUTPUT TO statement 7–2, 7–9
code page conversion 7–28
PAGED option 7–6

Overlay frames 19–31
using 19–30

OVERLAY option, DISPLAY statement 19–31

Overriding triggers 11–23

Overriding, internal procedures and user-defined functions 3–52

P

PAGED option, OUTPUT TO statement 7–6

Parameter files
- different systems 9–2
- invoking 9–2
- Parameter File (-pf) startup parameter 9–2

PARENT attribute 22–26

Parent frames 19–19
- defining 19–22

Parent windows 21–9

Parentheses ( ) 17–53

Password 14–2
- ENCODE function 14–2
- validating password 14–3

Period (.) 17–54

Permissions table 14–10
- adding records 14–10
- security 14–16
- using the Data Dictionary to create 14–10

PERSISTENT attribute, persistent procedures 3–40, 21–19

Persistent procedures
- advantages 3–38
- condition handling 3–46
- context 3–44
- creating 3–38
- CURRENT-LANGUAGE variable changes 3–46
- CURRENT-WINDOW attribute 21–20
- database disconnection 3–46
- deleting the context 3–41
- described 3–37
- dialog boxes 21–29
- DISABLE TRIGGERS statement 3–44
- encapsulation 3–38
  - *See also* Encapsulation
- examples 3–42
- handles 3–39
- instantiating 3–38
- NEXT-SIBLING attribute 3–39
- PERSISTENT attribute 21–19
- PREV-SIBLING attribute 3–39
- PRIVATE-DATA attribute 21–24
- QUIT condition 3–46
- run-time parameters 3–45
- schema changes 3–46
- schema locks 3–46
- SESSION handle, example 21–19, 21–24
- shared objects 3–44
- STOP condition 3–46
- transactions 3–45
- UNDO processing 3–45
- widget pools 20–15
- window management 21–16
  - *See also* Persistent windows

Persistent triggers 16–36

Persistent widget pools 20–14

PFCOLOR
- using 23–8
- widget assignment 23–4
- widget states 23–8

Physical Database Name (-db) startup parameter, connection parameter 9–2

Pixels 25–4
- conversion with character units 25–5

Plus sign (+) 17–54

Pop-up menus 22–3, 22–29
- dynamic pop-up menus 22–29
- static pop-up menus 22–11

Portable accelerators 22–20

Portable r-code A–17

Positioning
  character units 25–17
  pixels 25–17

Preparing input files 7–30

Preprocessors 8–1
  &ELSE directive 8–6
  &ELSEIF directive 8–6
  &ENDIF directive 8–6
  &GLOBAL-DEFINE directive 8–2
  &IF directive 8–6
  &MESSAGE directive 8–10
  &SCOPED-DEFINE directive 8–2
  &THEN directive 8–6
  &UNDEFINE directive 8–5
  {&BATCH-MODE} 8–14
  {&FILE-NAME} 8–14
  {&LINE-NUMBER} 8–14
  {&OPSYS} 8–15
  {&SEQUENCE} 8–15
    saving value 8–16
  {&WINDOW-SYSTEM} 8–15
  built-in names 8–14
    quoting 8–16
    references 8–14
  DEFINED function 8–6, 8–10
  expanding names 8–11
  expressions 8–7, 8–9
  name scoping 8–3
  names 8–2, 8–5, 8–10, 8–11
  nesting references 8–17
  operators 8–7
  referencing names 8–11

Preselect lists. *See* Results lists

PRESELECT option, browse widgets 10–4

PRESELECT statement, EACH option 3–10

Preselected queries 9–27

Presorted queries 9–26

PREV-SIBLING attribute, persistent procedures 3–39

PREV-TAB-ITEM attribute 25–44

Previous tab item 25–45

Printers
  redirecting output 7–3
  setting up on Windows 7–4, 7–5

Printing
  _osprint.p 7–11
  converting widgets 7–10
  solutions 7–11
  text files 7–11

PRIVATE-DATA attribute 16–35
  persistent procedures 3–40, 21–24

Procedure blocks 3–17
  described 3–2
  error property 5–3
  frame scoping 3–8
  infinite loop protection 5–4

Procedure call stack
  described 3–23
  internal procedures 3–37

Procedure context, defined 3–19

Procedure handles
  attributes 3–39
  compared to other handles 16–16
  defined 3–28
  TYPE attribute 3–29
  validity 3–29
    running internal procedures 3–32

Procedure overriding 3–52

Procedure parameters, run-time 3–23

PROCEDURE statement 3–30

Procedure triggers
applying events 3–29
defined 3–28

Procedure-driven programming 2–2
definition 2–2
example 2–8
flow of execution 2–2

Procedures
call stack 3–23
changing output destination 7–2
changing the input source 7–14
compiling 1–26
context 3–19
external procedures 3–23
internal procedures 3–33
continuing processing 6–29
errors 5–6
executing 1–26
external 3–2
See also External procedures
handles 3–28
purpose 16–16
internal 3–2, 3–30
See also Internal procedures
compared to triggers 16–43
monitoring keystrokes 6–3
non-persistent 3–37
See also Non-persistent procedures
parameters 3–23
persistent 3–37
See also Persistent procedures
properties 3–19
r-code integrity A–24
example A–25
running 1–26
scope 3–5
shared objects 3–25
sharing streams 7–22
super 3–52
using for security checking 14–12

Processes, input and output streams 7–25

Processing, all-or-nothing 12–3

Program execution. *See* Flow of control

Program structure
event-driven programs 2–9
procedure-driven programs 2–8

Programming models 2–1
event-driven 2–1
procedure-driven 2–1

Programming, multi-database, techniques, and considerations 9–75

Programs
event-driven, example 2–10
procedure-driven, example 2–8

Progress 4GL 1–12
connectivity 1–11
consistency 1–7
data model 1–8
default processing 1–10
encapsulation 1–9
event handler 1–9
features 1–6
flexibility 1–7
foundations 1–1
help facilities 1–10
internationalization 1–12
interoperability 1–10
localizability 1–12
portability 1–11
preprocessor 1–9
programming models 1–9
security 1–10
syntax and semantics 1–12
See also Syntax of the 4GL
user interface model 1–8

Progress database functions 9–8

Progress file, colors and fonts 23–2

progress.ini file, Use-3D-Size parameter 25–52

PROLIB utility A–5

PROMPT-FOR statement 9–17

_prostar.p procedure 14–3

PROTERMCAP file, colors and fonts 23–2

Prototyping, super procedures 3–55

PUBLISH statement 4–2

PUBLISHED-EVENTS attribute 4–2

Pull-down menus 22–2

PUT statement 7–39

PUT-BYTE statement 9–74

PUT-KEY-VALUE statement 23–21
  environment management 23–24
  example 23–22

Q

Queries 9–21
  See also Fetching records
  browse selection 10–11
  browse widgets 10–3
  field lists 9–33, 9–37
  results lists 9–26

Query object 20–15

Query, buffer, and buffer-field objects, handles, purpose 16–16

Query, dynamic query 20–15

QUIT condition 5–2, 5–19
  default handling 5–19
  overriding default handling 5–19
  persistent procedures 3–46

QUIT statement 5–19

QUOTER utility 7–31
  example procedure 7–34
  interactive 7–37
  options
    columns in fields 7–36
    field delimiter 7–36

R

R-code CRC
  schema triggers A–25

Radio set widget 17–19
  initial value 17–21
  options 17–20

Range, slider widget 17–10

Rapid prototyping 1–26

RAW data field 11–13

RAW data type 9–73, 11–10

RAW statement 9–74

RAW TRANSFER statement 11–10, 11–13, 11–14

R-code
  action code segment A–2, A–3
  CRC A–24
  debugger segment A–3
  defined A–1
  directory
    described A–7, A–11
    management A–13
  execution A–7
    available memory A–11
    environment limits A–10, A–13
    environment statistics A–14
    segment management A–10, A–13
    sequence A–9, A–12
  execution buffer, described A–7, A–11
  execution environment A–7, A–11
    typical layout A–8, A–12
  expression code segment A–2

factors affecting size A–3
features A–2, A–7
files
segment layout A–4
frame segment A–3
initial value segment A–2, A–4
memory-mapped execution environment A–11
portability A–17
segment descriptor table, described A–7, A–11
segment types A–2
standard execution environment A–7
structure A–2
text segment A–2

R-code CRC A–24, A–25
procedure integrity A–24, A–25
RCODE-INFO handle A–24
schema triggers
example A–27

RCODE-INFO handle, R-code CRC A–25

R-code directory
described A–7, A–11
management A–13
size A–10

R-code file, segment layout A–5

R-code segments A–2

READ-ONLY attribute, browse widgets 10–8

READ-ONLY menu items 22–25

Reading multiple tables. *See* Joins

Read-only browse, defined 10–5

Realizing widgets 16–26

Real-time 2–7
definition 2–7
event-driven programming 2–7

RECID data type
See also ROWID data type
converting to ROWID 9–25
DBRESTRICTIONS function 9–25

Record buffers 3–10, 9–16

Record locks
applying 13–6
AVAILABLE function 13–19
avoiding record conflicts 13–4
bleeding 13–15
bypassing 13–8
common problem 13–11
duration 13–10
LOCKED function 13–19
multi-user applications 13–4
multiple buffers 13–15
NO-LOCK 13–8
releasing 13–10, 13–12
resolving conflicts 13–16
changing size of transactions 13–22
transactions 13–10
types 13–13

Record phrase
EXCLUSIVE-LOCK option 13–5, 13–13, 13–16
field lists 9–33
NO-LOCK option 13–16
SHARE-LOCK option 13–6, 13–13, 13–16

Record scoping
free references 3–10
strong 3–10
weak 3–10

Records
adding 9–19
adding to a permissions table 14–10
deleting 9–19
fetching 9–21
locks 12–9, 13–3
releasing 3–17
retrieving 3–6
scoping 3–10
sharing 13–3

validation 3–17
writing 3–17

Rectangle widget 18–15 to 18–16
colors 18–15
example 18–16
proportions in character mode 18–16
TOOLTIP option 18–15

Redirecting I/O 7–25
input streams 7–14, 7–18
output streams 7–2, 7–7

References
free 3–10
resolving to fields 3–17
resolving to frame 3–7

Referencing widgets
dynamic 16–13
static 16–11

Registry
changing resources 23–21
colors and fonts 23–2

RELEASE statement 9–17, 13–10

Releasing records 3–17, 13–10, 13–12

Remote user-defined functions. *See* Persistent procedures

REPEAT blocks 3–17
described 3–2
error property 5–3
frame scoping 3–8
infinite loop protection 5–4

REPEAT statement
FOR option 3–10
frame allocation 3–7

REPLACE method 17–35, 17–45

Replication Changes Log. *See* Data replication

Replication configuration. *See* Data replication

REPLICATION-CREATE trigger 11–10, 11–12

REPLICATION-DELETE trigger 11–10, 11–12

REPLICATION-WRITE trigger 11–10, 11–12

Reports
control-break 15–3
cross-tab 15–13
spreadsheet 15–13
tabular 15–13
using work tables to write 15–3

REPOSITION statement 9–29
browse widgets 10–4, 10–11
ROWID 9–25

Resizable browse 10–40
additional attributes
programmatic manipulation 10–44
user manipulation 10–43
code example 10–45
events, user manipulation 10–44

Resize mode 24–11

Resizing 10–41
browse 10–40
browse-column 10–41
widgets 24–9, 24–16

Resource, definitions, changing 23–21

Results lists 9–26
navigating 9–29

RETAIN option, frame phrase 25–8

RETAIN-SHAPE, Image widget 18–10

Retaining data during RETRY 19–34

See also RETAIN option

Retrieving records. *See* Fetching records

RETRY function 5–4

RETRY processing 5–4, 5–7, 5–14
  infinite loops 5–4
  retaining data 3–8

RETURN event 18–9

RETURN key
  dialog boxes 18–9
  fill-in 17–5

RETURN processing 5–4

RETURN statement 5–26
  ERROR option 5–6

RETURN-VALUE function 5–4, 5–26

Returning ERROR 5–4

REVERT option 16–39

Reverting triggers 11–23, 16–39

RGB-VALUE function 23–2, 23–7
  ActiveX controls 23–7

Root frames 19–19

Root windows 21–9

Rounding errors, character units 25–5

ROW attribute 20–4

ROW frame phrase option 19–31

Row height, browse, changing 10–43

Row mode, character browse 10–46

ROW option, frame phrase 25–41

ROW-DISPLAY event 10–23

ROW-ENTRY event 10–23

ROW-LEAVE event 10–23

ROWID data type 9–23
  converting from RECID 9–25
  CREATE statement 9–25
  DataServer
    portability 9–25
    value construction 9–26
    versions 9–25
    views 9–25
  INSERT statement 9–25
  RECID 9–23
  REPOSITION statement 9–25
  returning record values 9–24
  storing values 9–24
  value scope 9–25

ROWID function 9–24

RULE menu items 22–25

RUN statement
  compared to COMPILE statement 1–26, 1–27
  for external procedures, persistent 3–38
  IN option 3–32
  instantiating 3–38
  internal procedures 3–31
  non-persistent external procedures 3–20

RUN-PROCEDURE option, SUBSCRIBE statement 4–2

Run-time connection, methods list 9–5

Run-time code 1–19
  execution 1–26
  interaction with compile-time 1–23
  object names 1–25
  syntax elements 1–22
  uses 1–27

Run-time parameters
persistent procedures 3–45
procedure 3–23

Run-time procedure execution 1–26

Run-time security 14–1

S

Sample code, resizable browse 10–45

SAVE CACHE statement 9–79

Schema Cache File (-cache) startup parameter 9–82

Schema cache files
building 9–78
creating 9–78
example 9–80
DataServers 9–79
described 9–78
saving entire schema cache 9–79
saving partial schema cache 9–80
startup 9–82

Schema changes, persistent procedures 3–46

Schema locks, persistent procedures 3–46

Schema security 14–1

Schema triggers 11–3, 11–4
auto-deployment example A–27
locating 11–27
R-code CRC A–25
RCODE-INFO handle A–25
setting A–24
validating A–25
r-code CRC A–24
deployment A–27

Scope 1–16, 3–5
block context 1–16, 3–5
compared to context 3–24
data and widgets 3–5
dynamic resources 1–17
frames 3–8
internal procedures 3–5
preprocessor names 8–3
records 3–10
triggers 3–5, 16–36
unscoped resources 1–16

Scoped shared objects 3–25

&SCOPED-DEFINE directive 8–2

Scoping, dynamic, internal procedures and user-defined functions 3–52

Screen
buffers 9–16
resolutions 25–4

SCREEN-IO option 7–10

SCREEN-VALUE attribute 17–35, 17–45, 20–4, 20–5

Screens
See also Frame(s)
designing frames 25–2
length in lines 25–3
width 25–3

Scroll bars
editor widgets 17–26
selection lists 17–32

Scroll mode, character interfaces 25–33

SCROLL-NOTIFY event 10–34
browse widget 10–20

SCROLL-TO-ITEM method 17–35

Scrollable frames 25–32
character interfaces 25–33

SCROLLABLE option 10–3

Scrolling
editor widgets 17–26

selection lists 17–32

Scrolling queries 9–26

SEARCH function 7–8

Searching, browse columns 10–15

Security
application 14–1
checking procedures 14–12
checking user IDs at run-time 14–7
checking, activities-based 14–10
compile-time 14–1
connection 14–1
creating, permissions table 14–10
password 14–2
validating password 14–3
permissions table 14–16
run-time 14–1
schema 14–1

Segment descriptor table, described A–7, A–11

Segment layout, r-code file A–5

Segment management, r-code A–10, A–13

Segment Statistics (-yd) parameter A–14

Segmented r-code A–2
types A–2

SELECT event 24–12

SELECT mouse button 6–27

Selecting widgets 24–6
box selecting 24–7
multiple 24–8

Selection list widgets 17–30 to 17–39
attributes 17–35
character interfaces 17–32
dragging 17–31, 17–32
example 17–33, 17–36
methods 17–35
options 17–31
scrolling 17–32

SELF system handle 16–35

Semantics of the 4GL
compile-time compared to run-time 1–19
context 1–15
dynamic resources 1–17
scope 1–16

SENSITIVE attribute 17–6, 19–49

Sequence definitions
creating and maintaining 9–67
storage 9–67

{&SEQUENCE} name reference 8–15
saving value 8–16

Sequence statements and functions 9–68

_Sequence table 9–67

Sequences
accessing values 9–68
attributes 9–64
compiling procedures 9–71
control tables 9–63
performance 9–65
storage limits 9–67
trade-offs 9–64
transaction handling 9–66
defined 9–62
dumping and loading 9–67
fields 9–63
security 9–67
storage 9–67
transactions 9–66
WHERE clause 9–72

Server handles, compared to other handles 16–16

Servers, starting 13–17

SESSION handle

frame spacing 25–41
persistent procedures 3–39
example 21–19, 21–24

Session triggers 11–3, 11–21

SET statement 9–17

SET-REPOSITIONED-ROW
event 10–25
method 10–25

SETUSERID function 14–5

SHARE-LOCK option, record phrase 13–13, 13–16
See also Record locking

Shared frames 19–35

Shared objects
call stack 3–44
defining and referencing 3–27
external procedures 3–25
internal procedures 3–33
persistent procedures 3–44
scoped compared to global 3–25

Shared queries, field lists 9–34

Sibling attributes, querying 22–26

SIDE-LABEL-HANDLE attribute 20–4

SINGLE option, selection lists 17–31

Single-select browse, defined 10–9

Single-window applications, described 21–14

SIZE phrase
button border thickness 18–6
buttons 18–6
combo boxes 17–40
editor widgets 17–24
fill-ins 17–3
NO-FOCUS option 18–6
selection lists 17–31
sliders 17–11

Sizing widgets 16–30

SKIP menu items 22–25

Slider widget 17–9
attributes 17–10
even marker movement, character mode 17–11
example 17–12
FREQUENCY option 17–10
LARGE-TO-SMALL option 17–9
manipulating 17–10
NO-CURRENT-VALUE option 17–10
range 17–10
size 17–11
TIC-MARKS option 17–10

Software failure, handling by Progress 5–24

Sort (.srt) files A–7, A–10
r-code execution A–9, A–12

SORT attribute 17–35, 17–45

SORT option, combo boxes 17–41

Sorting
comparing two methods 15–12
using work tables 15–9

Specifying default frames 19–7

Spreadsheet reports. *See* Cross-tab reports

.srt file. *See* Sort (.srt) file

Stack Size (-s) startup parameter 3–24

START-BOX-SELECTION event 24–14

START-MOVE event 24–13

START-RESIZE event 24–13

START-SEARCH event 10–19

Startup messages 5–25

Startup parameters
  connection 9–2
  European Numeric Format (-E) 17–56
  Field List Disable (-fldisable) 9–37
  grouping 9–2
  multi-database connection 9–2
  Number of Databases (-h) 9–2
  Physical Database Name (-db) 9–2
  Schema Cache File (-cache) 9–82
  Startup Procedure (-p) 9–7
  Time Stamp (-tstamp) 9–71
  Trigger Location (-trig) 9–72
  Triggers (-trig) 11–27

Startup procedure
  retrying 5–18
  specifying 9–7

Startup Procedure (-p) startup parameter 9–7

Statements
  4GL 1–12
  data handling 9–16
  DEFINE BUTTON 18–2
  DISPLAY 19–31, 25–8
  executable 1–19
  nonexecutable 1–19
  static and dynamic widgets 20–2
  VIEW 19–27, 25–17

Static frames 19–4

Static menu bars 22–7

Static menus 22–5
  enabling and disabling menu items 22–13
  nested submenus 22–12
  pop-up menus 22–11
  setting up menu bars 22–6

Static submenus, defining
  submenus 22–6

Static widgets
  creating 16–7
  defined 16–5
  describing 16–7
  instantiating 16–8
  triggers 16–34

Statistics (-y) startup parameter A–14

STOP condition 5–2, 5–18
  database disconnection 5–24
  default handling 5–18
  overriding default handling 5–18
  persistent procedures 3–46

STOP key 5–18

Stream I/O 7–10
  converting widgets 7–10

STREAM-IO option 7–10

Streams 7–2
  assignment 7–2
  closing 7–8
  default 7–2
  defining additional 7–18
  establishing 7–19
  input 7–2, 7–18
  internal procedures 3–33
  opening 7–21
  output 7–2, 7–18
  redirecting 7–15
    See also OUTPUT statements
  redirecting input. *See* OUTPUT statements
  shared 7–22
  unnamed 7–2

STRETCH-TO-FIT option
  Image widget 18–10

String conversion, ROWID data type 9–24

STRING function, ROWID 9–24

Strip menus 25–34

See also CHOOSE statement

Strong record scoping 3–10

SUB-MENU-HELP phrase 22–7

Submenus
dynamic 22–24
static submenus 22–6

Subprocedures, transactions 12–34

SUBSCRIBE statement 4–2

SUBSTRING function
QUOTER utility 7–35

Subtransactions 3–17, 12–16
mechanics 12–41

SUBTYPE attribute 17–3

Super procedure prototyping 3–55

Super procedures 3–52

Syntax of the 4GL 1–12
blocks 1–14
comments 1–13
statements 1–12

System failures
handling 12–15
handling by Progress 5–24

System handles 16–1, 16–17
ERROR-STATUS 5–6
SELF 16–35

SYSTEM-DIALOG COLOR
statement 23–11
environment management 23–24
example 23–13

SYSTEM-DIALOG FONT
statement 23–11
environment management 23–24
example 23–16

SYSTEM-DIALOG
PRINTER-SETUP statement 7–4, 7–5

T

-T startup parameter, temporary file directory 15–15

Tab
item 25–44
order 25–43

TAB-POSITION attribute 25–44

Tabbing
fill-in 17–5
fill-in RETURN 25–48

TABLE attribute 17–5

Tables
CRC calculation A–20
joining. *See* Joins
reading multiple 9–38
word break 9–47

Tab-order widget 25–44

Temporary Directory (-T) startup parameter 15–15

Temporary tables 15–15
See also Work tables
advantages over work tables 15–16
based on database tables 15–19
buffer size 15–15
characteristics 15–15
DEFINE TEMP-TABLE statement 15–16
deletion 15–19
differences from work tables 15–19
indexes 15–16, 15–22
internal procedures 3–33
joins 9–41
performance 15–19
sharing 15–19
similarities to work tables 15–19
using FIND statement 15–16

Terminals, screen formatting 25–2

Terminating sequences, resetting 9–72

Text segments
  r-code A–2
  translation A–4

Text widgets 17–6
  example procedure 17–7

&THEN directive 8–6

THIS-PROCEDURE handle 3–28

THREE-D attribute 25–49

Three-dimensional effects 25–49, 25–51
  example 25–50
  images 25–51
  layout considerations 25–52
  rectangles 25–51
  windows 25–52

TIC-MARKS attribute 17–10

TIC-MARKS option 17–10

Tilde (~), input files 7–42

Time Stamp (-tstamp) startup parameter 9–71

Time stamps
  compared to CRC A–21
  database changes affecting A–21
  deployment A–22
  validation A–19

TITLE option, DISPLAY statement 25–8

Title, frames 25–8

TO-ROWID function 9–24

Toggle box widget 17–14
  attributes 17–14

Toggle boxes, menu items 22–15

TOOLTIP attribute 18–2, 18–14, 18–15

TOOLTIP option 18–2, 18–14, 18–15

TRANSACTION function 12–40

TRANSACTION option, blocks 12–8, 12–18

Transactions 12–2
  active 12–40
  begin and end 12–8
  benefits of small 13–21
  blocks 3–17, 12–8
  changing size, resolve record locks 13–22 to 13–25
  controlling 12–18
  database connections 12–39
  defined 12–2
  handles, purpose 16–16
  handling 12–2
  improving efficiency 12–42
  input from a file 12–36
  making larger 12–19, 13–23
  making smaller 12–21, 13–24
  mechanics 12–41
  modeless model 12–28
  multi-windows 21–3
  multiple databases 12–39
  persistent procedures 3–45
  processing 12–3
  relationship to record locks 13–10
  specifying UNDO processing 12–15
  subprocedures 12–34
  subtransactions 12–16
  triggers 12–23
  variables 12–36

TRANSPARENT, Image widget 18–10

Trigger blocks 3–17
  described 3–3
  infinite loop protection 5–4

Trigger Location (-trig) startup parameter 9–72

Trigger-based replication 11–10

Triggers
database 11–1
defined 2–6
definitional 16–34
dynamic widgets 16–36
execution 16–36
frames 3–9, 19–17
LIKE option 16–35
NO-APPLY option 5–4
persistent 16–36
procedure 3–28
See also Procedure triggers
real-time response 2–8
reverting 16–39
run-time 16–34
schema 11–4
scope 16–36
static widgets 16–34
transactions 12–23
universal 16–40
user-interface 16–31

Triggers (-trig) startup parameter 11–27

Two-phase commit 12–39

TYPE attribute 3–29
handles 16–19

U

&UNDEFINE directive 8–5

UNDO keyword, usage rules 5–3

UNDO processing 5–7, 5–10, 5–14, 5–17, 5–18, 5–19, 12–2, 12–15, 12–36
database disconnection 5–24
persistent procedures 3–45

UNDO statement 12–17

UNFORMATTED option, IMPORT statement 7–41

Universal triggers 16–40

UNIX, processes as input and output streams 7–25

UNLOAD statement
scenario 23–24
using 23–23

Unnamed frame 3–7

Unnamed streams 7–2

Unscoped resources 1–16

UNSUBSCRIBE statement 4–2

UPDATE statement 9–17
use of FORMAT option 17–49
using VIEW-AS phrase 17–2

Updateable browse, defined 10–7

USE statement
scenario 23–24
using 23–23

Use-3D-Size parameter, progress.ini file 25–52

USE-DICT-EXPS option, frame phrase 19–50

USE-TEXT attribute 17–8

User count, exceeding 9–2

User IDs 14–2
blank 14–5
checking at run time 14–7
validating user ID 14–3, 14–9
validation list 14–9

User interfaces
character 2–12
graphical 2–12

User-defined functions 3–47

external. *See* Persistent procedures
local. *See* Persistent procedures
prototyping 3–56
remote. *See* Persistent procedures

User-interface components 16–1
See also Widgets

USERID function 14–9, 14–10

User-interface events 16–31

User-interface triggers 6–35
instead of EDITING blocks 6–37
monitoring keystrokes 16–31

V

VALID-HANDLE function 16–19
persistent procedures 3–39
procedure handles 3–29

VALIDATE method
data widgets 19–52
frames 19–49

VALIDATE option
DEFINE BROWSE statement 19–49
DEFINE FRAME statement 19–49
FORM statement 19–49
temporary tables 15–19

Validation
Data Dictionary
disabling 19–54
forcing 19–51
data widgets 19–52
field-level 7–30
frames 19–49
records 3–17

VALUE option
OUTPUT TO statement 7–8
run-time object names 1–25

VALUE-CHANGED event 16–33, 17–17, 17–22, 17–30, 17–32, 17–40, 17–44, 17–48, 22–16
browse widget 10–20

Variables, transactions 12–36

Version, of Progress, determining the current 6–19

VIEW statement 19–27, 25–17

VIEW-AS phrase 16–10, 17–2

Viewing, frames 3–8

VIRTUAL-HEIGHT-CHARS attribute 25–32

VIRTUAL-HEIGHT-PIXELS attribute 25–32

VIRTUAL-WIDTH-CHARS attribute 25–32

VIRTUAL-WIDTH-PIXELS attribute 25–32

Visual colors, Windows color dialog 23–12

W

WAIT-FOR condition 2–7

WAIT-FOR statement 2–9, 2–10
END-ERROR key 5–23

Weak record scoping 3–10

Widget attribute references 16–21

Widget handles
compared to other handles 16–15
defined 16–6

Widget method references 16–23

Widget pools 20–9
creating 20–9
deleting 20–9, 20–11, 20–14

named 20–10
persistent procedures 20–10
persistent 20–10
session pool 20–9
unnamed 20–14
persistent procedures 20–14

Widget types 16–3

WIDGET-HANDLE data type 16–6

WIDGET-HANDLE function 16–19

Widgets 16–1
atomic 16–3
attributes 16–20
box selecting 24–7
browse 10–1 to 10–53, 13–9
button 18–2, 18–9
DEFAULT option 18–9
combo box 17–39 to 17–49
attributes 17–45
example 17–43 to 17–46
item selection 17–39 to 17–48
methods 17–45
options 17–40
value settings 17–42
containers 16–3
converting for stream I/O 7–10
creating 16–6
data representation
choosing 17–1 to 17–5
defined 16–4
VIEW-AS phrase 17–2
default 17–2
default display formats 17–51
deleting dynamic 16–13
derealizing 16–29
describing static 16–7
DEFINE widget statement 16–10
frames and dialog boxes 16–10
VIEW-AS phrase 16–10
direct manipulation 24–1
dragging 24–8
dynamic 16–5
See also Dynamic widgets
editor 17–24
attributes 17–27
example 17–28
interactions 17–26
methods 17–27
options 17–24
save file portability 17–30
fill-in 17–2, 17–6
attributes 17–3
fonts used in Windows 17–5
format 17–49 to 17–57
character 17–51
date 17–58
decimal 17–53 to 17–56
integer 17–53 to 17–56
logical 17–57
grid alignment 24–17
handles 16–6
purpose 16–15
hierarchy 16–4
highlighting 24–6, 24–10
image 18–9 to 18–14
determining image file size 18–10
example 18–12
ignoring file at compile time 18–10
methods 18–11
options 18–9
instantiating static 16–8
instantiating, defined 16–6
methods 16–23
moving 24–8, 24–10, 24–16
overview 16–3
radio set 17–19
initial value 17–21
options 17–20
realizing 16–26
rectangle 18–15
EDGE-CHARS option 18–15, 18–16
EDGE-PIXELS option 18–15, 18–16
example 18–16
NO-FILL option 18–15
proportions in character mode 18–16
referencing dynamic 16–13
referencing static 16–11

resizing 24–9, 24–10, 24–16
selecting 24–6, 24–8
selection list 17–30 to 17–38
attributes 17–35
example 17–33, 17–36
methods 17–35
options 17–31
sizing 16–30
slider 17–9, 17–11
example 17–12
range 17–10
size 17–11
static 16–5
text 17–6
example procedure 17–7
toggle box 17–14
attributes 17–14
types 16–3
window 21–1 to 21–30
attributes 21–4
creating at run-time 21–8
events 21–8
maximizing 21–8
methods 21–5
minimizing 21–8
restoring 21–8
restrictions in character mode 21–8
working with static 16–10

WIDTH attribute 17–4

WIDTH option, browse widgets 10–14

WIDTH-CHARS attribute 17–11, 20–4

WIDTH-PIXELS attribute 17–11

Window families
defined 21–9
example 21–11, 21–29

Window size 25–5

Window widgets 21–1 to 21–30
attributes 21–4
creating at run-time 21–8
events 21–8
maximizing 21–8
methods 21–5
minimizing 21–8
restoring 21–8
restrictions in character mode 21–8

WINDOW-CLOSE event 16–33, 21–8

WINDOW-MAXIMIZED event 16–33, 21–8

WINDOW-MINIMIZED event 16–33, 21–8

WINDOW-RESTORED event 16–33, 21–8

{&WINDOW-SYSTEM} name reference 8–15

Windows
active 25–49
character simulated 25–3
clipboard output 7–6
color 23–9
Palette Manager 23–9, 23–10
compared to dialog boxes 21–3
current procedure 21–1
current session 21–1
default 21–1
directing I/O 21–16
families 21–9
multi-window pros and cons 21–2
multiple non-persistent 21–14, 21–15
multiple persistent 21–16
See also Multiple windows described 21–14
printer setup 7–4, 7–5
single, described 21–14
transactions 21–3
types of applications 21–14

Windows interface design options 25–52
Icon size 25–53

Progress support for Winhlp32.exe 25–53
ToolTips 25–52

WITH option, frame phrase 25–7

Word break tables 9–47
internationalization 9–47

Word indexing 9–47

Word wrap, editor widgets 17–26

Word-break, rules file
applying to database 9–56
compiling 9–55

Work tables 15–2
See also Temporary tables
based on database tables 15–19
characteristics 15–2
DEFINE WORK-TABLE statement 15–5, 15–8
differences from temporary tables 15–19
internal procedures 3–33
joins 9–41
number allowed in memory 15–2
similarities to temporary tables 15–19
using FIND FIRST statement to sort 15–10
using to collect data 15–7
using to create cross-tab reports 15–13
using to manipulate data 15–7
using to produce control-break reports 15–3
using to sort 15–9
using to write reports 15–3

WRITE trigger 11–3, 11–5

Writing records 3–17

X

X attribute 20–4

X, format character 17–52

Y

Y attribute 20–4

-y startup parameter, statistics A–14

-yd startup parameter, Segment Statistics A–14

Z

Z, format character 17–54

Z order 19–32