PROGRESS®

PROGRAMMING HANDBOOK

Copyright © 1990 Progress Software Corporation
617-275-4500

PROGRESS is copyrighted and all rights are reserved by Progress Software Corporation. This manual is copyrighted and all rights are reserved. This document 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 should not be construed as a commitment by Progress Software Corporation. Progress Software Corporation assumes no responsibility for any errors that may appear in this document.

PROGRESS® is a registered trademark of
Progress Software Corporation.

Printed in U.S.A.

December 1990

UNIX™ is a trademark of AT&T Bell Labs.
MS-DOS® and OS/2® are registered trademarks of Microsoft Corporation.
VMS™ is a trademark of Digital Equipment Corporation.
Convergent Technologies® and NGEN® are registered trademarks of Convergent Technologies.
Convergent™, Context Manager™, and CTOS™ are trademarks of Convergent Technologies.
BTOS™ and UNISYS™ are trademarks of Unisys Corporation.

CONTENTS

Preface .... **xvii**

THE ORGANIZATION OF THIS BOOK xvii

TYPOGRAPHICAL CONVENTIONS xix

THE SAMPLE PROCEDURES xix

OTHER USEFUL PUBLICATIONS xxii

Chapter 1: Introduction **1–1**

1.1 DATABASE FILES 1–2

1.1.1 Handling the Database Log File 1–4

1.2 DATA HANDLING STATEMENTS 1–4

1.3 PROGRESS MESSAGES 1–6

Chapter 2: The Keyboard, Editor, and Character Set **2–1**

2.1 USING PROGRESS KEYS 2–1

2.1.1 Continuation Lines in the Editor 2–9

2.2 KEY CODES AND KEY LABELS 2–9

2.3 DEFINING TERMINALS 2–18

2.3.1 Defining Your Terminal Type Under UNIX 2–19
2.3.2 Defining Your Terminal Type Under VMS 2–19
2.3.3 Defining Your Terminal Type Under BTOS/CTOS 2–20
2.3.4 Describing Terminal Characteristics 2–20
2.3.5 Creating a Protermcap Entry 2–21
2.3.6 Basic PROTERMCAP Entries 2–22
2.3.7 Additional PROTERMCAP Entries 2–25
2.3.8 UNIX stty Control Functions 2–30
2.3.9 Defining Special Keyboard Keys 2–31
2.3.10 Example PROTERMCAP Entry 2–33
2.3.11 Defining Video Display Attributes in PROTERMCAP 2–35
2.3.12 BTOS/CTOS Color Monitor Support 2–36

2.4 THE PROGRESS CHARACTER SET 2–36

2.4.1 Allowed Characters 2–37
2.4.2 Upper-Lower Case Conversions and Collating Sequence 2–37

2.5 EXTENDED ALPHABET SUPPORT 2–38

2.5.1 PROGRESS Language Syntax Extension 2–39
2.5.2 Terminal Definitions for non-English Languages 2–40
2.5.3 Using [NO-] MAP with PROGRESS INPUT and OUTPUT Statements 2–41
2.5.4 Extended Character Case Conversion and Collation 2–42
2.5.5 User-Defined Language Rules 2–44

Chapter 3: Designing and Changing Your Application Database 3–1

3.1 DESIGNING YOUR APPLICATION FILES AND FIELDS 3–1

3.1.1 Streamlining Your Data (Normalization) 3–4
3.1.2 Relating Files to One Another 3–5
3.1.3 Creating Database Files 3–5
3.1.4 Defining Large Records 3–11

3.2 INDEXING YOUR DATABASE 3–11

3.2.1 Why Define an Index? 3–11
3.2.2 Why Define an Index - Summary 3–14
3.2.3 Why Not Define an Index? 3–16
3.2.4 Maintaining Indexes 3–17
3.2.5 Indexes and Unknown Values 3–17
3.2.6 Indexes and Case-Sensitivity 3–17

3.3 CHANGING YOUR DATABASE DEFINITIONS 3–18

Table of Contents

3.3.1 Changing the Initial Value of a Field 3–20
3.3.2 Changing the Format of a Field 3–22
3.3.3 Changing the Data Type or Array Extent of a Field 3–23
3.3.4 Changing Index Definitions 3–24
3.3.5 Copying Data Definitions from One File to Another 3–24

3.4 DEFINING VALIDATION FOR FIELDS 3–26

3.4.1 Checking a Range of Values 3–27
3.4.2 Checking Against a List of Acceptable Values 3–27
3.4.3 Checking the Existence of a Related Record 3–28
3.4.4 Long Validation Expressions 3–29
3.4.5 Defining Validation Criteria Using the IF...THEN...ELSE Function 3–30
3.4.6 Notes on Defining Field Validation Criteria 3–31
3.4.7 Overriding Dictionary Validation Specifications 3–32

3.5 DEFINING FILE VALIDATION CRITERIA 3–33

3.6 DEFINING HELP FOR A FIELD 3–34

3.6.1 Overriding Dictionary Help Specifications 3–35

Chapter 4: Display Formats 4–1

4.1 DETERMINING THE DEFAULT DISPLAY FORMAT. 4–2

4.2 CHARACTER DISPLAY FORMATS 4–4

4.3 NUMERIC DISPLAY FORMATS 4–6

4.4 LOGICAL DISPLAY FORMATS 4–9

4.5 DATE DISPLAY FORMATS 4–9

4.6 TIME DISPLAY FORMATS 4–10

4.7 OVERRIDING DEFAULT DISPLAY FORMATS 4–12

4.8 TESTING DISPLAY FORMATS 4–12

Chapter 5: Block Properties 5–1

5.1 BLOCKS AND THEIR PROPERTIES 5–1

5.2 LOOPING 5-2

5.3 RECORD READING 5-3

5.4 FRAMES 5-4

5.4.1 Default Frames 5-4
5.4.2 Frame Scope 5-5
5.4.3 Frame Services 5-5

5.5 RECORD SCOPING 5-6

5.6 TRANSACTIONS 5-7

5.7 UNDO PROCESSING 5-8

Chapter 6: Monitoring andControlling Data Entry 6-1

6.1 USING KEYS WHILE RUNNING A PROCEDURE 6-2

6.2 CHANGING THE FUNCTION OF A KEY 6-3

6.2.1 Telling PROGRESS How to Continue Processing 6-5

6.3 MONITORING KEYSTROKES DURING DATA ENTRY 6-9

Chapter 7: Frame Design 7-1

7.1 THE PROGRESS SCREEN 7-1

7.2 HOW PROGRESS DESIGNS FRAMES 7-2

7.2.1 Frame-Level Design 7-4
7.2.2 Field- and Variable-Level Design 7-5

7.3 OVERRIDING DEFAULT FRAME DESIGNS 7-5

7.3.1 Format Phrases 7-6
7.3.2 Frame Phrases 7-8
7.3.3 The FORM Statement 7-10
7.3.4 Using Shared Frames 7-11
7.3.5 Using Strip Menus 7-15
7.3.6 Using Scrolling Frames 7-17

7.4 HOW STATEMENTS USE FRAMES 7-22

Table of Contents

7.5 PROGRESS FRAME SERVICES 7–24
7.5.1 Frame Scopes 7–24
7.5.2 Viewing and Hiding Frames 7–26
7.5.3 Advancing and Clearing Frames 7–26
7.5.4 Retaining Data During Retry of a Block 7–27
7.5.5 Repaint Optimization 7–27
7.6 FRAMES FOR NON-TERMINAL DEVICES 7–27
7.7 SPACE-TAKING AND NON-SPACE-TAKING TERMINALS 7–28

Chapter 8: Transactions and Error Processing 8–1

8.1 INTRODUCTION 8–1
8.2 TRANSACTIONS DEFINED 8–2
8.3 DOING ALL-OR-NOTHING PROCESSING 8–3
8.4 UNDERSTANDING WHERE TRANSACTIONS BEGIN AND END 8–7
8.4.1 The First Iteration of the Outer REPEAT Block 8–10
8.4.2 The Second Iteration of the Outer REPEAT Block 8–11
8.5 SPECIFYING HOW MUCH TO UNDO 8–13
8.6 CONTROLLING WHERE TRANSACTIONS BEGIN AND END 8–16
8.6.1 Making Transactions Larger 8–17
8.6.2 Making Transactions Smaller 8–20
8.7 USING TRANSACTIONS WITH SUBPROCEDURES 8–22
8.8 USING TRANSACTIONS WITH FILE INPUT 8–23
8.9 HANDLING OTHER ERRORS 8–23
8.9.1 How PROGRESS Handles Procedure Errors 8–24
8.9.2 How You Handle Procedure Errors 8–25
8.9.3 How PROGRESS Handles the Error Key 8–26
8.9.4 How You Handle the Error Key 8–29
8.9.5 How PROGRESS Handles The Endkey 8–29
8.9.6 How You Handle the Endkey 8–33
8.9.7 Rules About UNDO 8–33
8.9.8 How PROGRESS Handles the END-ERROR Key 8–34

8.10 HOW PROGRESS HANDLES SYSTEM AND SOFTWARE FAILURES 8-37

8.11 UNDERSTANDING HOW TRANSACTIONS AFFECT VARIABLES 8-37

8.12 DETERMINING WHEN TRANSACTIONS ARE ACTIVE 8-40

8.13 PROGRESS TRANSACTION MECHANICS 8-41

8.13.1 Transaction Mechanics 8-41
8.13.2 Subtransaction Mechanics 8-42

8.14 DOING EFFICIENT TRANSACTION PROCESSING 8-42

Chapter 9: Working With Input and Output in Your Procedures 9-1

9.1 UNDERSTANDING INPUT/OUTPUT 9-2

9.2 CHANGING A PROCEDURE'S OUTPUT DESTINATION 9-3

9.2.1 Using Multiple Output Destinations 9-3

9.3 CHANGING A PROCEDURE'S INPUT SOURCE 9-6

9.3.1 Using Multiple Input Sources 9-8
9.3.2 Preparing Input Files 9-10
9.3.3 Using Quoter 9-11
9.3.4 Other File Formats 9-15
9.3.5 Loading/Exporting DIF and SYLK Files 9-18
9.3.6 Loading DIF or SYLK Files 9-19
9.3.7 Exporting PROGRESS Database Files 9-20
9.3.8 Using Export 9-21
9.3.9 Using IMPORT 9-22

9.4 DEFINING ADDITIONAL STREAMS 9-23

9.5 SHARING STREAMS BETWEEN PROCEDURES 9-26

9.6 SUMMARY OF OPENING AND CLOSING STREAMS 9-27

9.7 PROCESSES AS INPUT AND OUTPUT STREAMS (UNIX) 9-29

9.8 TWO-WAY STREAMS: INPUT-OUTPUT THROUGH (UNIX) 9-29

9.9 I/O REDIRECTION FOR BATCH JOBS (UNIX and OS/2) 9-29

Chapter 10: Using PROGRESS Workfiles 10-1

10.1 WHAT ARE WORK FILES? 10-1

10.2 USING WORK FILES TO PRODUCE CATEGORIZED REPORTS 10-3

10.3 USING WORK FILES TO COLLECT AND MANIPULATE DATA 10-7

10.4 USING WORK FILES TO DO COMPLEX SORTING 10-8

10.5 USING WORK FILES TO DO "CROSS-TAB" REPORTS 10-12

Chapter 11: Providing Application Security 11-1

11.1 INTRODUCTION TO PROGRESS APPLICATION SECURITY 11-1

11.2 ESTABLISHING USERIDS AND PASSWORDS 11-2

11.2.1 Including Login Procedures in Your Application 11-3
11.2.2 Establishing a Userid for Batch Jobs 11-3

11.3 CHECKING USERIDS AT RUN-TIME 11-4

11.4 PERFORMING ACTIVITIES-BASED SECURITY CHECKING 11-6

11.4.1 Creating an Application Activity Permissions File 11-7
11.4.2 Adding Records to the Permissions File 11-8
11.4.3 Including Security Checking in Procedures 11-9
11.4.4 Protecting the Permissions File 11-12

11.5 USING FILE- AND FIELD-LEVEL SECURITY AT COMPILE TIME 11-14

11.5.1 File Permissions 11-18
11.5.2 Field Permissions 11-19
11.5.3 Example of Personnel Security 11-21

11.6 PERFORMING SECURITY ADMINISTRATION 11-22

11.6.1 Setting Up the _User File 11-23
11.6.2 Changing Your Password 11-27
11.6.3 Designating a Security Administrator 11-29
11.6.4 Determining the Privileges of the Blank Userid 11-30
11.6.5 Generating a Quick User Report 11-31
11.6.6 Freezing and Unfreezing Files 11-33

Chapter 12: Writing Multi-user Applications 12-1

12.1 THE MULTI-USER ENVIRONMENT 12-1

12.1.1 Starting a Multi-user Server 12-1
12.1.2 Starting Multi-User PROGRESS 12-3
12.1.3 Stopping a Multi-User Server 12-4

12.2 USING APPLICATIONS IN A MULTI-USER ENVIRONMENT 12-4

12.3 SHARING DATABASE RECORDS 12-6

12.3.1 Using Locks to Avoid Record Conflicts 12-6
12.3.2 How PROGRESS Applies Locks 12-9

12.4 HOW LONG DOES PROGRESS HOLD A LOCK? 12-10

12.5 RESOLVING LOCKING CONFLICTS 12-13

12.5.1 Changing the Size of Transactions 12-16
12.5.2 Making Transactions Larger 12-17
12.5.3 Making Transactions Smaller 12-18

12.6 AUTOMATICALLY GENERATING AN INDEX VALUE 12-19

12.6.1 Creating a System Control File 12-20
12.6.2 Writing the Procedure to Generate an Index Value 12-21

Chapter 13: Multi-database Programming 13-1

13.1 DATABASE CONNECTIONS 13-2

13.1.1 Connecting At Startup 13-4
13.1.2 Connecting From a Procedure with the CONNECT Statement 13-5
13.1.3 Connecting With the PROGRESS Data Dictionary 13-7
13.1.4 Connecting With Auto-Connect 13-8

13.2 GENERAL CONNECTION CONSIDERATIONS 13-10

13.2.1 Logical Database Names 13-10
13.2.2 Connection Modes 13-11
13.2.3 Database Types 13-13
13.2.4 Database Location 13-14

13.3 DISCONNECTING DATABASES 13-19

13.4 DEVELOPMENT CONNECTION CONSIDERATIONS 13-20

13.5 RUN-TIME CONNECTION CONSIDERATIONS 13-21

13.5.1 Creating Machine Independent Applications 13-22
13.5.2 Understanding and Managing Connection Failures and Disruptions 13-23
13.5.3 Conserving Database Connections
Versus Minimizing Connection Overhead 13-27

13.6 MULTI-DATABASE PROGRAMMING TECHNIQUES AND CONSIDERATIONS ... 13-28

13.6.1 Referencing Files And Field Names In Multiple Database Applications ... 13-28
13.6.2 Using Aliases ... 13-29

13.6.3 Using the Raw Datatype ... 13-34

13.6.4 PROGRESS Functions For Database Connection Information ... 13-37
13.6.5 Using the LIKE option ... 13-38
13.6.6 Understanding Transaction Behavior In Multiple Database Applications ... 13-38
13.6.7 Providing Help For Multi-database Applications ... 13-39
13.6.8 Implementing Run-time Security For Multi-database Applications ... 13-41

13.7 MULTI-DATABASE PROGRAMMING CASE STUDIES ... 13-45

13.7.1 Case Study 1 ... 13-47
13.7.2 Case Study 2 ... 13-49
13.7.3 Case Study 3 ... 13-52
13.7.4 Case Study 4 ... 13-54

Chapter 14: PROGRESS Programming Tips ... 14-1

14.1 DIFFERENCES BETWEEN OPERATING SYSTEMS RUNNING PROGRESS ... 14-1

14.2 WRITING TRANSPORTABLE APPLICATIONS ... 14-3

14.3 WRITING UNIX TRANSPORTABLE APPLICATIONS ... 14-5

14.4 WRITING VMS TRANSPORTABLE APPLICATIONS ... 14-6

14.5 TRANSPORTING A DATABASE ... 14-6

14.6 TESTING AND DEBUGGING ... 14-9

14.7 ACCESSING THE PROGRESS PROCEDURE LIBRARY ... 14-10

Chapter 15: PROGRESS/SQL ... 15-1

15.1 The PROGRESS/SQL Language ... 15-2

15.1.1 PROGRESS Extensions ... 15-3
15.1.2 Using Foreign Databases with SQL ... 15-3

15.2 PROGRESS/SQL LANGUAGE COMPONENTS 15-3

15.2.1 PROGRESS/SQL Reserved Words 15-3
15.2.2 SQL Statements 15-4
15.2.3 Null Values 15-7

15.3 SQL DATA MANIPULATION LANGUAGE 15-7

15.3.1 The SELECT Statement 15-7
15.3.2 Inserting Rows 15-16
15.3.3 Updating Rows 15-18
15.3.4 Deleting Rows 15-19
15.3.5 Working with Cursors 15-19
15.3.6 Cursor Statements 15-20
15.3.7 Selecting Single Rows (Singleton SELECT) 15-28
15.3.8 Transaction Processing 15-29
15.3.9 Privilege Checking at Compile-time and Runtime 15-29

15.4 DATA DEFINITION LANGUAGE 15-30

15.4.1 Tables 15-31
15.4.2 Data Types 15-34
15.4.3 Views 15-35
15.4.4 Indexes 15-39
15.4.5 Access Privileges 15-40
15.4.6 PROGRESS Schema Files 15-42

Chapter 16: PROGRESS and X Windows 16-1

16.1 INTRODUCTION 16-1

16.2 INSTALLATION CONSIDERATIONS 16-5

16.3 RUNNING PROGRESS IN X WINDOWS 16-5

16.3.1 Starting PROGRESS With Windows 16-6
16.3.2 Windows 16-13
16.3.3 Icons 16-13
16.3.4 Using A Mouse With PROGRESS 16-13

16.4 SUPPORTED CHARACTER SETS FOR THE X WINDOW ENVIRONMENT 16-17

Appendix A: Using Named Pipes A-1

A.1 AN INTRODUCTION TO NAMED PIPES (FIFOS) A-2

A.1.1 Creating a Named Pipe A–2
A.1.2 Deleting a Named Pipe A–3
A.1.3 Advantages and Disadvantages of Named Pipes A–5

A.2 NAMED PIPE EXAMPLES A–6

A.2.1 Example 1 – Creating and Using a Named Pipe at the Shell A–6
A.2.2 Example 2 – Using a Named Pipe with PROGRESS A–7

Index Index–1

FIGURES

Figure 1–1: PROGRESS Components 1–1
Figure 1–2: The Primary Data Handling Statements 1–6
Figure 2–1: Minimum protermcap Entry for DEC VT–100 2–23
Figure 2–2: Full protermcap Entry for DEC VT–100 2–33
Figure 7–1: Shared Frames 7–11
Figure 10–1: Processing a File for a Categorized Report 10–4
Figure 10–2: Using Work Files to Produce a Categorized Report 10–6
Figure 10–3: Maintaining Work Files in Sorted Order 10–10
Figure 10–4: Comparing Two Sorting Methods 10–11
Figure 13–1: A PROGRESS Multi-database Application 13–1
Figure 13–2: Connect A Database Window 13–7
Figure 13–3: Auto–Connect List Window 13–9
Figure 13–4: Federated Scenario 13–14
Figure 13–5: Distributed Scenario 13–16
Figure 13–6: Distributed/Simultaneous Networks Scenario 13–18
Figure 13–7: Machine Independent Applications 13–22
Figure 13–8: Positioning Database References 13–26
Figure 13–9: Case Study 2 Flow of Execution 13–50
Figure 13–10: Main Menu 13–51
Figure 15–1: Cursor Execution Sequence 15–22
Figure 16–1: Sample Motif Window 16–3
Figure 16–2: Sample OpenLook Window 16–4
Figure 16–3: PROGRESS and X Windows 16–5
Figure A–1: Typical Named Pipe Scenario A–1
Figure A–2: Writing Messages to a Named Pipe A–5
Figure A–3: Scenario Using Two Named Pipes A–10

Table of Contents

TABLES

Table 1–1: Data Handling Statements and Data Movement 1–5
Table 2–1: The PROGRESS Keyboard 2–2
Table 2–1: The PROGRESS Keyboard (continued) 2–3
Table 2–1: The PROGRESS Keyboard (continued) 2–4
Table 2–2: Key Codes and Key Labels 2–10
Table 2–2: Key Codes and Key Labels (continued) 2–12
Table 2–2: Key Codes and Key Labels (continued) 2–13
Table 2–2: Key Codes and Key Labels (continued) 2–14
Table 2–3: Alternate Key Labels 2–15
Table 2–4: BTOS/CTOS Key Labels 2–16
Table 2–4: BTOS/CTOS Key Labels (continued) 2–17
Table 2–5: Required PROTERMCAP Entries 2–23
Table 2–6: Additional PROTERMCAP Entries 2–25
Table 2–6: Additional PROTERMCAP Entries (continued) 2–26
Table 2–6: Additional PROTERMCAP Entries (continued) 2–27
Table 2–6: Additional PROTERMCAP Entries (continued) 2–28
Table 2–7: Variable-size Window Control Sequences 2–29
Table 2–8: Allowed Characters 2–37
Table 2–9: PROGRESS Extended Characters 2–39
Table 2–10: Special Character Substitutions Used in the PROGRESS Editor . .2–40
Table 2–11: Extended Character Case Conversion 2–43
Table 2–12: Special Collation Rules for Extended Characters 2–44
Table 2–13: Language Tables 2–45
Table 3–1: Reasons for Defining the Demo Database Indexes 3–15
Table 3–1: Reasons for Defining the Demo Database Indexes (continued) 3–16
Table 4–1: Default Display Formats for Expressions 4–2
Table 4–2: Default Formats for Data Types 4–2
Table 4–3: Character Display Format Examples 4–6
Table 4–4: Numeric Display Format Examples 4–8
Table 4–5: Logical Display Format Examples 4–9
Table 4–6: Date Display Format Examples 4–10
Table 4–7: Time Display Format Examples 4–11
Table 5–1: Block Properties 5–2
Table 5–2: Starting Transactions 5–8
Table 6–1: Keys You Can Use While Running Procedures 6–2
Table 6–2: Actions You Assign to Keys 6–5
Table 6–3: Key-Related Functions 6–7
Table 7–1: Format Phrase Options 7–6
Table 7–1: Format Phrase Options (continued) 7–7
Table 7–2: Frame Phrase Options 7–8

Table 7-2: Frame Phrase Options (continued) 7-9
Table 7-2: Frame Phrase Options (continued) 7-10
Table 8-1: Starting Transactions and Subtransactions 8-16
Table 9-1: Using Streams 9-28
Table 10-1: Database File and Work File Differences 10-2
Table 11-1: Access Restrictions for Files 11-19
Table 12-1: When PROGRESS Releases Record Locks 12-11
Table 13-1: Connection Parameters 13-3
Table 13-2: Connection Mode Parameters 13-12
Table 13-3: The Network (-N) Parameter 13-17
Table 13-4: Connection Failure Behavior 13-24
Table 13-5: PROGRESS Database Functions 13-37
Table 13-6: Default Userid Assignments For Run-time Connections 13-41
Table 13-7: Case Study Procedure Names 13-46
Table 15-1: PROGRESS/SQL and PROGRESS Terms 15-2
Table 15-2: PROGRESS/SQL Reserved Words 15-4
Table 15-3: Column, Table, and View Name Qualifiers 15-6
Table 15-4: Truth Values in Combined Comparison Expressions 15-7
Table 15-5: PROGRESS Record Locking Phrases 15-24
Table 15-6: SQL Statements and Associated Record Locks 15-24
Table 15-7: SQL Statements and Associated PROGRESS Privileges 15-41
Table 15-8: Fields in the _File Schema File 15-43
Table 15-9: Fields in the _Field Schema File 15-44
Table 15-7: Fields in the _Field Schema File (Continued) 15-45
Table 15-10: Fields in the _View Schema File 15-46
Table 15-11: Fields in the _View-Col Schema File 15-47
Table 15-12: Fields in the _View-Ref Schema File 15-47
Table 16-1: PROGRESS Startup Parameters for X Windows 16-6

Preface

This book explains advanced PROGRESS programming techniques and provides information about how to use your system with PROGRESS. Use this book if you have already read the *PROGRESS Language Tutorial*, and you are familiar with programming in PROGRESS.

In this book, the term PROGRESS refers to the PROGRESS 4GL/RDBMS (formerly Full PROGRESS). The PROGRESS Application Development System consists of the PROGRESS 4GL/RDBMS and the PROGRESS FAST TRACK Application Builder. For information about PROGRESS FAST TRACK, see the *PROGRESS FAST TRACK Tutorial* and the *PROGRESS FAST TRACK User's Guide.*

THE ORGANIZATION OF THIS BOOK

This book is organized as follows:

Chapter 1 — Introduction

Describes the basic components of PROGRESS, database files, data handling statements, and PROGRESS messages.

Chapter 2 — The Keyboard, Editor, Display Monitor, and Character Set

Explains how to handle keys, the editor, your terminal, and your particular character set while you use PROGRESS.

Chapter 3 — Designing and Changing Your Application Database

Explains advanced ways of working with database and data definitions, including how to change existing definitions. This chapter also covers designing help for your applications.

Chapter 4 — Display Formats

Describes PROGRESS display formats.

Chapter 5 — Block Properties

Explains looping, record reading, frames, record scoping, and transactions and undo processing.

Chapter 6 — Monitoring and Controlling Data Entry

Describes how to use and change the function of a key in a procedure, and how to monitor keystrokes during data entry.

Chapter 7 — Frame Design

Explains how PROGRESS designs frames and how you can override PROGRESS design defaults.

Chapter 8 — Transactions and Error Processing

Describes how to control and use transactions. This chapter also shows you how to control error processing.

Chapter 9 — Working With Input and Output in Your Procedures

Explains how to control input and output from procedures using streams, and by changing the input source or output destination of a procedure.

Chapter 10 — Using PROGRESS Work Files

Explains what work files are and how to use them.

Chapter 11 — Providing Application Security

Provides information on defining security for applications, including defining userids and passwords, security checking, and security administration.

Chapter 12 — Writing Multi-User Applications

Describes the issues involved in writing multi-user applications.

Chapter 13 — Multi-database Programming

Provides an introduction to multi-database programming with the PROGRESS 4th generation language (4GL).

Chapter 14 — PROGRESS Programming Tips

Provides tips about PROGRESS programming, including how to make your application transportable between different operating systems.

Chapter 15 — PROGRESS / SQL

Describes PROGRESS/Structured Query Language (SQL). PROGRESS/SQL is a relational database language based on the SQL86 database access standard of the American National Standards Institute (ANSI).

Chapter 16 — PROGRESS and X Windows

Explains how to use PROGRESS to develop applications that can run in an X Window environment and take advantage of many graphical user interface tools supplied in the X Window system.

TYPOGRAPHICAL CONVENTIONS

This document uses the following typographical conventions:

- **Bold typeface** indicates commands and characters you type. It also emphasizes important points.
- *Italic typeface* indicates a parameter or argument you supply. It also introduces new terms and manual titles.
- `Typewriter typeface` indicates system output and PROGRESS procedures. It also highlights file names, field names, command names, and menu options in text.

THE SAMPLE PROCEDURES

This book uses many sample procedures to show you how to use PROGRESS. All procedure examples appear in a shaded box with the name of the procedure in the upper right corner. For example:

p-demo1.p

```
FOR EACH customer:
   DISPLAY Cust-num name max-credit.
END.
```

The procedure name is the name of the file in which the procedure is stored. All procedure names start with a "p" for "programming handbook" and end with a ".p" for procedure.

In the sample procedures, words that are part of the PROGRESS language are in uppercase letters and words you supply are in lowercase letters. When you write procedures you can mix uppercase and lowercase in whatever way you want.

You can find all of the sample procedures for the Programming Handbook in the proguide subdirectory of the directory where you installed the PROGRESS software (by default, `\DLC\PROGUIDE` under DOS and OS/2, `[sys] <dlc>` under BTOS/CTOS, `/usr/dlc/proguide` under UNIX, and `[DLC.PROGUIDE]` under VMS). These procedures are stored in a "packed" format in a file named `PHPROC`. This packed format ensures that the many small procedures take up as little disk space as possible. You can "unpack" all of the Programming Handbook procedures into individual procedure files, or you can extract specific procedures one at a time.

To unpack these*procedures*, access the PROGRESS Procedure Library from the PROGRESS Help menu, as follows:

1. Press the HELP (F2) key to display the PROGRESS Help menu.
2. From the Help menu, select option **p**, Access the Procedure Library. The Main Menu of the PROGRESS Procedure Library appears on the screen.
3. From the Library Main Menu, select option **h**, Help, for an overview of the Library contents and instructions on how to use the Library.

From the Main Menu of the PROGRESS Procedure Library, you can also access the procedures used in the *PROGRESS Language Tutorial,* and the *PROGRESS Language Reference* by choosing option **u**, Unpack Utility.

4. When you are prompted for the name of the procedure to unpack, enter **all** and press RETURN. The procedure prompts you to choose the book whose procedures you want to unpack. Choose "Programming Handbook Procedures."

```
PROGRESS Unpack Utility

Please enter the procedure to be unpacked
 or enter "ALL" to unpack all procedures.
all

Reference Guide Procedures
Programming Handbook Procedures
Tutorial Procedures

Hit the space bar to scroll through the available choices.
Hit F1 to accept your choice, or F4c to abort.
(Prodemo procedures now accessible through PROGRESS
Procedure Library from the main help-menu.
Go back to the editor, press F2 and p.)
```

You are then asked to enter the name of the directory where you want the files to be written. If you enter blanks, they will be written to your working directory. If you unpack the procedures into the proguide directory, all users can easily access them.

- To unpack a *specific procedure*, when you are prompted for the name of the procedure to unpack, type the name of the procedure you want to use (e.g. `p-demo1.p`).

To put the procedure in your working directory, enter blanks when prompted for the name of the directory where you wish to place the procedure.

```
PROGRESS Unpack Utility

Please enter the procedure to be unpacked
 or enter "ALL" to unpack all procedures.
p-demo1.p

Please enter the directory where the file is to be written
________________________________
```

- All of the examples are based on the demo database. You make a working copy of this database by using the following commands:

Operating System	**To Copy the** demo **Database**
UNIX	`prodb` *database-name* `demo`
DOS and OS/2	`prodb` *database-name* `demo`
VMS	`PROGRESS/CREATE` *database-name* `demo`

BTOS/CTOS	`PROGRESS Create Database`
	`New Database Name` *database-name*
	`Copy From Database Name  demo`

OTHER USEFUL PUBLICATIONS

The following is a list of publications written by Progress Software Corporation which you may find useful:

PROGRESS Installation Notes

Contains step-by-step instructions for installing PROGRESS. Describes the prerequisites and procedures to get PROGRESS up and running on your machine.

PROGRESS Test Drive

Introduces new users to PROGRESS through a tutorial that describes a sporting goods distributor's inventory and order processing application.

PROGRESS Language Tutorial

Provides a "how-to" guide to PROGRESS fundamentals, designed for both novice and experienced programmers.

PROGRESS Language Reference

A detailed library of information on a number of PROGRESS topics. Provides descriptions and examples for each statement, function, phrase, and operator in the PROGRESS language.

System Administration I: Environments

Explains the DOS, OS/2, UNIX, VMS, and BTOS/CTOS concepts required to run PROGRESS, and provides information about running PROGRESS on networks.

System Administration II: General

Describes PROGRESS limits, disk and memory requirements, startup and shutdown procedures, backing up and restoring databases, and PROGRESS utilities. It also provides information about security administration, using multi-volume databases, and Roll Forward Recovery.

Pocket PROGRESS

Lets you quickly look up information about the PROGRESS language or programming environment. There is also a master index for the documentation in this quick reference.

Developer's Toolkit Manual

Explains how to use the PROGRESS Developer's Toolkit, a set of tools used to prepare PROGRESS applications for distribution.

Database Gateways

Provides information about how to use the PROGRESS 4th generation programming language on different relational database management systems other than PROGRESS RDBMS.

3GL Interface Guide

Supplies information about the PROGRESS Host Language Call (HLC), embedded SQL, and the Host Language Reference (HLI). This manual also contains information on how to use the PROBUILD utility.

Chapter 1
Introduction

PROGRESS is a relational database management system (RDBMS) and fourth-generation language that allows the application developer to build database applications for any kind of business need. As shown in Figure 1-1, PROGRESS consists of five components that allow you to create and maintain a database and its applications.

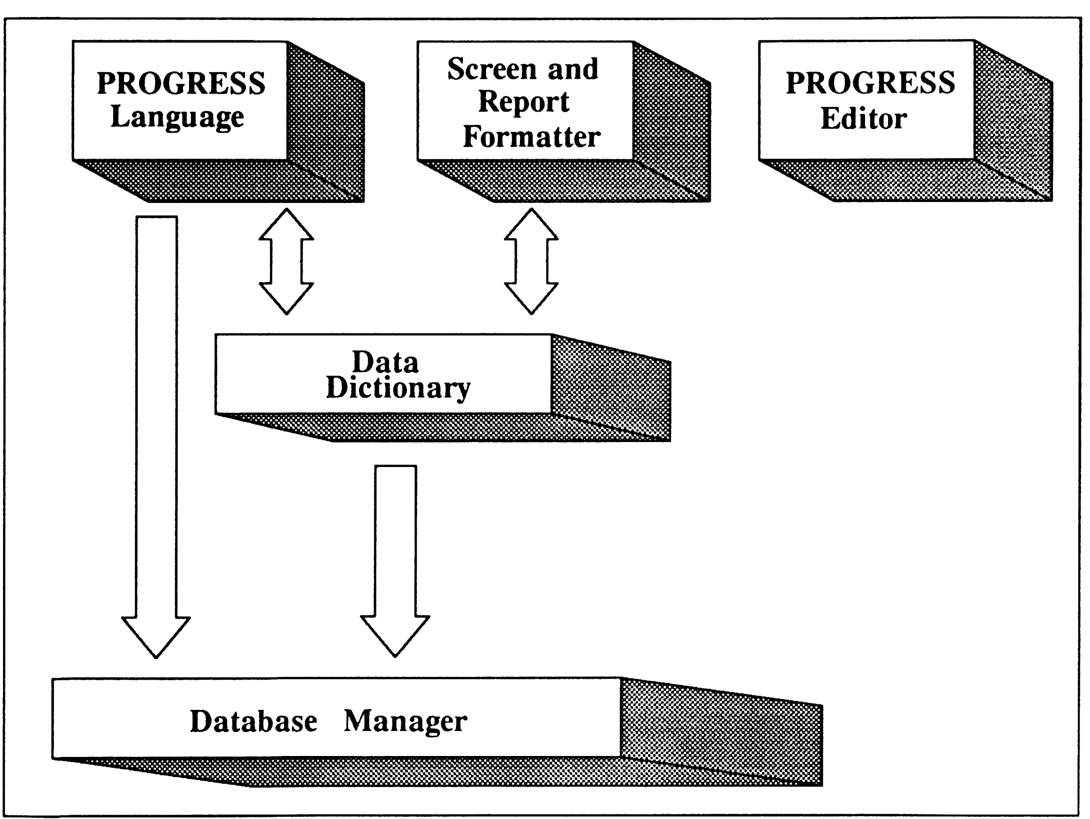


Figure 1-1: PROGRESS Components

Data Dictionary

The Data Dictionary is the PROGRESS component you use to manage the structure of a database. With the Data Dictionary, you can interactively create, modify, and delete database files, fields, and indexes. You can also use the Data Dictionary to dump and reload your database, define security measures, access the data exchange facility, and change your password.

Chapter 5 in the *PROGRESS Language Tutorial* provides introductory information about using the Data Dictionary to define files, fields, and indexes. Chapter 3 in this manual provides additional information about designing and changing a database. Refer to Chapter 11 in this manual and Chapter 5 in the *System Administration II: General* for the details of defining security measures.

PROGRESS Language

You use the PROGRESS programming language to write your application's procedures. This manual explains advanced features of the PROGRESS language. See the *PROGRESS Language Tutorial* for a hands-on approach to learning the PROGRESS language. Refer to the *PROGRESS Language Reference* for descriptions and examples of each element in the PROGRESS language.

PROGRESS Editor

The PROGRESS editor is the tool you use to enter and edit your procedures. The editor checks the syntax of your statements, and if it finds no errors, PROGRESS compiles the procedure. Once a procedure is compiled, you can run it from the editor. Chapter 2 in the *PROGRESS Language Tutorial* explains how to use the PROGRESS editor.

Screen and Report Formatter

The screen and report formatter provides default output formats that PROGRESS uses to organize the placement of elements on the screen or in printed reports. You can override the default formats in the Data Dictionary or within your applications. Chapter 11 in the *PROGRESS Language Tutorial* and Chapters 4 and 7 in this manual provide information about display formats.

Database Manager

The database manager handles the database interactions that you generate through the use of the Data Dictionary and your applications.

1.1 DATABASE FILES

A PROGRESS database is a single DOS, OS/2, UNIX, VMS or BTOS/CTOS file (with a .db file extension) containing all the files and indexes for an application. You access one database during a PROGRESS session.

PROGRESS uses the database file and the following related files:

- The before-image file for a database has a .bi extension. The before-image file ensures the integrity of your database by keeping track of the status of the database in the event of a system failure or an incomplete transaction.
- The lock control file for a database has a .lk extension (some systems don't have this file).
- The multi-user directory has a .ld extension. This is used only with UNIX.
- The file containing a log of status messages and database usage has a .lg extension.
- On VMS systems, the server has a separate log file called *database-name*sv.lg which contains messages about server activity.
- On VMS systems, there is a file with the extension .shm, which is used for shared memory when a server is running. The file is large and you can remove it when you are not running the server if you can afford the time to create it when you next start the server.

For example, the file names related to a database named "demo" are:

- The database file: demo.db
- The before-image file: demo.bi
- The lock file: demo.lk
- The multi-user directory: demo.ld
- The log file: demo.lg
- VMS server log file: demosv.lg
- VMS shared memory file: demo.shm

When you exit normally from PROGRESS, PROGRESS deletes the lock file and multi-user directory (if used), because they are necessary only during a PROGRESS session.

PROGRESS also deletes some other temporary database files it creates during your PROGRESS session. Under DOS, .pge, .trp, .lbi, and .srt are the extensions of these temporary files. If your system crashes and you subsequently do a DOS CHKDSK command, DOS may ask you whether to convert lost clusters or lost chains to files. Under OS/2, pg, sr, and lb are the prefixes of these temporary files. If your system crashes and you subsequently do an OS/2 CHKDSK command, OS/2 may ask you whether to convert lost clusters or lost chains to files. In the case of these temporary files, the lost data need not be saved and does not need to be converted to files.

On UNIX, VMS, or BTOS/CTOS, you must specify the Save Temporary Files (-t) option to view temporary files. Otherwise, they are created "unlinked" and deleted before you exit PROGRESS. If your system crashes, however, the file system recovery program, fsck, recovers your temporary files and may prompt you to delete them. If it does, delete the files.

The database lock file is a "flag" file that ensures that two users, both running PROGRESS in single-user mode, are not using the same database. The lock file is used on operating systems that do not support file locking. In any case, PROGRESS applications always handle locking at the record level.

If there is a system failure while PROGRESS is running and you try to enter PROGRESS again, you may get a message indicating that a lock file exists. If you are certain that the database is not in use, then erase the lock file. If you erase the lock file while the database is in use and then start PROGRESS, you will damage your database and will have to restore it from a backup. It is important that you do not remove any of the `.db` or `.bi` database files.

1.1.1 Handling the Database Log File

The database log (`.lg`) file expands as you use the database. If the log file begins to get too large, you can reduce its size by removing old log entries. To remove log entries from a `.lg` file, type the following commands in response to the operating system prompt:

SYNTAX (UNIX , DOS, and OS/2)

```
prolog database-name
```

In this command, *database-name* is the name of the database whose log file you want to reduce. The `prolog` command removes all but the most recent entries in the log file.

On VMS and BTOS/CTOS systems, the database log file is locked whenever the database is active and you are not able to read it.

On BTOS/CTOS systems, the log maintenance command removes all but the most recent entries from the PROGRESS log file. To enter the command, type `PROGRESS Log Maintenance`. This invokes the following form:

SYNTAX (BTOS/CTOS)

```
PROGRESS Log Maintenance
    Database Name
```

In this form, enter the database filename. This tells the log maintenance utility to truncate the log associated with the specified database. On BTOS/CTOS systems, if the `.lg` file is small and the resulting file is not trimmed, the command returns an error message.

1.2 DATA HANDLING STATEMENTS

When a procedure works with data, the statements in the procedure are moving the data from one location to another. The statements that move data are called *data handling statements*.

In PROGRESS, data can be in any of three places: the database, a record buffer, or a screen buffer. The database is where your permanent application data is stored. Record buffers are used to hold data while a procedure is using it. PROGRESS also uses a special record buffer to hold the values of any variables used in a procedure. Screen buffers hold data being displayed on the screen or being sent to another output destination, and data that is being entered from the terminal or another input source.

Table 1-1 shows how each of the data handling statements moves data.

Table 1-1: Data Handling Statements and Data Movement

	Database Record	Record Buffer	Screen Buffer	User
ASSIGN		◄	●	
CREATE	◄	●		
DELETE	◄	●		
DISPLAY		●	►	
FIND	●	►		
FOR EACH	●	►		
INSERT	 ◄	● ◄ ●	► ●◄	 ●
PROMPT-FOR			◄	●
RELEASE	◄	●		
SET		◄	●◄	●
UPDATE		● ◄	► ●◄	 ●

To use this chart, find the statement you are interested in. The black dot indicates where the statement gets data. The arrowhead indicates where the statement stores data. For example, the DISPLAY statement takes data from a record buffer, moves it into a screen buffer and displays the data on the screen.

This chart illustrates that some statements seem to do parts of the jobs of other statements. For example, the UPDATE statement does three steps: it moves data from the record buffer to the screen buffer, allows data to be entered into the screen buffer, and then moves it back to the record buffer. The SET statement does the second and third of these steps.

In fact, some statements are made up of combinations of other statements. You can use statements as building blocks, using only as many as you need to do specific kinds of work. For example, the INSERT statement is very powerful and combines several processing steps into one statement. But in some situations this is less flexible than using the CREATE, DISPLAY, and SET statements individually.

The following figure shows all the primary data handling statements and illustrates which statements make up other statements.

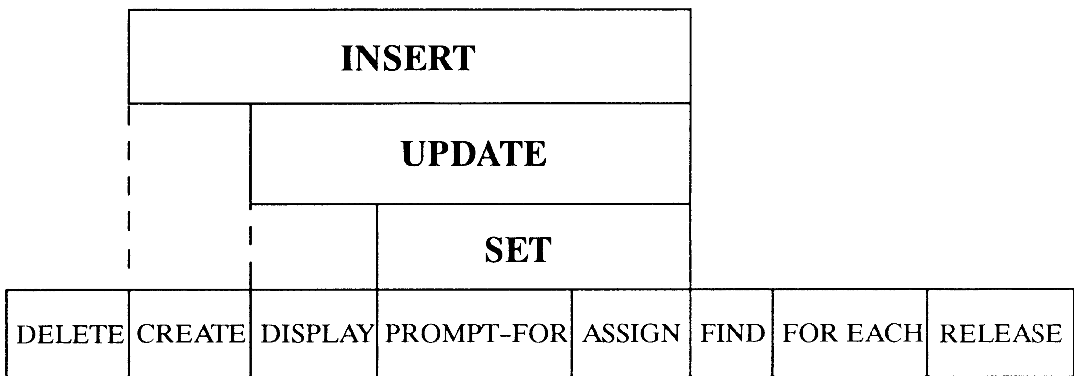


Figure 1–2: The Primary Data Handling Statements

For more information about each of these statements, see the *PROGRESS Language Reference* manual.

1.3 PROGRESS MESSAGES

PROGRESS and applications written under PROGRESS display several types of messages to inform you of both routine and unusual occurrences:

1. Execution messages inform you of errors encountered while PROGRESS is running a procedure (e.g., a record with a specified index field value could not be found).

2. Compile messages inform you of errors found while PROGRESS is reading and analyzing a procedure prior to running it (e.g., a reference to a file name that is not defined in the database).

3. Startup messages inform you of unusual conditions detected while PROGRESS is getting ready to execute (e.g., an invalid startup parameter was entered).

After it displays a message, PROGRESS proceeds in one of several ways:

1. 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.

2. Returns to the editor so that you can correct an error in a procedure. This is the usual action taken following compiler messages.

3. Halts processing of a procedure and returns immediately to the editor. This does not happen often.

4. Terminates the current session.

Use PROGRESS Help if you do not understand a message you see on your system. You access Help by pressing the HELP (F2) key when you are in the PROGRESS editor. 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. You can use the Help screens for explanations of any PROGRESS message.

Chapter 2
The Keyboard, Editor, and Character Set

This chapter covers the following topics:

- Using PROGRESS keys.
- Key codes and key labels.
- Defining Terminals.
- The PROGRESS character set.
- Extended alphabet support.

2.1 USING PROGRESS KEYS

There are special PROGRESS keys you can use when you are in the PROGRESS editor writing procedures and when you are running a PROGRESS procedure.

There are three ways you can use special PROGRESS keys:

1. Press a single key on the terminal keyboard. For example, most terminals have TAB.
2. Press CTRL or CODE and another key at the same time. CTRL-X, for example, usually performs the same function as GO.
3. Press a function key. Some terminals have numbered F or PF keys. Some have both F and PF keys, and some have neither.

In this manual and in the *PROGRESS Language Tutorial*, whenever you need to press a key, you see the name of the key followed by the label of the key as it probably appears on your keyboard (some keyboards may be slightly different). For example, we tell you to press GO (F1). This means press the key labeled F1 on your keyboard.

Table 2-1 lists the standard CTRL keys and function keys for each of the PROGRESS keys.

Table 2-1: The PROGRESS Keyboard

Key Function	Editor	Procedure Execution	Standard Keyboard Key	Standard Control Key			Standard Function Key			Allowed in ON
				DOS OS/2	UNIX VMS	BTOS/ CTOS	DOS OS/2	UNIX VMS	BTOS/ CTOS	
ABORT	✓	✓		Ctrl-Alt -Del	**	ACTION-/				
APPEND-LINE	✓			Ctrl-A	Ctrl-A	CODE-A	ALT-F2 F11 (OS2)	F12	SHIFT-F12	
BACK SPACE	✓	✓	BACK SPACE							✓
BACK-TAB	✓		SHIFT-TAB (DOS & OS/2)	Ctrl-U	Ctrl-U	CODE-U			CODE-TAB	✓
BELL										✓
BLOCK	✓			Ctrl-V	Ctrl-V	CODE-V	ALT-F4	F14	SHIFT-F4	
BOTTOM-COLUMN *						CODE-↓	ALT-B	ESC,↓ ESC,B		✓
BREAK-LINE	✓			Ctrl-B	Ctrl-B	CODE-B	ALT-F1 F11 (OS2)	F11	SHIFT-F1	
BTOS/CTOS-END		✓	EXIT EXEC(BTOS) or FINISH EXEC(CTOS) PROGRESS EXIT (PROGRESS Utility to quit CONTEXT with PAUSE.)							
CANCEL-PICK *							ALT-X	ESC,X	CODE-CANCEL	✓
CHOICES *						CODE-C	ALT-C	ESC,C		✓
CLEAR	✓	✓		Ctrl-Z	Ctrl-Z	CODE-Z	F8	F8	F8	✓
CURSOR-UP	✓	✓	↑	Ctrl-K	Ctrl-K					✓
CURSOR-DOWN	✓	✓	↓	Ctrl-J	Ctrl-J					✓
CURSOR-LEFT	✓	✓	←							✓
CURSOR-RIGHT	✓	✓	→	Ctrl-L	Ctrl-L					✓
DELETE-CHARACTER	✓	✓	DEL DELETE [1]							✓
DELETE-COLUMN *						CODE-SHIFT-D	ALT-Z	ESC,Z		✓
DELETE-FIELD *						CODE-J	ALT-D	ESC,D		✓

* See paragraph below for explanation

** UNIX: \ (Depends on UNIX stty setting for quit)
VMS: Ctrl-Y, STOP

1. BTOS/CTOS

(continued on next page)

Table 2–1: The PROGRESS Keyboard (continued)

Key Function	Editor	Procedure Execution	Standard Keyboard Key	Standard Control Key			Standard Function Key			Allowed in ON
				DOS OS/2	UNIX VMS	BTOS/ CTOS	DOS OS/2	UNIX VMS	BTOS/ CTOS	
DELETE-LINE	✓			Ctrl-D	Ctrl-D	CODE-D	F10	F10	F10	
DOS-END		✓		Type EXIT						
ENDKEY		✓								✓
END-ERROR	✓	✓	ESC (DOS) [1 2] CANCEL or FINISH	Ctrl-E	Ctrl-E	CODE-E	F4	F4	F4	✓
ERROR		✓								✓
FIND	✓			Ctrl-F	Ctrl-F	CODE-F	ALT-F3	F13	SHIFT-F3	
GET	✓			Ctrl-G	Ctrl-G	CODE-G	F5	F5	F5	
GO	✓	✓		Ctrl-X	Ctrl-X	CODE-X	F1	F1	F1	✓
GOTO LINE *						CODE-SHIFT-G	ALT-G	ESC,G		✓
HELP	✓	✓ ***	HELP	Ctrl-W	Ctrl-W	CODE-W	F2	F2	F2	✓
HOME	✓	✓	HOME					ESC,H	CODE-NEXT-PAGE	✓
INSERT-COLUMN *						CODE-SHIFT-C	ALT-N	ESC,N		✓
INSERT-FIELD *						CODE-I	ALT-I	ESC,I		✓
INSERT-FIELD-DATA *						CODE-SHIFT-L	ALT-F	ESC,F		✓
INSERT-FIELD-LABEL *							ALT-E	ESC,E		✓
INSERT-MODE	✓	✓	INSERT OVERTYPE[1]	Ctrl-T	Ctrl-T	CODE-T	F3	F3	F3	✓
LEFT-END	✓		CTRL- ← (DOS & OS/2) ESC - ← (UNIX & VMS)						CODE- ←	✓
MAIN-MENU *						CODE-M	ALT-M	ESC,M		✓
MOVE *						MOVE	ALT-V	ESC,V		✓
NEW-LINE	✓			Ctrl-N	Ctrl-N	CODE-N	F9	F9	F9	
OS/2 END		✓		Type EXIT						
PAGE-DOWN	✓		NEXT PAGE or PG DN				ALT-F6	16	SHIFT-F6	
PAGE-UP	✓		PREV PAGE or PG UP				ALT-F5	15	SHIFT-F5	
PICK *						CODE-K	ALT-P	ESC,P		✓

* See paragraph below chart for explanation

*** Allowed only if help procedure, `applhelp.p`, exists.

(continued on next page)

1. BTOS/CTOS
2. OS/2

Table 2-1: The PROGRESS Keyboard (continued)

Key Function	Editor	Procedure Execution	Standard Keyboard Key	Standard Control Key			Standard Function Key			Allowed in ON
				DOS OS/2	UNIX VMS	BTOS/ CTOS	DOS OS/2	UNIX VMS	BTOS CTOS	
PICK-AREA *						CODE-SHIFT-A	ALT-W	ESC,W		✓
PICK-LABEL-DATA *						CODE-Q	ALT-Q	ESC,Q		✓
PUT	✓			Ctrl-P	Ctrl-P	CODE-P	F6	F6	F6	
RECALL	✓	✓		Ctrl-R	Ctrl-R	CODE-R	F7	F7	F7	✓
REPAINT	✓						ALT-P	ESC,P		✓
REPORTS *						CODE-SHIFT-T	ALT-A	ESC,A		✓
RESUME DISPLAY	✓	✓		Ctrl-Q	Ctrl-Q					
RETURN	✓	✓	RETURN or NEXT	Ctrl-M	Ctrl-M					✓
RIGHT-END	✓		CTRL-→ (DOS & OS/2) ESC - → (UNIX & VMS)	Ctrl-E	Ctrl-E		F4	F4	CODE-→	✓
SCROLL-LEFT *							ALT-L	ESC,L	SHIFT-←	✓
SCROLL-RIGHT *							ALT-R	ESC,R	SHIFT-→	✓
SEARCH	✓					CODE-F	ALT-F	ESC,F		
SETTINGS *						CODE-S	ALT-S	ESC,S		✓
STOP		✓		BREAK	**** Ctrl-C	ACTION-CANCEL				✓
STOP-DISPLAY	✓	✓		Ctrl-S	Ctrl-S					
TAB	✓	✓	TAB	Ctrl-I	Ctrl-I					✓
TOP-COLUMN *							ALT-T	ESC, CURSOR UP ESC,T		✓
UNIX-END		✓			***** Ctrl-D					
VMS-END		✓			Type logout					

* See paragraph below for explanation
**** Depends on the UNIX stty setting for intr
***** Depends on the UNIX stty setting for eof

*These key functions do not have automatic actions associated with them when you use them in the editor or while running a procedure. However, these key functions are available for use with the ON statement together with the KEYFUNCTION and LASTKEY functions.

For example, one of the special key functions marked with an asterisk in the table above is CHOICES. The following statements check to see if the user pressed the CHOICES key. The example defines the F2 key as the CHOICES key. If the user presses F2, the procedure might then display a list of available choices.

```
ON F2 CHOICES.

READKEY.
IF KEYFUNCTION(LASTKEY) = "choices"
THEN DO:
              . . .
```

The following list describes the function of most of the keys listed in Table 2–1. Keys that you use while in the editor are followed by the word "Editor"; keys that you use while running procedures are followed by the word "Execution". Keys marked with a single asterisk (*) in the table are not described here because they do not have automatic actions associated with them.

- ABORT stops the current PROGRESS session. On DOS and OS/2, reboots the computer. On UNIX, BTOS/CTOS, or VMS, returns you to the operating system prompt. (Editor, Execution)
- APPEND LINE combines the line the cursor is on with the next line. (Editor)
- BACKSPACE deletes the character to the left of the cursor position. On some terminals (e.g., WYSE 50, WYSE 350), the backspace key sends the same code as the left arrow key. On such terminals the backspace key will not do a destructive backspace in PROGRESS, but will move the cursor left one position. (Editor, Execution)
- BACKTAB tabs backward to different locations. In the editor, tab stops are every four columns: 1, 5, 9, 13, and so on. During execution, each field on a frame is a tab position.
- BELL. This is not a key, but you can list it as an action in an ON statement.
- BLOCK marks the beginning of a block of text. You use the block key to move text around in a file. (Editor)

  To copy a block of lines from one part of a file to another part of a file, put the cursor at either the beginning or end of those lines and press BLOCK,(F14), or Ctrl-V . Move the cursor to the other end of the block. Press PUT (F6) to save the block. PROGRESS prompts you for the name of the file in which you want to store the lines. You can supply a file name or blanks. If you give a file name, the block of text is placed in that file. If you enter a blank name, the text is placed in a special save area called the "temporary file", replacing the previous contents of that file. Press RETURN or GO (F1). To copy the block in a new location in the file, move the cursor to the line above where you want to insert the text, press GET (F5), supply a file name or blanks, and press RETURN or GO (F1).

  To cut a block of lines from one part of a file and paste that block somewhere else in the file, put the cursor at either the beginning or end of those lines and press BLOCK,(F14), or Ctrl-V . Move the cursor to the other end of the block. Press DELETE LINE ,(F10), or Ctrl-D .

2 Keyboard and Editor

PROGRESS removes the lines and stores them in the temporary file. Move the cursor to the line above where you want to insert the lines, press GET (F5), supply a blank file name, and press RETURN or GO (F1). PROGRESS copies the block of lines back into the procedure.

- BREAK LINE breaks the current line into two lines beginning with the character the cursor is on. The cursor remains on the first line. (Editor)
- BTOS/CTOS END Typing EXIT EXECUTIVE (BTOS) or FINISH EXECUTIVE (CTOS) and PROGRESS EXIT quits the program.
- CLEAR clears the editing area or the current field. For character fields, the clearing takes place from the cursor position to the end of the field. If you clear the editing area and you have not saved your work, PROGRESS displays this message: "Discard your changes? (y/n)". If you enter no, PROGRESS returns you to the editing area so you can save your work. If you enter yes, PROGRESS clears the editing area and your work is not saved. (Editor, Execution)
- CURSOR UP, CURSOR DOWN, CURSOR RIGHT, and CURSOR LEFT move the cursor. (Editor, Execution)
- DELETE CHARACTER deletes the character at the cursor position. (Editor, Execution)
- DELETE LINE deletes the line the cursor is located on. (Editor)
- DOS END Typing EXIT at the DOS prompt returns you to PROGRESS after you have used the PROGRESS DOS statement to run DOS commands interactively. (Execution)
- OS/2 END Typing EXIT at the OS/2 prompt returns you to PROGRESS after you have used the PROGRESS OS2 statement to run OS/2 commands interactively. (Execution)
- ENDKEY always performs the ON ENDKEY processing for the current block. (See the ON ENDKEY phrase in the *PROGRESS Language Reference* manual for more information.) (Execution)
- END-ERROR on the first interaction during a block iteration, does the ENDKEY processing. On subsequent interactions, does the ERROR processing. (See the ON ENDKEY phrase and ON ERROR phrase in the *PROGRESS Language Reference* manual for more information.) (Execution)

  This key also moves the cursor to the right end of a line when you are in the PROGRESS editor. (Editor)
- ERROR always performs the ON ERROR processing for the current block. (See the ON ERROR phrase in the *PROGRESS Language Reference* manual for more information.) (Execution)

- FIND finds a string of characters. When you press FIND, PROGRESS prompts you for the string you want to search for. The search is case insensitive; the string you supply can be in upper or lowercase or both. PROGRESS finds matching strings regardless of case. Press the GO key to start the search. To repeat the search, press FIND and GO again. The search automatically wraps from the end of the file to the beginning of the file. (Editor)

- GET retrieves a PROGRESS procedure and puts it into the edit buffer. When you press GET, PROGRESS prompts you for the name of the file you want to retrieve. PROGRESS looks for the file in the directories listed in the PROPATH environment variable. If you do not name a file, PROGRESS retrieves the contents of the unnamed buffer. If the edit area is not clear, then the lines are inserted after the line the cursor is on. (Editor)

- GO compiles and runs the statements in the editing area. While a procedure is running, PROGRESS is in "execution mode" and the editing keys no longer apply. During execution, the GO key indicates that all data has been entered and that processing should resume. (Editor, Execution)

- GOTO LINE moves the cursor to a specified line in the current procedure in the edit buffer. When you press this key, PROGRESS prompts you to enter a line number. Enter a line number and press the GO key to move to the designated line. If you enter a line number larger than the number of lines in the current procedure, the cursor moves to the last line in the procedure. (Editor)

- HELP from the editor provides you with on-line information about different parts of PROGRESS. HELP while an application is running runs an application-specific procedure named `applhelp.p`. (Editor, Execution)

- HOME moves alternately between the first and last lines of the text in the editor. (Editor)

- INSERT MODE changes from insert mode to overtype mode or vice versa. When you are in insert mode, everything you type is inserted into the line at the cursor position. When you are in overtype mode, the character at the cursor is replaced by the one you type. If you are in insert mode in the editor and you fill up the current line, you must press BREAK LINE before you can add any more characters. (Editor, Execution)

- LEFT END moves the cursor to the left end of the current line on DOS & OS/2 systems. (Editor)

- NEW LINE inserts a blank line below the line the cursor is on and positions the cursor on that line. (Editor)

- PAGE DOWN moves down one page (approximately 20 lines) in the file you are editing. (Editor)

- PAGE UP moves up one page in the file you are editing. (Editor)
- PUT stores the contents of the editing area in a file. When you press PUT, PROGRESS prompts you for the name of the file where you want to store the edit buffer text. If you do not name a file, PROGRESS stores the contents of the edit buffer in the unnamed buffer. You can retrieve the contents of that buffer by pressing GET and entering a blank file name when PROGRESS prompts you for one. (Editor)

  Unless you specify a pathname when you name the file, PROGRESS stores the file in your current working directory.
- RECALL in the editor recalls into the edit area the last PROGRESS procedure you ran, omitting any blank lines that were at the end of that procedure when it was in the edit buffer. If you press RECALL while running a procedure, PROGRESS restores the frame field you are entering to its value at the beginning of the statement.
- REPAINT refreshes the information currently displayed in the edit buffer. (Editor)
- RESUME DISPLAY restarts a display that was stopped by pressing STOP DISPLAY. (Editor, Execution)
- RETURN moves the cursor to the next line (editor) or to the next field (execution). on the last field being entered is treated as GO. (Editor, Execution)
- RIGHT END moves the cursor to the right end of the current line in the PROGRESS editor. (Editor)
- SEARCH performs a search and replace operation on a procedure in the PROGRESS editor. When you press this key, PROGRESS prompts you to enter the string for which you are searching. The search is case insensitive; the string you supply can be in upper- or lowercase or both. You cannot use wildcard characters when specifying the search and replace strings. After you enter the string and press RETURN or GO (F1), PROGRESS prompts you to enter a replacement string. After you finish entering the replacement string and press RETURN or GO (F1), PROGRESS finds the first occurrence of the string in the file below the current cursor location.

  When a match is found for the search string, PROGRESS prompts you to enter Y, N, or G to designate the type of replacement operation. If you enter G (Global), PROGRESS replaces all occurrences of the search string with the replacement string in the current procedure. If you enter Y (Yes), the current occurrence of the search string is replaced with the replacement string and PROGRESS finds the next occurrence of the search string. If you enter N (No), the current occurrence of the search string is not replaced and PROGRESS finds the next occurrence of the search string. The search automatically wraps from the end of the file to the beginning of the file. (Editor)

- STOP stops processing a procedure, backs out the active transaction, and returns to the start-up procedure or to the PROGRESS editor. (Execution)
- STOP DISPLAY temporarily stops the display of information on the screen. (Editor, Execution)
- TAB tabs to different locations. In the editor, tab stops are every four columns: 1, 5, 9, 13 and so on. During execution, each field on a frame is a tab position. (Editor, Execution)
- UNIX END Pressing Ctrl-D at the UNIX prompt returns you to PROGRESS after you have used the PROGRESS UNIX statement to run UNIX commands interactively. (Execution)
- VMS END Typing LOGOUT at the VMS prompt returns you to PROGRESS after you have used the PROGRESS VMS statement to run VMS commands interactively. (Execution)

2.1.1 Continuation Lines in the Editor

The basic line width of the PROGRESS editor is 80 characters. However, you may occasionally want to edit files with lines longer than 80 characters or create files with long lines.

A tilde (~) as the last non-blank character on a line indicates that the line is continued beginning with column 1 of the next line on the screen. For example, these two lines in the editor:

```
DISPLAY "This is a long message ~
continued on the next line".
```

are written out to a file as this single line:

```
DISPLAY "This is a long message continued on the next line".
```

If you get a file into the editor that contains lines that are longer than 80 characters, PROGRESS splits the lines into 79 character segments and puts a tilde in column 80.

2.2 KEY CODES AND KEY LABELS

All of the PROGRESS keys have both a code and a label. The key code is a PROGRESS internal identifier for the key. You can use the KEYCODE function to determine the key code for a particular key. For example, the function

```
KEYCODE("del")
```

returns the value 127, which is the integer code assigned to the DEL key. You use the KEYLABEL function to determine the keyboard label for a particular key code. For example, the function

```
KEYLABEL(7).
```

returns the value CTRL-G. The following table lists the key codes and key labels for all the keys you can use with PROGRESS. It also indicates whether you can use each of those keys in an ON statement or with the GO-ON option.

Table 2-2: Key Codes and Key Labels

Key Code	Key Label	Allowed in ON Statement or GO-ON Phrase			
		DOS & OS/2	UNIX	VMS	BTOS/CTOS
0	CTRL-@	✓	✓ (1)	✓	
1 - 7	CTRL-A through CTRL-G	✓	✓ (1)	✓	
8	BACKSPACE	✓	✓ (1)	✓	✓
9	TAB	✓	✓ (1)	✓	✓
10 - 12	CTRL-J through CTRL-L	✓	✓ (1)	✓	
13	ENTER (DOS) RETURN (UNIX, BTOS/CTOS)	✓ ✓	✓ (1) ✓ (1)	✓ ✓	 ✓
14 - 26	CTRL-N through CTRL-Z	✓	✓ (1)	✓	
27	ESC	✓	✓ (1)	✓	
28	CTRL-\	✓	✓ (1)	✓	
29	CTRL-]	✓	✓ (1)	✓	
30	CTRL-^	✓	✓ (1)	✓	
31	CTRL-_	✓	✓ (1)	✓	
32 - 126	Corresponding extended ASCII character		✓ (1)	✓	
127	DEL	✓	✓ (1)	✓	
128 - 255	Corresponding extended ASCII character				
256 - 299	null string				
301 - 310	F1 - F10	✓	✓ (2)	✓(6)	✓
311 - 320	F11 - F20	✓ (3)	✓ (2)	✓	
321 - 330	F21 - F30	✓ (4)	✓ (2)	✓	
331 - 340	F31 - F40	✓ (5)	✓ (2)	✓	
341 - 399	F41 - F99		✓ (2)	✓	
400 - 499	PF0 - PF99		✓ (2)	✓	
501	CURSOR-UP	✓	✓ (2)	✓	✓
502	CURSOR-DOWN	✓	✓ (2)	✓	✓

(1) - Unless pre-empted by UNIX stty settings for intr, quit, stop display, resume display.
(2) - Only if key is defined in protermcap file.
(3) - Corresponds to Alt-F1 through Alt-F10 on the DOS keyboard.
(4) - Corresponds to Shift-F1 through Shift F-10 on the DOS keyboard.
(5) - Corresponds to Ctrl-F1 through Ctrl-F10 on the DOS keyboard.
(6) - Not allowed for vt100 terminals.

(continued on next page)

Table 2–2: Key Codes and Key Labels (continued)

Key Code	Key Label	Allowed in ON Statement or GO–ON Phrase			
		DOS & OS/2	UNIX	VMS	BTOS/CTOS
503	CURSOR-RIGHT	✓	✓ (2)	✓	✓
504	CURSOR-LEFT	✓	✓ (2)	✓	✓
505	HOME	✓	✓ (2)	✓	
506	END	✓	✓ (2)	✓	
507	PAGE-UP	✓	✓ (2)	✓	✓
508	PAGE-DOWN	✓	✓ (2)	✓	✓
509	BACK-TAB	✓	✓ (2)	✓	
510	INS		✓ (2)	✓	
511	HELP		✓ (2)	✓	✓
512	DEL-CHAR		✓ (2)	✓	✓
513	EXECUTE		✓ (2)	✓	✓
514	PAGE		✓ (2)	✓	
515	FIND		✓ (2)	✓	
516	INS-LINE		✓ (2)	✓	
517	DEL-LINE		✓ (2)	✓	
518	LINE-ERASE		✓ (2)	✓	
519	PAGE-ERASE		✓ (2)	✓	
520	CTRL-BREAK		✓ (2)	✓	
521	CTRL-ALT-DEL		✓ (2)	✓	
522	EXIT		✓ (2)	✓	
523	CTRL-RIGHT	✓	✓ (2)	✓	
524	CTRL-LEFT	✓	✓ (2)	✓	
525	U1		✓ (2)	✓	
526	U2		✓ (2)	✓	
527	U3		✓ (2)	✓	
528	U4		✓ (2)	✓	
529	U5		✓ (2)	✓	
530	U6		✓ (2)	✓	
531	U7		✓ (2)	✓	
532	U8		✓ (2)	✓	
533	U9		✓ (2)	✓	

(2) – Only if key is defined in protermcap file.

(continued on next page)

Table 2-2: Key Codes and Key Labels (continued)

Key Code	Key Label	Allowed in ON Statement or GO-ON Phrase			
		DOS & OS/2	UNIX	VMS	BTOS/CTOS
534	U10		✓ (2)	✓	
535	ERASE		✓ (2)	✓	
536	WHITE		✓ (2)	✓	
537	BLUE		✓ (2)	✓	
538	RED		✓ (2)	✓	
539	RESET		✓ (2)	✓	
540	ESC-F		✓ (2)	✓	
541	ESC-N		✓ (2)	✓	
542	ESC-1		✓ (2)	✓	
543	ESC-2		✓ (2)	✓	
544	ESC-3		✓ (2)	✓	
545	ESC-4		✓ (2)	✓	
546	ESC-5		✓ (2)	✓	
547	ESC-6		✓ (2)	✓	
548	ESC-7		✓ (2)	✓	
549	ESC-8		✓ (2)	✓	
550	ESC-9		✓ (2)	✓	
551	ESC-Z		✓ (2)	✓	
552	ESC-LEFT-ARROW		✓ (2)	✓	
553	ESC-UP-ARROW		✓ (2)	✓	
554	ESC-DOWN-ARROW		✓ (2)	✓	
555	ESC-V		✓ (2)	✓	
556	DO		✓	✓	
557	SELECT		✓	✓	
558	INSERT-HERE		✓	✓	
560 - 585	ALT-A to ALT-Z		✓	✓	
586	ESC-A		✓	✓	
587	ESC-B		✓	✓	
588	ESC-C		✓	✓	
589	ESC-D		✓	✓	

(2) - Only if key is defined in protermcap file.

(continued on next page)

Table 2–2: Key Codes and Key Labels (continued)

Key Code	Key Label	Allowed in ON Statement or GO–ON Phrase			
		DOS & OS/2	UNIX	VMS	BTOS/CTOS
590	ESC–E		✓	✓	
591	ESC–G		✓	✓	
592	ESC–I		✓	✓	
593	ESC–J		✓	✓	
594	ESC–K		✓	✓	
595	ESC–L		✓	✓	
596	ESC–M		✓	✓	
597	ESC–O		✓	✓	
598	ESC–P		✓	✓	
599	ESC–Q		✓	✓	
600	ESC–R		✓	✓	
601	ESC–S		✓	✓	
602	ESC–T		✓	✓	
603	ESC–U		✓	✓	
604	ESC–W		✓	✓	
605	ESC–X		✓	✓	
606	ESC–Y		✓	✓	
607	ESC–RIGHT–ARROW		✓	✓	
650	LEFT–MOUSE–UP		✓	✓	

PROGRESS provides more than one key label for certain key codes. The key label in Table 2-2 is the preferred label, but you can also use the labels in Table 2-3.

Table 2-3: Alternate Key Labels

Key Code	Alternate Key Labels
7	BELL
8	BS
10	LINEFEED, LF
12	FORMFEED, FF
13	RETURN (DOS & OS/2), ENTER (UNIX), CR, NEXT (BTOS/CTOS)
27	ESCAPE
127	CANCEL
501	UP
502	DOWN
503	RIGHT
504	LEFT
505	ESC-H
507	PGUP, PREV-PAGE, PREV-SCRN
508	PGDN, NEXT-PAGE, NEXT-SCRN
509	SHIFT-TAB
510	INSERT, INS-CHAR, INS-C, INSERT-HERE
512	DELETE, DELETE-CHAR, DEL-C
516	INS-L, LINE-INS
517	DEL-L, LINE-DEL

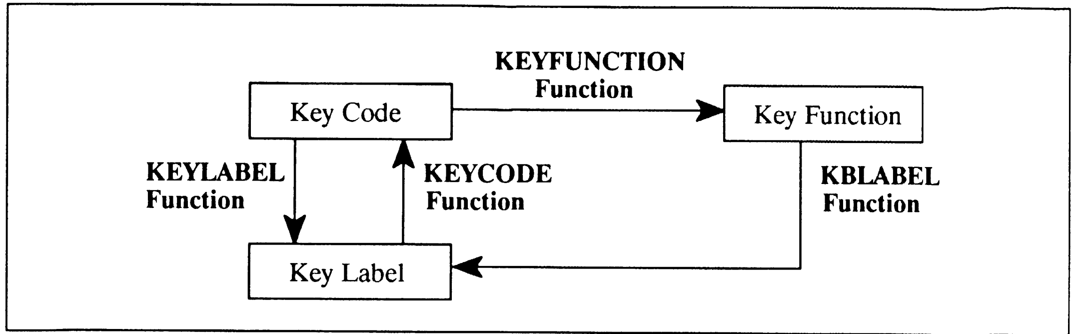
Table 2-4: BTOS/CTOS Key Labels

Key Function	BTOS/CTOS Keys	Key Code
ABORT	Action-/	515
APPEND-LINE	SHIFT-F2	214
APPEND-LINE	CODE-A	225
BACK-TAB	CODE-TAB	137
BACK-TAB	CODE-U	245
BACKSPACE	BACKSPACE	008
BLOCK	MARK	608
BLOCK	SHIFT-F4	216
BLOCK	CODE-V	246
BOTTOM-COLUMN	CODE ↓	139
BREAK-LINE	SHIFT-F1	213
BREAK-LINE	CODE-B	226
CANCEL-PICK	CODE-CANCEL	135
CHOICES	CODE-C	227
CLEAR	F8	308
CLEAR	CODE-Z	250
DELETE-CHARACTER	DELETE	512
DELETE-CHARACTER	DELETE-CHAR	008
DELETE-COLUMN	CODE-SHIFT-D	196
DELETE-FIELD	CODE-J	234
DELETE-LINE	F10	310
DELETE-LINE	CODE-D	228
END-ERROR	CANCEL	127
END-ERROR	FINISH	609
END-ERROR	F4	304
END-ERROR	CODE-E	229
FIND	SHIFT-F3	215
FIND	CODE-F	230
GET	F5	305
GET	CODE-G	231
GO	GO	513
GO	F1	301
GO	CODE-X	248
GOTO	CODE-SHIFT-G	199
HELP	HELP	511
HELP	F2	302
HELP	CODE-W	247
HOME	CODE-NEXT-PAGE	140
HOME	CODE-PREV-PAGE	133

Table 2-4: BTOS/CTOS Key Labels (continued)

Key Function	BTOS/CTOS Keys	Key Code
INSERT-COLUMN	CODE-SHIFT-C	195
INSERT-FIELD	CODE-I	233
INSERT-FIELD-DATA	CODE-SHIFT-F	198
INSERT-FIELD-LABEL	CODE-SHIFT-L	204
INSERT-MODE	OVERTYPE	510
INSERT-MODE	F3	303
INSERT-MODE	CODE-T	244
LEFT-END	CODE ←	142
LEFT-MOUSE-UP	LEFT MOUSE BUTTON	650
MAIN-MENU	CODE-M	237
MOVE	MOVE	610
NEW-LINE	F9	309
NEW-LINE	CODE-N	238
OPTIONS	CODE-O	239
PAGE-UP	PREV-PAGE	507
PAGE-UP	SHIFT-F5	217
PAGE-DOWN	NEXT-PAGE	508
PAGE-DOWN	SHIFT-F6	218
PICK	CODE-K	235
PICK-AREA	CODE-SHIFT-A	193
PICK-BOTH	CODE-SHIFT-B	194
PUT	F6	306
PUT	CODE-P	240
RECALL	F7	307
RECALL	CODE-R	242
REPORTS	CODE-SHIFT-T	212
RETURN	RETURN	013
RETURN	NEXT ←	013
RIGHT-END	CODE →	146
SCROLL-LEFT	SHIFT ←	206
SCROLL-RIGHT	SHIFT	210
SETTINGS	CODE-S	243
STOP	ACTION-CANCEL	373
TOP-COLUMN	CODE ↑	129
UNIX-END	PROGRESS EXIT	373

The key codes and key labels in Tables 2-2, 2-3, and 2-4 and the key functions in Table 2-1 are interrelated, and PROGRESS provides a set of built-in functions to convert from one to another. The following diagram shows how the conversions work, and which functions to use for each conversion.



See the *PROGRESS Language Reference* manual for more information on the KBLABEL, KEYCODE, KEYFUNCTION, and KEYLABEL functions.

2.3 DEFINING TERMINALS

So far, the keys you use when editing and running PROGRESS procedures have been explained without reference to a particular terminal keyboard. The DOS and OS/2 keys listed above apply to all PC keyboards. However, the UNIX keys listed above may vary from terminal to terminal. If you are using UNIX, PROGRESS makes certain assumptions about which keys on your terminal activate PROGRESS functions. You can modify those assumptions.

PROGRESS supports terminals based on definitions of their characteristics stored in a file called the protermcap file. The protermcap file supplied with PROGRESS 4GL/RDBMS, PROGRESS Query/Run-Time, and PROGRESS Run-Time has built-in definitions for many terminals.

It is likely that your terminal is included in the protermcap file supplied with PROGRESS. If this is the case, you can skip the rest of this chapter. If your terminal is not included in the protermcap file, you must make modifications to that file.

Note to DOS and OS/2 users: Because DOS and OS/2 do not support multiple terminal types, PROGRESS terminal definition files have only limited use. (The terminal definition file is pointed to by the PROTERMCAP environment variable, and it is `\DLC\PROTERM.CAP` by default.) `PROTERM.CAP` serves only to support the MAP option on INPUT FROM and OUTPUT TO statements. An entry in `PROTERM.CAP` can contain IN( ) and OUT( ) directives that map character values (typically, to support extended alphabetics). You might use `PROTERM.CAP` entries and the MAP option to translate extended alphabetic characters during output to a printer or when importing from a file. See also "Extended Alphabet Support" later in this chapter. (DOS and OS/2 users can skip all sections between here and "Extended Alphabet Support.")

2.3.1 Defining Your Terminal Type Under UNIX

When you first log in to UNIX, the UNIX shell variable TERM should be set to the type of terminal you are using. PROGRESS uses the value of this variable to determine the terminal type you are using.

To check the value of the TERM variable on UNIX, use this command:

```
echo $TERM
```

You can change this terminal type using normal system maintenance procedures, or by redefining the TERM shell variable. You can redefine the TERM variable either at UNIX level or by adding a line to your `.profile` file. For example,

```
TERM="wy75"; export TERM
```

defines the terminal type as `wy75`. The export command ensures that this value of TERM is available to all procedures throughout your UNIX session.

You can change the value of TERM during program execution with the PROGRESS statement TERMINAL= *termid*. See the TERMINAL function in the *PROGRESS Language Reference* manual.

2.3.2 Defining Your Terminal Type Under VMS

When you log in to VMS, your terminal characteristics are set by default. The characteristics include device or terminal type, which PROGRESS uses to determine the type of terminal you are using.

If you are using a non-standard Digital Equipment Corporation terminal (anything other than a VT100 or VT200 series), you can set the PROTERM logical to your terminal type. PROGRESS will use the PROTERM logical to look at your PROTERM.DAT to define your terminal type. To see the value of PROTERM, type:

```
SHOW LOGICAL PROTERM
```

You can change your terminal type using normal system maintenance procedures, or by redefining the PROTERM logical. You can redefine the PROTERM logical either in your `login.com` file or at the command line. For example:

```
DEFINE PROTERM "wy50"
```

defines the terminal type as `wy50`.

You can change the value of PROTERM during program execution with the PROGRESS statement TERMINAL= *termid*. See the TERMINAL function in the *PROGRESS Language Reference* manual.

2.3.3 Defining Your Terminal Type Under BTOS/CTOS

BTOS/CTOS and PROGRESS use several environment variables stored in a file. Environment variables are standard names for various information that PROGRESS needs to run.

The name of the environment variables file matches your user name plus the extension ".ENV". For example, if you sign on as Bill, PROGRESS looks for [Sys]<Sys>Bill.ENV. If the file does not exist, PROGRESS checks your .USER file (the user configuration file named *username*.USER). If the .USER file contains the entry :ProgressEnv: [Sys]<Bill>Bill.ENV, PROGRESS uses the environment file [Sys]<Bill>Bill.ENV. PROGRESS then tries to use the variables found in [Sys]<Sys>Progress.ENV. Otherwise, default values are used.

TERM = *The type of terminal you are using.*

Example: TERM = B26 (The default is B20.)

DLC = *The full pathname of the volume and directory that contains PROGRESS.*

Example: DLC = [D1]<DLC> (The default is [Sys]<DLC>.)

2.3.4 Describing Terminal Characteristics

To perform specific functions, appropriate codes must be sent to each terminal. The UNIX termcap file and the VMS SMGTERMS.TXT files contain a list of the codes to be sent to terminals in order to perform some of these functions. These functions include cursor addressing, erasing areas of the screen, and controlling the appearance of characters on the screen. The termcap file format is described in the UNIX PROGRAMMER'S MANUAL termcap(5) or in your VMS operating system manuals.

PROGRESS makes use of its own terminal definition file, which is usually stored in a file named protermcap (under UNIX and BTOS/CTOS) or PROTERM.DAT (under VMS) in the directory in which you installed PROGRESS. This file contains special feature definitions especially for use with PROGRESS. It may contain descriptions of how to draw the line graphics characters, keyboard function key assignments and labels for those keys.

Before creating a new entry in the protermcap or PROTERM.DAT file, examine it to see if your terminal type is already defined. If it is, be sure that you use your terminal's setup mode to specify the options described in the protermcap description for your terminal.

The file /etc/termcap is the standard terminal definition file used by several UNIX system programs such as the vi editor. This file is usually provided with your UNIX operating system. You can use this file as a start for creating your own protermcap entries.

The shell variable or logical PROTERMCAP is used by PROGRESS to find the protermcap file. By default, PROGRESS looks for this file wherever you start the software. However, you can redefine the PROTERMCAP variable/logical to define a new location for the protermcap or PROTERM.DAT file.

2.3.5 Creating a Protermcap Entry

If your terminal is already defined in the `protermcap` file, you may only have to make minor changes, if any. If it is not defined in the `protermcap` file, you must create a terminal description for use with PROGRESS.

To create a terminal description for use with PROGRESS, extract from the `/etc/termcap` file the definition, if any, for the terminal you are using or a similar terminal. Add to that entry as many of the entries described below as apply to the terminal. Put the result in a file named `protermcap` in your working directory.

After you create a `protermcap` file and store it in your working directory, you must set the PROTERMCAP variable to the name of your alternate `protermcap` file. If you are a UNIX user, include the following line in your `.profile` file or you may enter it online. If you are a BTOS/CTOS user, add the following line in your `.ENV` file.

Operating System	Setting your protermcap
UNIX	In `.profile` or online: `PROTERMCAP=protermcap;export PROTERMCAP`
VMS	`DEFINE PROTERMCAP PROTERM.DAT`
BTOS/CTOS	`In .ENV file add:` `PROTERMCAP=[Sys]<DLC>PROTERMCAP`

Now, run PROGRESS. If you left any required entries out of the `protermcap` file, PROGRESS will print an error message, and you can make corrections and try again. Otherwise, try some operations in the PROGRESS editor, such as insert line, delete line, insert character, delete character, and scroll up/down. Also run some procedures that require data entry and do displays in the message area. If these are successful, your terminal is properly defined.

To go back to using the default PROTERMCAP during a UNIX session in which you explicitly defined PROTERMCAP, use the UNIX `unset` command:

```
unset PROTERMCAP
```

If `unset` is not available on your machine, you may need to logout and enter UNIX again.

The corresponding command on VMS is `DEASSIGN`:

```
DEASSIGN PROTERMCAP
```

This feature is not supported for BTOS/CTOS operating systems.

Some versions of UNIX use an alternate form of `termcap` called `terminfo`. In this case you can start with the `terminfo` entry, but you will need to modify the `terminfo` keywords to match the `protermcap` keywords described below.

If you decide that you want to make modifications to the `protermcap` file generally available, you must add those modifications to the `protermcap` file in the directory where you installed PROGRESS. After making modifications, be sure to make a copy of the modified `protermcap` file: when you install a new version of PROGRESS, the installation procedure will replace your protermcap file with the one accompanying the new version.

2.3.6 Basic PROTERMCAP Entries

The PROTERMCAP (`protermcap`, `PROTERM.DAT`) file holds the terminal description for your particular monitor, telling PROGRESS how to interact with your individual screen. It specifies which characters to use to draw boxes, write in various video modes (inverse, blinking, etc.), how many lines there are on your screen, and what actions to perform for each key on your keyboard.

The PROTERMCAP file already includes the proper default configuration codes for all the monitors and keyboards mentioned in the installation pamphlet, *PROGRESS Installation Notes*. However, you may want to edit this file to tailor your own particular environment.

The next sections describe PROTERMCAP values that can be changed. BTOS/CTOS users should not change the following values; altering these entries may cause PROGRESS **not** to run, or at least not to run correctly. These entries are:

```
bc, cd, ce, cl, cm, do, kd, ku, kl,
kr, nd, se, so, ue, us, up, CF, and CN
```

NOTE: Please make a BACKUP copy of your PROTERMCAP file before attempting any modifications.

Table 2-5 lists the required PROTERMCAP entries for any terminal if that terminal is to be used to run PROGRESS. Default values can be used for all other entries, although you should use as many of the additional entries described in the previous section, "Creating a Protermcap Entry", as you can.

Table 2-5: Required PROTERMCAP Entries

Termcap Symbol	Description
cm	Cursor motion
up	Move cursor up a line
cd	Clear to end of screen
ce	Clear to end of line
cl (1)	Clear entire screen
al, dl (2) or sr, cs	Add line, delete line Scroll reverse, change scroll region

(1) On some terminals, cl (clear screen) must be defined as home cursor followed by clear to end of the screen.

(2) If you have the al and dl entries (add line and delete line), PROGRESS scrolls by adding and deleting lines. If you have the cs and sr entries (scroll region capabilities), PROGRESS uses the terminal's scrolling capability. If you have none of these, PROGRESS scrolls by repainting the screen (this action can be quite slow). These entries are useful, but not required.

Each terminal must be defined on a single logical line in the protermcap file. If you cannot fit all the entries on one line, you must use the line continuation character "\" at the end of each physical line. There is a 2048K byte limit on the size of one protermcap entry.

Within PROTERMCAP, you use a colon (:) as a separator. If the definitions of a particular code involves using a colon, then enter its octal equivalent, \072. To indicate a control character you enter a caret (^) followed by the character (e.g. ^N represents the single character Ctrl-N).

If you have trouble with a PROTERMCAP entry, first check to be certain that every line **EXCEPT THE LAST ONE** for a particular terminal ends with a backslash "\". Also be sure you have not entered a space after the backslash.

Figure 2-1 shows a sample minimum PROTERMCAP entry for a Digital Equipment Corporation VT-100 terminal in ANSI mode.

```
V1|vt100|dec vt100:\
        :cm=10\E[%i%d;%dH:up=\E[A:cd=10\E[J:ce=10\E[K:\
        :cl=10\E[H\E[J:cs=\E[%i%d;%dr:sr=\EM:
```

Figure 2-1: Minimum protermcap Entry for DEC VT-100

2 Keyboard and Editor

`v1` is a two character code to identify the VT-100; `vt100` is the standard abbreviation; `dec vt100` is the long description. Any of these values can be set in the TERM variable to access this entry. The following list explains the rest of the codes in this protermcap entry.

- `cm=10\E[%i%d;%dH` ➔ Cursor addressing code: *Escape* [*line-num* ; *column-num* `H`

  The symbol `\E` represents the escape code (octal 033, decimal 27). The 10 represents a padding constant indicating to wait 10 milliseconds after sending this code to allow the terminal to complete the operation. Whether a padding constant is required, and its value if it is needed, are usually determined by trial and error, and may be machine dependent.

- `up=\E[A` ➔ Cursor up code: *Escape* `[A`
- `cd=10\E[J` ➔ Clear to end of screen code: *Escape* `[J`
- `ce=10\E[K` ➔ Clear to end of line code: *Escape* `[K`
- `cl=10\E[H\E[J` ➔ Clear entire screen code: *Escape* `[H` *Escape* `[J`
- `cs=\E[%i%d;%dr` ➔ Define scroll region code: *Escape* `[` *top-line* ; *bottom-line* `r`
- `sr=\EM` ➔ Reverse scroll code: *Escape* `M`

In this coding *bottom-line, column-num, line-num, top-line* are replaced by decimal integers and *Escape* is replaced by the ASCII ESC code (octal 033 or decimal 27).

2.3.7 Additional PROTERMCAP Entries

Once your minimal PROTERMCAP file is working, you can add more entries to take full advantage of your terminal's features. Table 2–6 lists these features which makes screen handling faster and significantly improves screen displays.

Table 2–6: Additional PROTERMCAP Entries

protermcap Symbol	Meaning
am	Automatic margin. Indicates the terminal wraps past column 80. PROGRESS needs to know where the cursor goes after writing a character into column 80. Note that PROGRESS never writes into the lower right corner of the screen, unless you force it to by using PUT SCREEN. If the right-hand side of a frame that ends in column 80 is not displaying correctly, there may be an inconsistency between the automatic margin setting on the terminal and the protermcap am setting.
bc	Backspace character. Indicates how to move the cursor one position to the left. This is only required if the normal backspace character (Ctrl-H) deletes the character to the left of the cursor position. If the backspace character conflicts with another terminal function (e.g. Ctrl-H is sent by both the keyboard backspace key and the left-arrow key) then when that common code is received from the terminal, PROGRESS interprets it as the function other than BACKSPACE (e.g. in the case above, pressing either the back-space or the left-arrow key results in the cursor moving left one position and not doing the PROGRESS BACKSPACE function of deleting the character to the left of the cursor).
bw	Backspace wrap. Indicates that when a backspace is done with the cursor in column 1 of a line, the cursor will move to the last column of the previous line.
do	Move cursor down 1 line. The default is the newline character (\n).

Table 2–6: Additional PROTERMCAP Entries (continued)

protermcap Symbol	Meaning
dc im ei	Delete character, begin insert mode, and end insert mode. If all three are present, then the PROGRESS editor will use the terminal's insert and delete modes. If any are missing, then the PROGRESS editor will simulate both insert and delete characters.
ic	Insert character.
is	Terminal initialization. This code is sent to the terminal when PROGRESS starts, and before the Si code is sent.
ku kd kr kl	Define the codes sent by the up arrow, down arrow, right arrow, and left arrow cursor motion keys. These symbols are allowed for compatibility with UNIX termcaps, but the cursor motion keys are usually defined for the CURSOR-UP, CURSOR-DOWN, CURSOR-RIGHT, and CURSOR-LEFT functions as described in section 16.1.
k0 k1 k2 k3 k4 k5 k6 k7 k8 k9 k. k, k-	Define the codes sent by the numeric keypad keys if these are different from the codes sent by the standard 0–9, period, comma, and hyphen keys.
li co	The number of lines and columns on the screen. The default code is :li#24:co#80. All termcap definitions supplied with PROGRESS have li set to 24 and co set to 80. This is done so that a procedure developed on a terminal with more than 24 lines or more than 80 columns will run on other terminals. If this is not an important factor, you can change the li and co entries and PROGRESS will utilize the larger screen. PROGRESS uses co as the default maximum frame width for procedures run interactively, and uses 80 columns for procedures run in batch or background. This applies to all data formatting regardless of whether output is directed to a terminal. You can override the default by using the WIDTH parameter in the Frame phrase in all your procedures.
nd	Move cursor right. If you do not use this code, PROGRESS uses the cursor movement code (cm) which is slightly slower.
pc	Pad character (the default is null).
rp	Repeat the previous character n times. The encoding is like cm.

Table 2-6: Additional PROTERMCAP Entries (continued)

protermcap Symbol	Meaning
sf	Scroll forward. You may use this on terminals which have scrolling regions (e.g. terminals with cs and sr defined).
us ue	Underline start and end. PROGRESS uses these codes to indicate which fields are enabled for input. Although these codes are not required, if you do not enter them and do not enter a code for the INPUT attribute as described in Section 3.3.9, then fields enabled for input cannot be specially indicated on the screen. You do not have to specify codes that actually do under-lining, but you may instead use the codes for any video attribute or color that you want. The attribute specified will also be used whenever the attribute keyword INPUT is used in a COLOR-Phrase. On certain terminals, the control string to start a video attribute (such as underline) does not shut off the preceding attribute (such as reverse video), resulting in combined attributes (such as reverse video AND underline). In this case, you must start the string of control characters for a video attribute with the control sequence to stop other video attributes.
ug	Number of screen spaces used for the underline screen attribute. This code is only for terminals which need a character position to hold the attribute code. If more than one space is needed, PROGRESS will not use screen attributes. Setting this may affect frame formatting since extra spaces must be allocated for attributes.
so se	Enter and leave highlight mode (reverse video, high intensity, or whatever you choose). PROGRESS uses this attribute by default for the two line message area at the bottom of the screen. The attribute specified will also be used whenever the attribute keyword MESSAGES is used in a COLOR-Phrase. If you do not give these codes and do not enter a code for the MESSAGES attribute as described in Section 16.3.9, then messages will appear in normal video.
sg	Number of screen spaces used for the highlight screen attribute. The same rules as for the underline attribute apply (see ug).
ti te	Some terminals require initialization before any cursor movement can be done with cm. PROGRESS sends the ti string to the terminal when starting up and te when ending. Leave these out if you do not need them.

2 Keyboard and Editor

Table 2-6: Additional PROTERMCAP Entries (continued)

protermcap Symbol	Meaning
Se	String to send when PROGRESS terminates (done after te).
Si	String to send when PROGRESS starts (done after is).
BW	Blanks write. If your terminal has a control sequence that causes characters to be erased starting at the location of the cursor on the screen, the BW symbol will increase the speed at which PROGRESS clears the contents of the box in a frame. You have to experiment to see how this works with the video attributes you define.
BX	Draws a box on the Wyse 60 as a single operation.
CN CF	Cursor on, cursor off. PROGRESS turns the cursor off at the beginning of a session and then turns it on when prompting for data input, inside the procedure editor, during an escape to UNIX, and at the end of a session.
XC	Indicates that, on certain spacetaking terminals, the keyboard may "lock" when the cursor is positioned over a video attribute. You have to experiment to see how this works with the video attributes you define.
GS GE	Switch to and from the graphic character set if the terminal has one. GS and GE stand for graphics start and graphics end. PROGRESS switches to the graphic set when it starts to draw a box or wants to do underlining, and switches back when the box is done. If your terminal does not have an appropriate graphic character set, leave these entries out of protermcap. The hyphen and vertical bar characters will then be used to draw boxes.
GH GV	Horizontal line graphic character. Vertical line graphic character.
G2	Upper left corner line graphic character.
G1	Upper right corner line graphic character.
G3	Lower left corner line graphic character.
G4	Lower right corner line graphic character.

The following five protermcap sequences can be used to make PROGRESS adjust to the variable-size windows on some work stations. See the sun-v entry in PROTERMCAP for an example of their use.

Table 2-7: Variable-size Window Control Sequences

protermcap Symbol	Meaning
WS	Sequence sent to window to determine its size.
DL	Specifies the delimiter portion of a window size report string--that is, the character(s) that occur between the row and column numbers. PROGRESS assumes this is exactly one character long.
PF	The prefix portion of a window size report string.
RF	Specifies that when a window returns its size, the row value precedes the column value.
TM	The suffix portion of a window size report string.

The following paragraphs explain examples of modifications you can make to the PROTERMCAP file.

1. Number of lines per screen.

   BTOS/CTOS workstations typically support 29 lines per screen. In the UNIX and VMS environments, standard ASCII terminals have only 24 lines; in DOS and OS/2, 25 lines. If you are writing applications to run solely in the BTOS/CTOS environment you will probably want to take advantage of all the available screen space. If you are developing applications to run in any of the other PROGRESS supported operating environments, you may want to reset the #li parameter in your PROTERMCAP file so that all your applications will be easily ported, as well as look and run the same. Enter the number of lines in integer value. For example:

   ```
   #li=24;\
   ```

   or

   ```
   #li=25;\
   ```

2. Redefining Key Definitions.

   PROGRESS has particular actions associated with all of the function keys on your keyboard. For instance, striking the < GO > key causes PROGRESS to check, compile,

and run the procedure in the edit buffer; pressing the < HELP > key brings the user into the HELP portion of PROGRESS. See the Pocket PROGRESS Language Reference Guide for a complete listing of all key actions. By changing the actions associated with various keys in the PROTERMCAP file, you can tailor PROGRESS to reflect your own individual working habits.

For example, to define the SCROLL-DOWN key to act like the PAGE-DOWN key, you can change the following entries:

```
:PAGE-DOWN(SCROLL-DOWN)=\021:\
```

NOTE: These values must be specified in octal format; for BTOS/CTOS users, the key values outlined in the UNISYS and CONVERGENT documentation list the key codes in hexadecimal.

3. Redefining Border Characters.

By default, PROGRESS draws a "box" around any information you display to the screen. This box consists of single thin horizontal and vertical line characters, as well as square corner blocks. The BTOS/CTOS environment supports a variety of graphics characters; you may want to change the ones PROGRESS uses. Again, this may be done by changing values in the PROTERMCAP file.

For example, if you wish to use thick graphic lines rather than the thin line characters, you would make the following changes in octal format:

```
:GH=\316:\
:GV=\306:\
:G1=\354:\
:G2=\353:\
:G3=\351:\
:G4=\352:\
```

The GH variable is for the graphics horizontal line characters, the GV variable is for graphics vertical lines, and G1 through G4 values are for the four graphic corner characters.

2.3.8 UNIX stty Control Functions

The UNIX `stty` command, rather than `protermcap` entries, is used to define or redefine the control keys that invoke the three PROGRESS functions ABORT, STOP, and UNIX-END. For example, this `stty` command:

```
stty quit ^\ intr ^C eof ^D
```

specifies that PROGRESS should use Ctrl-\ for ABORT, Ctrl-C for STOP, and Ctrl-D for UNIX-END. In entering the above `stty` command, you type the control character by holding down the Ctrl key and pressing the specified key: you do not type a caret (^) followed by the key.

PROGRESS will automatically determine the key label of the key for ABORT, STOP, and UNIX-END from the `stty` setting for `quit`, `intr`, and `eof`. This label will be either "Ctrl-*x*" where *x* is the key pressed along with the Ctrl key, or "DEL" if the delete key (octal 177, decimal 127) is used. If you want to assign another label, use the following syntax in the `protermcap` file:

:*action*(*key-label*) = *code*:

Here, *action* is ABORT, STOP, or UNIX-END. *key-label* is a key label from Table 2-2 or Table 2-3, and *code* is the control key (e.g. ^C) or is \177 for the delete key. For example,

```
:STOP(CANCEL)=\177:
```

If an entry in your `protermcap` file conflicts with your `stty` setting, you see a warning message when you start PROGRESS. For example, "You cannot use DELETE for both DELETE-CHARACTER and STOP."

2.3.9 Defining Special Keyboard Keys

You can use special keys (e.g. function keys) on the terminal keyboard to indicate that a particular action is to be taken. For example, the F7 key usually is defined to be the RECALL function. In the editor, this causes the last procedure executed to be recalled into the edit area. During data entry it causes the field into which data is being entered to be restored to the value it had at the beginning of the data entry statement.

For each special key (e.g. F7) that you want to use, there must be an entry in the protermcap file that indicates two things:

- The code the terminal sends when the key is pressed.
- Which PROGRESS action (e.g. RECALL) should be taken when the key is pressed.

A special key is defined in the `protermcap` file with the following syntax:

:*action*(*key-label*) = *code*:

Here, *action* is the name of a PROGRESS key function from Table 2-1, *key-label* is the name of a PROGRESS keyboard key-label from Table 2-2 or Table 2-3, and *code* represents the character or characters that are transmitted when the key is pressed. For example, the entry:

```
:RECALL(F7)=^AF\r:
```

defines the F7 key as transmitting a Ctrl-A followed by a capital F followed by a carriage return, and associates use of the F7 key with the RECALL function.

You can refer to special keys in a PROGRESS procedure in the following ways:

- In the ON statement. For example, `ON F7 HELP` indicates that when you press the F7 key, PROGRESS runs the help procedure.

- In a GO-ON option in a PROMPT-FOR, SET, or UPDATE statement. For example, `SET name GO-ON(F7)` indicates that you can end data entry by pressing the F7 key, in addition to any of the default methods of ending data entry.

To use special keys in either of these ways, the code transmitted by the key must be indicated in the protermcap file. If the key is assigned to a standard action as described above, then the code is given as part of that definition. If you will only use the key in an ON statement or GO-ON option, and you do not need to assign it to a standard action, then use the following syntax in the protermcap entry:

:(*key-label*) = *code*:

Here, *key-label* is the label of the key and *code* represents the character or characters that are transmitted when the key is pressed. For example, the entry

```
:(F16)=O\r:
```

indicates that the F16 key on the keyboard sends a capital O followed by a carriage return.

If any of the control code sequences sent when a key on the keyboard is pressed begin with a control key (e.g. Ctrl-F), then you will not be able to use that control key on your keyboard and the key will not have its normal PROGRESS meaning (e.g. in the above case, you could not use Ctrl-F for FIND). You would need to assign another key (e.g. F13) and/or an alternate control key.

2.3.10 Example PROTERMCAP Entry

Figure 2-2 shows a full protermcap entry for a DEC VT-100 (ANSI mode).

```
V1|vt100|DEC VT-100 running in ansi mode:\
        :cd=10\E[J:\
        :ce=10\E[K:\
        :cl=10\E[H\E[J:\
        :cm=10\E[%i%d;%dH:\
        :co#80:\
        :cs=\E[%i%d;%dr:\
        :do=\n:\
        :kd=\E[B:\
        :kl=\E[D:\
        :kr=\E[C:\
        :ku=\E[A:\
        :li#24:\
        :nd=\E[C:\
        :se=\E[m:\
        :so=\E[7m:\
        :sr=\EM:\
        :Si=\E>0:\
        :Se=:\
        :ue=\E[m:\
        :up=\E[A:\
        :us=\E[7m:\
        :GS=^N:\
        :GE=^O:\
        :G1=k:\
        :G2=l:\
        :G3=m:\
        :G4=j:\
        :GH=q:\
        :GV=x:\
        :CN=\E[?25h:\
        :CF=\E[?25l:\
        :END-ERROR(PF4)=\EOS:\
        :GO(PF1)=\EOP:\
        :HELP(PF2)=\EOQ:\
        :HOME(ESC-H)=\EH:\
        :HOME(ESC-h)=\Eh:\
        :INSERT-MODE(PF3)=\EOR:
```

Figure 2-2: Full protermcap Entry for DEC VT-100

The following list explains the codes in this protermcap entry that are not covered in Section 2.3.4.

- kd, kl, kr, and ku

  Define the four cursor motion keys.

- li and co

  Explicitly specify a 24 line, 80 column display area.

- do and nd

  Make cursor addressing more efficient.

- se,so,us, and ue

  Define underlining for input fields and reverse video highlighting for the message area.

- Si and Se

  Represent terminal initialization strings. In this case Si assigns the graphics character set as the alternate character set. Se is null.

- GS and GE

  Indicate how to start and end line graphics mode (alternate character set).

- GH, GV, G1 - G4

  Define the line graphics characters.

- CN and CF

  Define how to turn the cursor on and off.

- GO (PF1), HELP (PF2), INSERT-MODE(PF3), END-ERROR (PF4)

  Defines the keyboard code PROGRESS expects for the GO, HELP, INSERT-MODE, and END-ERROR functions. The PF1 in parentheses is the keyboard key label for the GO key. The same applies for the [HELP], [INSERT MODE], and [END-ERROR] keys.

  In addition to the PF keys defined, you can use all of the control keys from Table 2-1. For example, [GO] can be indicated by PF1 or Ctrl-X, but only Ctrl-G can be used for [GET].

- HOME(ESC-H) and HOME(ESC-h)

  Define the two keystroke sequences of either ESC and H or ESC and h as being the [HOME] key.

2.3.11 Defining Video Display Attributes in PROTERMCAP

Within PROGRESS, you can control the color (or other video attribute such as reverse video, blinking, highlighted) in which data is displayed on the terminal.

When you refer to colors in a procedure, you use one of two basic methods:

- Specify the keywords NORMAL, INPUT, or MESSAGES. NORMAL corresponds to the standard video display for your terminal, INPUT to the color used when a field is enabled for input, and MESSAGES to the color used for text displayed in the message area.
- Specify the name of a color for which a definition is provided in the PROTERMCAP file. If, during execution, a procedure refers to a color that is not defined for the terminal being used, then normal video is assumed.

Use the following syntax to define a color in protermcap:

:COLOR *color-name = start-sequence:stop-sequence:attr-count:*

Here, *color-name* is the name of the color, *start-sequence* is the sequence of characters to be sent to the terminal to start the color, *stop-sequence* is the sequence of characters sent to the terminal to stop the color attribute, and *attr-count* is either 1 or 0, indicating whether or not this color attribute takes a space on the screen (similar to `ug`, `sg` in Table 2-6). For example, the entry:

```
:COLOR GREEN=\EGp:\EG0:1:\
```

defines the color GREEN in the `wy350 protermcap` entry. (In the case of the `wy350`, the terminal must also be initialized so that this attribute is in fact associated with the physical color green.)

You can use any value you want for *color-name* except a PROGRESS keyword. For example, on a monochrome terminal you may want to define REVERSE, BLINK, and BRIGHT to represent reverse video, blinking and high intensity.

You can use the special color names INPUT and MESSAGES to define the video attributes used by default when fields are enabled for input and for displays in the message area. If you do not define INPUT or MESSAGES, then these attributes are determined as described in Section 2.3.5.

When a frame is defined in a PROGRESS procedure, you explicitly or implicitly indicate whether or not extra spaces need to be reserved for video attributes or colors. If you indicate that extra spaces should not be reserved, and if you then color the field with a color that does require extra spaces (i.e. *attr-count* is 1 in the protermcap COLOR entry), then the field is displayed in normal video and is not colored.

The Wyse 75 terminal is spacetaking for some COLOR attributes and nonspacetaking for others. This difference interferes with resetting COLOR MESSAGE (nonspacetaking) back to COLOR NORMAL in a PUT SCREEN statement. If you use WHITE instead of NORMAL whenever you reset color attributes back to normal video attributes, the Wyse 75 will behave like other terminals.

2.3.12 BTOS/CTOS Color Monitor Support

The protermcap entry "CP" has been added to allow you to specify the eight colors that make up the color pallet used by PROGRESS. You must define all eight colors. For more details on using the color options while running on B20s or NGENS see the *System Administration I: Environments* manual.

Once the "CP" entry is defined for the terminal BTOS/CTOS PROGESS calls "ProgramColorMapper" to setup the color monitor. For more information see the previous section and your BTOS/CTOS manuals.

Example 1: The following defines the color yellow:

```
:cp=\017\003\014\010\077\052\060\074:\
:COLOR  YELLOW=\377B\301:\377AA:0:
```

Example 2: The following defines the color to be red and display in reverse video:

```
:COLOR   REV-RED=\377B\301:\377AA:0:
```

2.4 THE PROGRESS CHARACTER SET

The PROGRESS character set consists of 256 possible characters. Each character is represented by eight bits. In contrast, a standard ASCII character is represented by seven bits. As a result, the standard ASCII character set consists of 128 different characters, or one half the amount allowed by PROGRESS. To enable your keyboard and terminal to use these extra characters, you may have to change your `protermcap` file, using `IN` and `OUT` statements.

If you use DOS or OS/2, the PC keyboard can generate all 256 possible PROGRESS characters and the PC screen provides a unique display representation for each character.

If you use BTOS/CTOS, the keyboard can generate all 256 possible characters and the screen provides a unique display representation for each character. However, since PROGRESS uses characters 188 and 252 (decimal) internally, the values 188 and 252 cannot appear in the `protermcap` file.

If you use UNIX, the keyboard can generate 128 characters and the screen provides a unique display representation for 95 of them (ASCII 32-126). The other 33 characters (ASCII 0-31 and 127) represent control functions, such as TAB and RETURN.

Nevertheless, no matter what operating system you use, PROGRESS can enter, store, and display alphabetic characters in the range 128-255. As stated previously, however, to use characters in this range, you may need to change your protermcap file. See the section "Extended Alphabet Support" later in this chapter for more information.

2.4.1 Allowed Characters

When using the PROGRESS editor, you cannot directly enter certain characters into the editor or into a field during data entry. Table 2-8 shows how each character is treated depending on the context.

Table 2-8: Allowed Characters

Character	Editor		Execution	
	Keyboard Input	File Input (GET (F5))	Keyboard Input	File or Other Input
0 - 31, 127	Not allowed, or performs control functions.	Allowed on DOS. Translated to blanks on UNIX and BTOS/CTOS.	Not allowed, or performs control functions.	Allowed but may cause unexpected results on UNIX terminals.
32 - 126	Allowed	Allowed	Allowed	Allowed
128 - 255	Allowed (typically, generated directly only by DOS keyboards)	Allowed, but may cause unexpected results on UNIX terminals.	Allowed (can only be generated directly by DOS, BTOS/CTOS keyboards.	Allowed, but may cause unexpected results on UNIX terminals.

In the PROGESS editor, to enter control characters or character codes the keyboard cannot generate directly, type the character's three-digit octal code, preceded by a tilde (~).

2.4.2 Upper-Lower Case Conversions and Collating Sequence

Conversions between uppercase and lowercase characters are performed as follows:

- The CAPS and LC functions convert from lower to uppercase and vice-versa.
- Uppercase and lowercase letters are considered to be identical when character strings are compared (using for example, the comparison operators, the BEGINS function, or the MATCHES function). (Note that upper- and lowercase letters are **not** considered identical for fields and variables defined as *case-sensitive*. See also Chapter 5 in the *PROGRESS Tutorial* and the ANSI SQL (-Q) startup option for more information on case-sensitive fields.)
- When sorting character strings and when indexing character fields, all lowercase letters are converted to uppercase and the resulting strings are sorted according to their ASCII codes. (This case conversion does not apply to fields and variables defined as case-sensitive.) There are exceptions with non-English (extended) characters. Several extended characters index and sort at the end of the alphabet; others sort as 2-character

sequences. Most sort like their similar ASCII characters. See also the following section, "Extended Alphabet Support".

- When data is entered into a character field containing the ! format character, it is converted to uppercase.

2.5 EXTENDED ALPHABET SUPPORT

PROGRESS supports the 58 extended alphabetic characters in Table 2-9 (from IBM code page 850; see IBM publication GE19-5356-0). This section describes:

- A PROGRESS language syntax extension required to write procedures in non-English languages.
- Non-English language terminal definitions.
- Extended character case conversion and collation rules.

DOS and OS/2 keyboards can transmit these eight-bit character codes directly. UNIX keyboards typically do not. Therefore, storage and display of extended characters using PROGRESS on UNIX systems requires mappings between standard ASCII (seven-bit) and extended (eight-bit) characters both within PROGRESS and at the terminal. The PROGRESS language syntax extension accommodates remappings at the terminal, and terminal definitions provide the conversion mechanism for PROGRESS.

Table 2-9: PROGRESS Extended Characters

	8-	9-	A-	B-	C-	D-	E-	F-
0	Ç	É	á				Ó	
1	ü	æ	í				ß	
2	é	Æ	ó			Ê	Ô	
3	â	ô	ú			Ë	Ò	
4	ä	ö	ñ			È	õ	
5	à	ò	Ñ	Á			Õ	
6	å	û		Â	ã	Í		
7	ç	ù		À	Ã	Î		
8	ê	ÿ				Ï		
9	ë	Ö					Ú	
A	è	Ü					Û	
B	ï	ø					Ù	
C	î						ý	
D	ì	Ø					Ý	
E	Ä					Ì		
F	Å							

NOTE: Characters whose codes correspond to empty boxes in the above table can still be entered in a database. However, displaying and sorting values that contain these characters may yield unpredictable results.

NOTE: In the X window environment, PROGRESS supports the ISO Latin-1 character set for all display functions. PROGRESS remaps the ISO Latin-1 character set to the IBM PC Multi-lingual Code Page 850 (see IBM publication GE19-5356-0) for internal sorting and other purposes.

2.5.1 PROGRESS Language Syntax Extension

Many terminals that display non-English characters map the nine ASCII characters in the first row of Table 2-10 to extended characters (and the keyboard keycaps are often changed accordingly). When using the PROGRESS procedure editor on such a terminal, the nine ASCII characters are not available; use the appropriate two-character alternatives listed below (each begins with the semicolon character) where PROGRESS procedure code ordinarily calls for one of the special ASCII characters.

Table 2-10: Special Character Substitutions Used in the PROGRESS Editor.

Special Character	@	[	]	^	`	{	|	}	~
Alternative Representation	;&	;<	;>	;*	;'	;(	;%	;)	;?

You cannot type extended characters directly into the edit buffer. However, there are two ways to represent extended characters:

1. A 3-digit octal code preceded by a tilde (~**216**, for example).
2. ;*ASCII-character* (;[for example), where *ASCII-character* is the character mapped to the desired extended character by an `IN` statement in `protermcap` as described in the next section.

2.5.2 Terminal Definitions for non-English Languages

To use the non-English capabilities of a terminal with PROGRESS, you must make an entry in the PROTERMCAP file for that language **for that terminal.** (The PROTERMCAP file is the file pointed to by the PROTERMCAP environment variable. It defaults to `/dlc/protermcap` for UNIX, `\DLC\PROTERM.CAP` for DOS and OS/2, and to `DLC:PROTERM.DAT` for VMS.) For example, the following is a PROTERMCAP file entry for German language on a Wyse-60 terminal. (The terminal is assumed to be in German language mode.)

```
ger|german|wy60 in german mode:\
        :IN(\176)=\341:  \
        :IN(\133)=\216:  \
        :IN(\173)=\204:  \
        :IN(\134)=\231:  \
        :IN(\174)=\224:  \
        :IN(\135)=\232:  \
        :IN(\175)=\201:  \
```

The `IN` statement maps the ASCII character in parentheses to an extended character, specified in octal notation (*ddd*). The extended characters mapped in the above entry are German ß, Ä, ä, Ö, ö, Ü, and ü. When PROGRESS sees a [character (\133) on input, it is converted to Ä (\216). Likewise, when PROGRESS needs to send an Ä to the terminal, it sends [(\133). The terminal sees [and displays Ä.

Suppose the terminal cannot display an Ä, but the database includes Ä characters. Use the OUT statement in the PROTERMCAP entry to specify an appropriate character to display for Ä, capital A for example:

```
:OUT(\216)=A:
```

Given the proper PROTERMCAP entries for a language and terminal, set the TERM environment variable to *terminal/language* (TERM=wy60/german, for example). To set an I/O stream's character mapping during program execution, you can supply PROTERMCAP entries comprised of IN and/or OUT directives with the MAP option on INPUT and OUTPUT statements as described in the next section.

2.5.3 Using [NO-] MAP with PROGRESS INPUT and OUTPUT Statements

When PROGRESS encounters IN and OUT statements in a PROTERMCAP file, it creates a translation table to convert the mapped characters. The [NO-] MAP option enables you to change the translation table in effect for an I/O stream. The following PROGRESS statements accept the MAP option:

- INPUT FROM
- INPUT THROUGH
- INPUT-OUTPUT THROUGH
- OUTPUT THROUGH
- OUTPUT TO

The MAP option names a PROTERMCAP file entry that contains IN and OUT directives. If you use MAP, PROGRESS creates a translation table for the stream, based on the IN and OUT directives. Then, as you send or receive character data through an I/O stream, PROGRESS uses the table to convert characters as necessary.

However, if you use the NO-MAP option, PROGRESS does not map the characters at all. The data is sent and received directly. The MAP option is commonly used with the OUTPUT TO statement when directing output to a printer. (Remember that IN directives cause PROGRESS to map characters on both input and output, while OUT directives cause character translation on output only.)

Use NO-MAP to ignore the translation table currently in effect for an I/O stream.

2.5.4 Extended Character Case Conversion and Collation

Recall that when sorting character strings and when indexing character fields, all lowercase letters are converted to uppercase prior to sorting. (This case conversion does not apply to index fields and variables defined as case-sensitive. See Chapter 5 in the *PROGRESS Tutorial* and the ANSI SQL (-Q) startup option for more information on case-sensitive fields.) Several extended characters index and sort at the end of the alphabet, and others sort as two-character sequences. Table 2-11 shows the case conversion rules for extended characters.

Table 2-12 shows the special collation rules for the three *language groups*. The case conversion and collation rules are as based on IBM publication GE19-5356-0 (12/84), "Software without Frontiers." The collation rules used in your database depend on the language (group) specified. The database language must be set on an **empty** database with the PROGRESS Utilities command. By default, the "basic group" collating rules are in effect.

NOTE: If you are using DOS or OS/2 and you set your terminal to German mode (or any other language mode), PROGRESS uses the DOS or OS/2 code page for collation, regardless of the collation rules in effect.

OS	Command to Set Database Language
UNIX	`proutil` *database-name* `-C language` *language*
VMS	PROGRESS/UTILITIES=LANGUAGE_*language* *database-name*

The legal values for *language* are:

```
dutch,    english,  \                      danish      }  (Danish group)
french,   german,    } (Basic group)       norwegian   }
italian,  spanish,  |
swiss               /                      swedish     }  (Swedish group)
                                           finnish     }
```

Note that you can change the value of TERM at any time to change the character conversion rules, but the collation tables remain as set on the empty database.

Table 2-11: Extended Character Case Conversion

Lowercase Character	ASCII Value (decimal)	Octal	Corresponding **Uppercase** Character	ASCII Value (decimal)	Octal
ü	129	~201	Ü	154	~232
é	130	~202	É	144	~220
â	131	~203	Â	182	~266
ä	132	~204	Ä	142	~216
à	133	~205	À	183	~267
å	134	~206	Å	143	~217
ç	135	~207	Ç	128	~200
ê	136	~210	Ê	210	~322
ë	137	~211	Ë	211	~323
è	138	~212	È	212	~324
ï	139	~213	Ï	216	~330
î	140	~214	Î	215	~327
ì	141	~215	Ì	222	~336
æ	145	~221	Æ	146	~222
ô	147	~223	Ô	226	~342
ö	148	~224	Ö	153	~231
ò	149	~225	Ò	227	~343
û	150	~226	Û	234	~352
ù	151	~227	Ù	235	~353
ÿ	152	~230	Y	89	~131
ø (1)	155	~233	Ø	157	~235
á	160	~240	Á	181	~265
í	161	~241	Í	214	~326
ó	162	~242	Ò	224	~340
ú	163	~243	Ú	233	~351
ñ	164	~244	Ñ	165	~245
ã (3)	198	~306	Ã	199	~307
õ (2)	228	~344	Õ	229	~345
ý	236	~354	Ý	237	~355
ß	255	~377	ß	255	~377

(1) cent-sign except on Norway/Denmark PCs
(2) underlined-a except on Norway/Denmark PCs
(3) upper left corner graphic symbol except on Norway/Denmark PCs

When sorted or used in an index, extended characters are further converted to the standard ASCII characters from which they are derived. For example, â sorts like capital A, ñ like capital N, Ç like capital C, and so on. However, there are several exceptions depending on the language in use.

Table 2–12: Special Collation Rules for Extended Characters

Language Group	Special Collation Rules
Basic	German ß = SS
Danish	Ü = Y (capital Y) Æ = Ä; both sort after Z Ø = Ö; both sort after Æ Å sorts after Ø (Z < Æ/Ä < Ø/Ö < Å)
Swedish	Å sorts after Z Ä sorts after Å Ö sorts after Ä (Z < Å < Ä < Ö)

2.5.5 User-Defined Language Rules

PROGRESS enables you to define your own language and control its collation sequence and case conversion. PROGRESS provides this capability to support languages outside the Basic, Danish, and Swedish groups. The Greek language, for instance, has unique collation requirements that the three language groups do not address.

Each "language" that PROGRESS supports is made up of the following four tables. To create your own language, you must prepare an ASCII text file that includes all four tables and then compile the file. The four tables, and the steps required to prepare them, are described in the following table.

Table 2-13: Language Tables

Table Name	Function
user_uppercase	Assigns an upper-case value to each ASCII character
user_lowercase	Assigns a lower-case value to each ASCII character
user_weight	Assigns a sort weight to each ASCII character
user_cs_weight	Assigns a case-sensitive sort weight to each ASCII character

user_uppercase Table. The following figure shows the format of a sample user_uppercase table:

```
user_uppercase =
{                           "A" in 66th position
            .
            .
        64,  65,    66,   67,   68,   69,   70,   71,
            .
            .
        96,  65,    66,   67,   68,   69,   70,   71,
            .
            .
};                          "A" in 98th position
```

There are 256 positions, ranging from 0 to 255, in all four tables. Each position stores an ASCII value. PROGRESS determines the upper-case value of a character by indexing the character against this table. For example, the character "a" has an ASCII value of 97. PROGRESS converts this character to the upper-case character "A," which has an ASCII value of 65, by going to the 98th position in the user_uppercase table. (Since the table starts at position 0, ASCII value 97 is indexed by the 98th position.)

Likewise, positions 99 through 104 hold the ASCII values 66 to 71 ("B" to "G").

user_lowercase Table. This table's format is identical to the `user_uppercase` table. However, instead of converting characters the their upper-case value, this table converts characters to their lower-case value.

To convert the upper-case character "A" (ASCII 65) to the lower-case character "a" (ASCII 97), PROGRESS goes to the 66th position of the table. The number 97 is in this position, and therefore PROGRESS converts "A" to "a."

user_weight Table. This table determines the order in which PROGRESS collates or sorts the characters. PROGRESS indexes the table in the same way as the other tables, using the ASCII value of each character. PROGRESS locates the sort weight of "a" (ASCII 97) by going to the 98th position in the table.

Generally, you will want upper-case and lower-case characters to sort together. Therefore, you will want to give "A" and "a" the same sort weight. This sort weight should be the ASCII value of upper-case "A," or 65. When PROGRESS indexes the `user_weight` table, it finds the number 65 in the 66th and 98th positions.

user_cs_weight Table. PROGRESS uses the `user_cs_weight` table to do case-sensitive sorting. The `user_cs_weight` table has the same format as all three of the other tables, but it enables you to give each character a unique sort weight. In this table, you want to give "A" and "a" different sort weights. You can make PROGRESS sort "A" using its ASCII value (65), and sort "a" using its ASCII value (97). In this example, "A" sorts before "a."

Building Your Tables. You can manipulate the tables to sort the characters in any order you want. However, when you build your tables, you must conform to the following rules:

- The tables must be in this order: `user_uppercase`, `user_lowercase`, `user_weight`, and `user_cs_weight`.
- Each table must have 256 positions separated by commas.
- Each cell in the table must contain an ASCII value.
- You must use the proper syntax

NOTE: The file `procoll.eng` in your installation directory is a sample version of a language table file.

You can use any ASCII text editor to prepare your tables. To make the tables more readable, you can place comments in the tables and represent the ASCII values four different ways:

- By typing the ASCII value in decimal.
- By typing the ASCII value in Hexidecimal (place a 0x before the hexidecimal number).
- By typing the actual ASCII character, surrounded by single quotes.
- By giving the ASCII value a name, using the #define directive.

Therefore, 88, 'X', and 0x58 all represent the same character.

Naming an ASCII Value. The first line in the following figure demonstrates how to assign a name to an ASCII value. (The /* and */ characters enclose a comment.)

```
#define AtildeU 199   /* A (Uppercase) with tilde "~" */
user_uppercase =
{
            .
            .
        192,  193,  194,  195,  196,  197,  AtildeU, AtildeU,
            .
            .
};
```

Each name must begin with a letter. Also, each name must be unique to the first 16 characters.

In addition, there is one symbol that must have a predefined value, ProcessSSharp. You must define this symbol as either 1 or zero. Set the value to 1 (#define ProcessSSharp 1) to have PROGRESS convert the German SSharp character to a double "S." Set the value to 0 (#define ProcessSSharp 0) if your language tables do not support the German SSharp.

In the directory where you installed PROGRESS (usually DLC), there is a procoll.eng file. Read this file to see a completed set of language tables. This file also illustrates how you can use define statements and comments.

Activating Your Tables. Once build a file that contains your completed tables, you must compile it with the following command:

OS	Command to Create Binary File procoll.dat
UNIX DOS and OS2	`proutil -C collation-compiler` *language-source-file*
VMS	`PROGRESS/UT=COLLATION/INPUT_FILE=`*language-source-file*

where *language_source_file* is the name of the ASCII file containing your completed language tables. PROGRESS creates a binary file named `procoll.dat`.

Once you have a `procoll.dat` file, you can begin to use your language. Generally, PROGRESS reads `$DLC/procoll.dat` at startup. However, if you want to store `procoll.dat` in a different directory and you want to give it a different name, you can use the PROCOLL environment variable to direct PROGRESS to the file's new location. For example:

```
PROCOLL=/usr/foo/mylang.dat.
```

Enter the following command to specify your new language. You must specify your new language on an **empty** database.

OS	Command to Specify Database Language
UNIX DOS and OS2	`proutil` *database-name* `-C language user-defined`
VMS	`PROGRESS/UTILITIES=LANGUAGE_USER` *database-name*

Also, after you create an index and add records to your empty database, be careful about changing your user_weight table. Changing it could corrupt your indexes. If this happens, run the Index Rebuild utility to repair the corrupted indexes.

Multiple Databases. When you use multiple databases, each database can have its own language. If you compare records from two databases, PROGRESS uses the language of the first connected database. You can override this default with the -xc *language* startup option.

Chapter 3
Designing and Changing Your Application Database

In the *PROGRESS Language Tutorial*, you learned how to use the PROGRESS Data Dictionary to define the files, fields, and indexes for your database. This chapter discusses the issues involved in designing your database and, because initial designs are rarely perfect, how to make changes to existing database definitions.

Designing your database involves deciding:

- How to divide your application data into different files.
- What the fields in each file should be.
- What the indexes should be on each file.
- How to use validation for files and fields.
- How to use PROGRESS Help with your application.

The next sections explore each of these areas.

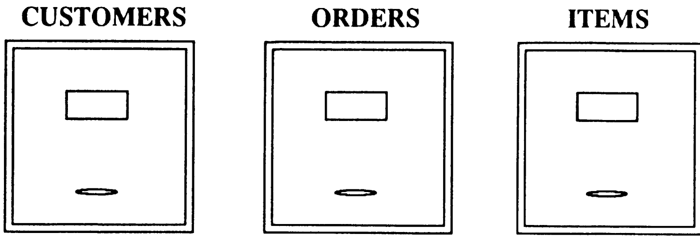
3.1 DESIGNING YOUR APPLICATION FILES AND FIELDS

When designing your database, the most logical thing to do is to think about what your database might look like if you were using a manual filing system rather than a computer. As an example, we'll use a model of a simple order entry system.

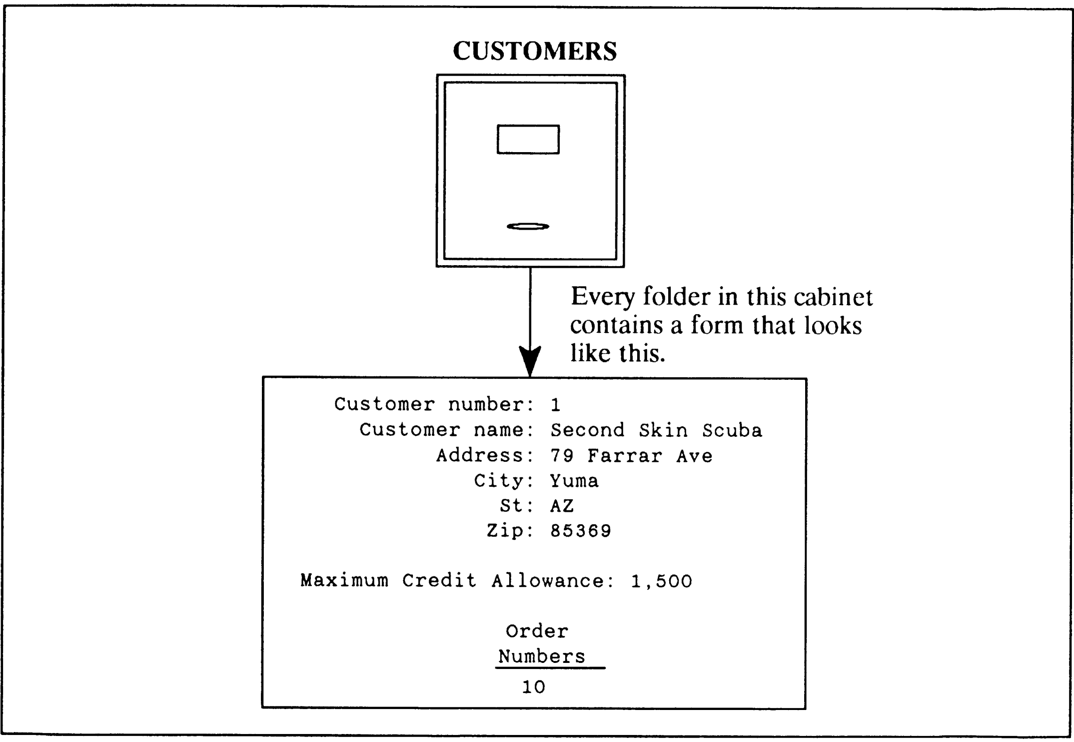
A simple order entry system has three basic parts:

- Customers who have placed orders or who are planning on placing orders.
- Orders that have been placed and possibly shipped.
- Items that have been ordered or are in inventory.

Suppose that you have three separate filing cabinets, one for each category of information.



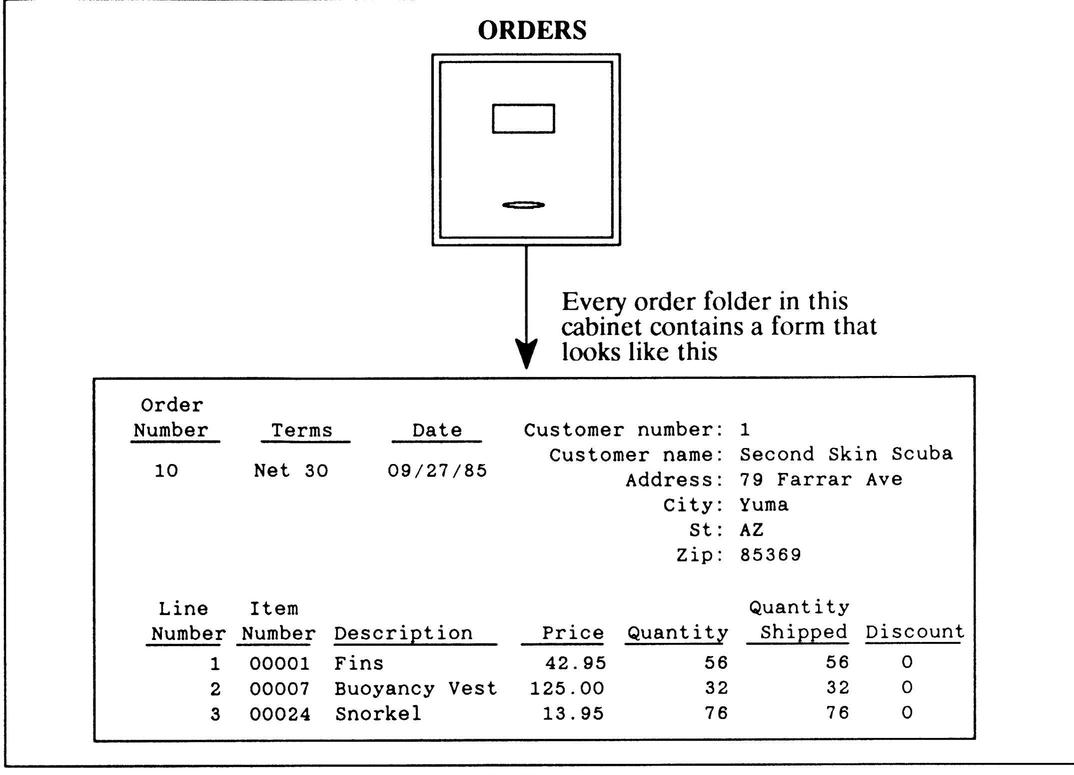
Let's look at the kind of data that might be kept in each of these cabinets.



Here, the information about a customer includes:

- Customer specific information such as a name and address.

- A list of the orders placed by the customer. If this information wasn't here, you would have to look at every form in the order filing cabinet until you found all the forms that said "Second Skin Scuba".



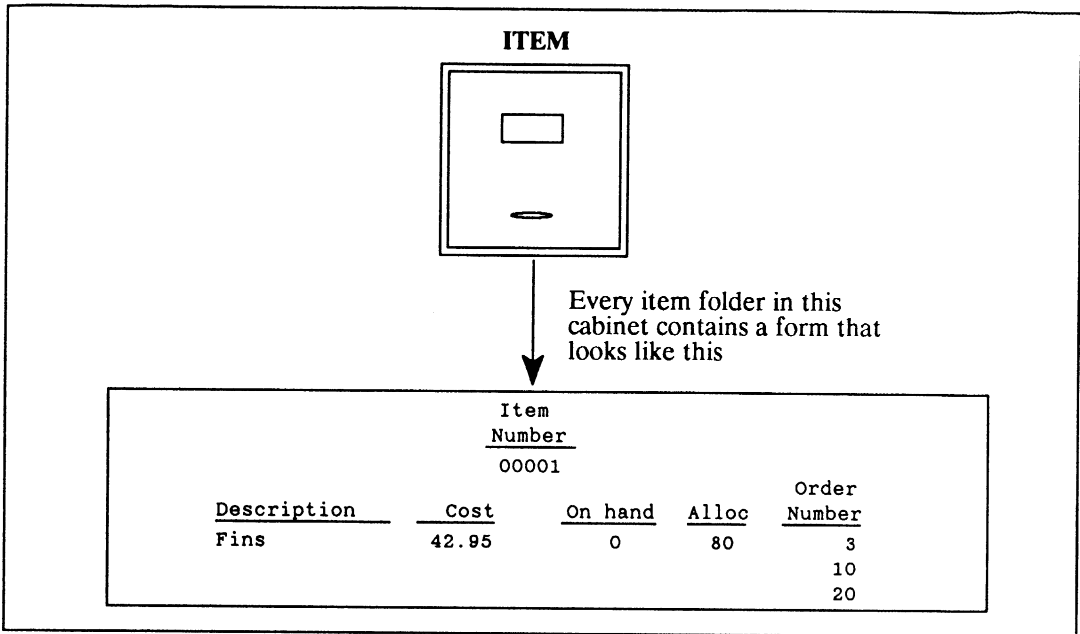
Order Number	Terms	Date
10	Net 30	09/27/85

Customer number: 1
Customer name: Second Skin Scuba
Address: 79 Farrar Ave
City: Yuma
St: AZ
Zip: 85369

Line Number	Item Number	Description	Price	Quantity	Quantity Shipped	Discount
1	00001	Fins	42.95	56	56	0
2	00007	Buoyancy Vest	125.00	32	32	0
3	00024	Snorkel	13.95	76	76	0

Here, the information for every order includes:

- Order-specific information such as the order number and the date.
- Information about the customer who placed the order. If this information wasn't here, you would have to look at every customer folder to find the customer that placed this order.
- Information about each of the items on the order. If this information wasn't here, you would have to look at forms in the item file to find information about the item.



Here, the item information includes:

- Item-specific information such as the number and description.
- Information about which orders requested the item. If this information weren't here and you wanted to know which orders used the item, you would have to look at every form in the order cabinet to find those that ordered this item.

3.1.1 Streamlining Your Data (Normalization)

You probably noticed that, in our manual filing system, some pieces of information are included in more than one file:

- Customer number, name, address, city, st, and zip all appear in both the customer file and the order file.
- The order number appears in all three files.
- The item number appears in two files.

In database terminology, this repetition is known as **redundancy**. Minimizing this redundancy is called **normalizing** your data.

Now, forget about those filing cabinets - after all, you have PROGRESS! How would you go about setting up the same files, keeping in mind that you want to reduce the redundancy you just saw?

3.1.2 Relating Files to One Another

You can significantly reduce data repetition across files by considering how you might relate those files to one another. There are three kinds of file relationships that are important for you to understand:

- One-to-Many

  A record in one file is related to many other records in another file or many records in one file are related to a single record in another file.

- One-to-One

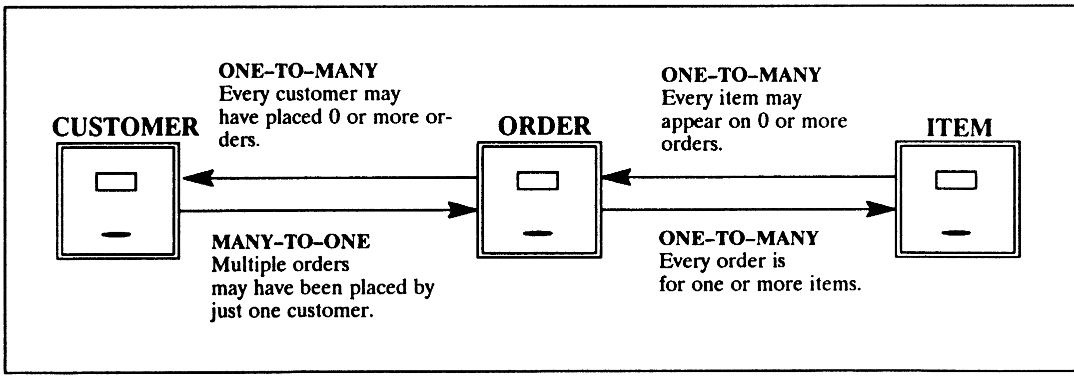
  One record in one file is related to one record in another file. (This is far less common than the one-to-many relationship).

- Many-to-Many

  Many records in one file are related to many records in another file.

See Chapter 6 of the *PROGRESS Language Tutorial* for a more detailed discussion of these relationships.

Let's look at how we can apply these relationships to our manual filing system:



Now that we know how the customer, order, and item files are related, we need to keep less information in each file. However, you need some other information to create a database file.

3.1.3 Creating Database Files

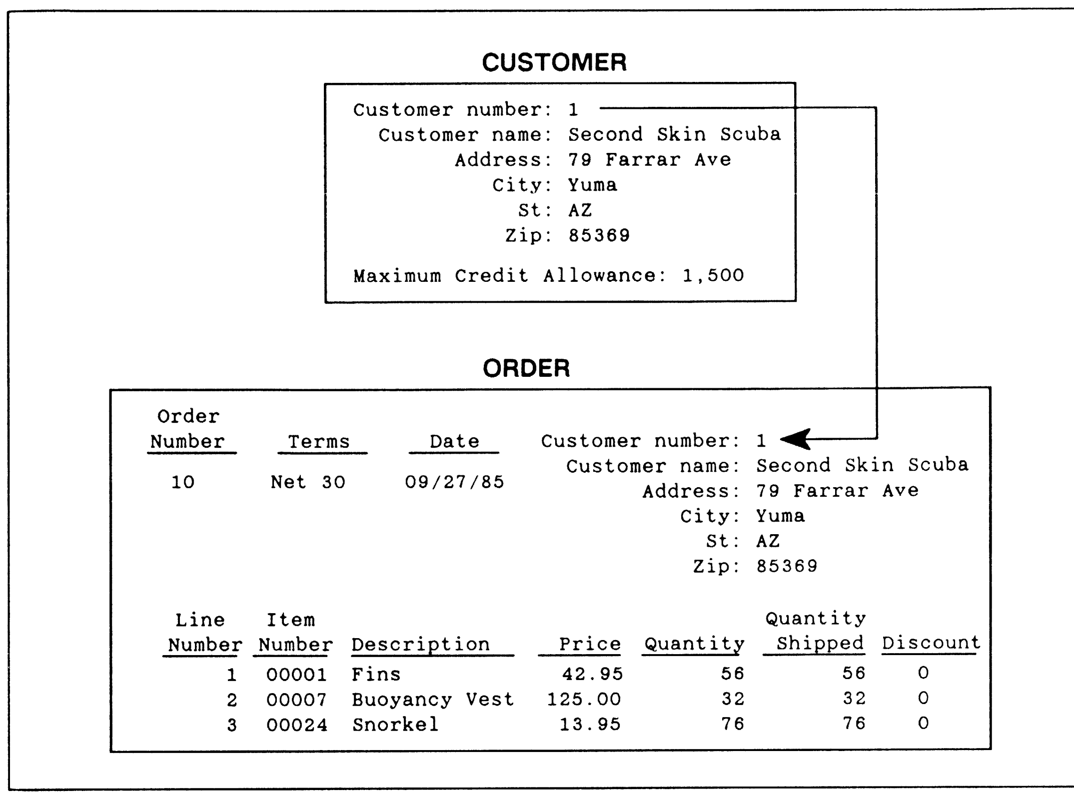
Think of each piece of paper as a record. That is, every piece of paper in the customer filing cabinet represents a record in the customer database file. You know that every record in a file must contain the same fields. That means that, for every piece of information on the paper form, there must be a field in the corresponding record.

Let's try to put this concept to work by designing the customer file.

```
Customer number: 1
  Customer name: Second Skin Scuba
        Address: 79 Farrar Ave
           City: Yuma
             St: AZ
            Zip: 85369

Maximum Credit Allowance: 1,500

           Order
          Numbers
            10
```

Now, can you turn this piece of paper directly into a record? To do that, you would have to name one field for each piece of information. That will work just fine in this case; the fields in the customer file can be: customer number, name, address, city, state, zip, and max-credit.

Now, do you need order number information in the customer file? You know that one customer can have many orders and you also know that the order file includes the customer number. So use the customer number field as a way to access the appropriate records in the order file:



CUSTOMER

Customer number: 1
Customer name: Second Skin Scuba
Address: 79 Farrar Ave
City: Yuma
St: AZ
Zip: 85369

Maximum Credit Allowance: 1,500

ORDER

Order Number	Terms	Date
10	Net 30	09/27/85

Customer number: 1
Customer name: Second Skin Scuba
Address: 79 Farrar Ave
City: Yuma
St: AZ
Zip: 85369

Line Number	Item Number	Description	Price	Quantity	Quantity Shipped	Discount
1	00001	Fins	42.95	56	56	0
2	00007	Buoyancy Vest	125.00	32	32	0
3	00024	Snorkel	13.95	76	76	0

By using a single piece of information that is common to both customers and orders, the customer number, we are able to remove all of the order information from the customer file. Now let's take a look at the order file.

Order Number	Terms	Date	Customer Number
10	Net 30	09/27/85	1

Line Number	Item Number	Description	Price	Quantity	Quantity Shipped	Discount
1	00001	Fins	42.95	56	56	0
2	00007	Buoyancy Vest	125.00	32	32	0
3	00024	Snorkel	13.95	76	76	0

Can you do the same thing with this form that you did with the customer form? That is, can you create a record that has a field for every piece of information on the paper? To do this, your fields would have to look something like this:

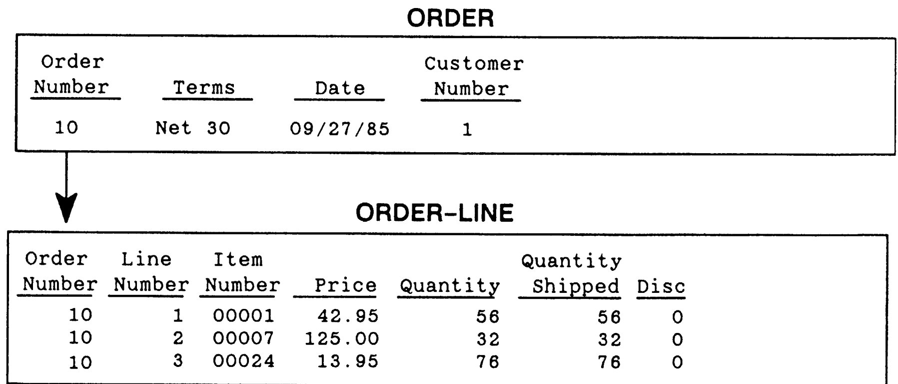
- Item number for line number 1

- Item number for line number 2
- Item number for line number 3
- Description for line number 1
- Description for line number 2
- Description for line number 3
- And so on.

You would have to have enough fields to hold values for each of the separate order lines. If you wanted to allow many items to be ordered on one order, you would need to allow for hundreds of fields on each order. There is an easy way to resolve this. Simply create a separate file called an `order-line` file. In that file you can keep individual records for each of the lines on an order. Here is what a single record in the `order-line` file might look like:

Order Number	Line Number	Item Number	Description	Price	Quantity	Quantity Shipped	Discount
10	1	00001	Fins	42.95	56	56	0

Since the `order-line` file contains the order number, and the order file contains the order number, you can use that field to relate the two files to one another:



ORDER

Order Number	Terms	Date	Customer Number
10	Net 30	09/27/85	1

ORDER-LINE

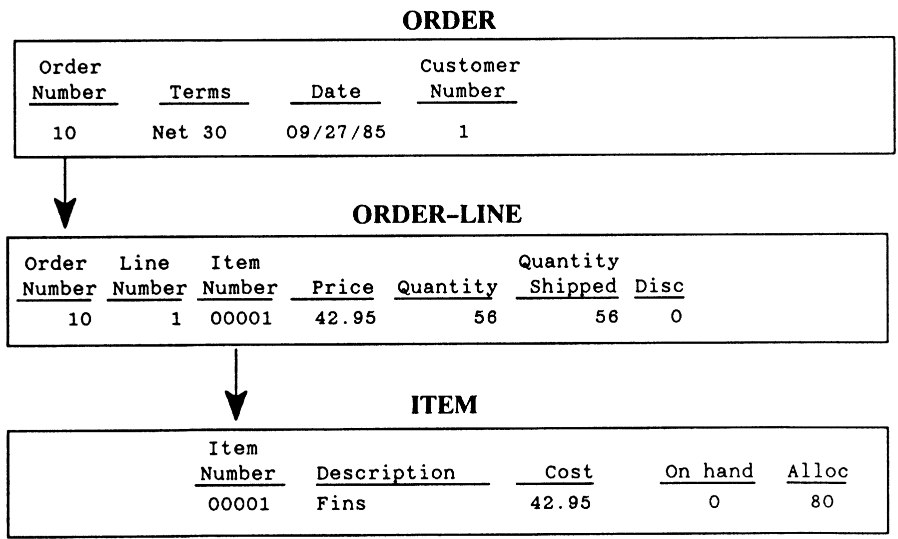
Order Number	Line Number	Item Number	Price	Quantity	Quantity Shipped	Disc
10	1	00001	42.95	56	56	0
10	2	00007	125.00	32	32	0
10	3	00024	13.95	76	76	0

Now that we've got the order file information organized, what about the item file?

		Item Number			
		00001			
Description	**Cost**		**On hand**	**Alloc**	**Order Number**
Fins	42.95		0	80	3
					10
					20

You can easily assign a field in an item record each of the pieces of data shown here. The only data that would give you any trouble would be the list of order numbers. Because the item might appear on more than one order, there might be multiple order numbers, as for item number 1. If you try to define order number as an array field, there is no way of telling how many orders an item might appear on. What it comes down to is that defining a field for the order numbers is not very practical.

What's the answer then? Every item appears on a specific line of an order. So why not use the order-line file as a link between the order and item files. That way, you don't even need the order number data in the item record:



ORDER

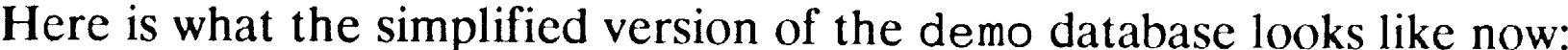
Order Number	Terms	Date	Customer Number
10	Net 30	09/27/85	1

ORDER-LINE

Order Number	Line Number	Item Number	Price	Quantity	Quantity Shipped	Disc
10	1	00001	42.95	56	56	0

ITEM

Item Number	Description	Cost	On hand	Alloc
00001	Fins	42.95	0	80



Here is what the simplified version of the demo database looks like now:

CUSTOMER

```
Customer number: 1
  Customer name: Second Skin Scuba
        Address: 79 Farrar Ave
           City: Yuma
             St: AZ
            Zip: 85369

Maximum Credit Allowance: 1,500
```

A2 A1

ORDER

Order Number	Terms	Date	Customer Number
10	Net 30	09/27/85	1

B2 B1

ORDER-LINE

Order Number	Line Number	Item Number	Description	Price	Quantity	Quantity Shipped	Disc
10	1	00001	Fins	42.95	56	56	0

C1 C2

ITEM

Item Number	Description	Cost	On hand	Alloc
00001	Fins	42.95	0	80

This new design has established relationships among the `customer`, `order`, `order-line`, and `item` files. Specifically, you can determine:

- All orders placed by a customer (**A1**).
- For an order, the corresponding customer information (**A2**).
- For an order, what was ordered (order-line) (**B1**) and information about each item ordered (**C1**).
- For an item, all the places it was ordered (**C2**), the order it was on (**B2**), and the customer who placed the order (**A2**).

3.1.4 Defining Large Records

When defining records with large numbers of fields or with long fields, keep in mind the 32,000 byte record size limit. This limit refers to the actual data stored in a record. (However, performance may be adversely affected when record size exceeds 2K bytes.)

You may also want to consider creating parallel records in two separate files. This approach may be more efficient if one of the files can group fields that are infrequently used. You can use the FIND statement with the OF option, if both files have indexes with the same name.

3.2 INDEXING YOUR DATABASE

Chapter 5 of the *PROGRESS Language Tutorial* explained how to define indexes for your database. The next sections tell you when you should and shouldn't define indexes and explain some of the more detailed points of index definition.

3.2.1 Why Define an Index?

There are at least four benefits to defining an index for a file:

- Rapid retrieval of one or more records. When you use the FIND statement, PROGRESS automatically uses the index to quickly read one record or a set of records. For example:

p-index.p

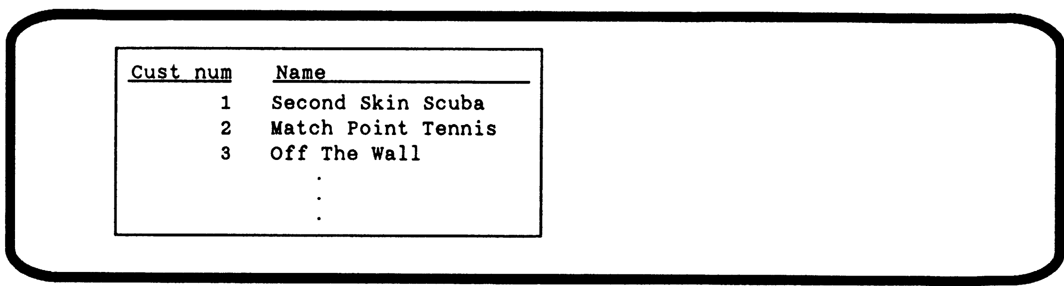
```
PROMPT-FOR customer.cust-num.
FIND customer USING cust-num.
DISPLAY customer WITH 2 COLUMNS.
```

Here, PROGRESS uses the cust-num index in the customer file to quickly locate customer records.

- Automatic ordering of records. PROGRESS uses the index to automatically sort records. For example:

p-index1.p

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



You can see that the customer records are automatically put in order by the `cust-num` field. The primary index of the `customer` file in this database is based on `cust-num`. If you wanted to see the records in some other order, you could have named another index in the FOR EACH statement.

- Enforcing uniqueness. When you define an index for a file as a unique index, you ensure that no two records can have the same value for that index. For example, the `cust-num` index is a unique index. To demonstrate, run this procedure:

p-index2.p

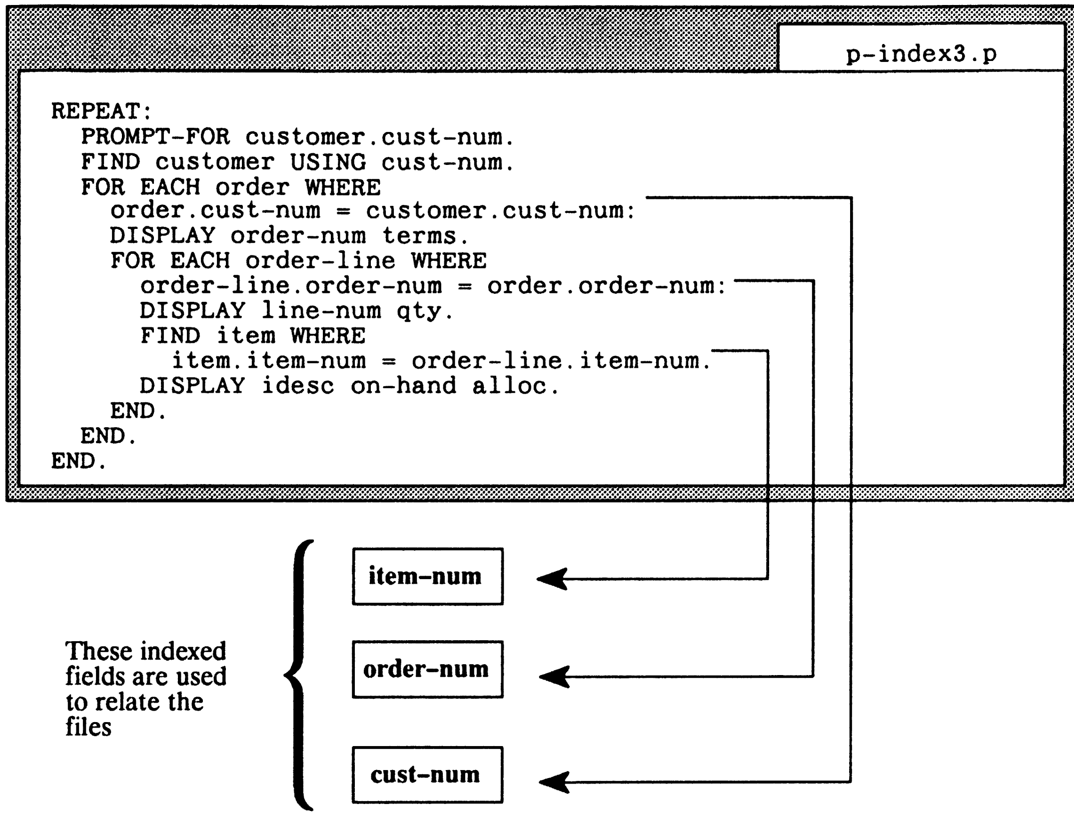
```
REPEAT:
  CREATE customer.
  UPDATE cust-num name.
END.
```

Enter the following data and press RETURN.

```
Cust num   Name
       1   Marco Polo Pools

** customer already exists with Cust num 1.
```

You get an error message saying that customer number 1 already exists. You get that message because `cust-num` is a unique index for the customer file.

- Rapid processing of inter-file relationships. Two files are related if there is a field (or fields) in one file that is used to access a record in another file. If the file being accessed has an index based on the corresponding field then the record access will be much more efficient. For example:



Here, both FOR EACH statements and the FIND statement use the WHERE option to locate one or more related records. The fields used in each WHERE option are indexed, which means that PROGRESS can find the appropriate record or records without having to scan an entire file. For example:

```
FOR EACH order WHERE
  order.cust-num = customer.cust-num:
```

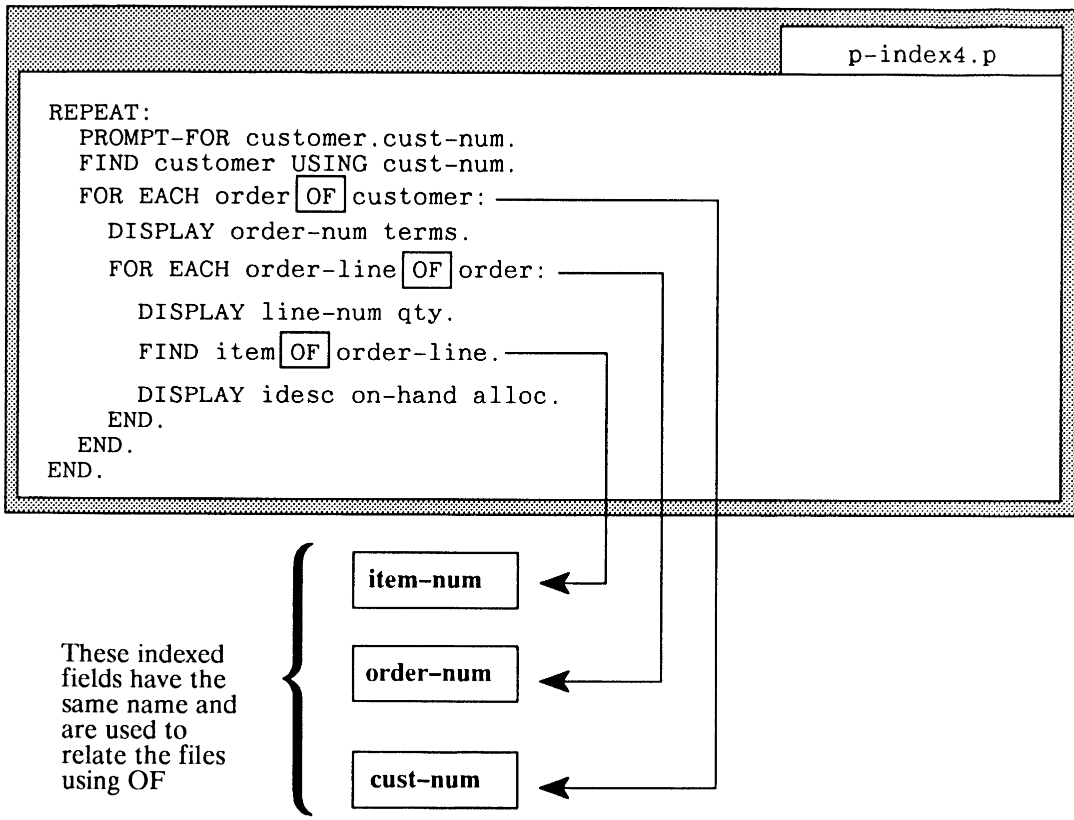
tells PROGRESS to find all the orders for a specific customer. This can be processed efficiently if there is an index in the order file based on cust-num or that has cust-num as the first field in a multi-field index.

It is important to understand that the field you use to relate two files to one another need not have the same name in both files. For example:

```
FOR EACH order WHERE
  order.c-num = customer.cust-num:
```

This works as long as c-num is indexed in the order file.

Because relating records by using indexed fields is something you will do quite often in your application, PROGRESS provides a shortcut: the OF keyword. Here is the procedure you just saw but with this shortcut incorporated:



3.2.2 Why Define an Index - Summary

Once again, the reasons for defining an index are:

- Rapid retrieval of one or more records.
- Automatic ordering of records.

- Enforcing uniqueness.
- Rapid processing of inter-file relationships.

Table 3-1 lists the indexes for the demo database, showing which of the above four reasons was used for defining each index.

Table 3-1: Reasons for Defining the Demo Database Indexes

<table>
<tr><th>File</th><th>Index Name</th><th>Index Field(s)</th><th>Primary</th><th>Unique</th></tr>
<tr><td>customer</td><td>cust-num</td><td>cust-num</td><td>YES</td><td>YES</td></tr>
<tr><td></td><td colspan="4">Why the Index Was Defined
1. Rapid access to a customer given a customer's number.
2. Reporting customers in order by number.
3. Ensuring that there is only one customer record for each customer number (uniqueness).
4. Rapid access to a customer from an order, using the customer number in the order record.</td></tr>
<tr><td></td><td>name</td><td>name</td><td>NO</td><td>NO</td></tr>
<tr><td></td><td colspan="4">Why the Index Was Defined
1. Rapid access to a customer given a customer's name or the first letters in the customer's name.
2. Reporting customers in order by name (could also be done using sorting in the FOR EACH statement).</td></tr>
<tr><td></td><td>zip</td><td>zip</td><td>NO</td><td>NO</td></tr>
<tr><td></td><td colspan="4">Why the Index Was Defined
1. Rapid access to all the customers with a given zip code or in a zip code range.
2. Reporting customers in order by zip code (could also be done using sorting in the FOR EACH statement).</td></tr>
<tr><td>item</td><td>item-num</td><td>item-num</td><td>YES</td><td>YES</td></tr>
<tr><td></td><td colspan="4">Why the Index Was Defined
1. Rapid access to an item given an item number.
2. Reporting items in order by number.
3. Ensuring that there is only one item record for each item number (uniqueness).
4. Rapid access to an item from an order-line, using the item-num field in the order-line record.</td></tr>
</table>

(continued on next page)

Table 3–1: Reasons for Defining the Demo Database Indexes (continued)

File	Index Name	Index Field(s)	Primary	Unique
order	order-num	order-num	YES	YES
	Why the Index Was Defined 1. Rapid access to an order given an order number. 2. Reporting orders in order by number. 3. Ensuring that there is only one order record for each order number(uniqueness). 4. Rapid access to an order from an order-line, using the order-num field in the order-line record.			
	cust-order	cust-num order-num	NO	YES
	Why the Index Was Defined 1. Rapid access to all the orders placed by a customer. Without this index, all of the records in the order file would be examined to find those having a particular value in the cust-num field. 2. Ensuring that there is only one record for each customer/ order combination (uniqueness). 3. Rapid access to the order numbers of a customer's orders.			
	order-date	odate	NO	NO
	Why the Index Was Defined 1. Rapid access to all the orders placed on a given date or in a range of dates.			
order-line	order-line	order-num line-num	YES	YES
	Why the Index Was Defined 1. Ensuring that there is only one order-line record with a given order number and line number. The index is based on both fields together since neither alone need be unique. 2. Rapid access to all of the order-lines for an order, ordered by line number.			
	item-num	item-num	NO	NO
	Why the Index Was Defined 1. Rapid access to all the order-lines for a given item.			

3.2.3 Why Not Define an Index?

Even though indexes give you the benefits described in the last section, there are two things to keep in mind when defining indexes for your database:

- Indexes take up disk space.
- There is extra processing overhead associated with every record creation and deletion, and every update to an index field value.

Define the indexes your application requires, but avoid indexes that provide minor benefit or are infrequently used. For example, unless you display data in a particular order (such as by zip code) very frequently, then you are usually better off sorting the data when you display it rather than defining an index to do that sorting automatically.

3.2.4 Maintaining Indexes

As you work with your application, you will want to know when PROGRESS creates and updates indexes. PROGRESS first puts a record into an index at the earlier of:

- The end of any statement following which values have been assigned to all components of the index.
- The end of the closest iterating subtransaction block in which the record is created. For more information on subtransaction blocks, see Chapter 8.
- The processing of a VALIDATE statement.
- The release of the record from the record buffer.
- The end of the transaction in which the record was created.

PROGRESS updates an index at the end of any statement in which the values for one or more index fields are changed. Because PROGRESS updates indexes immediately (at the end of an UPDATE statement), PROGRESS finds records in the order of the new index while the data in the found record is unchanged. PROGRESS changes the data in the record at the end of the scope of the record or when the record is released.

Note that PROGRESS does not update an index if the value you try to assign to the index is the same as the current value of the index.

3.2.5 Indexes and Unknown Values

If an index contains an unknown value (?), PROGRESS sorts that value higher than any other value. In a unique index, there can be any number of records with unknown values in index fields.

3.2.6 Indexes and Case-Sensitivity

When a field is indexed, the index values are normally stored as all upper-case letters. This ensures that character values are sorted properly without regard to case. For example, the character values "JOHN", "John", and "john" are treated the same. Also, if "JOHN" was already in a unique index, then any attempt to insert "John" would be rejected.

Starting with Version 6, if a case-sensitive field is indexed, the values are stored as exactly as entered. This means that in the above example, "John" would be accepted as a different value. Also, when sorted the uppercase values appear first, then lowercase. So to follow through with the example, "JOHN", "John", and "john" all appear in a different order.

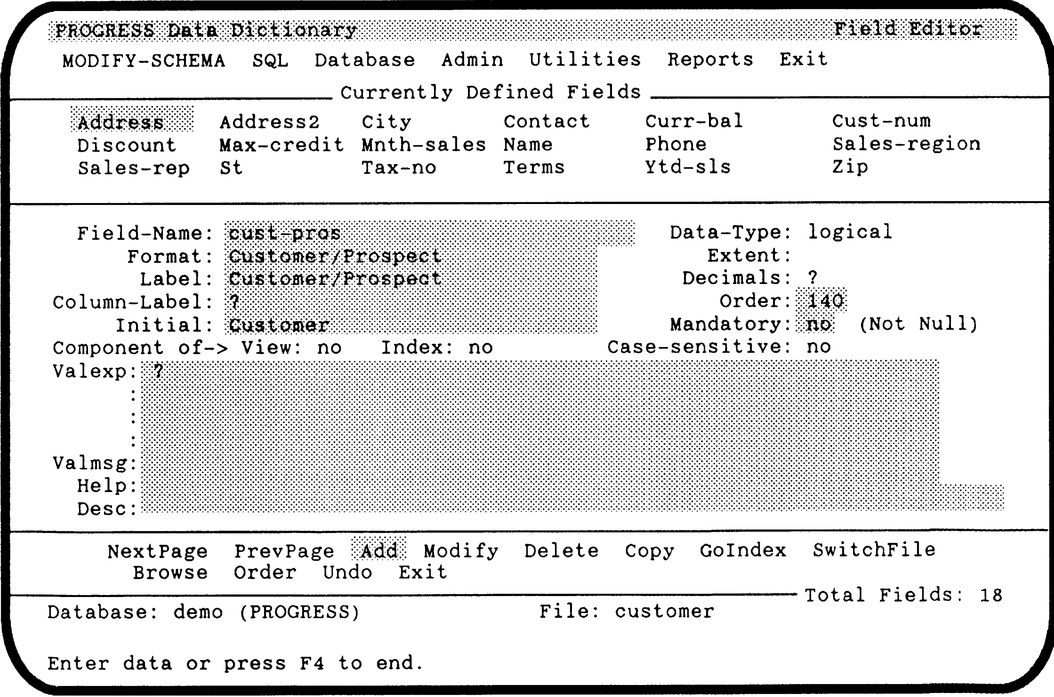
Case-sensitivity is a characteristic of the field not the index. Therefore, if an index contains some fields that are case-sensitive and some that are not, then the different sorting rules apply.

3.3 CHANGING YOUR DATABASE DEFINITIONS

With many database systems it is often very difficult, if not impossible, to go back and change existing database definitions. Not so with PROGRESS. Database definitions are easy to change and changes do not affect the integrity of your database data. Keep in mind however that, if you are working with a very large database, some of these changes may take time to process.

Suppose you have created a database and realize that there is a critical field missing from one or more database files. For example, suppose that in the demo database you want to add a field to the customer file that identifies whether the customer is really a customer or just a hot prospect.

Go ahead and start the Data Dictionary and add the following field definition to the customer file:



After you have created the new field definition, press [GO] (F1) and then leave the Dictionary. Now run the following procedure:

p-adfld.p

```
FOR EACH customer:
  DISPLAY name cust-pros.
END.
```

When you run this procedure, you can see from the display that PROGRESS has added the field to all the existing customer records and has initialized the field to the initial value you defined in the Dictionary.

When you add a field to an existing file, PROGRESS adds that field to all existing records in that file, and initializes the field to the specified initial value. All new records will naturally include the new field.

3.3.1 Changing the Initial Value of a Field

Suppose now that you want to change the initial value of a field. Say you decide that you want all customers to have an initial credit limit of $100. The current initial value of the max-credit field is 0. Change the max-credit field to have an initial value of 100.

```
PROGRESS Data Dictionary                                      Field Editor
 MODIFY-SCHEMA  SQL  Database  Admin  Utilities  Reports  Exit
_________________________ Currently Defined Fields _______________________
   Address       Address2   City        Contact      Curr-bal    Cust-num
   Cust-pros     Discount   Max-credit  Mnth-sales   Name        Phone
   Sales-region  Sales-rep  St          Tax-no       Terms       Ytd-sls
   Zip
__________________________________________________________________________
    Field-Name: max-credit                     Data-Type: decimal
        Format: ->,>>>,>>9                        Extent:
         Label: Max cred                        Decimals: 2
  Column-Label: ?                                  Order: 105
       Initial: 100                            Mandatory: no  (Not Null)
  Component of-> View: no    Index: no    Case-sensitive: no
  Valexp: max-credit >= 0 and max-credit <= 9999999
         :
         :
         :
  Valmsg:Max credit must be >= 0 and <= 9,999,999.99
    Help:Please enter a credit limit
    Desc:Maximum credit
__________________________________________________________________________
      NextPage  PrevPage  Add  Modify  Delete  Copy  GoIndex  SwitchFile
        Browse  Order  Undo  Exit
_____________________________________________________ Total Fields: 18
 Database: demo (PROGRESS)               File: customer

 Enter data or press F4 to end.
```

After you have changed the initial value, press GO (F1) and then leave the Dictionary. Now, what would you want to happen in this situation? You probably want all future customers to have an initial max-credit of 100. But would you want to go and change all the existing customers' max-credit to 100? Probably not. If all those records were changed, any existing credit information would be discarded.

Run this procedure to see what PROGRESS does when you change the initial value of a field:

p-adfld2.p

```
CREATE customer.
SET cust-num.
FOR EACH customer:
  DISPLAY cust-num name max-credit.
END.
```

```
Cust num
     999

Cust num   Name                      Max cred
       1   Second Skin Scuba            1,500
       2   Match Point Tennis           1,970
       3   Off The Wall                   685
                     .
                     .
                     .
     999                                  100
```

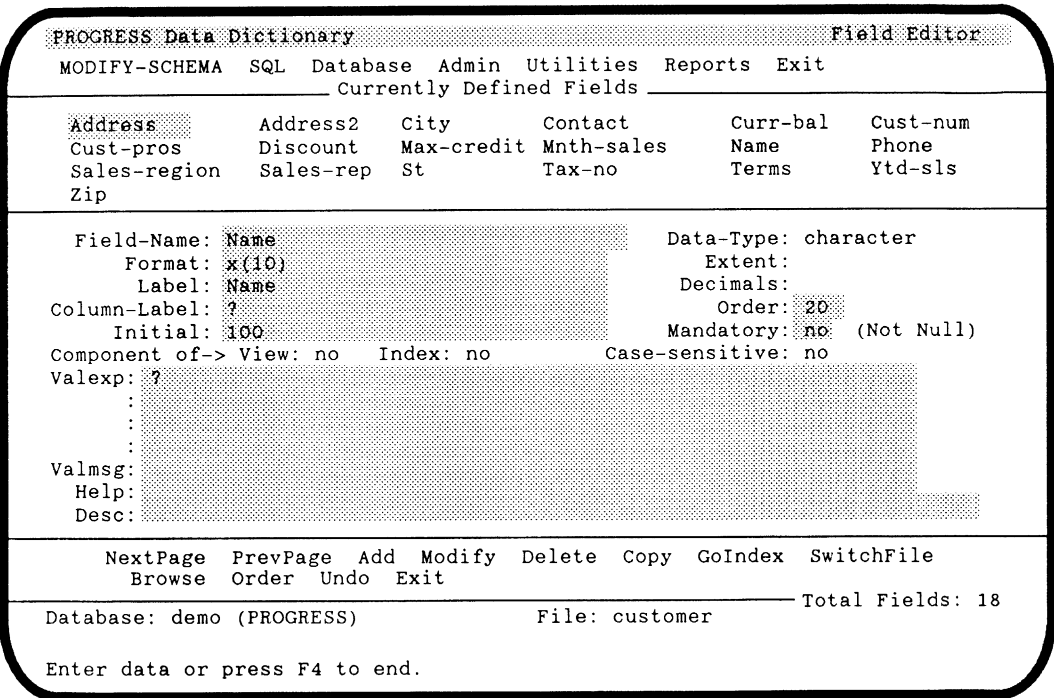
You can see that the `max-credit` value for all existing records was unchanged. But the new customer record has a `max-credit` of 100.

When you change the initial value of a field, no existing records containing that field are changed. However, all new records in the file use the new initial value.

3.3.2 Changing the Format of a Field

It is important to remember that when you define the format of a field, you are simply describing the way you want PROGRESS to **display** that field. The format has nothing to do with the actual data that is stored in the database. Chapter 4 describes display formats in detail.

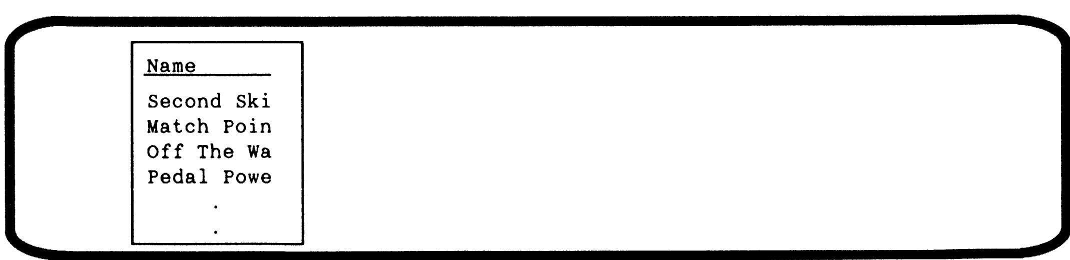
Let's try changing the format of the name field in the customer file. The current format of the field is `x(20)`. Change it to `x(10)`:



After you have changed the format, press GO (F1) and then leave the Dictionary. Now run this procedure to look at the data:

p-adfld.p

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



You can see that the names now take only 10 spaces. But that doesn't mean that the remainder of each name has been removed from the database. Run the following procedure, a modified version of the previous procedure:

p-adfld4.p

```
FOR EACH customer:
  DISPLAY name FORMAT "x(20)".
END.
```

```
Name
Second Skin Scuba
Match Point Tennis
Off The Wall
Pedal Power Cycles
        .
        .
```

Changing the format of a field does not affect the data stored in the database. If you really do want to truncate existing data values, you must write a procedure and use functions such as SUBSTR, ROUND, and TRUNCATE to process the data.

3.3.3 Changing the Data Type or Array Extent of a Field

Imagine for a moment that you want to be able to start using characters in each customer's customer number. That is, instead of using customer numbers such as 1, 2, and so on, you wanted to use number such as 1A and 2X. In order to make this change, you have to change the data type of the field.

Get into the Dictionary and choose the option that lets you update the `cust-num` field of the customer file. You will notice that the Data Type area is not highlighted. This indicates that you cannot change the data type of the field. Why?

Suppose PROGRESS did allow you to change the data type of the field. In many cases (such as changing a character field to an integer field) all the existing data in that field could be invalid. Therefore, to change the data type of a field, you should:

- Create a new field with the appropriate data type.
- Write a procedure to convert the old field data to the new field. For example:

p-adfld5.p

```
FOR EACH customer:
  new-cust-num = STRING(cust-num).
END.
```

  This procedure stores in the new-cust-num field the converted data from the cust-num field. The STRING function converts the data in the cust-num field to character data so PROGRESS can store the data in the new-cust-num field which has a character data type.

- Delete the old field.
- Rename the new field to the name of the old field.

If there is no data in the old field (or none that you care about), you can simply delete the old field and then redefine that field. Be careful when you do this; you may have defined the field or referred to it in other programs.

3.3.4 Changing Index Definitions

You can change the name of an index at any time. You can also delete non-primary indexes. However, suppose you want to delete the primary index. Before letting you delete a primary index, PROGRESS requires that you designate another index as the primary index before deleting the original primary index. If there is only one index, then you must create a new index before you delete the index.

You cannot change any of the component definitions of an index. Instead, you must delete the index and recreate it, using the modified component definitions.

Remember that PROGRESS assumes that the first index you create is the primary index, so try to create your primary index first.

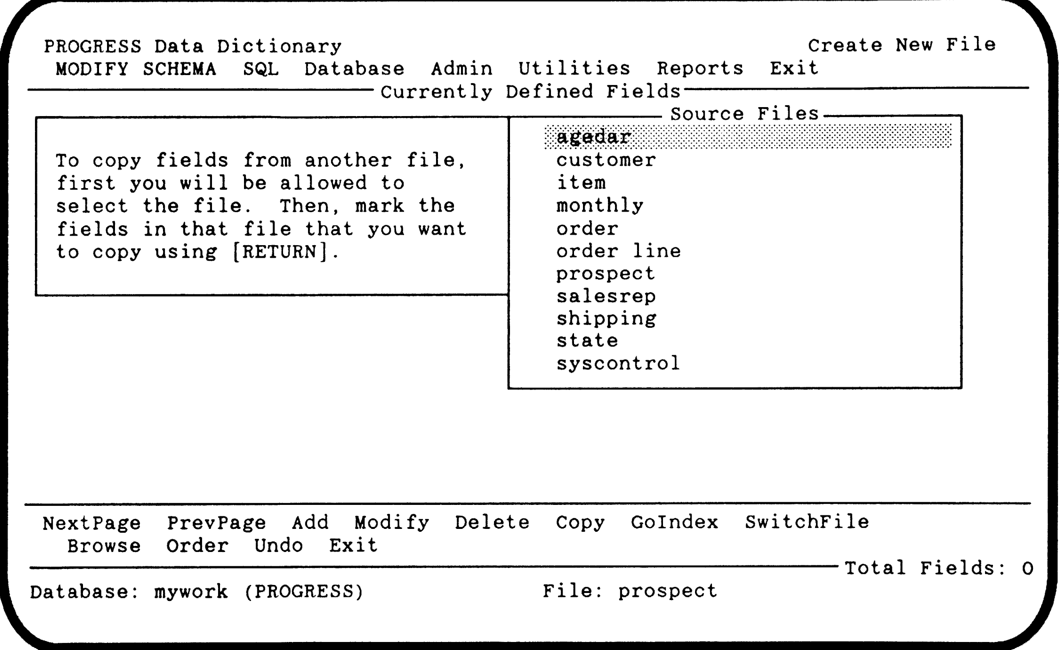
3.3.5 Copying Data Definitions from One File to Another

In the course of developing your application, you may find that you want to create a file that contains the same fields as another file. You can use the Dictionary to easily copy one file's field definitions to another file.

Suppose you want to create a Prospect file to keep track of all potential customers. The structure of this prospect file is basically the same as the customer file. To create a prospect file that looks just like the customer file:

1. Access the Data Dictionary Main Menu and type **m** to select `Modify Schema`. Type **c** to select `Create New File`. Name the new file "prospect" and press GO. A submenu appears at the bottom of your screen.

2. At the submenu, select `Copy` and press RETURN. The following screen appears:



3. Select `customer` as the file whose definitions you want to copy and press RETURN. PROGRESS displays the names of all the fields in customer. Press RETURN after each file name to select all the files you want to copy and press GO. PROGRESS displays the field names as they are copied from `customer` to `prospect`.

4. At the submenu, use `Add` and `Modify` to to make any modifications to the field definitions and use `Delete` to delete those fields you don't want in `prospect`. **Be sure to define appropriate indexes because PROGRESS does not copy index definitions.**

3.4 DEFINING VALIDATION FOR FIELDS

Often when a user is updating a field, you want to make sure valid data is being entered for that field. There are two ways to specify validation criteria for a field:

- Use the VALIDATE format phrase option in the procedure.
- Use the Data Dictionary to specify validation criteria for the field.

There are three common kinds of validation criteria:

- Checking a range of values.
- Checking a value against a list of values.
- Checking the existence of a related record.

PROGRESS validates the user's entry in a field when the user tries to leave that field. Note that PROGRESS checks validation for a field **only** when you update that field.

You are not limited to the three kinds of validation expressions mentioned here. The validation expressions you use can be any expression that results in a logical value and can involve multiple fields and files.

3.4.1 Checking a Range of Values

Suppose that you want to be sure that all customers' `max-credit` values are greater than or equal to 0 but less than a million dollars. You include the following validation criteria in the Data Dictionary definition for the `max-credit` field:

```
PROGRESS Data Dictionary                                              Field Editor
 MODIFY-SCHEMA  SQL  Database  Admin  Utilities  Reports  Exit
──────────────────────── Currently Defined Fields ─────────────────────────────
  Address        Address2  City        Contact      Curr-bal   Cust-num
  Cust-pros      Discount  Max-credit  Mnth-sales   Name       Phone
  Sales-region   Sales-rep St          Tax-no       Terms      Ytd-sls
  Zip
───────────────────────────────────────────────────────────────────────────────
   Field-Name: max-credit                       Data-Type: decimal
       Format: ->,>>>,>>9                          Extent:
        Label: Max cred                          Decimals: 2
 Column-Label: ?                                    Order: 105
      Initial: 100                              Mandatory: no  (Not Null)
 Component of-> View: no   Index: no       Case-sensitive: no
 Valexp: max-credit >= 0 and max-credit <= 9999999
       :
       :
       :
 Valmsg:Max credit must be >= 0 and <= 9,999,999.99
   Help:Please enter a credit limit
   Desc:Maximum credit
───────────────────────────────────────────────────────────────────────────────
    NextPage  PrevPage  Add  Modify  Delete  Copy  GoIndex  SwitchFile
      Browse  Order  Undo  Exit
─────────────────────────────────────────────────────────── Total Fields: 18
 Database: demo (PROGRESS)              File: customer

 Enter data or press F4 to end.
```

The "valexp", or validation expression, in the Dictionary is a logical expression. That is, its result is either true or false. If it results in a true value, the field passes the validation test. If it results in a false value, the field does not pass the validation test and PROGRESS displays the "valmsg" or validation message.

3.4.2 Checking Against a List of Acceptable Values

If you know that, for a particular field, there are a set number of acceptable values, you can put those values into a list and then check input values against that list. You can use the LOOKUP function to check input values against a list.

By now you know that there are three acceptable sales-rep values: BBB, DKP, or SLS. We could put those values into a list and then use the LOOKUP function to test that list in the valexp for the sales-rep field:

```
 PROGRESS Data Dictionary                                        Field Editor
  MODIFY-SCHEMA  SQL  Database  Admin  Utilities  Reports  Exit
 ________________________ Currently Defined Fields ______________________________
   Address        Address2    City        Contact       Curr-bal    Cust-num
   Cust-pros      Discount    Max-credit  Mnth-sales    Name        Phone
   Sales-region   Sales-rep   St          Tax-no        Terms       Ytd-sls
   Zip
 ________________________________________________________________________________
    Field-Name: sales-rep                      Data-Type: character
        Format: !(3)                              Extent:
         Label: Sls rep                         Decimals:
  Column-Label: ?                                  Order: 85
       Initial: ?                              Mandatory: no  (Not Null)
  Component of-> View: no   Index: no      Case-sensitive: no
  Valexp: LOOKUP(sales-rep, "BBB,DKP,SLS") <> 0
        :
        :
        :
  Valmsg:Sales rep must be one of BBB, DKP, or SLS
    Help:Enter initials for a sales rep
    Desc:
 ________________________________________________________________________________
      NextPage  PrevPage  Add  Modify  Delete  Copy  GoIndex  SwitchFile
        Browse  Order  Undo  Exit
 ____________________________________________________________ Total Fields: 18
  Database: demo (PROGRESS)              File: customer

  Enter data or press F4 to end.
```

The LOOKUP function returns the position of the input sales-rep value in the list BBB, DKP, SLS. If you entered SLS, LOOKUP returns a 3. Therefore, if LOOKUP returns a 0, that indicates that the input value does not exist in the list. Here the validation criteria tests to be sure that the LOOKUP returns any value other than 0.

3.4.3 Checking the Existence of a Related Record

In the demo database, you probably want to be sure that, when a new order is created, the customer number entered corresponds to an existing customer record. If you didn't check the existence of the customer record, you could end up with an order record that didn't belong to any particular customer.

Here is the "valexp" for the cust-num field of the order record:

```
PROGRESS Data Dictionary                                        Field Editor
  MODIFY-SCHEMA  SQL  Database  Admin  Utilities  Reports  Exit
_______________________ Currently Defined Fields ___________________________

  Address       Address2   City        Contact     Curr-bal    Cust-num
  Cust-pros     Discount   Max-credit  Mnth-sales  Name        Phone
  Sales-region  Sales-rep  St          Tax-no      Terms       Ytd-sls
  Zip
____________________________________________________________________________
   Field-Name: Cust-num                         Data-Type: integer
       Format: >>>>9                               Extent:
        Label: Cust-num                          Decimals:
Column-Label: ?                                     Order: 20
      Initial: 0                                Mandatory: no   (Not Null)
Component of-> View: no   Index: no        Case-sensitive: no
Valexp: can-find(customer of order)
      :
      :
      :
Valmsg:Customer must already exist
  Help:Enter an existing customer number
  Desc:
____________________________________________________________________________
    NextPage  PrevPage  Add  Modify  Delete  Copy  GoIndex  SwitchFile
      Browse  Order  Undo  Exit
                                                     Total Fields: 18
  Database: demo (PROGRESS)              File: order

  Enter data or press F4 to end.
```

3.4.4 Long Validation Expressions

Suppose you want to write a validation expression that will not fit on the line provided in the Data Dictionary. You can write the expression in an include file. The salesrep.v file is a validation expression for the salesrep field of the customer file. You write and save the validation expression in the PROGRESS editor, following the format for valexp in the Dictionary.

salesrep.v

```
CAN-FIND(salesrep WHERE salesrep.sales-rep = customer.sales-rep)
          AND INPUT st <> "WI"
```

Then you return to the Data Dictionary (customer file, sales-rep field) and enter the name of the include file between curly braces in the valexp field.

```
PROGRESS Data Dictionary                                          Field Editor
  MODIFY-SCHEMA  SQL  Database  Admin  Utilities  Reports  Exit
──────────────────────── Currently Defined Fields ────────────────────────
  Address       Address2   City        Contact      Curr-bal   Cust-num
  Cust-pros     Discount   Max-credit  Mnth-sales   Name       Phone
  Sales-region  Sales-rep  St          Tax-no       Terms      Ytd-sls
  Zip
──────────────────────────────────────────────────────────────────────────
  Field-Name: sales-rep                     Data-Type: character
      Format: !(3)                             Extent:
       Label: Sls rep                        Decimals:
Column-Label: ?                                 Order: 95
     Initial: ?                             Mandatory: no  (Not Null)
Component of-> View: no   Index: no      Case-sensitive: no
Valexp: {salesrep.v}
       :
       :
       :
Valmsg:The sales rep must be one that already exists
  Help:Enter initials for a sales rep
  Desc:
──────────────────────────────────────────────────────────────────────────
    NextPage  PrevPage  Add  Modify  Delete  Copy  GoIndex  SwitchFile
      Browse  Order  Undo  Exit
────────────────────────────────────────────────────── Total Fields: 18
 Database: demo (PROGRESS)               File: customer

 Enter data or press F4 to end.
```

When a user enters information in the `sales-rep` field, PROGRESS uses the `salesrep.v` file to check if the user's entry is valid.

3.4.5 Defining Validation Criteria Using the IF...THEN...ELSE Function

What if you want to define longer and more complex validation criteria than any we have discussed? Here is a procedure that uses the IF...THEN...ELSE function to define complex validation criteria.

p-fldval.p

```
FOR EACH customer:
  UPDATE cust-num name max-credit terms
    VALIDATE(IF (INPUT max-credit > 1000) AND
                LOOKUP(terms,"net30,net45")< > 0 THEN TRUE
             ELSE IF(INPUT max-credit <= 1000) AND
                    LOOKUP(terms,"net30,net60")< > 0 THEN TRUE
             ELSE FALSE,"Terms are incorrect").
END.
```

In this procedure, the user is allowed to update the `max-credit` for a customer if certain terms exist. If the user enters a `max-credit` of more than 1000, the procedure checks to see if the customer has terms of net 30 or net 45. If the user enters a `max-credit` of less than 1000, the procedure checks to see if the customer has terms of net 30 or net 60. If the customer does not have terms that match either statement, the user sees a message that says "Terms are incorrect." The INPUT function on the VALIDATE statement directs PROGRESS to read the value of `max-credit` the user just entered, not the value of `max-credit` stored in the database.

The IF...THEN...ELSE function allows you to create this kind of complex validation expression. You can use the IF...THEN...ELSE function with the VALIDATE statement in a procedure or in the Data Dictionary.

3.4.6 Notes on Defining Field Validation Criteria

In addition to any validate expression you specify, PROGRESS validates field values based on the data type and format of the field. For example, if a numeric format does not provide for a sign, then you cannot put a negative value in the field.

If you change the `valexp` of a field, the Dictionary does not validate the existing field values in the database again. The validation is only applied to data entry statements such as PROMPT-FOR, SET and UPDATE.

PROGRESS makes the following assumptions about valexp:

- If you are validating x with the valexp `x > y`, PROGRESS assumes that x is an input field. However, it does not assume y is an input field. If you want to validate the input value of x against the value of y being entered in the same statement, use the valexp `x > input y`.

- If you name an array in valexp, the element being referred to in the frame is validated. For example,

  ```
  mtd-sales > 0
  ```

  Here, PROGRESS validates the array element being used in the current frame field (`mtd-sales[1] > 0`, `mtd-sales[i] > 0`, etc.).

- If you have used the DEFINE BUFFER statement to define a buffer for a file, PROGRESS assumes, for any valexp, that you mean the buffer for the field being validated. For example, suppose your procedure says

  ```
  UPDATE cusx.
  ```

  and `cusx` is a buffer you have defined for the customer file. If the `max-credit` field in the Dictionary uses the valexp

```
max-credit > 100.
```

PROGRESS assumes that you mean the max-credit field in the cusx buffer. If there are other fields named in the valexp, PROGRESS does not assume that those fields belong to the cusx buffer. Because PROGRESS makes this assumption about the subject field, it is best not to use fully qualified field names in a validate expression. For example if the above valexp were

```
customer.max-credit > 100
```

PROGRESS would not assume that the max-credit field is the one in the cusx buffer.

If you change the validation expression for a field in the Dictionary, PROGRESS does not recompile any object versions of procedures that use that field. To use the changed validation expression, you must recompile the procedures.

3.4.7 Overriding Dictionary Validation Specifications

Just as you can override Dictionary Help specifications in individual procedures, you can override Dictionary validation criteria. For example, here is a procedure that overrides the validation criteria we defined for the customer file in many of the examples above:

p-valid.p

```
FOR EACH customer:
  DISPLAY cust-num name WITH FRAME a.
  UPDATE sales-region sales-rep WITH NO-VALIDATE FRAME a.
  UPDATE max-credit VALIDATE(max-credit > 500,
         "Max-credit must be greater than 500") WITH FRAME b.
END.
```

The NO-VALIDATE option tells PROGRESS to ignore any validation criteria specified for any fields in the frame. Here, the Dictionary validation criteria for the sales-rep field is ignored. To test this, run the procedure and enter "xxx" in the sales-rep field. According to the Dictionary criteria, this is an invalid sales-rep value. But PROGRESS accepts the value because of the NO-VALIDATE option.

The Dictionary validation criteria for the max-credit field state that the max-credit value can be greater than or equal to 0 but less than or equal to 999999.99. The VALIDATE option overrides that criteria and says that the max-credit value must be over 500. Run the procedure and try entering 300 as the max-credit value. According to the Dictionary validation criteria that value is valid. But PROGRESS does not accept the value because of the VALIDATE option used with the max-credit field in the procedure.

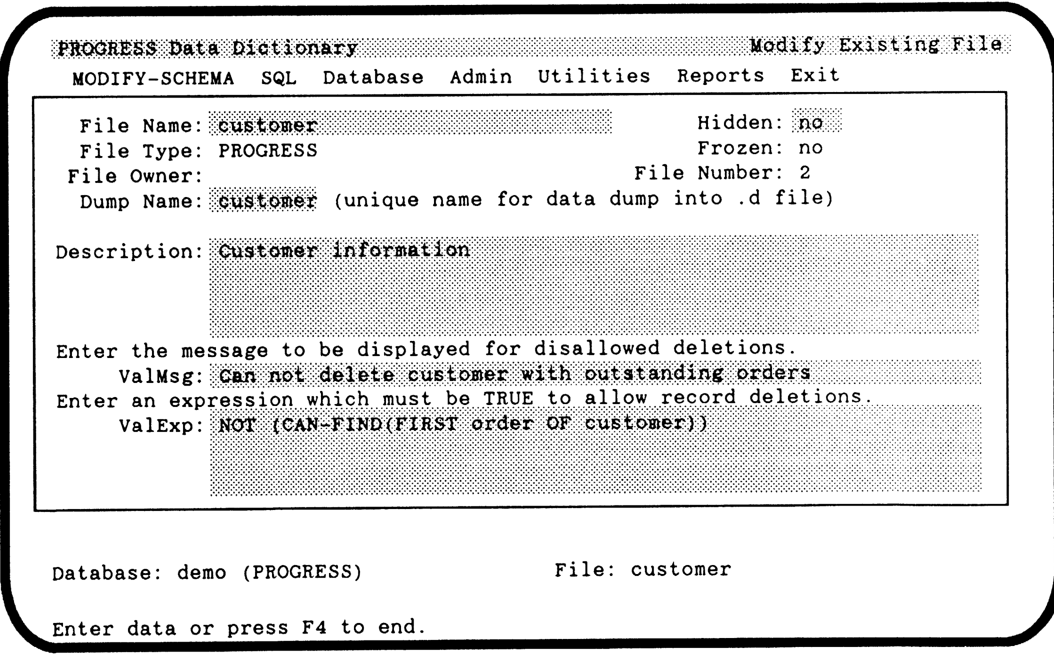
3.5 DEFINING FILE VALIDATION CRITERIA

You can define delete validation criteria for files the same way that you can define validation criteria for fields. Delete validation is criteria that must be met before PROGRESS deletes a record in a file.

Refer to the previous sections on defining validation criteria for a field to learn the different ways to define delete validation for a file (such as using include files). This section describes one way to define delete validation for a file.

When a user deletes a file, you often want to make sure that other associated files are deleted first. For example, if a user tries to delete a customer, you want to make sure that the customer does not have any outstanding orders. You can define delete validation that checks for outstanding orders, sends a message to the user indicating that outstanding orders exist, and prevents the delete.

You can include validation criteria when you create a file or you can add validation criteria for a file at a later time by selecting the "Modify Existing File" option from "Modify-Schema" in the Data Dictionary. Select this option look at the validation criteria in the valexp field for the customer file.



If a user tries to delete a customer, PROGRESS checks whether the customer has orders. If the customer does have orders, PROGRESS displays the message "`Cannot delete customer with outstanding orders`." If the customer does not have orders, PROGRESS deletes the customer.

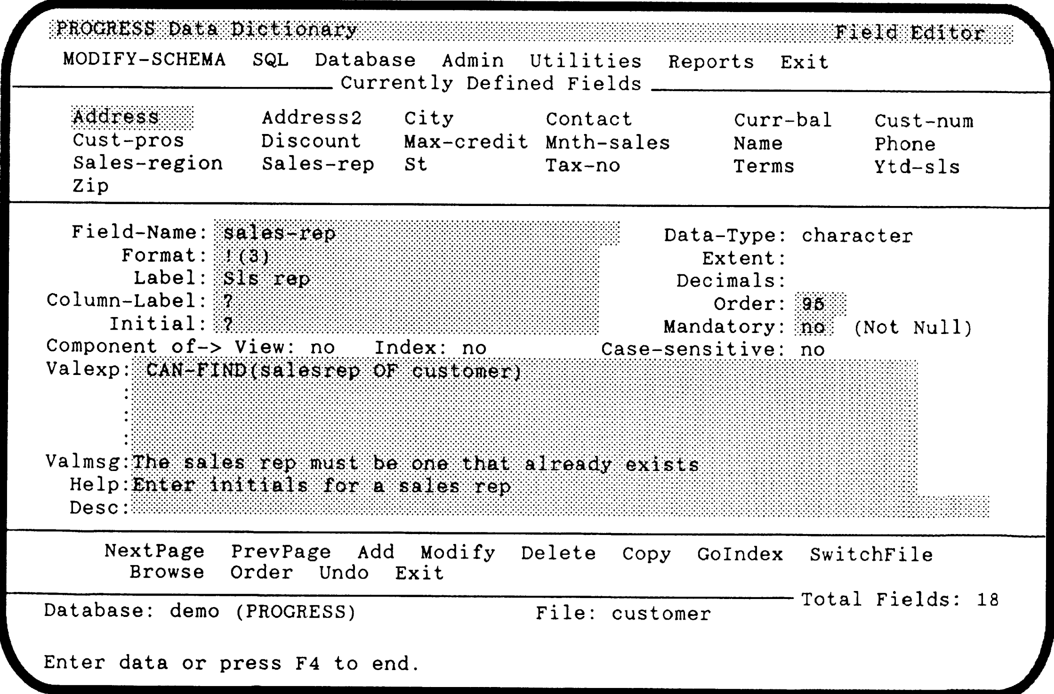
3.6 DEFINING HELP FOR A FIELD

It is often useful to provide field-specific help information to a user. That way, when the user is asked to enter data into a field, there is some information available as to what kind of data is appropriate. For example, if you prompt the user for a value for the sales-rep field in the customer file, it may not be obvious that the initials of the sales rep rather than the first name or last name should be entered.

There are several ways you can provide help information about a field:

- Use a very descriptive label. This is not always possible because of space limitations.
- Include information about that field in your applhelp procedure. For more information about writing an applhelp procedure, see Chapter 15 of the *PROGRESS Language Tutorial*.
- Use the HELP Format phrase option in the procedure.
- Use the Data Dictionary to define help for the field.

Access the Dictionary and choose the option that lets you modify the sales-rep field in the customer file:

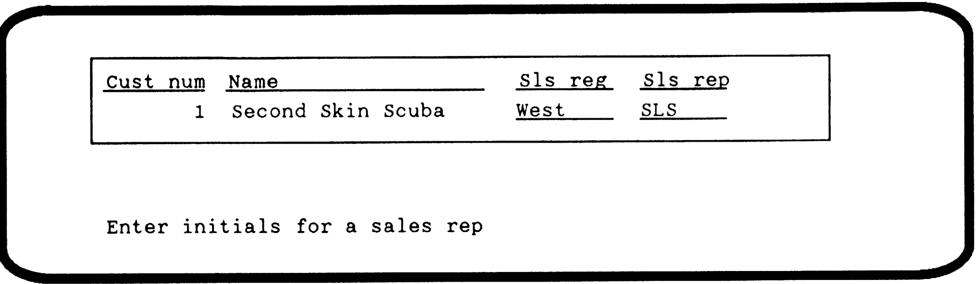


Any time the user updates the sales-rep field, PROGRESS displays the help message "Enter initials for a sales rep". Run this procedure:

p-help.p

```
FOR EACH customer:
  DISPLAY cust-num name.
  UPDATE sales-region sales-rep.
END.
```

On the first customer, tab over to the sales-rep field and notice the help message that PROGRESS displays at the bottom of the screen.



3.6.1 Overriding Dictionary Help Specifications

You can override the Dictionary help specification by using the HELP option in a PROMPT-FOR, SET, or UPDATE statement:

p-help2.p

```
FOR EACH customer:
  DISPLAY cust-num name.
  UPDATE sales-region sales-rep
    HELP "Enter one of SLS, BBB, or DKP".
END.
```

Chapter 4
Display Formats

PROGRESS uses a display format to determine how to display a field or variable either for a simple display (DISPLAY) or for input (SET, UPDATE, PROMPT-FOR, INSERT). A display format tells PROGRESS how many spaces there should be in the display, whether there should be dollar signs, decimal points, or other special characters. PROGRESS supplies default display formats for any field, variable, or expression you display.

As with all other PROGRESS defaults, you can override these default display formats.

The chapter covers the following topics:

- Determining the default display format.
- Character display formats.
- Numeric display formats.
- Logical display formats.
- Date display formats.
- Time display formats.
- Overriding default display formats.
- Testing display formats.

4.1 DETERMINING THE DEFAULT DISPLAY FORMAT

Table 4–1 describes how PROGRESS determines the default display format for different kinds of expressions.

Table 4–1: Default Display Formats for Expressions

Type of Expression	Default Format
Field	Format from the Dictionary.
Variable	Format from variable definition.
Constant character	Length of character string.
Other	Default format for the data type of the expression. Table 4–2 shows these default formats.

Table 4–2 shows the default formats for data types.

Table 4–2: Default Formats for Data Types

Data Type of Expression	Default Format
Character	`x(8)`
Date	`99/99/99`
Decimal	`->>,>>9.99`
Integer	`->,>>>,>>9`
Logical	`yes/no`

Now, take a look at this procedure:

p-ftchp.p

```
     DEFINE VARIABLE nsalesrep AS CHARACTER.
     DEFINE VARIABLE answer AS LOGICAL.

     REPEAT:
1.     PROMPT-FOR customer.sales-rep
         LABEL "Sales rep's initials" WITH FRAME a.
       FOR EACH customer WHERE customer.sales-rep =
                                  input customer.sales-rep:
2.       DISPLAY name WITH 1 DOWN.
3.       SET answer LABEL
           "Change the sales rep on this account?"
           WITH FRAME b SIDE-LABELS NO-BOX.
         IF answer THEN DO:
4.         SET nsalesrep
             LABEL "Enter new sales rep's initials"
             WITH FRAME c SIDE-LABELS NO-BOX.
           customer.sales-rep = nsalesrep.
         END.
       END.
     END.
```

In this example, there are four spots where PROGRESS must display data on the screen. Here is how display formats are determined for each of those:

1. customer.sales-rep is a field. According to Table 4-1, when displaying fields, PROGRESS uses the format from the Dictionary. The Dictionary format defined for the customer.sales-rep field is !(3). This format indicates that there are three alphabetic characters in the field and that lowercase characters are converted to uppercase as you enter them. Based on this definition, PROGRESS displays a field large enough to hold three characters.

2. name is a field. The Dictionary format defined for the field in the customer file is x(20). This format indicates that there can be up to 20 characters displayed for this field. Based on this definition, PROGRESS reserves a frame field large enough to hold 20 characters.

3. answer is a variable. According to Table 4-1, PROGRESS takes the display format from the variable definition. Since the DEFINE VARIABLE answer statement does not specify a format, PROGRESS must use the format associated with the data type of the variable. Table 4-2 shows the default formats for each of the five PROGRESS data types.

   The data type of the answer variable is logical. The default format for a logical variable is yes/no. PROGRESS displays the variable, leaving enough room for either a yes or no answer.

4. nsalesrep is a variable. Again, the DEFINE VARIABLE nsalesrep statement contains no format definition. Therefore, PROGRESS uses the default display format for a character data type, x(8).

The display format of a field simply tells PROGRESS how to display the data in that field: it has nothing to do with the actual data that is stored in the database. For example:

p-ftchp2.p

```
DEFINE VARIABLE test AS CHAR FORMAT "x(10)".
REPEAT:
  SET test FORMAT "x(20)".
  DISPLAY test WITH FRAME aaa.
  DISPLAY test FORMAT "x(20)" WITH FRAME bbb.
END.
```

Here, the test variable is defined to have a format of x(10). The SET statement lets you supply up to 20 characters. The two DISPLAY statements show how the display format of a field or variable does not affect the value that PROGRESS actually stores. The first DISPLAY statement uses the default display format defined in the DEFINE VARIABLE statement. So, no matter how many characters you supplied to the SET statement, this first DISPLAY statement displays only 10 of those characters. The second display statement can display up to 20 characters. If you supplied 15 characters, the first DISPLAY statement makes it appear as if only 10 of those have been stored, but the second DISPLAY statement shows that PROGRESS has stored all 15 characters.

To test different display formats on your own, you can use the datafmt.p procedure supplied with PROGRESS in the prodemo subdirectory. To use this procedure, press GET (F5) and type datafmt.p as the name of the procedure (you may have to unpack it first). Then press GO (F1) to run the procedure.

4.2 CHARACTER DISPLAY FORMATS

The following figure shows the syntax you use to specify the format of a character field or variable.

```
{ { X        } [(n)] } ...
{ { N        }       }
{ { A        }       }
{ { !        }       }
{ { 9        }       }
{ fill character     }
```

The default display format for character fields is x(8).

The following are descriptions of each of the format symbols you use when you define a format for a character field or variable. PROGRESS uses these symbols as constraints when the user is entering data into the field or variable. They have no explicit effect on the appearance of data that is only being displayed.

In these constraint definitions, a digit means 0 through 9. On UNIX, BTOS/CTOS, DOS, and OS/2 a letter means a–z, A–Z and foreign alphabetic characters. On VMS a letter means a–z and A–Z.

X — This character represents any character.

N — This character represents a digit or a letter. A blank (space) is not allowed.

A — This character represents a letter. A blank is not allowed.

! — This character represents a letter that is converted to uppercase during input. A blank is not allowed.

9 — This character represents a digit. A blank is not allowed.

(n) — A number indicating how many times to repeat the previous format character. For example, !(5) is the same as !!!!! and represents 5 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***.

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 non-fill character, you must precede it with a tilde (~). If you use these 5 characters as fill characters in a format specification in a procedure, then enter two tildes so that the PROGRESS editor will interpret the tilde literally and not as an escape lead-in for the following character (e.g. specify a format of x999, where the x is a fill character, as FORMAT "~~x999").

Fill characters are not stored in the database; they are supplied when the field is formatted for display. This permits a field to be displayed 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 blank is truncated to the null value. You can use the TRIM function to truncate leading and trailing blanks.

Table 4–3 shows some examples of how PROGRESS displays a character value using different formats.

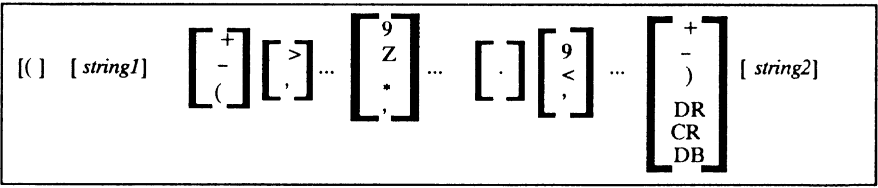
Table 4–3: Character Display Format Examples

FORMAT	VALUE IN FIELD	DISPLAY
xxxxxxxx	These are characters	These ar
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
!!!-9999	abc1234	abc-1234 (1)
(999) 999-9999	6176635000	(617) 663-5000

1) If you enter the characters abc1234 into a field defined with the format !!!-9999, then PROGRESS displays ABC-1234 on the screen and stores ABC1234 in the field. The hyphen is a fill character that is displayed only when the data is displayed in this particular format.

4.3 NUMERIC DISPLAY FORMATS

As with character data, the format you specify for a numeric value determines how PROGRESS displays that value, but not how it stores that value. The following figure shows the syntax you can use when specifying the format of a numeric field or variable:



( )
: 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 negative 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. The position of the plus sign in the number format can also hold a digit if the number is positive.

-	When you use this character to the left of the decimal point, PROGRESS replaces this character 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 pulled 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 one of > or Z. 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.
9	PROGRESS replaces this character with a digit, including cases where the digit is a leading zero.
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 4–4.
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 to be 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 4-4 shows some examples of how PROGRESS displays numeric values using different formats:

Table 4–4: Numeric Display Format Examples

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
$>,>>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

1. This display value is right-justified if it has a column label, left-justified if 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 simply too large to fit in the display format of $ >, > >9.99. The result is a string of question marks.

4. "Floating decimal" display format. The < symbols must follow the decimal point and be balanced by an equal or greater number of > symbols.

NOTE: If you use the European Numeric Format (-E) startup option, PROGRESS interprets commas as decimal points and decimal points as commas when displaying or prompting for numeric values. However, formats in procedures and the Data Dictionary are always entered as described above.

4.4 LOGICAL DISPLAY FORMATS

Logical fields or variables can contain a value that is either true or false. You can define any strings to represent those true and false values. For example, you might define a logical variable whose format is on/off. Here, on represents the true value, while off represents the false value.

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 4–5 shows some examples of how PROGRESS displays a logical value with different formats.

Table 4–5: Logical Display Format Examples

FORMAT	TRUE	FALSE
yes/no	yes	no
Yes/no	Yes	no
true/false	true	false
shipped/waiting	shipped	waiting

4.5 DATE DISPLAY FORMATS

The default display format for a date field or variable is `99/99/99`. Here is the syntax for specifying a date display format:

```
9.../9.../9...    OR    9...-9...-9...    OR    999999    OR    99999999
```

PROGRESS determines where to put the month, day, and year, values based on any Date Format (-d) startup option you may have used. That startup option lets you specify a date format for your application. The default format is `mm/dd/yy`. Table 4–6 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 4–6: 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.

If you enter a single digit for year in a date field, PROGRESS will use the current decade with the single digit.

4.6 TIME DISPLAY FORMATS

The following is the syntax you can use with the STRING function when specifying time formats.

HH:MM:SS[*any characters*]	hh:mm:ss[*any characters*]
HH:MM[*any characters*]	hh:mm[*any characters*]

If you specify an A or a as part of *any-characters* after the basic format, then PROGRESS converts it to a P or p if the time is after 11:59:59 and PROGRESS uses 12-hour time format. Otherwise, PROGRESS uses 24-hour format.

You can use the TIME function to return the number of seconds since midnight. For example:

```
DISPLAY TIME.
```

This statement might return a value such as 60000. To convert the value returned by this statement into the time of day, you use the STRING function. For example,

```
STRING(TIME,"HH:MM:SS AM")
```

Table 4–7 shows some examples of how PROGRESS displays a time value using different formats.

Table 4–7: Time Display Format Examples

FORMAT	VALUE	DISPLAY
hh:mm:ss	60000	16:40:00
HH:MM:SS AM	60000	4:40:00 PM
hh:mm	60000	16:40
hh:mm am	60000	4:40 pm
hh:mm am	3600	1:00 am

To see the entire result of STRING, you may have to use the FORMAT option when displaying the result of the STRING function. For example:

```
DISPLAY STRING(TIME,"HH:MM:SS AM") FORMAT "x(12)".
```

Because STRING is a function, the default display format is "x(8)" and any characters after the first eight are dropped if you do not supply an explicit format.

4.7 OVERRIDING DEFAULT DISPLAY FORMATS

To override the default display format, use the FORMAT option with the field, variable, or expression. For example:

p-ftchp3.p

```
DEFINE VARIABLE nsalesrep AS CHARACTER FORMAT "x(3)".
DEFINE VARIABLE answer AS LOGICAL FORMAT "Change/Keep".

REPEAT:
  answer = false.
  PROMPT-FOR customer.sales-rep
    LABEL "Sales rep's initials" WITH FRAME a.
  FOR EACH customer WHERE customer.sales-rep =
                          input customer.sales-rep:
    DISPLAY name FORMAT "x(30)" WITH 1 DOWN.
    UPDATE answer LABEL "Change the salesrep on this account?
           (change/keep)" WITH FRAME b SIDE-LABELS NO-BOX.
    IF answer THEN DO:
      SET nsalesrep LABEL "Enter new sales rep's initials"
          WITH FRAME c SIDE-LABELS NO-BOX.
      customer.sales-rep = nsalesrep.
    END.
  END.
END.
```

In this example, the `DEFINE VARIABLE nsalesrep` statement uses the FORMAT option to define a display format of `x(3)` for the `nsalesrep` variable. (The default format for a character variable is `x(8)`.) The `DEFINE VARIABLE answer` statement uses the FORMAT option to define a display format of `Change/Keep`. (The default format for a logical variable is `yes/no`.) Also, the `DISPLAY of name` specifies a format of `x(30)`, overriding the format for the `name` field that was in the Data Dictionary.

4.8 TESTING DISPLAY FORMATS

To test different display formats on your own, you can use the `datafmt.p` procedure supplied with PROGRESS in the Procedure Library. To use this procedure, press (F2) and choose the option for access to the Procedure Library, then go to the Convert Library.

Chapter 5
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:

1. You can use blocks to apply a set of characteristics to a group of statements rather than having to do it on a statement by statement basis.
2. PROGRESS provides a wide range of automatic processing services for blocks.

This chapter describes:

- The different kinds of blocks you can use.
- The properties of those blocks.

5.1 BLOCKS AND THEIR PROPERTIES

There are four different kinds of blocks: DO blocks, FOR EACH blocks, REPEAT blocks and Procedure blocks. A Procedure block consists of all the statements within a procedure. PROGRESS provides a different set of default processing services for each type of block. Table 5–1 summarizes these blocks and their default processing services.

Table 5-1: Block Properties

PROPERTY	REPEAT Implicit	REPEAT Explicit	FOR EACH Implicit	FOR EACH Explicit	DO Implicit	DO Explicit	Procedure Implicit	Procedure Explicit
Looping	YES	WHILE TO/BY	YES	WHILE TO/BY	NO	WHILE TO/BY	NO	NO
Record reading	NO	NO	YES	Record-Spec	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	NO
Record Pre-selection	NO	PRESELECT	NO	NO	NO	PRESELECT	NO	NO
UNDO	YES	NO	YES	NO	NO	TRANS-ACTION ON ERROR	YES	NO
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
System Transaction	YES	TRANS-ACTION	YES	TRANS-ACTION	NO	TRANS-ACTION ON ERROR*	YES	NO

(*) Only if DO block contains database updates or reads with exclusive locks.

The next sections explain each of the block properties.

5.2 LOOPING

Looping, or iteration, is one of the automatic processing services that PROGRESS provides for blocks. Specifically, REPEAT blocks and FOR EACH blocks implicitly (automatically) iterate. For example:

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 file
   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.
```

5.3 RECORD READING

Only one kind of block, the FOR EACH block, implicitly reads records each time it iterates. The records it reads are those in the file or files you name in the FOR EACH block header. For example:

p-bkchp4.p

```
/* This FOR EACH block reads the records from the customer file
   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-spec 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 max-credit greater than
   500. */
FOR EACH customer WHERE max-credit > 500:
  DISPLAY customer WITH 2 COLUMNS.
END.
```

5.4 FRAMES

Each statement that displays data or requests data to be entered, uses a frame for display or input of that data. A procedure can use many different frames. The frame a particular statement uses to display data is determined by two things:

- First, if the statement explicitly names a frame, the statement uses that frame to display data.
- If the statement does not explicitly name a frame, the display is done using the default frame for the block in which the statement occurs.

5.4.1 Default Frames

PROGRESS automatically creates a frame for REPEAT, FOR EACH, and procedure blocks. That frame is the default frame for the block and is called the **unnamed** frame. If you name a frame in the block header, that frame becomes the default frame for the block and an unnamed frame is not created for that block. That frame is also scoped to the block if the block is the first one to reference the frame. For example:

```
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; in
   stead 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 example, the default frame for the procedure block is unnamed. The default frame for the REPEAT block is frame bbb.

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

Separate frames are created for DO blocks only if a frame phrase is specified 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.

5.4.2 Frame Scope

The scope of a frame is the first block in which that frame is used. For example:

p-bkchp7.p

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

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.
```

REPEAT, FOR EACH and Procedure blocks automatically have frames scoped to them. PROGRESS only scopes frames to DO blocks if you indicate a new frame in a frame phrase on the DO statement.

Every frame is scoped to a block. A frame can be scoped to a block and not be the default frame for that block.

5.4.3 Frame Services

PROGRESS provides these services for each of the frames in a block:

- Advancing
- Hiding and viewing
- Clearing

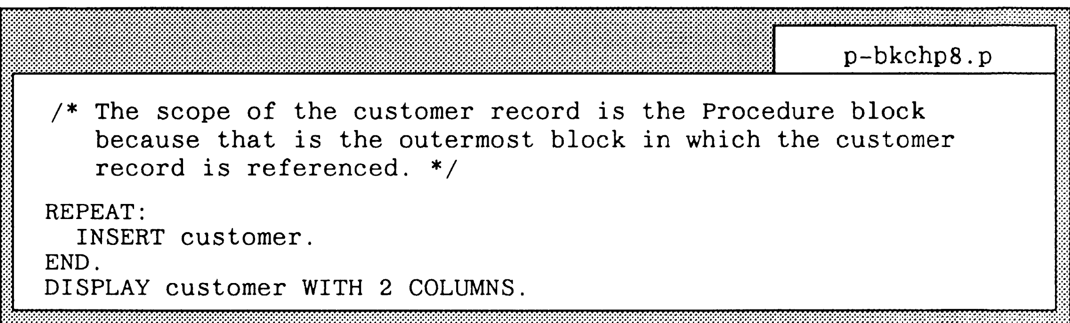
- Retaining previously entered data during RETRY of a block
- Repaint optimization

PROGRESS determines when to provide these services based on the block to which a frame is scoped. For more information, see Chapter 7.

5.5 RECORD SCOPING

When a FIND or FOR EACH statement reads a record from the database, it stores that record in a record buffer. The data in that record buffer is available to a procedure during the **scope** of the record.

By default, the scope of a record is the smallest FOR EACH, REPEAT, or procedure block that encompasses all references to the record.



p-bkchp8.p

```
/* The scope of the customer record is the Procedure block
   because that is the outermost block in which the customer
   record is referenced. */
REPEAT:
  INSERT customer.
END.
DISPLAY customer WITH 2 COLUMNS.
```

PROGRESS automatically scopes records to REPEAT, FOR EACH, and Procedure blocks. You can explicitly scope records to REPEAT and DO blocks by using the FOR option in the block header. For example:

p-bkchp9.p

```
/* The FOR option in the block header explicitly scopes the
   customer record to the DO block. */
DO FOR customer:
  PROMPT-FOR cust-num.
  FIND customer USING cust-num.
  DISPLAY customer WITH 2 COLUMNS.
END.
```

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

- When to write the record to the database. PROGRESS automatically writes a record that has been modified to the database at the end of the record scope. For example:

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 in which the record is
referenced.  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 or upon leaving the block. 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 8 (Transactions and Error Processing) 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 file) to refer to fields that have the same name in other files in the database. You have to include the file name when you reference fields of the same name from two or more files.

5.6 TRANSACTIONS

A transaction is a set of changes to the database, which should either be completely done or should leave no modification to the database. In a multi-user environment, one user's transaction should see a complete set of changes made by another transaction or see none of them. Table 5.2 shows which kinds of blocks start transactions and when they start them.

Table 5-2: Starting Transactions

Type of Block	Transaction Not Active	Transaction Active
DO TRANSACTION FOR EACH TRANSACTION REPEAT TRANSACTION Any DO ON ERROR, FOR EACH, REPEAT, or procedure block containing 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

As you can see from this table, PROGRESS starts "subtransactions" in some cases. 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.

For more information about transactions, see Chapter 8.

5.7 UNDO PROCESSING

There are several ways a transaction or a subtransaction can be undone:

1. You can use the UNDO statement to undo a transaction or subtransaction.

2. There is a procedure-generated error: for example, a record find fails, or a procedure tries to create a duplicate entry in a unique index. In this case, PROGRESS looks for the closest block that has the "error" property and undoes and retries that block. You can override this default processing by using the ON ERROR phrase in the block header and specifying a different action to take in the event of an error.

   Blocks that have the error property by default are: FOR EACH blocks, REPEAT blocks, and procedure blocks.

3. The user presses END-ERROR (F4). If the user presses this key during the first screen interaction (opportunity to enter data) of a block iteration, PROGRESS looks for the closest block that has the "endkey" property and undoes and leaves that block.

4. Blocks that have the endkey property by default are: FOR EACH blocks, REPEAT blocks, and procedure blocks.

5. If the user presses this key after the first screen interaction of a block iteration, PROGRESS looks for the closest block that has the "error" property and undoes and retries that block.

6. The user presses a key that tells PROGRESS to do error processing. There is no default "error" key on the keyboard. However, you can use the ON statement to assign a key as the error key. If the user presses that key, PROGRESS looks for the closest block that has the "error" property and undoes and retries that block. You can override this default processing by using the ON ERROR phrase in the block header and specifying a different action to take when the error condition occurs.

7. The user presses a key that tells PROGRESS to do endkey processing. There is no default "endkey" key on the keyboard. However, you can use the ON statement to assign a key as the endkey key. If the user presses that key, PROGRESS looks for the closest block that has the "endkey" property and undoes and leaves that block. You can override this default processing by using the ON ENDKEY phrase in the block header and specifying a different action to take when the endkey condition occurs.

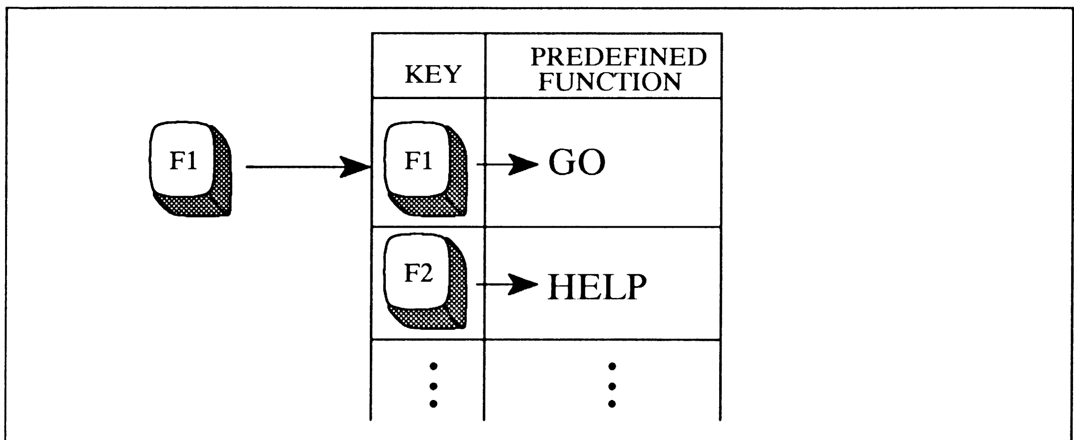
8. The user presses STOP (CTRL-BREAK on DOS and OS/2; CTRL-C on many UNIX and VMS systems; Action-Cancel on BTOS/CTOS). After the current transaction is undone, control then returns to the startup procedure if one is still active (if you started PROGRESS with the -p startup option), otherwise to the editor.

9. Following a system hardware or software failure (a "crash"), any partially completed transactions are backed out for all users. This is the fundamental difference between transactions and subtransactions: a partially completed transaction is backed out after a system failure, including work done in any completed or uncompleted subtransactions that are encompassed within the transaction. Subtransactions can be undone in cases 1 through 5 above without undoing previously completed subtransactions that are all within one transaction.

For more information about transactions and error processing, see Chapter 8.

Chapter 6
Monitoring and Controlling Data Entry

When you run a PROGRESS procedure, PROGRESS processes each of your keystrokes. That is, when you press a key while running a procedure, PROGRESS immediately performs the defined function of that key.



Sometimes you want to know right away when a user has pressed a key. For example, every time the user presses a certain key, you may want to display a message or take some special action. This is called "monitoring" a user's keystrokes. PROGRESS provides a set of keystroke tools to do this.

This chapter:

- Reviews the keys you normally use when running a PROGRESS procedure and how PROGRESS handles each of those keys.
- Explains how you can redefine the functions of different keys and how you can include in a procedure statements that monitors keystrokes.

In this chapter, the focus is on data that you enter from the keyboard. Chapter 9 explains how to write procedures that accept input from a file.

6.1 USING KEYS WHILE RUNNING A PROCEDURE

By now you are probably familiar with what different keyboard keys do while you are running a procedure. Table 6-1 lists keys and their functions.

Table 6-1: Keys You Can Use While Running Procedures

Key Function	Standard Keyboard Key	Standard Control Key			Standard Function Key		
		DOS & OS/2	UNIX, VMS	BTOS/ CTOS	DOS OS/2	UNIX, VMS	BTOS/ CTOS
ABORT		Ctrl-Alt-Del	\ *	ACTION-/			
BACKSPACE	BACKSPACE						
BACK-TAB	SHIFT-TAB (DOS & OS/2)	Ctrl-U	Ctrl-U	CODE-U			CODE-TAB
CLEAR		Ctrl-Z	Ctrl-Z	CODE-Z	F8	F8	F8
CURSOR-UP	↑	Ctrl-K	Ctrl-K				
CURSOR-DOWN	↓	Ctrl-J	Ctrl-J				
CURSOR-LEFT	←						
CURSOR-RIGHT	→	Ctrl-L	Ctrl-L				
DELETE CHARACTER	DEL *** DELETE						
END-ERROR	ESC (DOS & OS/2) *** CANCEL or FINISH	Ctrl-E	Ctrl-E	CODE-E	F4	F4	F4
GO		Ctrl-X	Ctrl-X	CODE-X	F1	F1	F1
HELP	HELP	Ctrl-W	Ctrl-W	CODE-W	F2	F2	F2
HOME	HOME		ESC-H				****
INSERT MODE	INSERT	Ctrl-T	Ctrl-T	CODE-T	F3	F3	F3
RECALL		Ctrl-R	Ctrl-R	CODE-R	F7	F7	F7
RETURN	RETURN or NEXT ***	Ctrl-M	Ctrl-M				
STOP		Ctrl-Break	Ctrl-C**	ACTION-CANCEL			
TAB	TAB	Ctrl-I	Ctrl-I				

* - Depends on UNIX stty setting for quit
** - Depends on UNIX stty setting for intr
*** - BTOS/CTOS
**** - CODE-NEXT-PAGE or CODE-PREV-PAGE

NOTE: PROGRESS also supports the LEFT-MOUSE-UP key for applications in windowed environments. For information about about running PROGRESS with windows, see *System Administration I: Environments.*

Let's look at a simple procedure to review a few of these keys:

```
p-intro.p

FOR EACH customer:
  DISPLAY cust-num.
  UPDATE name max-credit sales-rep.
END.
```

Cust num	Name	Max cred	Sls rep
1	Second Skin Scuba	1,500	SLS

To get from one frame field to another, you press TAB or RETURN. If you press RETURN when you are in the last frame field you are updating, PROGRESS continues on to the next statement in the procedure as if you had pressed GO (F1).

Whenever you tab into a field and type any character, PROGRESS clears that frame field and displays the character you typed. If, before typing a character, you use any of the cursor keys to move around within the field, PROGRESS will not clear that field but will simply enter the characters you type. If you want to replace just the first character of a field, move the cursor one character to the right, then move back to the first character and type over that character.

In numeric fields, PROGRESS is automatically in insert mode when you are entering data to the left of the decimal point and in overtype mode when you are entering data to the right of the decimal point.

To clear the value from a frame field, press CLEAR (F8). If you press CLEAR (F8) while in a character field, PROGRESS starts from the cursor position and clears to the end of the field. If you press CLEAR (F8) while in a non-character field, PROGRESS clears the entire field.

To recall the original value of the field, before you made any modifications, use the RECALL (F7) key.

6.2 CHANGING THE FUNCTION OF A KEY

Although PROGRESS has defined functions for several of the keys on your keyboard, you can redefine those keys to perform other functions. In addition, you can assign functions to any of the other keyboard keys.

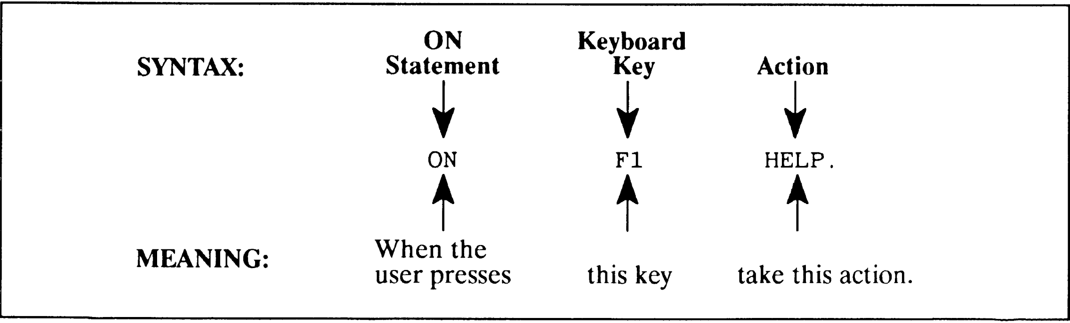
Suppose your users are accustomed to pressing the F1 key to get help information and the F2 key to signal that they are finished entering data. Although PROGRESS usually treats F1 as GO and F2 as HELP, you can switch the functions of those keys to do what your users expect.

p-action.p

```
ON F1 HELP.
ON F2 GO.
ON CTRL-X BELL.

FOR EACH customer:
  DISPLAY cust-num.
  UPDATE name max-credit sales-rep.
END.
```

In this procedure, the ON statements redefine the function of the F1 key, the F2 key, and Ctrl-X. Go ahead and run this procedure. Press F2. You can see that the F2 key now performs the GO function, normally performed by the F1 key. If you press Ctrl-X, PROGRESS rings the terminal bell. The new key definitions are in effect for the duration of the procedure, unless you redefine them. Also, any key label you use in an ON statement must have an entry in the `protermcap` file for the terminal you are using.

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 a user presses that key.



The `p-action.p` procedure uses just three actions: GO, HELP, and BELL. There are many other actions you can use when redefining the function of a key. Table 6–2 lists those actions.

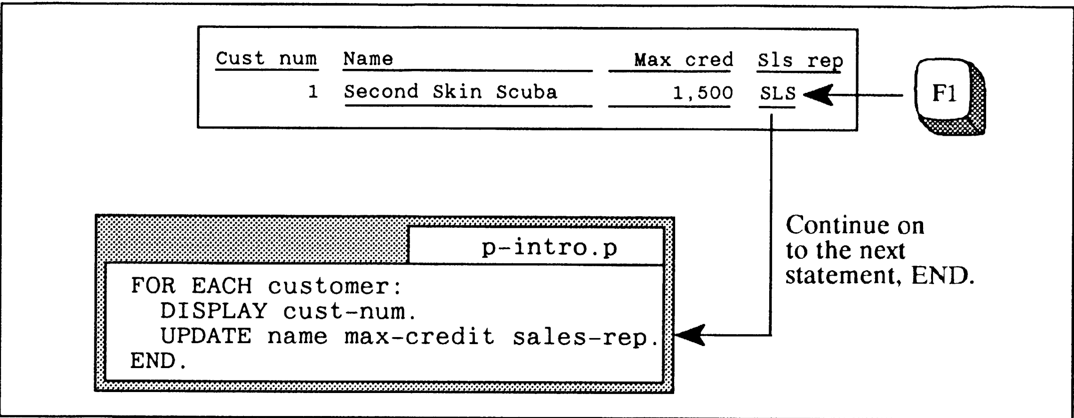
Table 6–2: Actions You Assign to Keys

Actions	Key That Normally Performs this Function
BACKSPACE	BACKSPACE
BACK-TAB	SHIFT-TAB, Ctrl-U, or CODE-TAB
BELL	none
CLEAR	F8, Ctrl-Z, or CODE-Z
CURSOR-UP	↑ or Ctrl-K
CURSOR-DOWN	↓ or Ctrl-J
CURSOR-LEFT	← or Ctrl-H
CURSOR-RIGHT	→ or Ctrl-L
DELETE-CHARACTER	DELor DELETE(BTOS/CTOS)
ENDKEY	none
END-ERROR	F4, Ctrl-E, CODE-E, or, on DOS only ESC
ERROR	none
GO	F1, Ctrl-X, or CODE-X
HELP	F2 or HELP key or Ctrl-W or CODE-W
HOME	HOME or, on UNIX only, ESC-H
INSERT-MODE	F3 or INSERT key or Ctrl-T or OVERTYPE
RECALL	F7, Ctrl-R, or CODE-R
RETURN	RETURN or NEXT (BTOS/CTOS)
STOP	Ctrl-Break (DOS), Ctrl-C (UNIX and VMS) ACTION-CANCEL (BTOS/CTOS)
TAB	TAB or Ctrl-I

6.2.1 Telling PROGRESS How to Continue Processing

When you are modifying a field or variable, pressing GO (F1) (or RETURN on the last field) tells PROGRESS to accept the data in all the fields and variables being modified in the current statement and to go on to the next statement in the procedure.

In the following procedure, if you press F1 while the cursor is in the name, max-credit, or sales-rep field, 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.



You may want to define function keys that tell PROGRESS to continue processing data a certain way. Use the GO-ON phrase with the SET or UPDATE statement to do this. For example:

p-go-on.p

```
DISPLAY "You may update each customer." SKIP
        "After making your changes, press one of:" SKIP(1)
        " F1  - Make the changes permanent" SKIP
        " F4  - Undo changes and exit     " SKIP
        " F9  - Undo changes and try again" SKIP
        " F10 - Find next customer"         SKIP
        " F11 - Find previous customer" WITH CENTERED FRAME instr.

FIND FIRST customer.
REPEAT:
  UPDATE cust-num name address st
         GO-ON(F9 F10 F11) WITH 1 DOWN CENTERED.
  IF LASTKEY = KEYCODE("F9")
  THEN UNDO, RETRY.
  ELSE
  IF LASTKEY = KEYCODE("F10")
  THEN FIND NEXT customer.
  ELSE
  IF LASTKEY = KEYCODE("F11")
  THEN FIND PREV customer.
END.
```

In this example, if the user presses F9, F10, or F11 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 associated with it a code, a function, and a label. The code of a key is an integer value that PROGRESS uses to identify that key. For example, the code of the F1 key 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 is GO. The label of a key is the actual label that appears on the keyboard key. The label of the F1 key is F1. Table 6–3 lists the three PROGRESS functions you use to determine the code, function, or label, of a key.

Table 6–3: Key-Related Functions

FUNCTION	PARAMETER	RETURNS
KEYCODE	*key-label*	An integer keycode.
	EXAMPLE: DISPLAY KEYCODE("F1") RESULT: 301	
KEYFUNCTION	*expression* (value of the expression is an integer keycode)	The PROGRESS function associated with the key.
	EXAMPLE: DISPLAY KEYFUNCTION(301) RESULT: GO	
KEYLABEL	*key-code*	The keyboard label of the key.
	EXAMPLE: DISPLAY KEYLABEL(301) RESULT: F1	

In addition to these functions, the LASTKEY function returns the key code of the last key pressed.

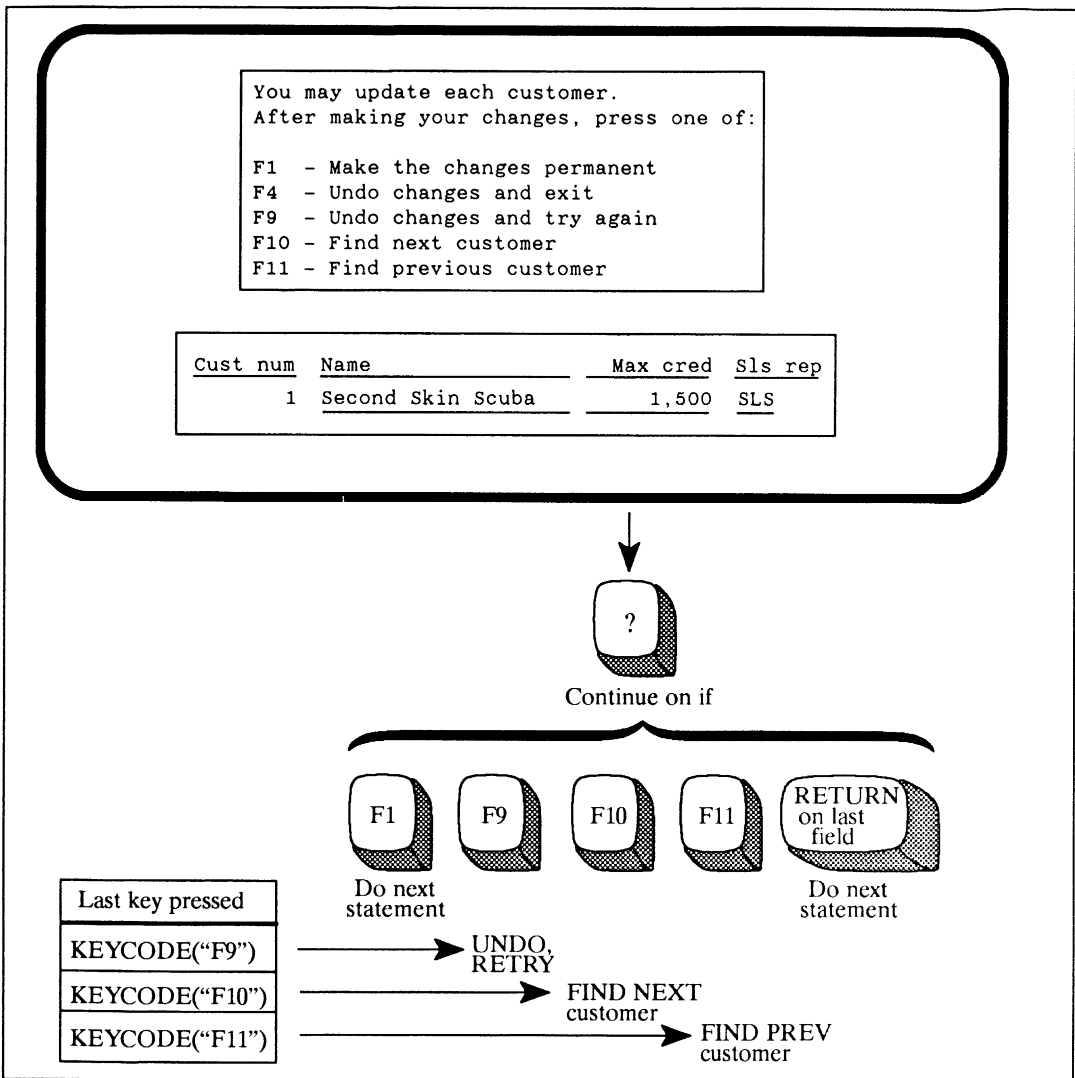
You can use the functions described in this table to monitor the keys being pressed. For example:

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 key functions translate keys you press.

Now, run the p-go-on.p procedure. The following screen is displayed:



If you press F9, F10, F11, or use either of the standard ways of signaling the end of data entry, while updating the customer information, 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 (F4), then the default ENDKEY processing of UNDO, LEAVE is performed. Chapter 8, "Transactions and Error Processing," gives more information on ENDKEY processing.

If PROGRESS does continue on to the next statement in the procedure, a series of IF...THEN...ELSE statements determines what action to take by checking the value of the last key pressed.

6.3 MONITORING KEYSTROKES DURING DATA ENTRY

In the last section, you saw how to check for a certain key as a signal to continue on to the next statement in the procedure. What you were actually doing was monitoring the user's keystrokes to see when the user pressed a certain key.

There are other reasons you might want to monitor a user's keystrokes besides just to signal a continuation to the next statement in a procedure. For example, suppose a user is updating a field and does not know what values are acceptable for that field. You can tell the user to press a certain key to scroll through acceptable values. By monitoring all the user's keystrokes, you will know when the user has pressed that special key.

Performing this detailed processing based on individual keystrokes is an advanced technique which you may find useful in certain situations. Most of the time you will find that the usual statement by statement control that you have seen up to now will do everything you need.

To monitor a user's keystrokes while they are being made, you use an EDITING phrase within a PROMPT-FOR, SET, or UPDATE statement. Within that EDITING phrase, you use the READKEY statement to read each keystroke and then use other PROGRESS statements to take action based on the key that was pressed.

For example:

p-kystrk.p

```
REPEAT:
  PROMPT-FOR customer.cust-num.
  FIND customer USING cust-num.
  UPDATE name SKIP address SKIP city SKIP st SKIP sales-rep HELP
         "Use the space bar to select a sales-rep"
         sales-region WITH SIDE-LABELS
    EDITING:
 ->   READKEY.
      IF FRAME-FIELD <> "sales-rep" THEN DO:
        APPLY LASTKEY.
        IF GO-PENDING
        THEN LEAVE.
        ELSE NEXT.
      END.
      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
                slsrgn @ sales-region.
        NEXT.
      END.
      IF LOOKUP(KEYFUNCTION(LASTKEY), "TAB,BACK-TAB,GO,RETURN,
                END-ERROR") > 0
      THEN APPLY LASTKEY.
      ELSE BELL.
  END.
END.
```

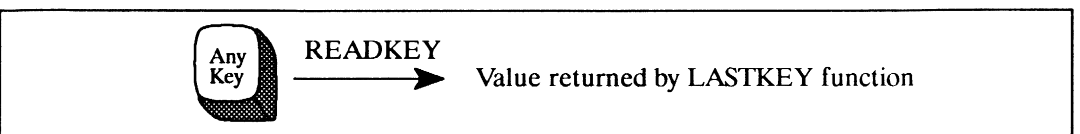
This procedure prompts for a customer number and lets you update information about that customer. While you are updating information, the EDITING phrase monitors your actions. An EDITING phrase is much like a block:

- You follow the word EDITING with a colon (:) rather than with a period (.).
- An EDITING phrase, like a block, can contain any number of statements to be executed during the EDITING phrase.
- An EDITING phrase iterates, or repeats, until the end of the data handling statement being used to modify data.

- You end the EDITING phrase with the END statement.

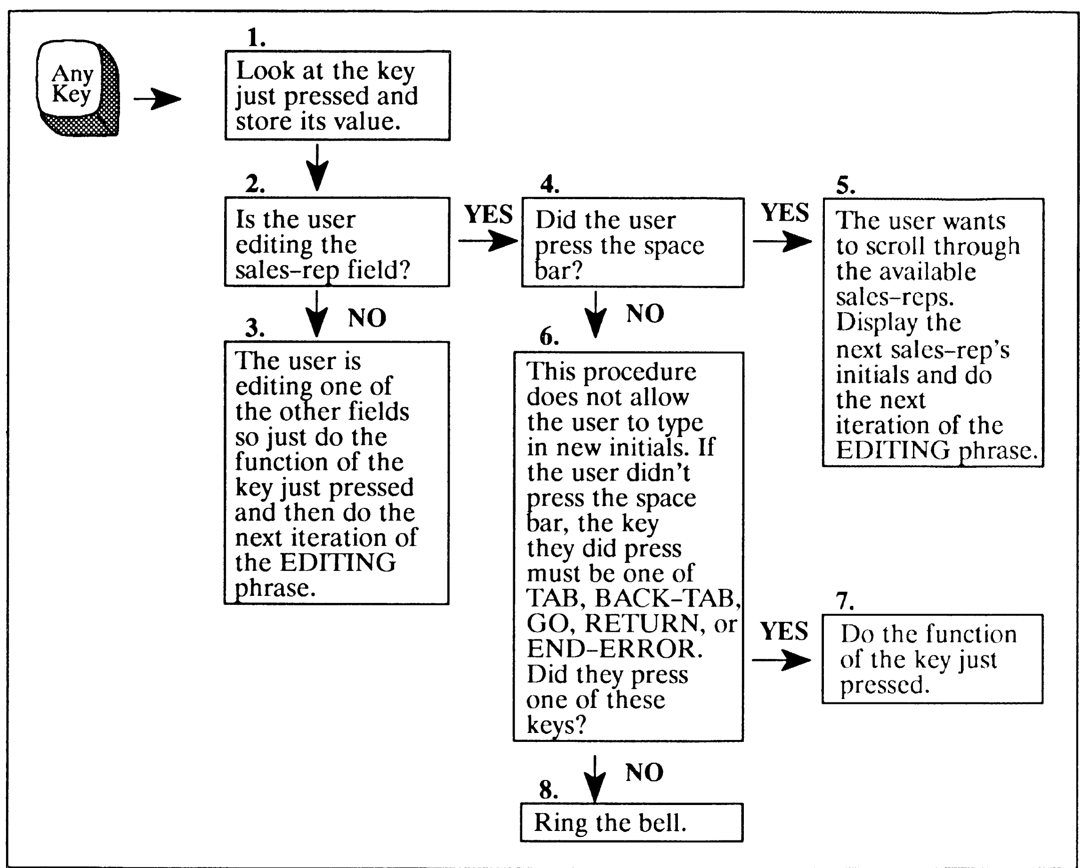
If you are monitoring a user's actions while they are changing data, you are typically interested in monitoring their keystrokes. Therefore, the first statement you usually use in an EDITING phrase is the READKEY statement.

The READKEY statement reads the key just pressed and stores that key as the value returned by the LASTKEY function.



In the `p-kystrk.p` procedure, the user is updating customer information. When the user gets to the `sales-rep` field, you want him to be able either to accept the current `sales-rep` value or to see other possible values for that field. The user should not be able to enter new values for `sales-rep`.

You use an EDITING phrase, in the UPDATE statement, to keep tabs on where the user is in the UPDATE statement and exactly what he is typing. Look back at the EDITING phrase of the UPDATE statement in the `p-kystrk.p` procedure. Here is the work the EDITING phrase does:



In PROGRESS terms:

1. The READKEY statement reads the key just pressed.
2. The FRAME-FIELD function returns the name of the frame field containing the cursor.
3. If your cursor is not currently positioned in the `sales-rep` field, then
   - The APPLY statement applies the last key you pressed. That is, it performs the function of that key. Using the APPLY statement makes it very easy for you to process the standard cases and to then concentrate on the conditions where you want to perform special processing.
   - The GO-PENDING function returns a value of true if, within an editing phrase, an APPLY statement results in a GO action. When you use the LEAVE statement in an editing phrase, PROGRESS leaves the EDITING phrase and does the assignment part of the SET or UPDATE statement.
   - The NEXT statement goes on to the next iteration of the EDITING phrase to read the next keystroke, and cancels any pending GO.
4. If your cursor is currently positioned in the `sales-rep` field, then the LASTKEY function checks to see if the key you pressed was the space bar (" ").
5. The FIND statement reads a record from the `salesrep` file and displays the `sales-rep` field from that record in the `customer.sales-rep` frame field and the `slsrgn` field from that record in the `customer.sales-region` frame field.
6. The NEXT statement goes back to the beginning of the EDITING phrase to read your next keystroke.
7. The KEYFUNCTION function determines the function of the last key you pressed. The LOOKUP function then tests to see if that function is one of TAB, BACK-TAB, GO, RETURN, or END-ERROR. By using KEYFUNCTION we avoid the need to test for individual keys that may all do the same function. For example, by default both F1 and CTRL-X are used to indicate GO; on some terminals additional keys such as EXECUTE may be assigned to GO as well.
8. The APPLY statement performs the function of the key.
9. The BELL statement rings the bell.

You use EDITING phrases to bypass the way PROMPT-FOR, SET, and UPDATE statements handle data entry. Because using EDITING phrases (with READKEY statements) requires extra coding effort, makes your procedures longer, and can slow your system considerably, use EDITING phrases only if you cannot accomplish your objective using standard PROGRESS statements or the GO-ON phrase.

Chapter 7
Frame Design

PROGRESS uses a "frame" to display data. A frame describes the way data appears on a terminal screen or any other output device such as a printer. Statements that display data must use a frame or the PUT statement to display that data. This chapter explains the following topics:

- How statements use frames.
- How frames behave when displaying data.
- How PROGRESS designs frames.
- How you can override default frame designs.

See also the Frame Phrase and various FRAME functions in the *PROGRESS Language Reference.*

7.1 THE PROGRESS SCREEN

Any frames displayed in a procedure can use all but the bottom three lines of the terminal screen. Two of these lines are used for PROGRESS error messages or messages produced by a procedure using the MESSAGE statement, while the last line is used for status and help messages. Most terminals have 24 display lines (leaving 21 lines available for frames); some terminals have 25 display lines (leaving 22 lines available for frames).

The length of any frame you use must be less than or equal to the length of the display area available on the screen. If you think some users might run your procedures on terminals that have 24 lines and some on terminals that have 25 lines, then design your frames to fit on the smaller screen. That way, you can be sure that displays work the way you planned.

Most terminal screens are 80 characters wide, but some are wider. You can take advantage of the entire width of your terminal screen (up to 255 characters) when developing your application. However, be sure to keep in mind the width of the terminal screens on which the application will ultimately be running. For example, if you write the application on a terminal whose screen is more than 80 characters wide, and you take advantage of the full screen width, the application displays will either not fit or will not look the same on a screen with a smaller width.

7.2 HOW PROGRESS DESIGNS FRAMES

There are two levels of frame design:

1. Formatting of specific fields and variables.

2. General frame characteristics.

PROGRESS automatically designs frames at both of these levels. You can override PROGRESS's frame designs at either or both levels.

PROGRESS determines the design of a frame at compile time. As PROGRESS encounters fields, variables, and expressions and their related format specifications in a top to bottom pass of the procedure source, it adds them 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 those fields.

If you are using the DISPLAY statement with an IF...THEN statement, you may get some results you did not expect. Look at this example:

p-frchp.p

```
REPEAT:
  FIND NEXT customer.
  IF max-credit > 500 THEN DO:
    DISPLAY name WITH NO-LABELS NO-UNDERLINE.
    DISPLAY "Extend Special Offer".
  END.
  ELSE DO:
    DISPLAY name WITH NO-LABELS NO-UNDERLINE.
    DISPLAY "Recheck credit".
  END.
END.
```

Because none of the DISPLAY statements names a specific frame, they all use the default display frame for the REPEAT block. When PROGRESS compiles this procedure, it designs a single frame, allocating different areas for each object the procedure might display.

In this example, three different things could possibly be displayed: the customer name, the message "Extend special offer", and the message "Recheck credit". Therefore, PROGRESS reserves one line with space in each of three columns in the frame for each iteration of the REPEAT block.

Here is part of the screen display:

```
Second Skin Scuba     Extend special offer
Match Point Tennis    Extend special offer
Off The Wall          Extend special offer
Pedal Power Cycles                          Recheck credit
Flying Fat Aerobics   Extend special offer
Lift Line Skiing      Extend special offer
Fallen Arch Running   Extend special offer
                                .
                                .
                                .
```

Notice that there are three columns areas on each line allotted for each customer. If a customer has a maximum credit of more than $500, the display takes two columns (the name and "Extend special offer"). If a customer has a maximum credit limit that is less than $500, PROGRESS displays the customer name and "Recheck credit", taking two columns. But regardless of a customer's credit rating, all three columns are available for the display of the customer and unused columns are left blank.

Here is a modified version of the procedure that takes care of this problem:

p-frchp2.p

```
REPEAT:
  FIND NEXT customer.
  IF max-credit > 500 THEN DO:
    DISPLAY name WITH NO-LABELS NO-UNDERLINE.
    DISPLAY "Extend special offer"
      @ msg-area AS CHARACTER FORMAT "x(25)".
  END.
  ELSE DO:
    DISPLAY name WITH NO-LABELS NO-UNDERLINE.
    DISPLAY "Recheck credit" @ msg-area.
  END.
END.
```

In this procedure, PROGRESS creates a frame with enough lines for each customer in the database. The frame has two columns. The customer name is displayed in the first column and "Recheck credit" or "Extend special offer" is displayed in the second column. The procedure defines a special message area called `msg-area` to display one of the two messages. `Msg-area` is a *base-field* with the format `x(25)`. See the *PROGRESS Language Reference* manual for more information about using *base-field* with a DISPLAY statement.

7.2.1 Frame-Level Design

To understand how PROGRESS designs a frame, look at this procedure:

p-frchp3.p

```
DEFINE VARIABLE answer AS LOGICAL LABEL
  "Do you want to delete this customer?".

REPEAT:
  PROMPT-FOR customer.cust-num WITH FRAME a.
  FIND customer USING cust-num.
  DISPLAY name.
  SET answer WITH FRAME a.
  IF answer THEN DELETE customer.
END.
```

Every data handling statement in a procedure serves two purposes:

1. To specify frame contents and layout at compile time.
2. To cause frame activity at run time.

When PROGRESS compiles this procedure, it designs frames as follows:

- The REPEAT block automatically gets a frame to use. 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 big enough to hold the `cust-num` field.
- The DISPLAY statement doesn't name a specific frame so it uses the default frame for the REPEAT block. PROGRESS makes enough room in that frame for the name field.
- The SET statement also names frame a. PROGRESS makes more room in that frame (it initially was only big enough to hold the `cust-num` field) to hold the label "`Do you want to delete this customer?`" and the `answer` variable.

So, 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 the `answer` variable.

Here are the defaults PROGRESS uses when designing a frame:

- Starts the display in column 1, at the first free row on the screen.

- Surrounds the display with a box.
- Displays field labels above fields.
- Underlines labels.
- Makes the frame as wide as is necessary to accommodate all the items in the frame, continuing on to additional lines if everything won't fit on one line.

7.2.2 Field- and Variable-Level Design

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. These labels are determined at compile time. Subscripts are omitted if the array subscript is variable or if a label was specified 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.
- See the DISPLAY statement in the *PROGRESS Language Reference* manual for information about field and variable display formats.

7.3 OVERRIDING DEFAULT FRAME DESIGNS

There are several ways you can override the default design of a frame:

- Use Format phrases with individual fields or variables.
- Use the Frame phrase to specify general frame characteristics.
- Use the FORM statement to specify both field and variable level characteristics as well as frame-level characteristics.
- Use the DEFINE SHARED FRAME statement to specify shared frames.
- Use the CHOOSE and SCROLL statements to create strip menus and scrolling frames.

7.3.1 Format Phrases

Table 7-1 lists the frame field characteristics you can use to describe individual fields or variables. You can use these options with any data handling statement that displays or prompts for data, or with the FORM statement.

Table 7-1: Format Phrase Options

FORMAT PHRASE	PURPOSE	DEFAULT
AT *n*	Starts the frame field in column *n* within the frame.	Starts the display in the next available column.
AS *datatype*	Creates a frame field and variable with the specified data type.	Does not create a new frame field and variable.
ATTR-SPACE	Reserves spaces for field attributes such as underlining and highlighting.	Depends on the type of terminal, unless you specify ATTR-SPACE or NO-ATTR-SPACE in one of several other ways.
AUTO-RETURN	When you fill the field with input data, PROGRESS automatically moves to the next input field.	Does not automatically move out of the field. Use TAB or RETURN to move on to the next field.
BLANK	Tells PROGRESS to display blanks for the frame field instead of the actual value.	Displays the frame field value.
COLON *n*	Positions the frame field so the colon between the label and the data is in column *n*.	The placement of the colon is dependent on the default placement of the label.
COLUMN-LABEL *label*	Names the label you want to display above the field, optionally using more than one line.	Displays the label above the field, using one line.
DEBLANK	Removes leading blanks from character fields on input.	Does not remove leading blanks.

(continued on next page)

Table 7–1: Format Phrase Options (continued)

FORMAT PHRASE	PURPOSE	DEFAULT
FORMAT *string*	Defines the format in which values will be displayed or entered.	A field uses the format in the Dictionary, a variable uses the format of the variable definition, and an expression uses the default format for its data type.
HELP *string*	Defines the help message to be displayed when the user is entering data into the frame field.	PROGRESS displays a help string specified in the Dictionary, if any.
LABEL *string*	Specifies the label for a field, variable, or expression.	Uses the label from the Dictionary or variable definition.
LIKE *field*	Creates a frame field and variable with the same definition as a specified field.	Does not create a new frame field and variable.
NO-ATTR-SPACE	Does not reserve spaces for field attributes such as underlining and highlighting.	Depends on the type of terminal, unless you specify ATTR-SPACE or NO-ATTR-SPACE in one of several other ways.
NO-LABEL	Does not use a label.	Uses the default label.
TO *n*	Positions the frame field so the last position is in column *n*.	The column in which the display ends is dependent on the column in which the display began and the display format.
VALIDATE	Validate the frame field value against specified criteria and displays a message if the validation fails.	Performs no validation unless validation was specified in the Dictionary.

7.3.2 Frame Phrases

Table 7–2 lists the frame phrases you can use to override global frame characteristics. The frame phrase describes overall properties of the frame. A frame phrase option applies to the entire frame even if it appears on only one data handling statement. See the *PROGRESS Language Reference* manual for details on options with the frame phrase.

Table 7–2: Frame Phrase Options

WITH	PURPOSE	DEFAULT
ATTR-SPACE	Reserves spaces for field attributes such as underlining and highlighting.	Depends on the type of terminal, unless you specify ATTR-SPACE or NO-ATTR-SPACE in one of several other ways.
CENTERED	Centers the frame on the terminal screen.	Does not center the frame.
COLOR	Specifies a video attribute or color to use for the background of the frame.	NORMAL. See the *PROGRESS Language Reference* manual for complete syntax on this option.
COLUMN *expression*	Positions the frame in a specified column on the screen.	Positions the frame in column 1.
n COLUMNS	Formats data into *n* columns with side-labels.	Places fields in the frame across the screen, wrapping onto as many lines as needed to accommodate all the fields. Labels are placed above the fields.
DOWN	Displays multiple instances or repetitions of the data in the frame.	Displays multiple instances of the data if the frame is the default frame for the innermost iterating block(s) of the procedure. Otherwise, displays a single instance in the frame.

(continued on next page)

Table 7-2: Frame Phrase Options (continued)

WITH	PURPOSE	DEFAULT
expression DOWN	Displays a specified number of records in a single frame.	Displays multiple instances of the data if the frame is the default frame for the innermost iterating block(s) of the procedure. Otherwise, displays a single instance in the frame.
FRAME *frame*	Names a frame.	Uses the default frame for a block.
NO-ATTR-SPACE	Does not reserve spaces for field attributes such as underlining and highlighting.	Depends on the type of terminal, unless you specify ATTR-SPACE or NO-ATTR-SPACE in one of several other ways.
NO-BOX	Does not display a box around the frame.	Displays a box around the frame.
NO-HIDE	Suppresses the automatic hiding of the frame.	Automatically hides the frame according to a set of default hiding rules.
NO-LABELS	Does not display labels.	Displays labels.
NO-UNDERLINE	Does not underline labels appearing above fields.	Underlines labels appearing above fields.
NO-VALIDATE	Disregards all validation conditions specified in the Dictionary for the fields in the frame.	Validates all fields in the frame according to specifications in the Dictionary.
OVERLAY	Indicates that the frame can overlay any other frame that does not use the TOP-ONLY option.	Does not overlay other frames.
PAGE-BOTTOM	Displays the frame at the bottom of the page each time a page is filled.	Does not display the frame at the bottom of the page each time a page is filled.

(continued on next page)

Table 7-2: Frame Phrase Options (continued)

WITH	PURPOSE	DEFAULT
PAGE-TOP	Displays the frame each time output begins on a new page.	Does not display the frame again on each page.
RETAIN *n*	Retains a specified number of iterations when the frame scrolls.	Does not retain on the screen any iterations already displayed.
ROW *expression*	Positions the frame in a specified row.	Positions the frame in the next available row on the screen.
SCROLL *n*	Displays a scrolling, rather than a paging, frame.	No scrolling.
SIDE-LABELS	Displays field labels to the left of the data.	Displays field labels above the corresponding field.
TITLE	Displays a title as part of the top line of the box around a display frame.	No title.
TOP-ONLY	Indicates that no other frame can overlay this frame.	Other frames can overlay this frame.
WIDTH *n*	Specifies the number of columns in a frame.	Uses a width based on the fields you are displaying and width of the terminal you are using.

7.3.3 The FORM Statement

Often you want to describe characteristics of a frame but don't 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 statement to describe the layout and processing properties of a certain frame. Specifically, you use the FORM statement to:

- Describe frame headers.
- Lay out frame fields in one order when you are going to process them in another order.
- Design a frame that doesn't necessarily match the flow of your procedure.

It is important to remember that the FORM statement just describes the layout of a frame. It does not actually bring that frame into view. To see the frame, you must either use a data handling statement that uses that frame or you must use the VIEW statement.

A FORM statement can cause a frame to be scoped to a particular block, so you must be sure to place any FORM statements you use within the appropriate block or blocks of a procedure.

7.3.4 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 one 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 a shared frame in a FORM statement.

Suppose you write a procedure to display customer information and you write another procedure to update order information. However, you want to display all the information in the same frame. One way to do this is to define a shared frame.

Figure 7-1 shows how two procedures can use the same frame.

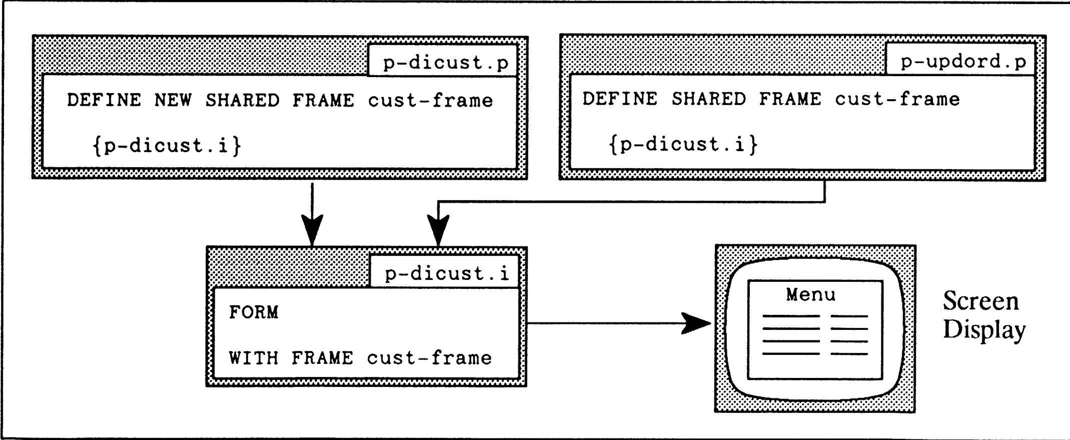


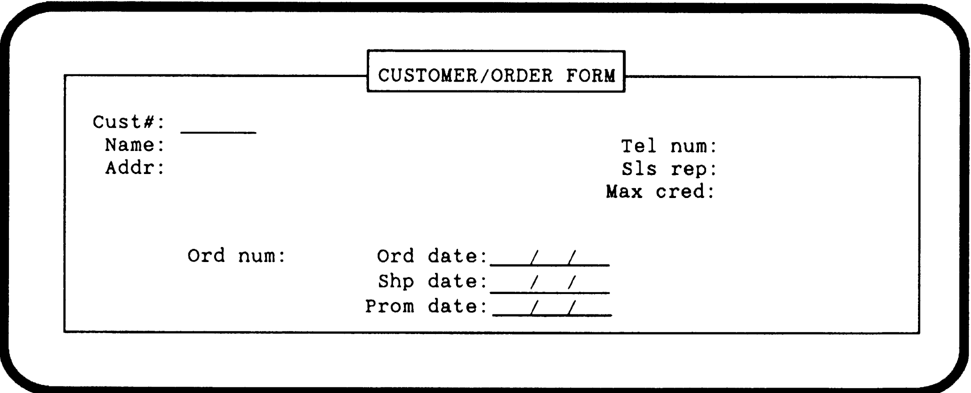
Figure 7-1: Shared Frames

For simplicity, the FORM statement in our 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.max-credit  COLON 50
  SKIP(2)
  order.order-num   COLON 10  order.odate       COLON 30
  order.sdate       COLON 30
  order.pdate       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 just a FORM statement, the file describes the following frame:



Here is the procedure that displays customer information:

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 + ", " + st + " " + STRING(zip) @ csz
          max-credit 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, in which to store customer information. (You can only use 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.

p-updord.p

```
DEFINE SHARED FRAME cust-frame.
DEFINE SHARED VARIABLE csz AS CHARACTER FORMAT "x(22)".
DEFINE 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.odate order.sdate order.pdate 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 in which 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 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 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 describe a shared frame in a FORM statement. All the procedures that use the shared frame must use the same form statement.
- A shared frame is scoped to the procedure (or 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.

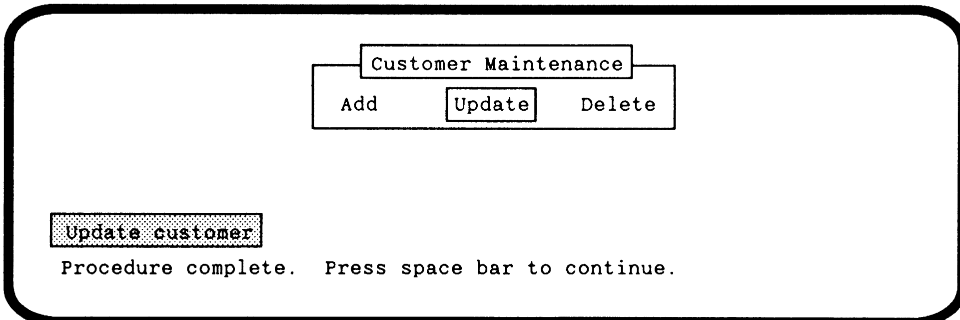
See the *PROGRESS Language Reference* manual for more information on the DEFINE SHARED FRAME statement.

7.3.5 Using Strip Menus

Sometimes you want to create a simple strip menu that allows a user to select an item by moving a highlight bar to the item. Here is a procedure to do this.

```
/* p-choose.p */

DEFINE VARIABLE abc AS character EXTENT 3
  INITIAL ["Add", "Update", "Delete".]

DISPLAY abc NO-LABELS WITH ROW 8 CENTERED
  TITLE "Customer Maintenance" ATTR-SPACE.
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.
```

Run the procedure and select update. You see this screen.



The p-choose.p procedure defines the array abc with an extent of three 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 is holding in the frame. In your own application you write PROGRESS code to 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 so the user can select an action based on information displayed on the screen. For example, look at the following procedure.

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 10.

REPEAT:
  VIEW FRAME instruc.
  DISPLAY menu WITH NO-LABELS ROW 21 NO-BOX ATTR-SPACE
    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.
```

When you run the p-chsmnu.p procedure you see this screen.

```
              Use the sample strip menu to select an action.

        Browse  Create  Update  Exit
        The program brws.p does not exist.
        Please make another choice.
        Press space bar to continue.
```

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, the only item that performs an action is exit. The other selections return a message, like that in the screen pictured above.

Now look back at the p-chsmnu.p procedure. The procedure defines two arrays with an extent of four. One array holds the items for selection on the menu, the other 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 (within the array) associated with the item selected, then the program associated with that number in the proglist array. PROGRESS runs the program if it exists, and displays a message and allows the user to select another item from the menu if the program does not exist.

In your own application, and in the examples below, actions are associated with items selected using the CHOOSE statement. 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.

7.3.6 Using Scrolling Frames

Suppose you want to display some information about every customer in your database, and you want to be able to scroll through all the information, and you want to change, delete, or add customers to the list while it is on your screen. You use the SCROLL statement together with the CHOOSE statement to perform these functions.

The SCROLL *statement* opens a row and moves data in a frame with multiple rows. (The SCROLL *option* on the frame phrase creates a scrolling frame.) You use the SCROLL statement whenever you want to scroll data up or down to display another line in a frame. For example, run the following procedure.

p-chose1.p

```
    DEFINE VARIABLE counter AS INTEGER
    DEFINE VARIABLE oldchoice AS CHARACTER.
    DEFINE VARIABLE null AS CHARACTER INITIAL "".

    FORM customer.cust-num customer.name customer.address
         customer.city customer.zip
1.       WITH FRAME cust-frame SCROLL 1 5 DOWN ATTR-SPACE.

    FIND FIRST customer.
    REPEAT counter = 1 TO 5:
      DISPLAY cust-num name address city zip 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.
    oldchoice = "".

    REPEAT:
      STATUS DEFAULT
        "Use up and down arrows. Enter C to create, D to delete".
2.    CHOOSE ROW customer.cust-num NO-ERROR GO-ON(CURSOR-RIGHT)
        WITH FRAME cust-frame.
      COLOR DISPLAY NORMAL customer.cust-num WITH FRAME cust-frame.

      /* After choice */
      IF FRAME-VALUE = ""
      THEN NEXT.
        /* Force user to press END or move cursor to valid line */
      IF FRAME-VALUE <> oldchoice
      THEN DO:
        oldchoice = FRAME-VALUE.
        FIND customer WHERE cust-num = INTEGER(FRAME-VALUE).
      END.
```

(Continued on next page.)

p-chose1.p (Continued)

```
/* 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 zip 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 zip WITH FRAME cust-frame.
  NEXT.
END.

/*CHOOSE selected a valid key.  Check which key. */

IF LASTKEY = KEYCODE("c")
THEN DO:                     /* Open a space in the frame. */
3.  SCROLL FROM-CURRENT DOWN WITH FRAME cust-frame.
  CREATE customer.
  UPDATE cust-num name address city zip WITH FRAME cust-frame.
  oldchoice = INPUT cust-num.
  NEXT.
END.

IF LASTKEY = KEYCODE("d")
THEN DO:          /* Delete a customer from the database. */
  DELETE customer.
  FIND NEXT customer NO-ERROR.
  /* Move to correct position in database. */
  IF NOT AVAILABLE customer
  THEN DO:
     CLEAR FRAME cust-frame.
     UP WITH FRAME cust-frame.
     NEXT.
   END.
```

(Continued on next page.)

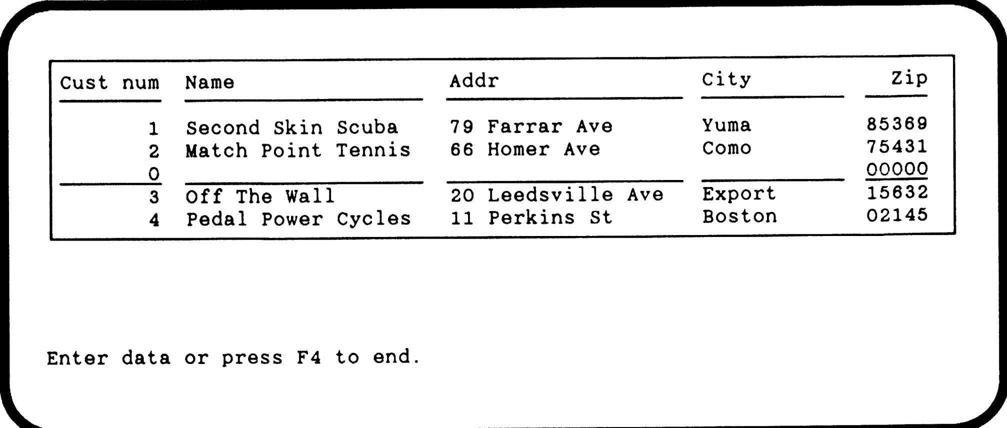
p-chosel.p (Continued)

```
    IF FRAME-LINE(cust-frame) = FRAME-DOWN(cust-frame)
    /* If last screen line deleted. */
    THEN DO:
      DISPLAY cust-num address city zip WITH FRAME cust-frame.
      NEXT.
    END.
4.  SCROLL FROM-CURRENT WITH FRAME cust-frame.
    REPEAT counter = 1 TO 100
    WHILE FRAME-LINE(cust-frame) < FRAME-DOWN(cust-frame).
      FIND NEXT customer NO-ERROR.
      IF NOT AVAILABLE customer
      THEN LEAVE.
      DOWN WITH FRAME cust-frame.
      IF INPUT cust-num = ""
      THEN DISPLAY cust-num name address city zip
                   WITH FRAME cust-frame.
    END.
    UP counter - 1 WITH FRAME cust-frame.
    oldchoice = INPUT cust-num.
  END.
END.
STATUS DEFAULT.
```

When you run the p-chosel.p procedure, you see this screen.

```
Cust num  Name                 Addr                 City        Zip
--------  -------------------  -------------------  ----------  -----
       1  Second Skin Scuba    79 Farrar Ave        Yuma        85369
       2  Match Point Tennis   66 Homer Ave         Como        75431
       3  Off The Wall         20 Leedsville Ave    Export      15632
       4  Pedal Power Cycles   11 Perkins St        Boston      02145
       5  Flying Fat Aerobics  39 Dalton St         Hartfield   14728

Use up and down arrows. Enter C to create, D to delete.
```

Use the arrow keys to move the highlighted bar to the third customer. Press **c** for create.



Cust num	Name	Addr	City	Zip
1	Second Skin Scuba	79 Farrar Ave	Yuma	85369
2	Match Point Tennis	66 Homer Ave	Como	75431
0				00000
3	Off The Wall	20 Leedsville Ave	Export	15632
4	Pedal Power Cycles	11 Perkins St	Boston	02145

Enter data or press F4 to end.

At this point you can add a new customer to the database by typing the customer information on the open line. The `p-chose1.p` procedure uses SCROLL and CHOOSE to allow the user to browse through information, then perform actions with the information.

The `p-chose1.p` procedure creates a scrolling frame of five fields. The frame displays the `cust-num`, `name`, `address`, `city`, and `zip` for each customer. The status default message displays "`Enter C to create, D to delete`" as long as the procedure is running. You use arrow keys to move the highlighted cursor bar through the database, and to add or delete customers from the database. The CHOOSE statement lets you create this style menu.

The SCROLL statement controls the scrolling action in the frame when you create and delete customers. You add a customer to the database by entering "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 enter "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.

Let's look at the way CHOOSE and SCROLL operate in the `p-chose1.p` procedure in more detail. Refer back to the procedure to follow the numbers below.

1. The SCROLL option on the frame phrase creates a scrolling frame for the customer information.

2. 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.

3. When the user presses "c" for create, the SCROLL FROM-CURRENT DOWN statement opens a space for another customer above the highlighted customer.

4. When the user presses "d" for delete, the SCROLL FROM-CURRENT statement closes the space left by the deleted customer.

In the `p-chose1.p` procedure, the code that performs the delete function is complex. You can perform the same function with fewer statements if you do not use the SCROLL statement. You can substitute the procedure segment below into the `p-chose1.p` procedure to perform the delete function.

p-chose2.p

```
                    .
                    .
                    .
IF LASTKEY = KEYCODE("d")
THEN DO:    /* Delete a line from the frame. */
  DELETE customer.
  REPEAT counter = 1 TO 100 WHILE frame-line(cust-frame)
         <= frame-down(cust-frame).
    FIND NEXT customer NO-ERROR.
    IF AVAILABLE customer
    THEN DISPLAY cust-num name address city zip
                 WITH FRAME cust-frame.
    ELSE CLEAR FRAME cust-frame.
    DOWN WITH FRAME cust-frame.
  END.
  UP counter - 1 WITH FRAME cust-frame.
  oldchoice = INPUT cust-num.
END.
                    .
                    .
                    .
```

You can see the entire `p-chose2.p` procedure on-line. This example above shows only the portion that is different from the `p-chose1.p` procedure.

7.4 HOW STATEMENTS USE FRAMES

A statement that displays data uses these rules to determine which frame to use:

- First, if the statement explicitly names a frame in a Frame phrase, the statement uses that frame to display data.

- If the statement does not explicitly name a frame, the display is done using the default frame for the block containing the statement.

PROGRESS automatically provides a default frame to FOR EACH blocks, REPEAT blocks, and procedure blocks. (The default frame used by a DO block is the default frame for the enclosing FOR EACH, REPEAT, or Procedure block unless you explicitly name a frame in the DO block header.) For example:

p-frchp4.p

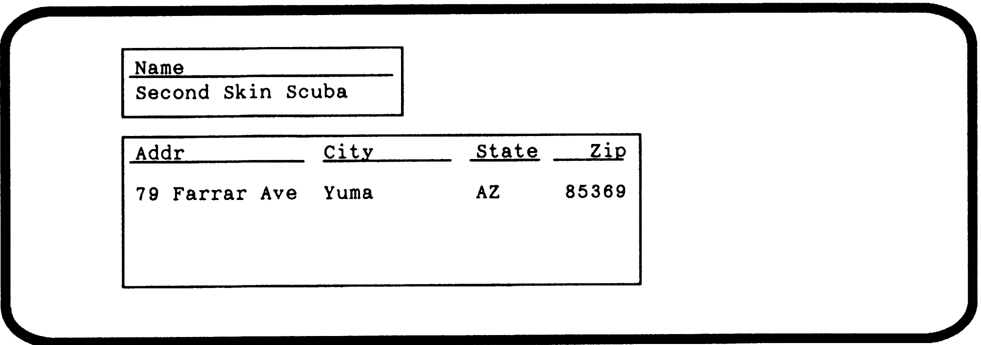
```
FOR EACH customer:
  DISPLAY name address city st zip.
END.
```

In this example, PROGRESS automatically creates a frame for the FOR EACH block. Since the DISPLAY statement does not name a frame, it uses the FOR EACH block's default frame. Look at this example:

p-frchp5.p

```
FOR EACH customer WITH FRAME b:
  DISPLAY name WITH FRAME a.
  DISPLAY address city st zip.
END.
```

In this example, the first DISPLAY statement names its own frame but the second DISPLAY statement does not. Therefore, there are two frames in this procedure, the default frame for the FOR EACH block (FRAME b because that is explicitly named in the FOR EACH block header) and the frame named in the first DISPLAY statement. Here is the output of this procedure:



7.5 PROGRESS FRAME SERVICES

PROGRESS provides these services for each of the frames in a block:

- Viewing and hiding.
- Advancing and clearing.
- Retaining previously entered data during RETRY of a block.
- Screen repaint optimization.

PROGRESS determines when to provide these services based on the scope of a frame.

7.5.1 Frame Scopes

The scope of a frame is the first REPEAT, FOR EACH, Procedure, or DO WITH FRAME block in which that frame is referenced (uses in HIDE statements do not count as frame references in this context). A frame can only be scoped to one of these blocks:

- REPEAT
- FOR EACH
- DO WITH FRAME
- Procedure

For example:

p-frchp6.p

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

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.
```

p-frchp7.p

```
/* The scope of frame aaa is the procedure block because that is
   the first block in which frame aaa is referenced; the scope
   of frame bbb is the REPEAT block.  The DISPLAY statement in
   the REPEAT block uses that frame because it is the default
   frame for the REPEAT block.  (The DISPLAY statement could
   override this by explicitly naming a frame: DISPLAY WITH
   FRAME...) */

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.
```

The FORM statement counts as a use of a frame. For example:

p-frchp8.p

```
/* Now the scope of frame bbb is the procedure block because that
   is the block in which it is first referenced: by the FORM
   statement. */

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.
```

Every frame can be scoped to one and only one block. Also, you cannot reference or use a frame outside its scope except in HIDE and VIEW statements. For example:

p-frchp9.p

```
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.
```

In this procedure, the scope of frame a is the first REPEAT block since that is the first block that references frame a. If you try to run this procedure, you get an 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 by adding a DO WITH FRAME a block that encompasses both the REPEAT and FOR EACH blocks.

Note that a frame defined as SHARED is scoped to the procedure (or block within the procedure) where that frame is defined as NEW SHARED. See the section of this chapter on shared frames for more information about the scope of shared frames.

7.5.2 Viewing and Hiding Frames

PROGRESS brings a frame into view when you display data into that frame. You can also use the VIEW statement to explicitly bring a frame into view.

When you want to display a frame and there is not enough room on the screen, PROGRESS starts from the bottom of the screen and erases (hides) existing frames to make room. If PROGRESS is ready to hide a frame and there has not been any user interaction since data was last displayed in the frame, PROGRESS automatically pauses and displays the message "Press space bar to continue".

If you want to display, in a position where a frame is already displayed, a frame that has the OVERLAY option, PROGRESS displays the frame over the other frame (unless that other frame has the TOP-ONLY option). PROGRESS displays a message before it displays the overlay frame.

You can use the HIDE statement to explicitly hide a frame.

7.5.3 Advancing and Clearing Frames

Frames can either be **single** frames or **down** frames. A single frame (which is the same as a "1 DOWN" frame) means that just one iteration of data is being shown in the frame. A down frame can display multiple iterations of data.

Each frame is 1 down unless it is scoped to, and is the default frame for, the innermost iterating block or an iterating block that does not itself contain iterating blocks with frames of their own. Such a frame is a down frame by default.

When a block iterates, any DOWN frames scoped to the block are advanced to the next display position and the field values in the screen buffer for all 1 DOWN frames scoped to the block are cleared. These actions take place only if data has been displayed or entered 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.

When a block is in RETRY, frames scoped to the block are not advanced and are not cleared.

7.5.4 Retaining Data During Retry of a Block

When a block is being retried, either because of an implicit or explicit UNDO, RETRY, PROGRESS does not clear the frame scoped to that block. That way, the data is still available when the user is entering new data or changing the data entered on the first try.

7.5.5 Repaint Optimization

When a block iterates (but not when it exits), single (1 DOWN) frames scoped to that block or nested blocks are tagged for hiding. The hiding occurs on the first frame activity for a frame scoped to that block or to a nested block that occurs during the iteration of the block. The hiding is optimized to avoid repainting the same frame. You can suppress this hiding by using the NO-HIDE frame phrase.

When a procedure run from another procedure ends or when a block ends, all its frames which are still in view are left on the screen and are then associated with the enclosing block. PROGRESS hides them when that block iterates or when additional space is needed on the screen for display of other frames.

7.6 FRAMES FOR NON-TERMINAL DEVICES

PROGRESS designs frames independently of the output destination of the frame, which may not be determined until the procedure runs, rather than when it is compiled. However, there are special considerations when you design frames that will be displayed on something other than a terminal screen. If you are displaying frames to non-terminal 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 a data handling statement that uses another frame is executed or until a PUT statement is executed. All changes to the frame are collected together and only one copy of the frame is sent to the output destination.

- The ROW frame phrase option is ignored. See the PUT statement in the *PROGRESS Language Reference* manual for more information about controlling the format of output to non-terminal devices.

7.7 SPACE-TAKING AND NON-SPACE-TAKING TERMINALS

There are two different ways a terminal can handle screen formatting. It can either:

- Reserve a character position, on both sides of every field, for special screen field attributes, such as underlining or highlighting. These terminals are called "spacetaking" because they reserve spaces on the terminal screen for these attributes.
- Not reserve a character position for special field attributes. These are called "nonspacetaking" terminals.

Before designing your frames, you should know whether your procedures will be running on spacetaking, nonspacetaking, or both kinds of terminals. Frames you design to run on spacetaking terminals will run on all terminals. However, if you assume that your procedures will always be run on nonspacetaking terminals, frames you design for those procedures may not fit or operate properly on spacetaking terminals. Remember, spacetaking terminals require extra spaces for special characteristics.

You specify that spaces should be reserved in one of four ways:

1. Use the ATTR-SPACE option with the name of a specific frame field.
2. Use the ATTR-SPACE option as part of a frame phrase that describes an entire frame.
3. Use the ATTR-SPACE option with the COMPILE statement.
4. Use a spacetaking terminal.

Regardless of whether you use the ATTR-SPACE option with the COMPILE statement, PROGRESS never reserves extra spaces around a title (that you specified with the Frame phrase) for display attributes such as reverse video and highlighting. If the title is at least two characters shorter than the width of the box and the attributes require extra spaces, PROGRESS can still apply those attributes.

You specify that spaces should not be reserved in one of four ways:

1. Use the NO-ATTR-SPACE option with the name of a specific frame field.
2. Use the NO-ATTR-SPACE option as part of a frame phrase that describes an entire frame.
3. Use the NO-ATTR-SPACE option with the COMPILE statement.
4. Use a nonspacetaking terminal or a batch job.

Note that you can use the IS-ATTR-SPACE function in a procedure to determine whether the current terminal type is a spacetaking or non-spacetaking terminal.

Chapter 8
Transactions and Error Processing

This chapter covers the following topics:

- Transaction basics.
- Using transactions.
- Error processing and recovery.

8.1 INTRODUCTION

Imagine this situation: you are entering new customer records into your database. You have 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:

1. The first 98 records are in the database.
2. The partial 99th record is discarded.

These two steps ensure the integrity of your database. Valid data that you enter is saved while partial data is discarded.

This is just one very simple scenario. Suppose the procedure had been updating multiple files. You want to make sure that any completed changes are saved while partial changes are discarded **in all files.**

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 ensuring that completed data is not discarded and incomplete data is not stored in the database. PROGRESS uses **transactions** to automatically handle this processing.

8.2 TRANSACTIONS DEFINED

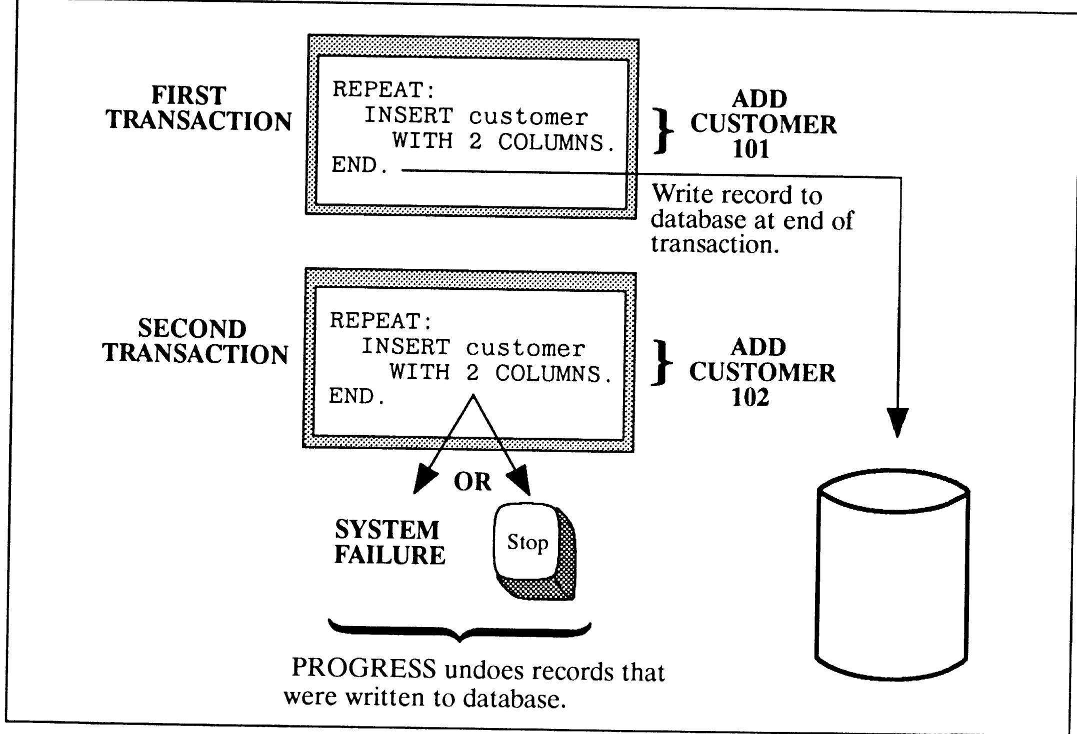
A transaction is a set of changes to the database, which should either be completely done or should leave no modification to the database. The commonly used terms, "physical transaction" and "commit unit" refer to the same concept as the PROGRESS transaction. For example, in the above scenario where you are adding customer records, each customer record you add is a transaction:

Each iteration of this REPEAT block is a **TRANSACTION.**

```
REPEAT:
   INSERT customer WITH 2 COLUMNS.
END.
```

The transaction is undone or "backed out" if:

- The system goes down or "crashes".
- The user presses STOP (Ctrl-Break on DOS and OS/2, usually Ctrl-C on UNIX or VMS, and Action-Cancel on BTOS/CTOS).



In either of these cases, PROGRESS undoes all work that has been done since the start of the transaction.

So far, you have seen how a transaction can be useful in a situation involving only a single file. Transactions take on additional importance when you are making database changes in multiple files or records.

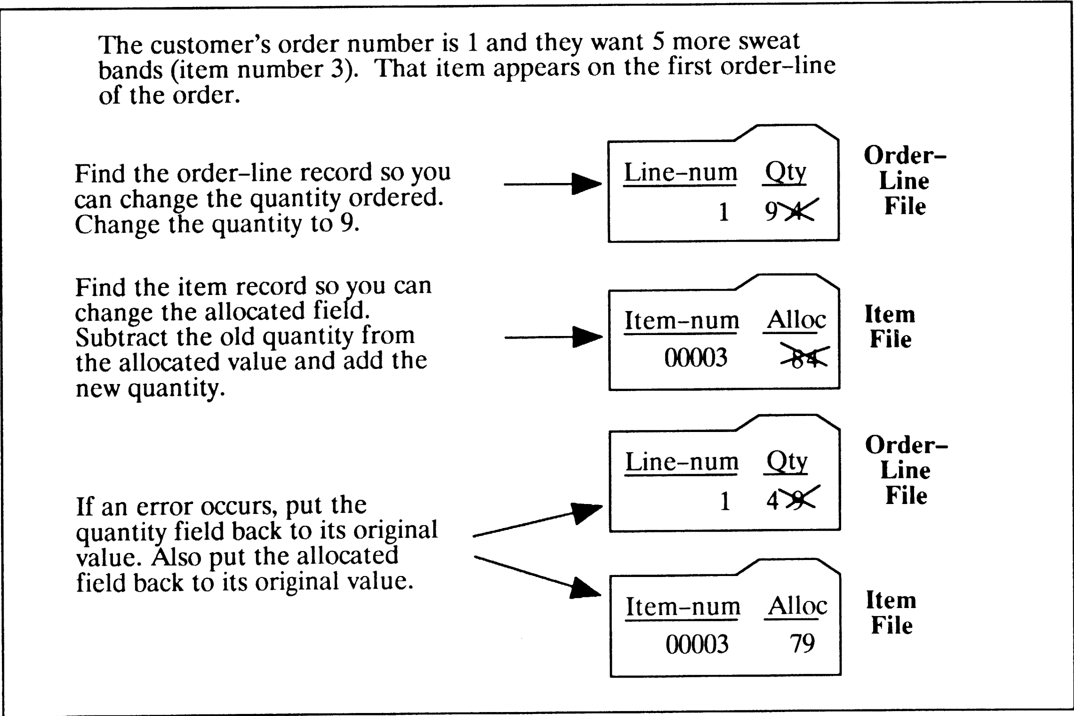
8.3 DOING ALL-OR-NOTHING PROCESSING

Suppose a customer calls to change an order for 4 sweat bands to 9 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 alloc (allocated) field in the record for that item in the item file.

What if you have changed the quantity field in the order-line record and are in the midst of changing the allocated field in the item record, and the machine goes down? You want to restore the records to their original state. That is, you want to be sure that either both records are changed or neither is changed.

Here is what this scenario looks like:



Here is the procedure that does this work:

p-txn.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.idesc item.alloc.
TRANSACTION  <    SET newqty.
              |   item.alloc = item.alloc - order-line.qty + newqty.
              |   PAUSE.
              |   order-line.qty = newqty.
              |   DISPLAY item.alloc @ new-alloc
              |     LIKE item.alloc 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.

Go ahead and run this procedure. Enter 1 as the order number and 1 as the line number as shown here:

```
       Order-num:1                  Line num:1
             Qty:4                  Item num:00003
            Desc:Sweat Band            Alloc:79
    New Quantity:                  New Alloc:
```

Now enter 9 as the New Quantity and press RETURN. You have reached the first PAUSE statement. Press the space bar to continue.

After you enter the new quantity value and the new allocated value is displayed, you have reached the PAUSE statement just before the end of the REPEAT block:

```
       Order-num:1                  Line num:1
             Qty:4                  Item num:00003
            Desc:Sweat Band            Alloc:79
    New Quantity:9                 New Alloc:84
```

You can see that the quantity has been changed to 9 and the amount allocated changed to 84.

p-txn.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.idesc item.alloc.
               ⎪  SET newqty.
TRANSACTION    ⎨  item.alloc = item.alloc - order-line.qty + newqty.
               ⎪  PAUSE.
               ⎪  order-line.qty = newqty.
               ⎪  DISPLAY item.alloc @ new-alloc
               ⎪    LIKE item.alloc LABEL "New Alloc" SKIP(1).
You are here ──►  PAUSE.
               ⎩END.
```

Press the space bar. Now the procedure is prompting you for another order number and line number. Enter a 1 for the order number and a 2 for the line number.

```
     Order-num:1                       Line num:1
           Qty:4                       Item num:00003
          Desc:Sweat Band                 Alloc:79
  New Quantity:9                      New Alloc:84


     Order-num:1                       Line num:2
           Qty:2                       Item num:00004
          Desc:Cycle Helmet               Alloc:0
  New Quantity:                       New Alloc:
```

On this order line, suppose you want to change the number of ordered cycle helmets to 12. Go ahead and enter a 12.

```
         Order-num:1                              Line num:1
               Qty:4                              Item num:00003
              Desc:Sweat Band                        Alloc:79
      New Quantity:9                             New Alloc:84

         Order-num:1                              Line num:2
               Qty:2                              Item num:00004
              Desc:Cycle Helmet                      Alloc:0
      New Quantity:12                            New Alloc:10
```

You can see that the quantity has been changed to 12 and the amount allocated has been increased to 10 (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, or backs out, 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.

```
RUN p-txn.p
Enter PROGRESS procedure. Press F1 to run.
```

Remember that you had changed the quantity to 9 on line 1 of order 1. Enter a 1 as the order number and a 1 as the line number. Here is the screen display:

```
         Order-num:1                              Line num:1
               Qty:9                              Item num:00003
              Desc:Sweat Band                        Alloc:84
      New Quantity:                              New Alloc:
```

You can see that this first change you made is safely stored in the database. Both the order-line file and the item file reflect the new values of 9 and 84. Press GO (F1) to continue on until you reach the PAUSE statement just before the end of the REPEAT block. Then press the space bar to continue on to the next iteration.

Enter a 1 for the order number and a 2 for the line number.

```
      Order-num:1                              Line num:1
            Qty:4                              Item num:00003
           Desc:Sweat Band                        Alloc:79
   New Quantity:9                             New Alloc:84


      Order-num:1                              Line num:2
            Qty:2                              Item num:00004
           Desc:Cycle Helmet                      Alloc:0
   New Quantity:                              New Alloc:
```

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 2 and 0 respectively.

Remember that each iteration of the REPEAT block started a new transaction. Once the first iteration (the sweat bands) had 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.

How did PROGRESS know where to start the transaction and how much work to undo or back out?

8.4 UNDERSTANDING WHERE TRANSACTIONS BEGIN AND END

For any given procedure, the transaction is one iteration of the outermost FOR EACH, REPEAT or procedure block or blocks that contain direct updates to the database. In other words, 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 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 12 for more information about multi-user applications.

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.

Notice that DO blocks do not automatically have the transaction property. Also, if the procedure or transaction you are looking at is called by another procedure, you must check the calling procedure to determine if it starts a transaction.

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. For example:

p-txn2.p

Each iteration of this REPEAT block is a **TRANSACTION.**

```
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 file will not cause PROGRESS to automatically start a transaction for a work file.

Let's look at another example:

p-txn3.p

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.
```

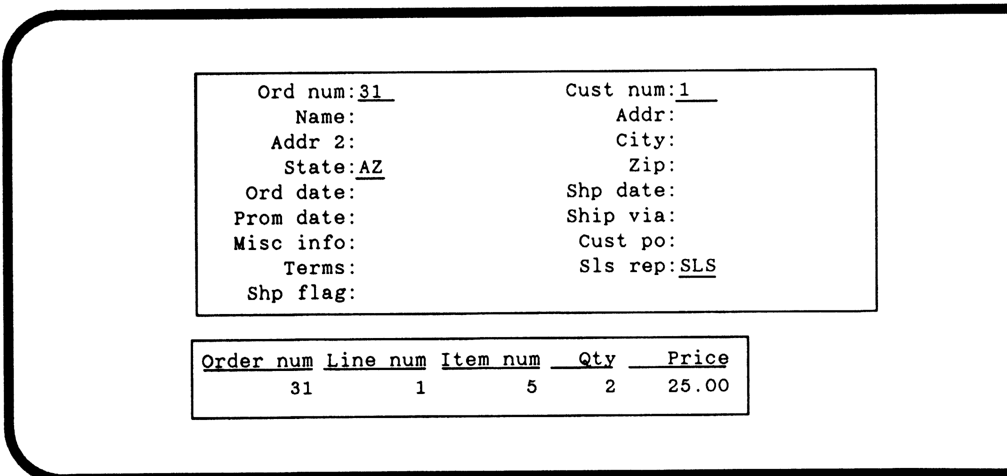
TRANSACTION

```
FOR EACH salesrep:
  DISPLAY slsname slstitle.
  UPDATE slsrgn.
END.
```

This procedure has four blocks:

- The **procedure** block. There are no statements in this block so PROGRESS does not start a transaction at the start of the procedure.
- The **outer REPEAT** block. This block is an 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.
- The **inner REPEAT** block. This 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.
- The **FOR EACH** block. This block is an outermost block that directly updates the database (UPDATE slsrgn). 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. Suppose you add the data shown in the following screen.



After you enter the order information, press GO (F1), then enter the data for line number 1 of the order and press GO (F1). The procedure prompts you for the next order line. Press END-ERROR (F4) to tell the procedure that you are finished entering order lines for order 31. The procedure prompts you for the next order number.

Enter the following data:

```
  Ord num:32                    Cust num:1
     Name:                          Addr:
   Addr 2:                          City:
    State:AZ                         Zip:
 Ord date:                      Shp date:
Prom date:                      Ship via:
Misc info:                       Cust po:
    Terms:                       Sls rep:SLS
 Shp flag:

Order num Line num Item num   Qty   Price
       31        1    00002     3    0.00
```

After you enter the quantity and are being prompted for the price, press STOP. PROGRESS returns you to the procedure editor.

Remember that either a system failure or 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.

p-txn3.p

```
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.

TRANSACTION {  FOR EACH salesrep:
                 DISPLAY slsname slstitle.
                 UPDATE slsrgn.
               END.
```

8.4.1 The First Iteration of the Outer REPEAT Block

1. On the first iteration of the REPEAT block, you entered data for order 31. Because PROGRESS automatically starts a transaction for a REPEAT block if that 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 31. Even though this is a REPEAT block and does directly update the database (`UPDATE line-num order-line.item-num qty price.`), 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 end 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 was written to the database. It should still be there, right? Let's check and see. Run this procedure to see if order 31 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 31 and its single `order-line`, proving that the data is in the database.

8.4.2 The Second Iteration of the Outer REPEAT Block

1. On the second iteration of the REPEAT block, you entered data for order 32. Because PROGRESS automatically starts a transaction for a REPEAT block if that 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 (`UPDATE line-num order-line.item-num qty price`), a transaction is already active. Therefore, PROGRESS does not start a transaction for this inner REPEAT block and you are still working within the initial transaction. In the midst of the transaction, you pressed STOP which backed out the current transaction.

3. Run the `p-check.p` procedure again, this time supplying 32 as the order number. PROGRESS displays the message "`order record not on file`".

4. This message confirms that the data for order 32 was not stored in the database. But remember the rule about all-or-nothing processing. When working with multiple files in a single transaction, it is important that either all the changes to all the files are completed or none of the changes to any of the files are completed.

5. 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 31 wasn't stored in the database, run this procedure (supply 32 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.
```

6. The 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 doesn't 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.

7. When you run the `p-check2.p` procedure and supply 32 as the order number and 1 as the order-line number, PROGRESS displays the message "order-line record not on file".

8. Not only did PROGRESS undo the order information you had entered for order 32 but it also undid the partial order-line information you had entered.

If you hadn't pressed STOP but your machine had crashed, you would have seen exactly the same behavior.

8.5 SPECIFYING HOW MUCH TO UNDO

You've 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.

Let's take a look at a different version of the p-txn3.p procedure:

p-txn3a.p

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.
```

8 Transactions and Errors

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:

p-txn5.p

```
DEFINE VARIABLE extension LIKE price.
DEFINE VARIABLE tot-order LIKE 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 exeeded $500".
      MESSAGE "No more order-lines allowed".
      UNDO o-l-block.
    END.
  END.
END.
```

SUBTRANSACTION

TRANSACTION

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.

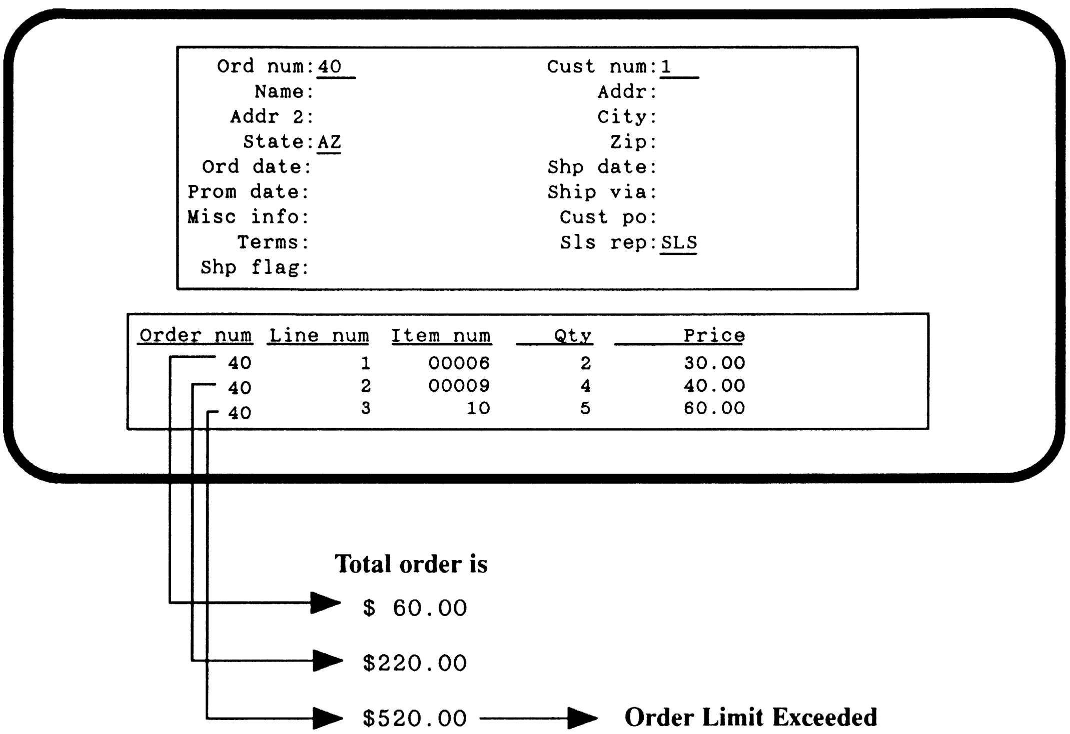
If an error occurs during a subtransaction, all the work done since the beginning of the subtransaction is undone. Subtransactions can be nested within other subtransactions.

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 following data:



After you enter the second order line, the amount of the order is $220. 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 8–1 shows when transactions and subtransactions are started.

Table 8–1: 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 directly containing 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 regular file will not cause PROGRESS to automatically start a transaction for a work file.

8.6 CONTROLLING WHERE TRANSACTIONS BEGIN AND END

You may find that, because of the kind of work a procedure is doing, you want to start or end a transaction 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, a transaction is started on each iteration of the block regardless of whether the block contains statements that directly update the database. Of course, a transaction is not started 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 in a later section of this chapter.

8.6.1 Making Transactions Larger

In the `p-txn3.p` procedure, the outermost REPEAT block is the transaction block. That means that when the transaction is undone, PROGRESS undoes any work done in the current iteration of the outer REPEAT block. So, only the current order is undone, while all other orders are safely stored in the database.

p-txn3a.p

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.
```

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:

p-txn4.p

TRANSACTION

```
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. That is, even though an outermost REPEAT, FOR EACH, or procedure block that contains direct database updates is normally automatically 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:

```
  Ord num:31                    Cust num:1
     Name:                          Addr:
   Addr 2:                          City:
    State:AZ                         Zip:
 Ord date:                      Shp date:
Prom date:                      Ship via:
Misc info:                       Cust po:
    Terms:                       Sls rep:SLS
 Shp flag:

Order num Line num Item num   Qty    Price
       31        1    00002     3    70.00
```

After you enter all the information for the first order line, press GO (F1). The procedure prompts you for more order-line information. Press END-ERROR (F4) to indicate that there are no more order lines to enter for this order. The procedure is prompting you for the next order.

Enter the following data:

```
  Ord num:32                    Cust num:1
     Name:                          Addr:
   Addr 2:                          City:
    State:AZ                         Zip:
 Ord date:                      Shp date:
Prom date:                      Ship via:
Misc info:                       Cust po:
    Terms:                       Sls rep:SLS
 Shp flag:

Order num Line num Item num   Qty    Price
       32        1    00008     2    15.00
       32 _______2 ___00009 ____4 ________
```

Before entering the price for order-line 2, press STOP. PROGRESS backs out the transaction, returning you to the procedure editor.

Because the transaction encompassed all iterations of the outer REPEAT block, you would expect that all work done in those iterations would have been backed out, right? Let's make sure that happened. Run this procedure 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 31 and 32. 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.

8.6.2 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 the STOP key, 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

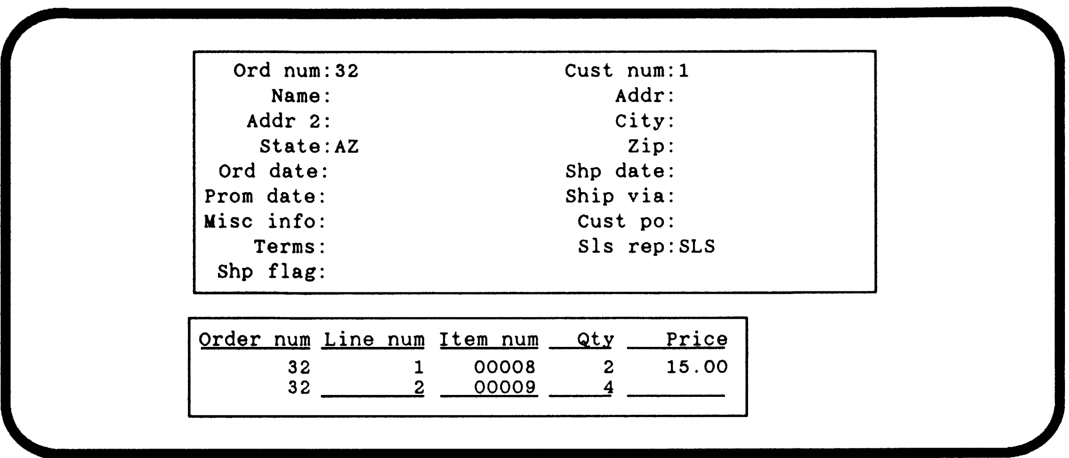
```
              REPEAT:
              { DO TRANSACTION:
TRANSACTION   {    INSERT order WITH 2 COLUMNS.
              {    FIND customer OF order.
              { END.
                REPEAT TRANSACTION:
              {    CREATE order-line.
              {    order-line.order-num = order.order-num.
TRANSACTION   {    DISPLAY order-line.order-num.
              {    UPDATE line-num order-line.item-num qty price.
              { END.
              END.
```

Here, 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 (`UPDATE line-num order-line.item-num qty price`).

The outer REPEAT block is not the transaction block because it does not contain any direct updates to the database. Run this procedure, entering the following data:



```
  Ord num:32                   Cust num:1
     Name:                         Addr:
   Addr 2:                         City:
    State:AZ                        Zip:
 Ord date:                     Shp date:
Prom date:                     Ship via:
Misc info:                      Cust po:
    Terms:                      Sls rep:SLS
 Shp flag:
```

Order num	Line num	Item num	Qty	Price
32	1	00008	2	15.00
32	2	00009	4	

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. That means that order 32 and the first order line should be in the database. Run this procedure to check:

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.

8.7 USING TRANSACTIONS WITH SUBPROCEDURES

If you start a transaction in a main procedure, that transaction remains active even while that main procedure is running called procedures. For example:

p-txn11.p

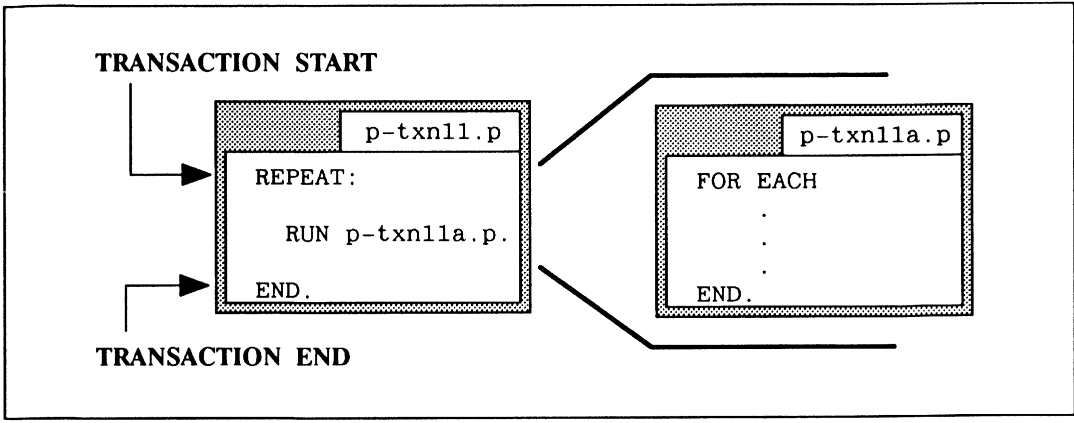
```
DEFINE VARIABLE answer AS LOGICAL.
DEFINE NEW SHARED VARIABLE cust-num-var AS RECID.

REPEAT WITH 1 DOWN:
  PROMPT-FOR customer.cust-num.
  FIND customer USING cust-num.
  DISPLAY name sales-rep.
  UPDATE max-credit curr-bal.
  SET answer LABEL "Do you want to do order processing?"
    WITH FRAME a NO-HIDE.
  IF answer THEN DO:
    cust-num-var = RECID(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 RECID.

HIDE ALL.
FIND customer WHERE RECID(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 p-txn11.p procedure is the transaction block for that procedure: it contains a direct update to the database (UPDATE max-credit curr-bal). The transaction starts 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.



If a system error occurs while you are doing order processing for a customer, PROGRESS undoes all the order processing work you've done for that customer as well as any changes you made to the customer record itself.

8.8 USING TRANSACTIONS WITH FILE INPUT

You must be especially careful when you are processing transactions that get input from an ASCII file. If a crash occurs, the active transaction will be backed out of the database but you will not automatically know how far processing was in the ASCII 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 `.bi` file for the transaction).
- Have a way of knowing 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 file name and the last line number in a database file since changes in this status information can be synchronized with the corresponding database updates.

8.9 HANDLING OTHER ERRORS

You've seen how PROGRESS handles errors such as the STOP key or a machine crash: the current transaction is backed out, undoing any database changes performed during the transaction. There are five other kinds of errors that may occur:

- There is a procedure-generated error. For example, a record find fails, or the procedure tries to create a duplicate entry in a unique index.
- The user presses a key that tells PROGRESS to do error processing.

- The user presses a key that tells PROGRESS to do endkey processing.
- The user presses the END-ERROR (F4) key.
- There is a system or software failure.

Let's look at these five cases in detail.

8.9.1 How PROGRESS Handles Procedure Errors

There are two very common ways a procedure can generate an error:

1. The procedure tries to create a duplicate entry in a unique index. For example, the procedure might try to create a customer record using a customer number that already existed in the database.

2. A FIND statement tries to read a record that does not exist.

In either of these cases, PROGRESS:

1. Looks for the closest block that has the "error" property.

2. Undoes and retries that block.

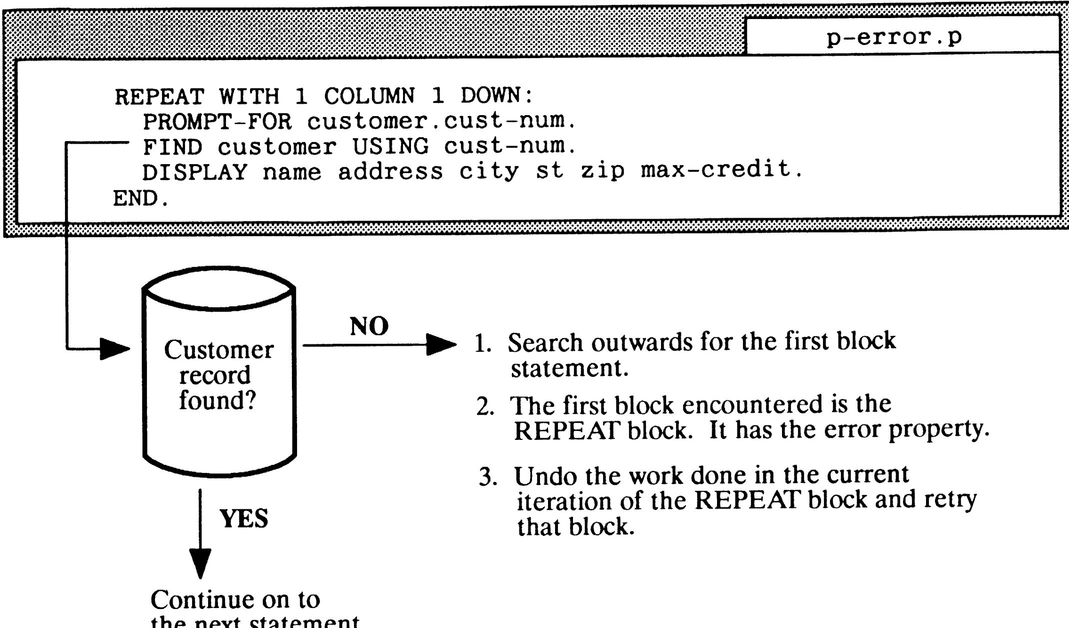
Blocks that automatically have the "error property" are: FOR EACH blocks, REPEAT blocks, and procedure blocks. By "having the error property" we mean that each of these blocks implicitly has the ON ERROR UNDO, RETRY phrase attached to it. For example:

```
FOR EACH customer:
```

is the same as:

```
FOR EACH customer ON ERROR UNDO, RETRY:
```

Take a look at an example so you can see how PROGRESS handles a processing error.

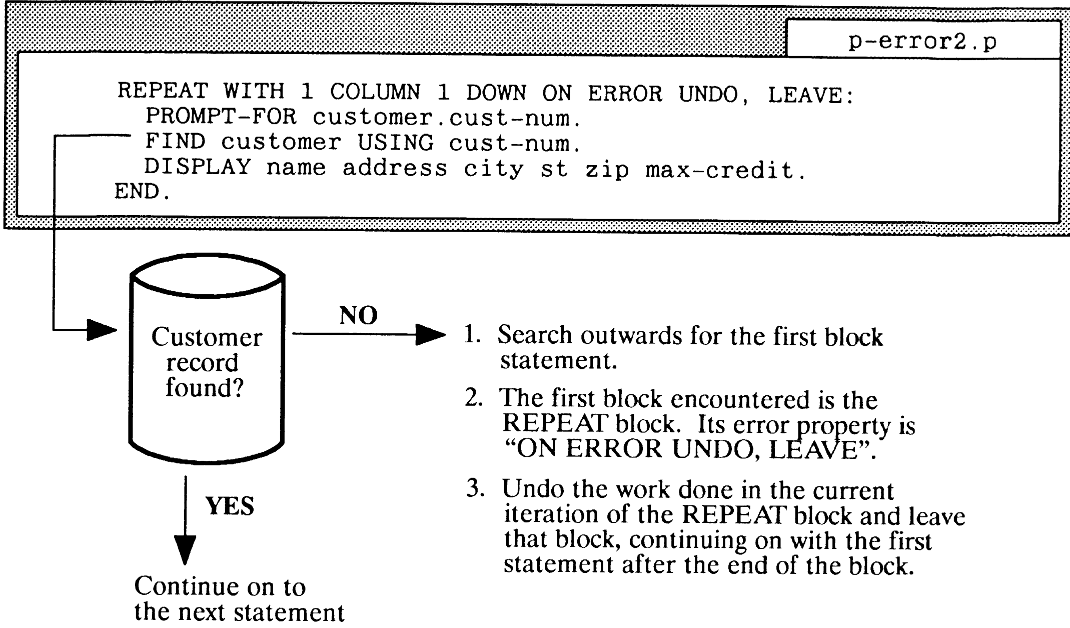


If PROGRESS is not handling a specific processing error the way you want, you can override automatic error handling by handling the error yourself.

8.9.2 How You Handle Procedure Errors

When a processing error occurs, the first thing PROGRESS does is begin working its way back through the procedure, looking at each block header until it finds a block that has the "error" property. This property is ON ERROR UNDO, RETRY by default for FOR EACH, REPEAT, and procedure blocks. You can override that property by using different ON ERROR phrases on a DO, FOR EACH, or REPEAT block statement.

Take the last procedure as an example. In the `p-error.p` 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. Here is the modified `p-error.p` procedure:



8.9.3 How PROGRESS Handles the Error Key

When you are running a PROGRESS procedure, there is no ERROR key defined on the keyboard. However, you can define any control (CTRL) key or other special key as the ERROR key. Then, when the user presses that key when prompted for input, the procedure does error processing. That is, PROGRESS:

1. Looks for the closest block that has the "error" property.

2. Undoes and retries that block.

Blocks that have the "error property" by default are: FOR EACH blocks, REPEAT blocks, and procedure blocks. By "having the error property" we mean that each of these blocks implicitly has the ON ERROR UNDO, RETRY phrase attached to it. For example, saying

```
FOR EACH customer:
```

is the same as:

```
FOR EACH customer ON ERROR UNDO, RETRY:
```

Take a look at an example so you can see how PROGRESS handles the 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.
    price = cost.
    UPDATE price.
  END.
END.
```

SUBTRANSACTION

TRANSACTION

The ON statement defines the F9 key as the error key. (If you do not have an F9 key on your terminal, you can modify this procedure to name another key such as F2 or PF2. Or, if you terminal has no F or PF keys, specify ON CTRL-W ERROR to assign ERROR to the CTRL-W key.) That is, when the user presses F9, PROGRESS raises the error condition. That means that PROGRESS searches for the closest block that has the error property and undoes and retries that block. Let's see if that's what happens. Go ahead and run the p-txn6.p procedure, entering the following data:

```
  Ord num:50                     Cust num:1
     Name:____________              Addr:____________
   Addr 2:____________              City:________
    State:AZ                         Zip:_____
 Ord date:  /  /                Shp date:  /  /
Prom date:  /  /                Ship via:____________
Misc info:____________           Cust po:____________
    Terms:____________           Sls rep:SLS
 Shp flag:___
```

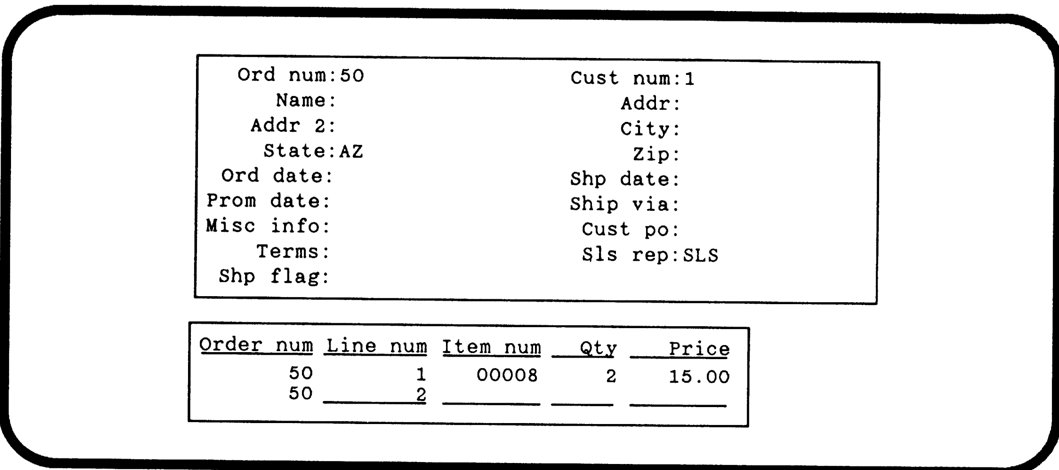
Before entering a sales rep value, press F9. PROGRESS:

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.

Run the `p-txn6.p` procedure again, entering the following data:



Press F9 while entering the second order line. PROGRESS:

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 inner REPEAT block.

3. Undoes all work done in the current iteration of the REPEAT block and retries that iteration of the REPEAT block.

8.9.4 How You Handle the Error Key

Suppose you wanted PROGRESS to undo more than the current iteration of the inner REPEAT block. That is, 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. That's easy enough to do:

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.
    price = cost.
    UPDATE 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. It also happens if there is some sort of processing error such as the FIND statement's being unable to locate a record.

8.9.5 How PROGRESS Handles The Endkey

When you are running a PROGRESS procedure, there is no ENDKEY key defined on the keyboard. However, you can use the ON statement to assign a key as the endkey, or you can define any control (CTRL) key or other special key as the ENDKEY. Then, when the user presses that key, the procedure does endkey processing. That is, PROGRESS:

1. Looks for the closest block that has the "endkey" property.

2. Undoes and leaves that block.

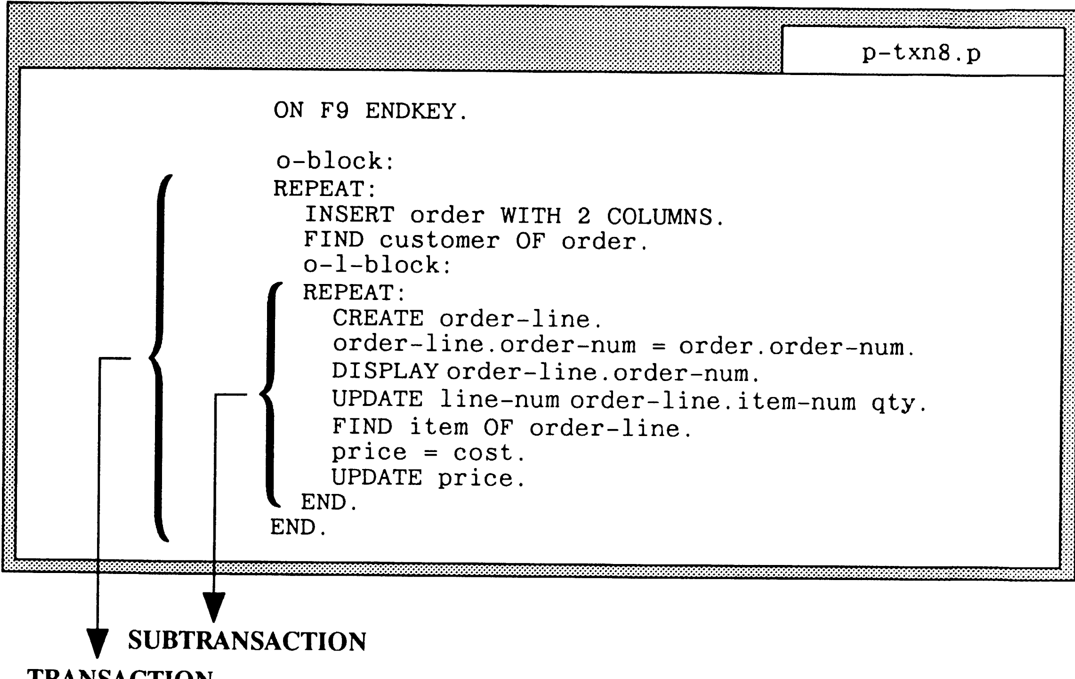
Blocks that have the "endkey property" by default are: FOR EACH blocks, REPEAT blocks, and procedure blocks. By "having the endkey property" we mean that each of these blocks implicitly has the ON ENDKEY UNDO, LEAVE phrase attached to it. For example:

```
FOR EACH customer:
```

is the same as:

```
FOR EACH customer ON ENDKEY UNDO, LEAVE:
```

Take a look at an example so you can see how PROGRESS handles the ENDKEY key:




```
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.
    price = cost.
    UPDATE price.
  END.
END.
```

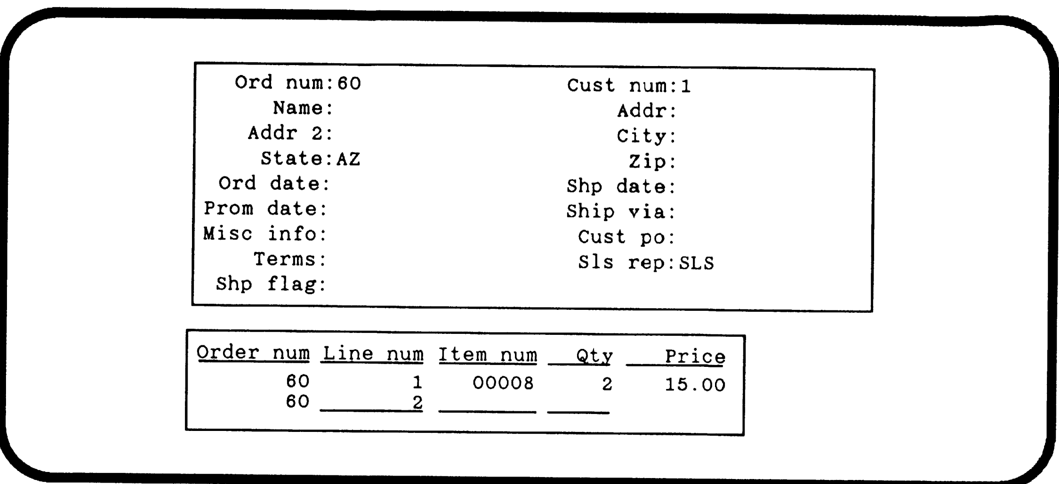
The ON statement defines the F9 key as the ENDKEY. That is, when the user presses F9, PROGRESS raises the endkey condition. That means that PROGRESS searches for the closest block that has the endkey property and undoes and leaves that block. Let's see if that's what happens. Go ahead and run the p-txn8.p procedure, entering the following data:

```
  Ord num:60              Cust num:1
     Name:____________        Addr:____________
   Addr 2:____________        City:________
    State:AZ                   Zip:______
 Ord date: / /            Shp date: / /
Prom date: / /            Ship via:____________
Misc info:____________     Cust po:____________
    Terms:____________     Sls rep:SLS
 Shp flag:___
```

Before entering a sales rep value, press F9. PROGRESS:

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.

Run the `p-txn8.p` procedure again, entering the following data:



Press F9 while entering the second order line. PROGRESS:

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.

8.9.6 How You Handle the Endkey

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 altogether. That's easy enough to do:

p-txn9.p

```
ON F9 ENDKEY.

o-block:
REPEAT:
  INSERT order WITH 2 COLUMNS.
  FIND customer OF order.
  o-l-block:
  REPEAT ON ENDKEY UNDO o-block, LEAVE o-block:
    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.
    price = cost.
    UPDATE price.
  END.
END.
```

Here, the inner REPEAT block explicitly says that, in the event of the endkey condition, PROGRESS should not undo the current REPEAT block but the outer `o-block` REPEAT block. In addition, it tells PROGRESS to leave the `o-block` REPEAT block.

8.9.7 Rules About UNDO

When you use the ON ERROR or ON ENDKEY phrase with the UNDO option, you can:

- Name the current block or any enclosing block with the UNDO.
- LEAVE or NEXT the current block or any enclosing block.
- RETRY only the block you are undoing.

Any block you name with the UNDO option must be the current block or a block that contains the current block.

For more information about UNDO, see the *PROGRESS Language Reference* manual.

8.9.8 How PROGRESS Handles the END-ERROR Key

There is a key on your keyboard that PROGRESS calls the END-ERROR key (usually F4). The reason it has this name is that sometimes PROGRESS treats it as an error and other times treats it as an endkey. Take a look at this example:

p-error3.p

```
REPEAT WITH 1 COLUMN 1 DOWN:
  PROMPT-FOR customer.cust-num.
  FIND customer USING cust-num.
  UPDATE name address city st zip max-credit.
END.
```

Go ahead and run this procedure.

```
Cust num:____
    Name:
    Addr:
    City:
   State:
     Zip:
Max cred:
```

At this point you are at the PROMPT-FOR statement in the procedure. Press END-ERROR (F4). The procedure returns you to the procedure editor. Run the procedure again, entering a 1 for cust num:

```
Cust num:1
    Name:Second Skin Scuba
    Addr:79 Farrar Ave
    City:Yuma
   State:AZ
     Zip:85369
Max cred:1,500
```

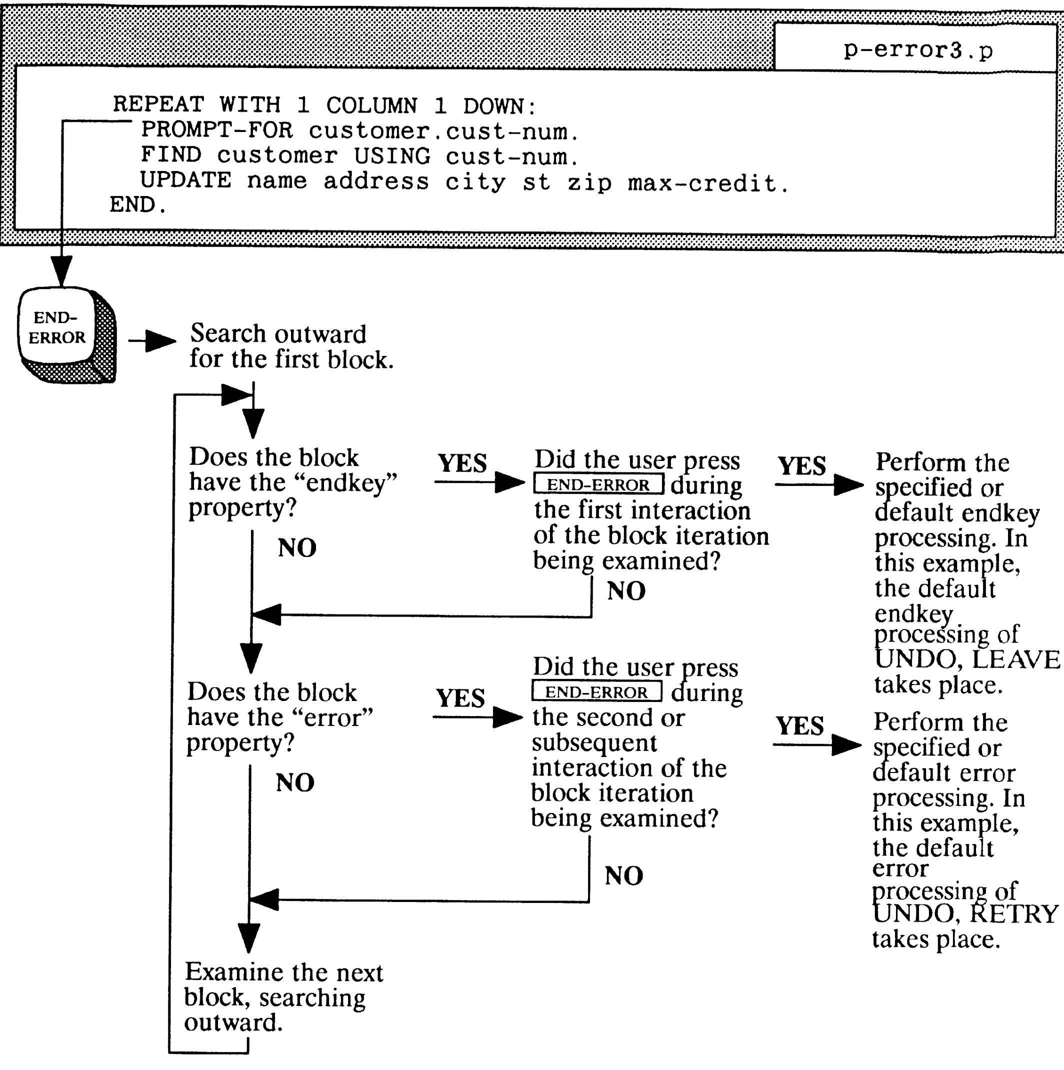
At this point you are at the UPDATE statement in the procedure. Press END-ERROR (F4). The procedure returns you to the PROMPT-FOR statement and lets you enter another customer number.

Why does PROGRESS treat the same key differently under different circumstances? Two good reasons:

1. Usually if you press END-ERROR (F4) 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 because you have decided you don't want to complete the task you started. In this case, PROGRESS undoes whatever you've done in the current iteration of the block and leaves the block.

2. If you press END-ERROR (F4) on the second or later interaction of a block iteration, it is usually because you have made a mistake or want to want to retry the work you've done. In this case, PROGRESS undoes whatever you've done and retries.

Run the `p-error3.p` procedure again. The following figure shows what happens when you press END-ERROR (F4).



In determining whether the user pressed END-ERROR (F4) during the first or subsequent interaction in a block iteration, PROGRESS does not count the following as interactions:

- Prior keyboard entries to end a pause ("Press space bar to continue")
- Prior keyboard entries in response to a READKEY statement.

8.10 HOW PROGRESS HANDLES SYSTEM AND SOFTWARE FAILURES

Following a system hardware or software failure (a "crash"), PROGRESS backs out partially completed transactions for all users. This is the fundamental difference between transactions and subtransactions: a partially completed transaction is backed out after a system failure, including work done in any complete or incomplete subtransactions that are encompassed within the transaction. Individual subtransactions can be undone without undoing previously completed subtransactions that are all within one transaction.

Changes made in a completed system transaction cannot be undone, otherwise changes made subsequently by other users could be inadvertently backed out also.

Also, if the user presses STOP (CTRL-BREAK on DOS, OS/2; CTRL-C on many UNIX and VMS systems; Action-Cancel on BTOS/CTOS), the current transaction is undone and control returns to the startup procedure if one is still active, otherwise to the editor.

8.11 UNDERSTANDING HOW TRANSACTIONS AFFECT VARIABLES

We've talked quite a bit about how transactions are backed out and how database changes are undone. But what happens to any 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. (See the section of this chapter called "Specifying How Much To Undo" for more information on subtransactions.) 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 keeping 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.
```

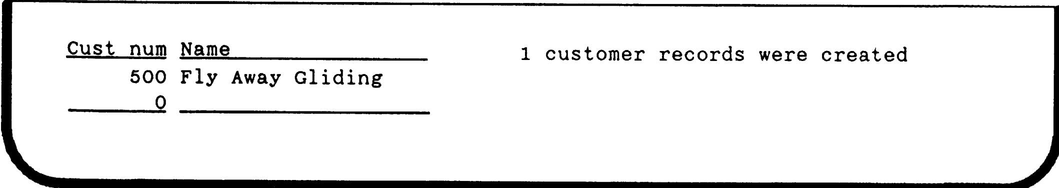
Run this procedure, entering the following data:

```
Cust num Name
     500 Fly Away Gliding
       0
```

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 (F4). 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), 1. The DISPLAY statement tells you that just one customer has been created.



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:

p-tnchp.p

```
DEFINE VARIABLE i AS INTEGER.
DEFINE VARIABLE j AS INTEGER.

cusloop:
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 (i.e. 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 you should consider using the NO-UNDO option on those variables if you have no need for the undo services on those variables.

8.12 DETERMINING WHEN TRANSACTIONS ARE ACTIVE

You can use the p-istran.p utility procedure to determine if a transaction is active in a procedure. The p-istran.p procedure helps you identify the transaction and subtransaction blocks within a procedure. Here is the p-istran.p procedure:

p-istran.p

```
DEFINE VARIABLE istrans AS LOGICAL INITIAL YES.
DO ON ERROR UNDO:
   istrans = no.
   UNDO, LEAVE.
END.

/* If variable was undone within DO ON ERROR, then a */
/* transaction was active when this procedure was called. */
/* Use argument to procedure in the message to identify */
/* where called from to aid in testing. */

IF istrans
THEN MESSAGE "Transaction active at" "{1}".
ELSE MESSAGE "Transaction not active at" "{1}".
```

You can find the p-istran.p procedure in packed form in the proguide directory on your system. To use the p-istran.p procedure, modify the procedure you are writing to include the line:

```
RUN p-istran.p "argument".
```

Run the following procedure to see how p-istran.p works.

p-nord2.p

```
RUN p-istran.p "procedure".

ordblock:
REPEAT:
  RUN p-istran.p "ordblock".
  CREATE order.
  UPDATE order-num cust-num odate.

  olblock:
  REPEAT:
    RUN p-istran.p "olblock".
    CREATE order-line.
    order-line.order-num = order.order-num.
    SET line-num qty item-num price.
  END.
END.
```

Each time `p-istran.p` runs within `p-nord2.p`, you see a message on the message line of your screen telling you whether a transaction is active or inactive. You can get more information about transaction activity by using the listing option on the COMPILE statement. See the *PROGRESS Language Reference* manual for more information on the COMPILE statement.

8.13 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.

8.13.1 Transaction Mechanics

During a transaction, information about all database activity occurring during that transaction is written to a **before-image** (.bi) file. PROGRESS maintains one before image file for each database. The information being written to the before-image file is carefully coordinated with the timing of the data being written to the actual database file. 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.

The amount of disk space used by the before-image file depends on these factors:

- In single-user mode, every transaction reuses the space used by previous transactions. The before-image file occupies as much space as needed to accommodate the largest transaction in the session.

- In multi-user mode, before-image space is reused whether or not transactions are active.

8.13.2 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 local before image 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 local-before-image file as a record.

Because the local-before-image information is not needed for crash recovery, it does not need 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 local-before-image file depends on the number of subtransactions started that are subject to being undone.

8.14 DOING EFFICIENT TRANSACTION PROCESSING

Here are a few guidelines to follow to improve the efficiency of transaction processing procedures:

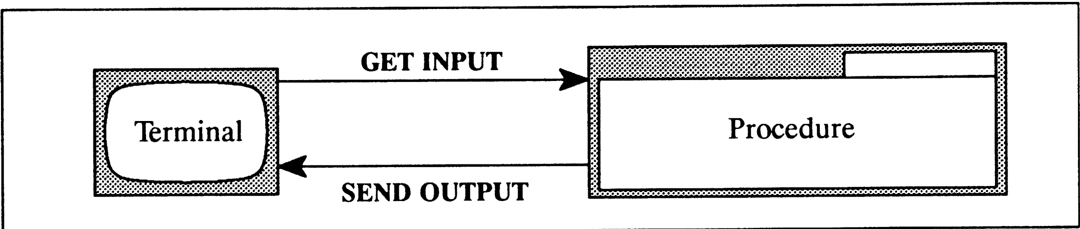
- If you are doing extensive calculations with variables, and you don't need to take advantage of undo processing for those variables, use the NO-UNDO option when defining the variables.

- 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.

Chapter 9
Working With Input and Output in Your Procedures

Most of the procedures you've seen in this book have used the terminal as the input source (the user types in data) and have used the terminal as the output destination (data is displayed to the user).



Up till now, this has been fine. But you have probably already thought of situations in which you may want to get data from and send data to locations other than the terminal. For example, it's likely that you will want to send reports to your printer.

This chapter explains how to use PROGRESS to change where a procedure gets its input and sends its output. It covers the following topics:

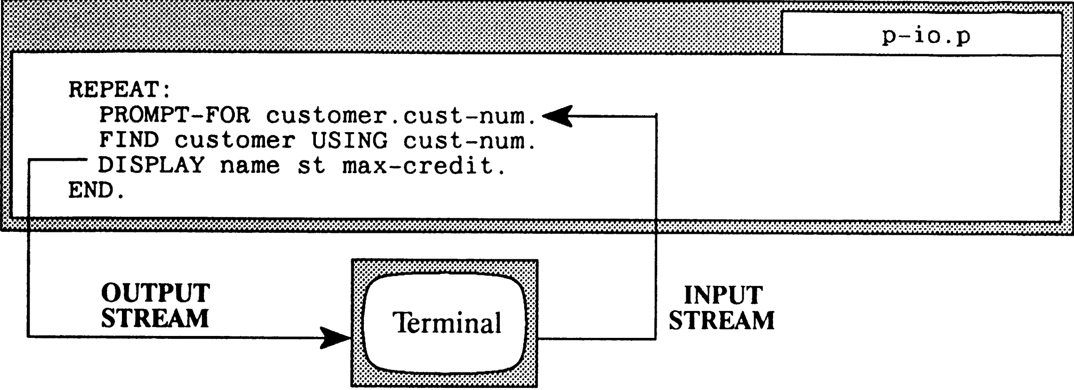
- Input/output basics.
- Changing a procedure's output destination.
- Changing a procedure's input source.
- Using the quoter program.
- Importing and exporting non-PROGRESS file formats.
- Defining additional streams

9.1 UNDERSTANDING INPUT/OUTPUT

When a PROGRESS procedure gets input from the terminal, it is using an **input stream**. Similarly, when the procedure sends output to the terminal, it is using 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.

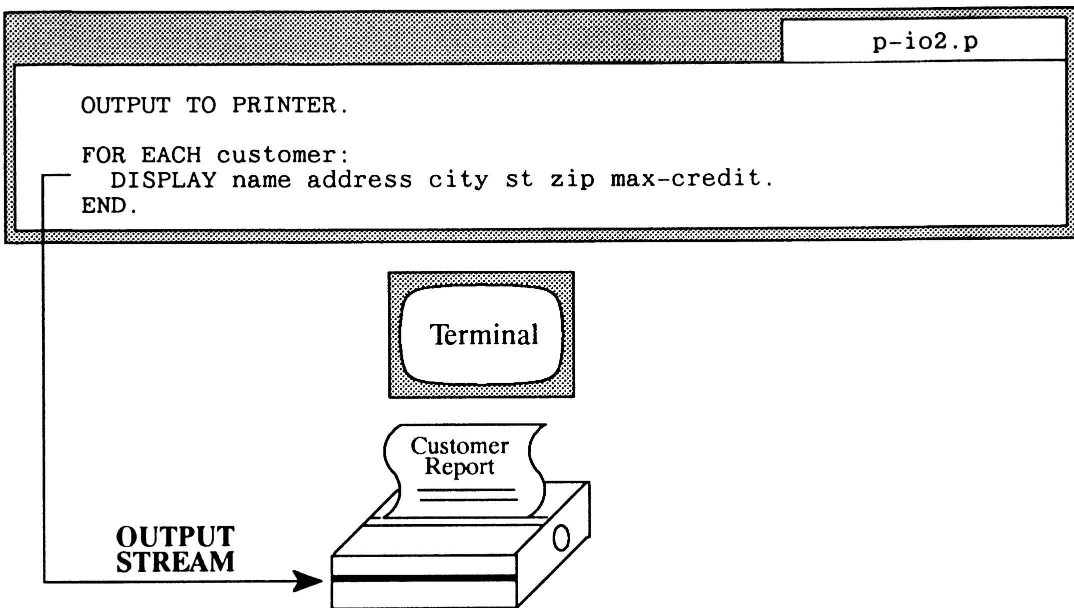
Look at this procedure:



The procedure shown in this illustration contains no special syntax about where to get and send data. So PROGRESS automatically assigns the input stream to the terminal and the output stream to the terminal.

9.2 CHANGING A PROCEDURE'S OUTPUT DESTINATION

You use the OUTPUT TO statement to name an output destination other than the terminal. All statements that output data use that new output destination. For example:



This procedure redirects the output stream to the printer. If you run this procedure, you will see nothing displayed on your terminal because all the output from the DISPLAY statement is going to the printer. You could also direct the report to a standard ASCII file by replacing the statement

```
OUTPUT TO PRINTER.
```

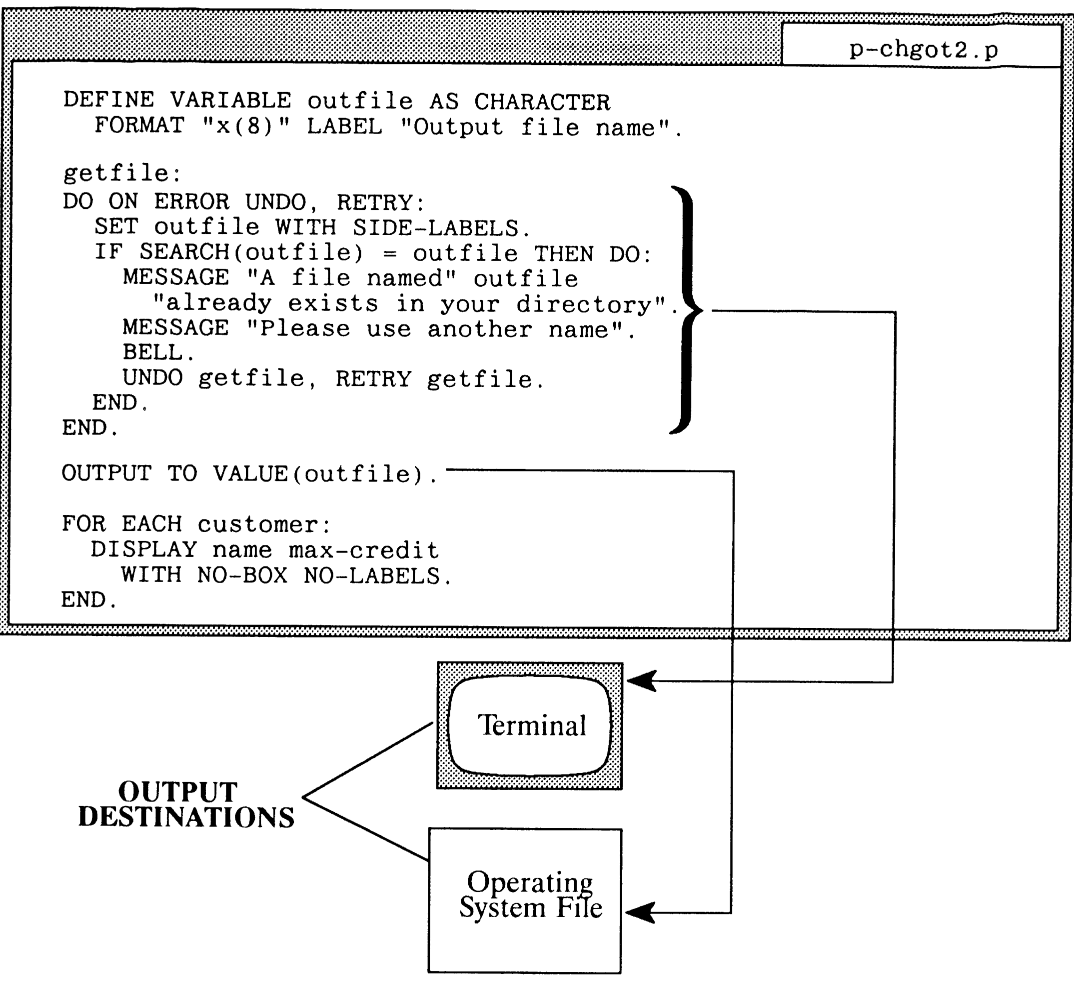
with

```
OUTPUT TO rpt-out PAGED.
```

In this OUTPUT TO statement, `rpt-out` is the name of the file to which output is being directed. Under UNIX, the file name is case sensitive. That is, UNIX treats `rpt-out` and `RPT-OUT` as two different files. 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 then PROGRESS sends the data to the file continuously and does not use any page break control characters.

9.2.1 Using Multiple Output Destinations

Suppose that, at some points in a procedure, you want to send output to the terminal, but at other points you want to send output to a file. You are not restricted to just one output destination per procedure.



This procedure lets you supply the name of the file to which you want to send a customer report and then, if that file doesn't already exist, sends the report to that file. Here are the specific steps the procedure takes:

- The SET statement asks for the name of a file.
- The SEARCH function searches for the file, returning the file name if the file is found.
- If the file is found, the procedure:
  - Displays a message telling you the file already exists and to use another file name.
  - Rings the bell.
  - Undoes the work done in the DO block and retries the block, giving you the opportunity to supply a different file name.

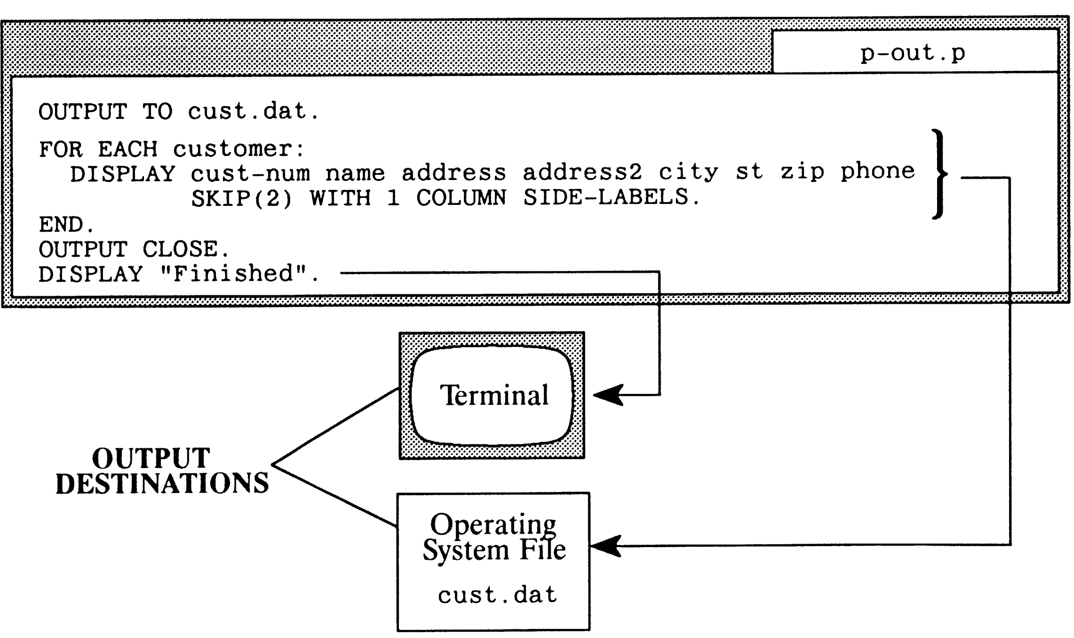
- If the file is not found, the procedure uses the statement :

  ```
  OUTPUT TO VALUE(outfile).
  ```

  to redirect the output to the file you specified. You have to use the VALUE keyword here instead of just saying OUTPUT TO outfile. If you say `OUTPUT TO outfile`, PROGRESS assumes that `outfile` is the name of the ASCII file you want to send output to. In fact, `outfile` is the name of the variable that contains the file name. The VALUE option tells PROGRESS to use the value of the `outfile` variable rather than the name "outfile" itself.

  Also, the procedure, runs a customer report listing each customer's name and maximum credit.

You use the OUTPUT CLOSE statement to stop sending output to a destination. Output sent after the OUTPUT CLOSE statement goes to the destination named prior to 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:

- The OUTPUT TO statement redirects output so all statements that normally send output to the terminal send output to the `cust.dat` file.

- The FOR EACH customer and DISPLAY statements produce a report listing each customer's name, address, city, state, zip, and phone number. The procedure sends the report to the `cust.dat` file.

- The OUTPUT CLOSE statement resets the output destination for the procedure from the `cust.dat` file to the terminal.

- The last DISPLAY statement displays the message "Finished" on the terminal screen.

9.3 CHANGING A PROCEDURE'S INPUT SOURCE

You use the INPUT FROM statement to name an input source other than the terminal. All statements that require data use that new input source. Suppose you have a data file (not a database file) that contains information about new customers. It might look something like this:

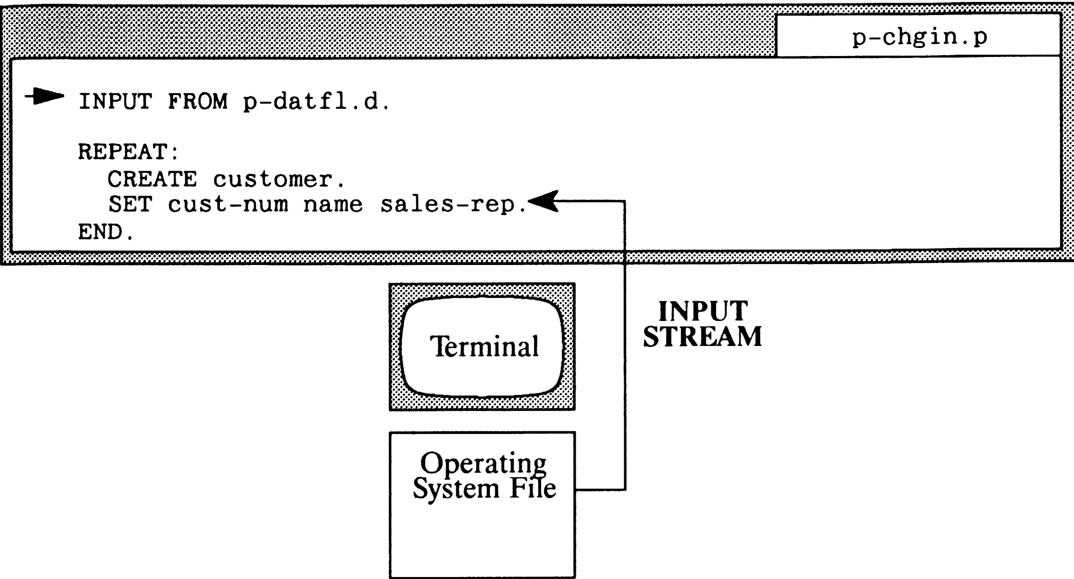
p-datfl.d

```
60 "Wind Chill Hockey" "BBB"
61 "Low Key Checkers" "DKP"
62 "Bing's Ping Pong" "SLS"
```

This file is in standard PROGRESS format. That is, field values are separated by blanks. Field values containing embedded blanks are surrounded by quotation marks (""). Later sections in this chapter discuss using alternate formats.

Press GET (F5) to retrieve this file and put it in the editor (provided you have unpacked it). Now press PUT (F6) to store the procedure in your own directory. Do the same for the following data files: `p-datfl2.d`, `p-datfl3.d`, `p-datfl4.d`, `p-datfl5.d`, and `p-datfl6.d`, so they will be available in your working directory for use later in this chapter. All the procedures in this book are stored in packed form with the Programming Handbook procedures that come with PROGRESS. See the Preface for directions on unpacking the procedures.

You can write a PROGRESS procedure and tell that procedure to get its input from the p-datfl.d file. For example:



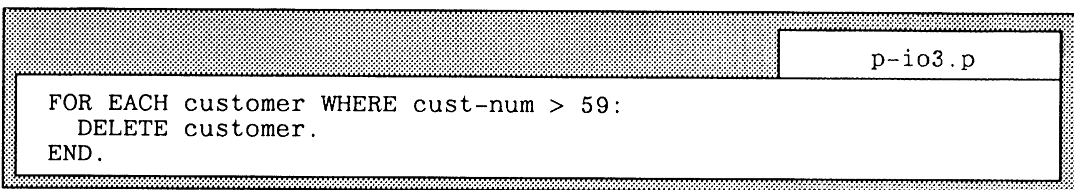
The SET statement, which normally gets its input from the terminal, gets its input from the p-datfl.d file. The cust-num field uses the first data item, 60. The name field uses the next quoted data item, "Wind Chill Hockey", and so on. Each time PROGRESS processes a data entry statement, one line is read from the file.

9.3.1 Using Multiple Input Sources

Suppose that, at some points in a procedure you want to get input from the terminal but at other points you 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-datfl.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. So you need input from the terminal. If the user wants to create customer records for the customers in the p-datfl.d file, you also need input from the file.

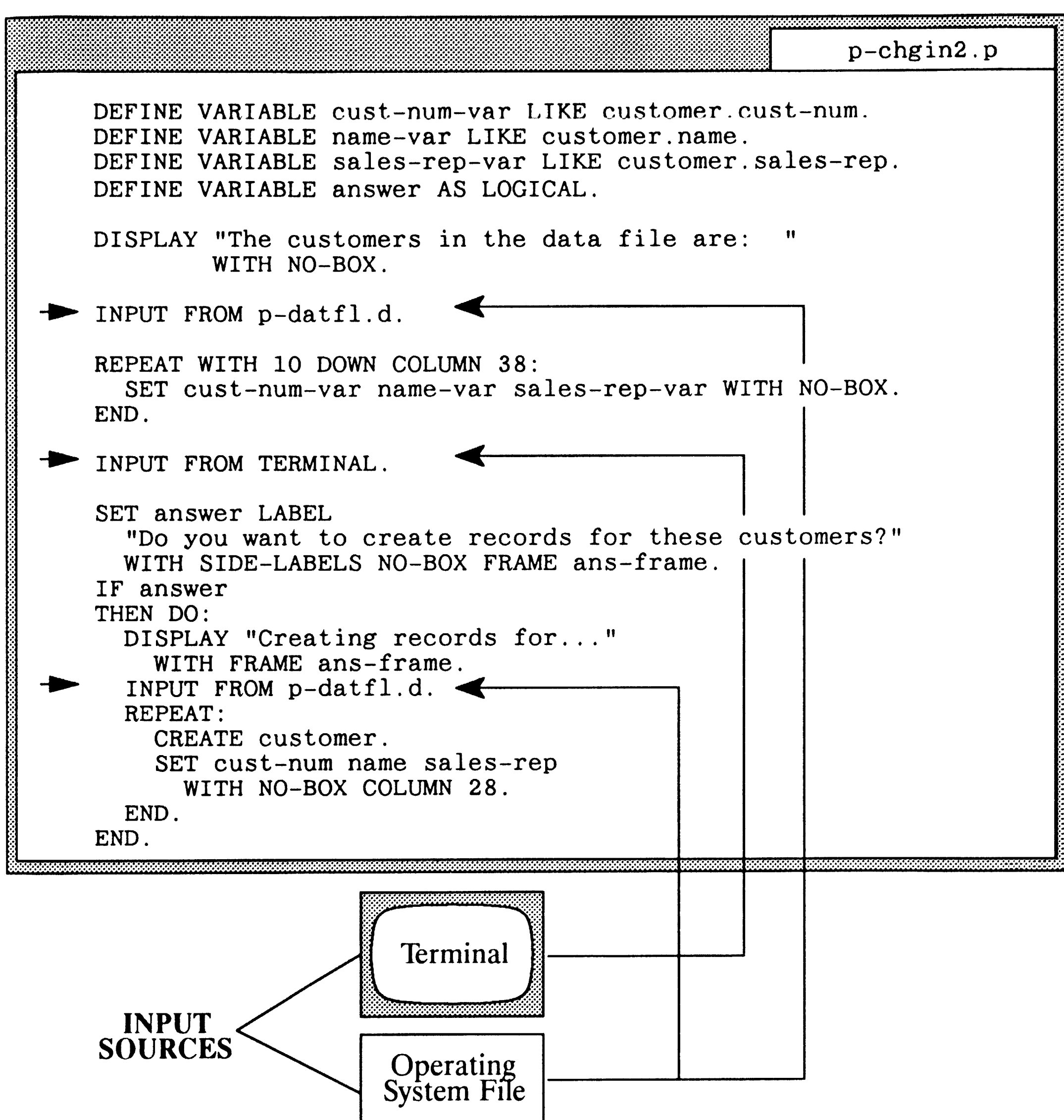
The p-chgin2.p procedure uses multiple input sources to perform the work described above. Because p-chgin2.p uses the same data file (p-datfl.d) we used in the previous section to create customer records, you must delete customers 60, 61, and 62 from your database before you run p-chgin2.p. Use this procedure to delete the customers:



p-io3.p

```
FOR EACH customer WHERE cust-num > 59:
  DELETE customer.
END.
```

Here is the `p-chgin2.p` procedure:



```
DEFINE VARIABLE cust-num-var LIKE customer.cust-num.
DEFINE VARIABLE name-var LIKE customer.name.
DEFINE VARIABLE sales-rep-var LIKE customer.sales-rep.
DEFINE VARIABLE answer AS LOGICAL.

DISPLAY "The customers in the data file are:  "
        WITH NO-BOX.

INPUT FROM p-datfl.d.

REPEAT WITH 10 DOWN COLUMN 38:
  SET cust-num-var name-var sales-rep-var WITH NO-BOX.
END.

INPUT FROM TERMINAL.

SET answer LABEL
  "Do you want to create records for these customers?"
  WITH SIDE-LABELS NO-BOX FRAME ans-frame.
IF answer
THEN DO:
  DISPLAY "Creating records for..."
    WITH FRAME ans-frame.
  INPUT FROM p-datfl.d.
  REPEAT:
    CREATE customer.
    SET cust-num name sales-rep
      WITH NO-BOX COLUMN 28.
  END.
END.
```


Here are the specific steps the procedure takes:

- The DISPLAY statement displays some text.
- The first INPUT FROM statement redirects the input source to the `p-datfl.d` file.

- The SET statement assigns the values in the `p-datfl.d` file to the cust-num-var, name-var, and sales-rep-var variables. As the values are assigned to these variables, they are echoed on the terminal screen.

- 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. It would have closed the input stream coming from the file, resetting the input source to what it was at the start of the procedure, the terminal. However, since this procedure could have been called from another procedure, it is better in this case to be explicit about the input source you want to use.

- The SET answer statement asks whether you want to create database records for the customer data just displayed. If you say yes, the procedure:

  - Redirects the input source to come from the beginning of the `p-datfl.d` file.

  - On each iteration of a REPEAT block, the procedure creates a customer record and assigns values to the `cust-num`, `name`, and `sales-rep` fields in that record.

  - The REPEAT block ends when the SET statement reaches the end of the input file.

9.3.2 Preparing Input Files

The input file used in the previous examples contained data that was in a very specific format:

p-datfl.p

```
60 "Wind Chill Hockey" "BBB"
61 "Low Key Checkers" "DKP"
62 "Bing's Ping Pong" "SLS"
```

When using a data file as an input source, PROGRESS 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 ("").

This does not mean that you have to manually convert all your data files to that format before you can use them with PROGRESS. PROGRESS provides several ways for you to prepare your data files for input to PROGRESS procedures.

9.3.3 Using Quoter

Provided with PROGRESS is a utility program called `quoter`. The `quoter` utility formats data in a file to the standard format so it can be used by a PROGRESS procedure. By default, `quoter` does the following:

- Places quotes (") at the beginning and the end of each line in the file.
- Replaces any already existing quotes (") with two quotes ("").

For example, suppose your data file looked like this:

p-datfl2.d

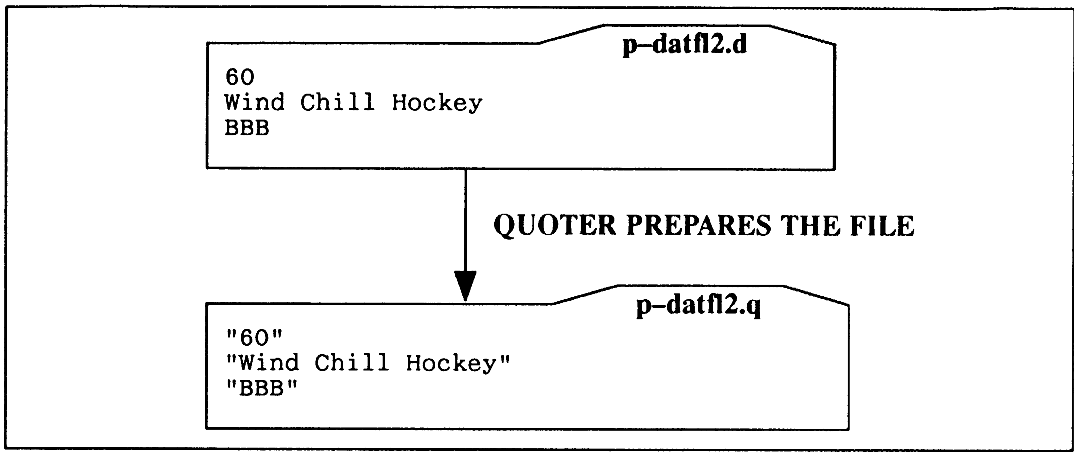
```
60
Wind Chill Hockey
BBB
61
Low Key Checkers
DKP
62
Bing's Ping Pong
SLS
```

`Quoter` is an executable program. To run `quoter`, use the following commands from the PROGRESS editor:

Operating System	Using Quoter – Examples
UNIX	`UNIX VALUE(SEARCH("quoter")) p-datfl2.d >p-datfl2.q`
DOS & OS/2	`DOS quoter p-datfl2.d >p-datfl2.q`
VMS	`VMS PROGRESS/TOOLS=QUOTER/OUTPUT=p-datfl2.q p-datfl2.d`
BTOS/CTOS	`BTOS VALUE (SEARCH("quoter.run")) p-datfl2.d >p-datfl2.q`

For the log maintenance command, `p-datfl2.d` is the name of the data file you are supplying to `quoter` while `p-datfl2.q` is the name of the file in which you want to store `quoter's` output.

In the UNIX example, the SEARCH function locates the directory where you installed PROGRESS, which is where the `quoter` program is stored. This way of accessing `quoter` will work even if the PROGRESS directory is not in your UNIX search path. The UNIX statement must be followed by either a UNIX command or the value of an expression. Since the SEARCH function returns an expression, you must use the VALUE keyword to indicate you want to use the value of that expression.



Here is the `p-datfl2.q` file:

p-datfl2.q

```
"60"
"Wind Chill Hockey"
"BBB"
"61"
"Low Key Checkers"
"DKP"
"62"
"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? That is, your data file looks like this:

```
p-datfl3.p

60 Wind Chill Hockey BBB
61 Low Key Checkers  DKP
62 Bing's Ping Pong  SLS
```

Suppose you wanted to use this file as the input source to create customer records for customers 60, 61, and 62. Here is the procedure that does that:

```
p-chgin3.p

   DEFINE VARIABLE data AS CHARACTER FORMAT "x(80)".

1.  IF OPSYS = "MSDOS"
      DOS quoter p-datfl3.d > p-datfl3.q.
2. ELSE IF OPSYS = "OS2"
      OS2 quoter p-datfl3.d > p-datfl3.q.
   ELSE IF OPSYS = "UNIX"
      UNIX VALUE(SEARCH("quoter")) p-datfl3.d > p-datfl3.q.
   ELSE IF OPSYS = "VMS"
      VMS "PROGRESS/TOOLS=quoter/OUTPUT=p-datfl3.d p-datfl3.q".
   ELSE IF OPSYS = "BTOS"
      BTOS VALUE (SEARCH("quoter.run")) p-datfl3.d >p-datfl3.q.
3. INPUT FROM p-datfl3.q NO-ECHO.

   REPEAT:
4.    CREATE customer.
5.    SET data WITH NO-BOX NO-LABELS NO-ATTR-SPACE WIDTH 80.
   {  cust-num = INTEGER(SUBSTRING(data,1,2)).
6. {  name = SUBSTRING(data,4,17).
   {  sales-rep = SUBSTRING(data,22,3).
   END.

7. INPUT CLOSE.
```

Here is a step-by-step description of what this procedure is doing:

1. The OPSYS function returns a value of either MSDOS, OS2, UNIX, VMS, or BTOS depending on the operating system you are using.

   If you are using MSDOS, OS/2, UNIX, VMS, or BTOS the procedure runs `quoter`, using the `p-datfl3.d` file as input and sends `quoter's` output to the `p-datfl3.q` file in your working directory.

By using the DOS, OS/2, UNIX, VMS, or BTOS statements, you can run quoter directly within the procedure. The process of reformatting the data is transparent. This procedure will also work on DOS, OS/2, UNIX, VMS, or BTOS without modifications.

2. `Quoter` takes the data from the `p-datf13.d` file and produces data that looks like this:

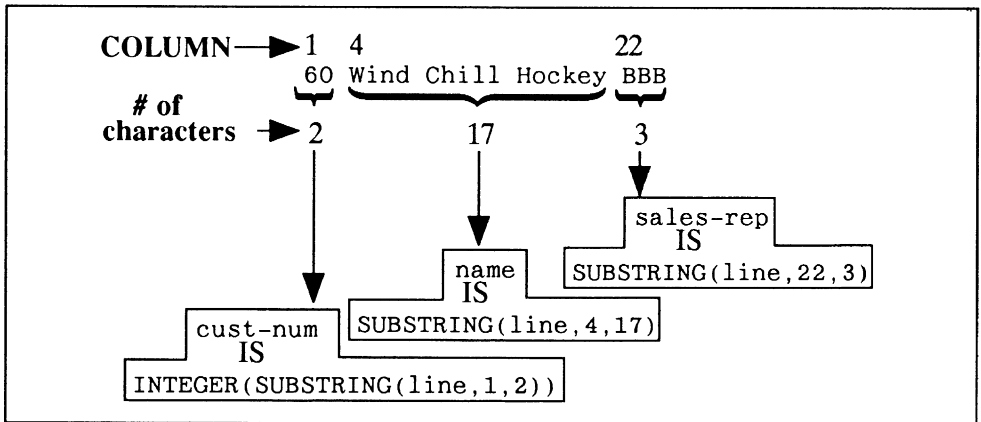
```
p-datf13.q

"60 Wind Chill Hockey BBB"
"61 Low Key Checkers  DKP"
"63 Bing's Ping Pong  SLS"
```

3. The INPUT FROM statement redirects the input stream to get input from the `p-datf13.q` file.

4. The CREATE statement creates an empty customer record.

5. The SET statement uses the first quoted line in the `p-datf13.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.

6. The SUBSTRING functions take the appropriate pieces of the data in the line variable and store that data in the `cust-num`, `name`, and `sales-rep` fields.



7. Because PROGRESS assumes that all the data in the `p-datf13.q` file is character data, you must use the INTEGER function to convert the `cust-num` data to an integer value.

8. The INPUT CLOSE statement closes the input stream coming from the `p-datf13.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. Refer to Chapter 3 of the *System Administration II: General* for information about options for the `quoter` utility.

9.3.4 Other File Formats

We have looked at how to use quoter to prepare files that have data in the following formats:

```
60 Wind Chill Hockey BBB
```

```
60
Wind Chill Hockey
BBB
```

You can use quoter to prepare files that are 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 (,):

p-datfl4.d

```
60,Wind Chill Hockey,BBB
61,Low Key Checkers,DKP
62,Bing's Ping Pong,SLS
```

You can use a special option, -d in UNIX, BTOS/CTOS, DOS, or OS/2, and /DELIMITER= in VMS, to tell quoter what character separates fields. Note that in VMS you must use quotation marks (" ") around the delimiter character. For example:

p-chgin4.p

```
 IF OPSYS = "MSDOS"
 DOS quoter -d , p-datfl4.d > p-datfl4.q.
ELSE IF OPSYS = "OS2"
 OS2 quoter -d , p-datfl4.d > p-datfl4.q.
ELSE IF OPSYS = "UNIX"
 UNIX VALUE (SEARCH("quoter")) -d , p-datfl4.d > p-datfl4.q.
ELSE IF OPSYS = "VMS"
 VMS "PROGRESS/TOOLS=quoter/OUTPUT=p-datfl4.q/DELIMITER="","""
       p-datfl4.d".
ELSE IF OPSYS = "BTOS"
 BTOS VALUE (SEARCH("quoter.run")) -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 (UNIX, BTOS/CTOS, DOS, and OS/2) or the DELIMITER qualifier (VMS) tells quoter that a comma (,) is the delimiter between each field in the data file. Here is the output of quoter:

p-datfl4.q

```
"60" "Wind Chill Hockey" "BBB"
"61" "Low Key Checkers" "DKP"
"62" "Bing's Ping Pong" "SLS"
```

This data file is in the standard blank delimited PROGRESS format. If your data file doesn't 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 (DOS, OS/2, UNIX, and BTOS/CTOS) or /COLUMNS (VMS).

You use the -c option or /COLUMNS to identify the columns in which fields begin and end. Note that in VMS you must use quotation marks (" ") around the numbers that identify the columns. For example, suppose your file looks like this:

p-datfl5.d

```
60 Wind Chill Hockey BBB
61 Low Key Checkers  DKP
62 Bing's Ping Pong  SLS
```

Here is the procedure that uses this data file to create customer records:

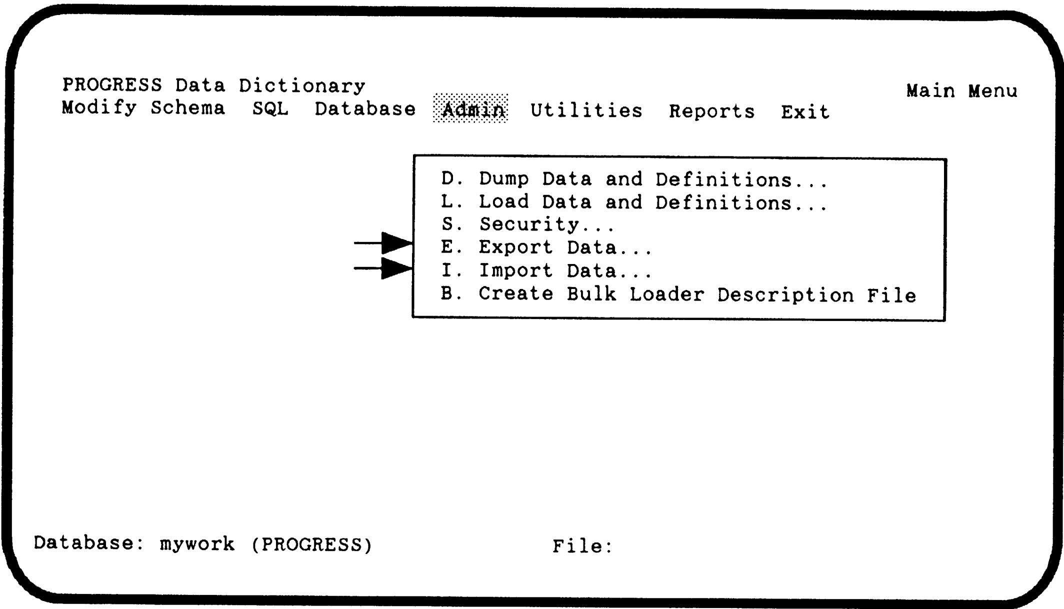
p-chgin5.p

```
IF OPSYS = "MSDOS"
 DOS quoter -c 1-2,4-20,22-24 p-datf15.d > p-datf15.q.
ELSE IF OPSYS = "OS2"
 OS2 quoter -c 1-2,4-20,22-24 p-datf15.d > p-datf15.q.
ELSE IF OPSYS = "UNIX"
 UNIX VALUE(SEARCH("quoter")) -c 1-2,4-20,22-24 p-datf15.d > \
     p-datf15.q.
ELSE IF OPSYS = "VMS"
 VMS "PROGRESS/TOOLS=quoter/COLUMNS=""1-2,4-20,22-24""
     /OUTPUT=p-datf15.q p-datf15.d".
ELSE IF OPSYS = "BTOS
 BTOS VALUE (SEARCH("quoter.run")) -c 1-2,4-20,22-24
     p-datf15.d >p-datf15.q.
INPUT FROM p-datf15.q NO-ECHO.
REPEAT:
  CREATE customer.
  SET cust-num name sales-rep.
END.

INPUT CLOSE.
```

You can also use quoter interactively to reformat your data. To use quoter interactively, access the Data Dictionary Main Menu, type u to select Utilities, and type q to select Quoter Functions.

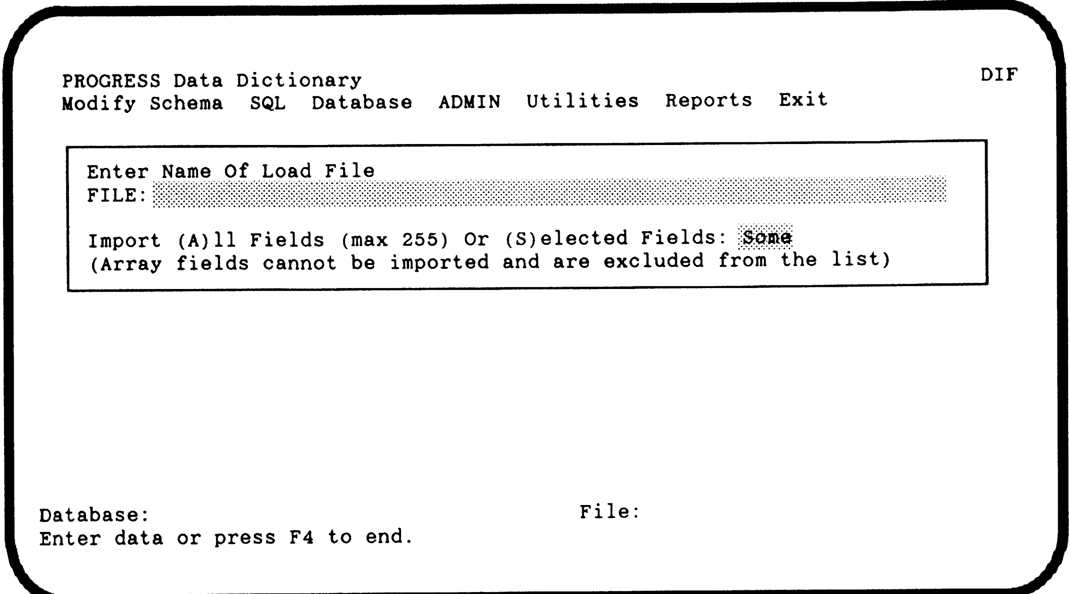
9.3.5 Loading/Exporting DIF and SYLK Files

The DIF (Data Interchange Format) and SYLK (SYmbolic LinK) file formats are used to store spreadsheets (Lotus 1-2-3 or Multiplan, for example) external to the host program. You can use the PROGRESS Data Dictionary to transfer data between PROGRESS databases and spreadsheets that support the DIF or SYLK formats. From the Data Dictionary Main Menu, type **a** to select the `Admin` submenu. The following screen appears:



9.3.6 Loading DIF or SYLK Files

To convert DIF or SYLK files to PROGRESS database files, type **i** to select Import Data. Type **d** to select DIF or **s** to select SYLK. The Data Dictionary displays a list of all your database files. Choose a file to hold the incoming DIF or SYLK file and press RETURN. The following screen appears:



In the FILE field, enter the name of the DIF or SYLK file to be imported (the default value for the FILE field is the name of the file you selected with a .dif or .sl extension) and press RETURN.

In the Import (A)ll Fields (max 255) Or (S)elected Fields field, type **a** if you want to import all the fields from your DIF or SYLK file or type **s** if you want to select only some of the fields (the default value for this field is Some) and press RETURN.

If you select All, the Data Dictionary loads the worksheet data. If you select Some, the Data Dictionary lists all the fields in the DIF or SYLK file. To select a field, use your space bar to highlight the field and press RETURN. Select the fields in the order you want them to be loaded into your PROGRESS database file. To skip a column in the worksheet, do not press RETURN while the field is highlighted. Press GO to load the fields.

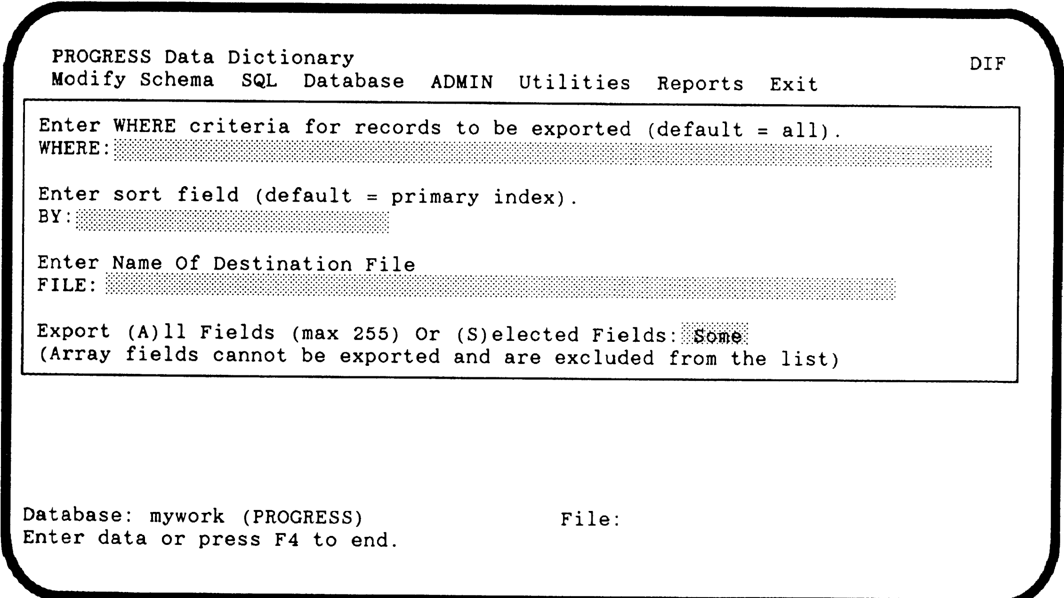
Data can be loaded in up to 39 non-array fields. Each row in the worksheet becomes a record in the database file. For each row, worksheet values are loaded column by column into the specified record fields.

NOTE: When DIF or SYLK file data is loaded, new database records are created. That is, existing records cannot be updated. If an existing PROGRESS file has a unique index, an attempt to load into that file a record having the same index key as an existing record fails; no fields are loaded for this record.

9.3.7 Exporting PROGRESS Database Files

To convert PROGRESS database files to DIF or SYLK format, type **e** at the `Admin` submenu to select `Export Data`. Type **d** to select `DIF` or **s** to select `SYLK`. The Data Dictionary lists all the file names in your PROGRESS database.

Use you space bar to highlight the file you want and press RETURN. The following screen appears:



Specify record selection criteria (for example, WHERE: cost > 100) and press RETURN. A record becomes a row in a worksheet, and the worksheet starting point is always row 1, column 1. Specify how to sort the selected records (for example, sort BY: name) and press RETURN.

Name the destination file (the default value is the name of the file you selected with a `.dif` or `.sl` extension) and press RETURN. The DIF or SYLK file is created in the current directory.

In the `Export (A)ll Fields (max 255) Or (S)elected Fields` field, type **a** if you want to import all the fields from your PROGRESS file or type s if you want to select only some of the fields (the default value for this field is `Some`) and press RETURN.

Each field corresponds to a worksheet column. You can export a maximum of 39 fields, and array fields cannot be exported. PROGRESS exports only the first 39 non-array fields if you attempt to export all fields.

If you select `All`, the Data Dictionary exports all the non-array fields in your PROGRESS file. If you select `Some`, the Data Dictionary lists all the fields in the DIF or SYLK file. To select a field, use your space bar to highlight the field and press [RETURN]. Select the fields in the order you want them to be exported into your destination file. To skip a column in the worksheet, do not press [RETURN] while the field is highlighted. Press [GO] to export the fields.

9.3.8 Using EXPORT

Sometimes you send data to a file knowing that later that data will be used by a PROGRESS procedure. If so, then you also know that the data file must be in standard format, 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, use 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

```
OUTPUT TO p-datf16.d.

FOR EACH customer:
  EXPORT cust-num name sales-rep.
END.
```

Here is the output of this procedure:

p-datf16.d

```
1 "Second Skin Scuba" "SLS"
2 "Match Point Tennis" "DKP"
3 "Off The Wall" "BBB"
4 "Pedal Power Cycles" "BBB"
              .
              .
```

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.

If you need to prepare a data file in a fixed format, perhaps for use by another system, you can do that as well. For example:

p-putdat.p

```
OUTPUT TO p-datfl7.d.

FOR EACH customer:
  PUT cust-num AT 1 name AT 10 sales-rep AT 40 SKIP.
END.
```

Here is the output produced by this procedure:

p-datfl7.d

```
1        Second Skin Scuba                   SLS
2        Match Point Tennis                  DKP
3        Off The Wall                        BBB
4        Pedal Power Cycles                  BBB
                  .
                  .
                  .
```

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.

9.3.9 Using IMPORT

The IMPORT statement is the counterpart of the EXPORT statement. It reads an input file into PROGRESS procedures, one line at a time. The files must be in standard format, that is, all character fields enclosed in quotes. Typically these files are created with the EXPORT statement, but you can also read files formatted by quoter.

The following example shows IMPORT reading the file exported by procedure p-export.p shown in the previous section.

p-import.p

```
INPUT FROM p-datfl6.d.

FOR EACH customer:
IMPORT cust-num name sales-rep.
END.
```

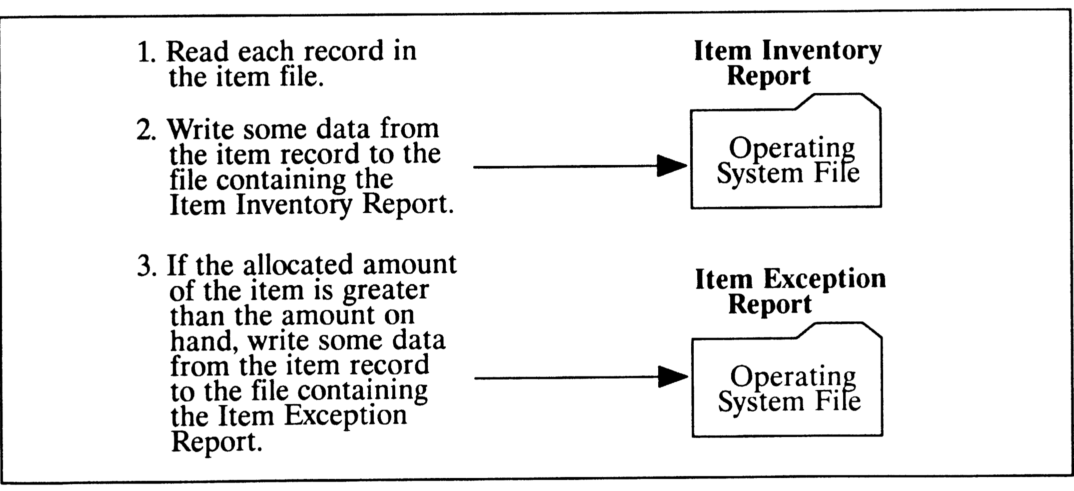
For more information about IMPORT, see the *PROGRESS Language Reference* manual.

9.4 DEFINING ADDITIONAL 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 may find that having just one input stream and one output stream is not enough for particular procedures. That is, you may 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. Here is what the scenario looks like:



As you can see in this diagram, in the case of 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, or other non-terminal devices. They cannot be the terminal; i.e. you cannot say OUTPUT STREAM a TO TERMINAL. Here is the procedure for the double report example we just discussed:

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.
4.  DISPLAY STREAM exceptions "Item Exception Report"
      SKIP(2) WITH CENTERED NO-BOX FRAME except-frame.

5.  FOR EACH item:
6.    IF on-hand < alloc THEN DO:
7.      DISPLAY STREAM exceptions item-num idesc
          on-hand alloc WITH FRAME exitem DOWN.
8.      excount = excount + 1.
9.      exception = TRUE.
      END.
10.   DISPLAY STREAM rpt item-num idesc
        WITH NO-LABELS NO-BOX.
11.   IF exception THEN
        DISPLAY STREAM rpt "See Exception Report".
      exception = FALSE.
    END.
12. DISPLAY STREAM exceptions SKIP(1) excount
      WITH FRAME exc SIDE-LABELS.
13. DISPLAY STREAM rpt WITH FRAME exc.

14. OUTPUT STREAM rpt CLOSE.
    OUTPUT STREAM exceptions CLOSE.
```

The numbers listed on the left side of the shaded box are not part of the procedure. They are associated with each of the items in the following list.

Here is a step-by-step description of what this procedure is doing:

1. The SET statement asks which names 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.
3. 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.
4. 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 has the text "`Item Inventory Report`" in it.
5. This DISPLAY statement also displays text but it uses the `exceptions` stream. The file you named for the Item Exception Report has the text "`Item Exception Report`" in it.
6. The FOR EACH block reads a single item record on each iteration of the block.
7. If the allocated amount of an item is larger than the on-hand amount of that item:
   - The DISPLAY statement displays, to the `exceptions` stream, some item data. After this DISPLAY statement finishes, the file you named for the Item Exception Report has item data for a single item in it.
   - The `excount` counter variable, defined at the start of the procedure, is incremented by 1. The value of this variable will be displayed at the end of the procedure so that you will know the total number of exception items in inventory.
   - The `exception` logical variable, defined at the start of the procedure, is set to TRUE.
8. The DISPLAY statement displays some item data to the `rpt` stream. After this DISPLAY statement finishes, the file you named for the Item Inventory Report has item data for a single item in it.
9. 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.
10. The DISPLAY statement displays, to the `exceptions` stream, the value of the `excount` variable. The value of this variable is the total number of exception items.

11. This DISPLAY statement displays, to the rpt stream, the value of the excount variable. Although the DISPLAY statement does not explicitly say what is being displayed, it does name the same frame, exc, as was used to display excount in the previous DISPLAY statement. That means that the exc frame already has the excount value in it. So all this second DISPLAY statement need do is name the same frame. It does not have to name the excount variable because the prior DISPLAY statement put the value in the frame.

12. The OUTPUT STREAM CLOSE statements close the rpt and exception streams, redirecting all further output to the default output destination.

9.5 SHARING STREAMS BETWEEN PROCEDURES

There are cases where you want two or more procedures to share the same input or output streams. The procedures below 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.
  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 st:
  DISPLAY STREAM phonelist cust-num name city st phone
          WITH NO-LABELS.
END.
```

The p-dispho.p procedure defines the SHARED STREAM phonelist, and displays the information from that stream on the screen. (The FOR EACH and DISPLAY statements in the p-dispho.p procedure should be put in the p-sstrm.p procedure for efficiency. It is in a separate procedure here to illustrate shared streams.)

You can see that 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.
- 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 it is defined.
- You define the stream as NEW GLOBAL when you want that stream to remain available even after the procedure containing the DEFINE NEW GLOBAL SHARED STREAM statement ends.

9.6 SUMMARY OF OPENING AND CLOSING STREAMS

Table 9-1 describes how you establish, open, use, and close default streams and streams you name.

Table 9–1: Using Streams

	UNNAMED STREAMS	NAMED STREAMS
How you 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: DEFINE STREAM, DEFINE NEW SHARED STREAM, DEFINE SHARED STREAM, DEFINE NEW GLOBAL SHARED STREAM.
How you 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 INPUT STREAM *name* FROM.
How you use the stream	All data handling statements use the stream by default.	Name the opened stream in the data handling statement that you want to use the stream.
How you 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. Also closed if you do another open.	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.

9.7 PROCESSES AS INPUT AND OUTPUT STREAMS (UNIX)

On UNIX systems, you can use the INPUT THROUGH statement to import data into PROGRESS from another process. Similarly, you can use the OUTPUT THROUGH statement to pipe data from PROGRESS to another UNIX process. For more information, see the INPUT THROUGH and OUTPUT THROUGH sections of the *PROGRESS Language Reference* manual.

9.8 TWO-WAY STREAMS: INPUT-OUTPUT THROUGH (UNIX)

You use the INPUT-OUTPUT THROUGH statement to pipe the output of a UNIX 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, see the INPUT-OUTPUT THROUGH section of the *PROGRESS Language Reference* manual.

9.9 I/O REDIRECTION FOR BATCH JOBS (UNIX and OS/2)

You can the < and > symbols on the PROGRESS command line to redirect I/O for batch jobs. For details, see the Batch (-b) startup option in Chapter 3 of *System Administration II: General.*

Chapter 10
Using PROGRESS Workfiles

By now, you have used the Dictionary to define database files, fields, and indexes. You have probably also written procedures that use those definitions. In general, you will find that you are able to accomplish most of your application development work using standard database files, fields, indexes and variables.

There are some cases, however, where you may want to consider using temporary files, called "work files". This chapter tells you how to use work files to:

- Produce categorized reports in which you do not know the value of each possible category until you have made a pass through the file.
- 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.

This chapter describes work files and how to use them to do the listed kind of work.

NOTE: Work files cannot be accessed using SQL statements.

10.1 WHAT ARE WORK FILES?

In essence, work files are to database files what variables are to database fields. Like a variable, a work file:

- Is a temporary file that is stored in memory rather than in the database and can either be local to a single procedure or shared between multiple procedures.
- Must be defined in any procedure that uses it.

- Can be defined to be LIKE a database file (much like a variable can be defined to be LIKE a database field).
- Workfiles do not have indexes, therefore you cannot do a "FIND *workfile*". However, you can do a "FIND FIRST *workfile*." The figures in this chapter illustrate this feature.

Table 10-1 lists the differences between database files and work files.

Table 10-1: Database File and Work File Differences

PROGRESS FEATURE	DATABASE FILE	WORK FILE
Database manager	PROGRESS uses the database manager and server (for multi-user systems) when working with database files.	PROGRESS does not use the database manager or server (for multi-user systems) when working with work files.
Indexes	You can define indexes for database files.	You cannot define indexes for work files.
Record deletion	To remove database records from the database, you must use the DELETE statement to explicitly delete those records.	If you do not explicitly delete the records in a work file, PROGRESS discards those records and the work file at the end of the procedure which initially defined the work file.
Record movement	A database file can move data between a record buffer and a database record.	Work files cannot move data between a record buffer and a database record.
Multi-user record access	Multiple users can access the same database file at the same time.	Users do not have access to each other's work files.
Transactions	Automatically started for certain blocks and statements.	You must explicitly start transactions.

NOTE: Because PROGRESS stores and processes work files in memory, the number of work files and the number of records in a work file that you can use is limited by the Local Buffer Size (-l) startup option. Use a value of 10 with the -l option to run the examples in this chapter. For more information about this startup option, see Chapter 2 of *System Administration II: General*.

10.2 USING WORK FILES TO PRODUCE CATEGORIZED REPORTS

Chapter 12 of the *PROGRESS Language Tutorial* explained how to write reports, including "control break" reports. Control break reports list information broken down into categories. For example, every item in the demo database belongs to a certain product line. Every item also has an on-hand value (how many items are in inventory) and a cost value associated with it. Suppose you want to produce a report that lists the value of the inventory for each product line in the item file.

Here is a procedure that produces that report:

p-wrk1.p

```
FOR EACH item BREAK BY prod-line:
  ACCUMULATE cost * on-hand (SUB-TOTAL BY prod-line).
  IF LAST-OF(prod-line)
  THEN DISPLAY prod-line
               ACCUM SUB-TOTAL BY prod-line (cost * on-hand).
END.
```

Here is some of the output produced by this procedure:

Product line	TOTAL
A10	383.52
A11	306.00
A3	1,653.30
A4	2,250.00
A5	4,625.00
A7	7,856.00
A8	2,555.78
A9	172.27
B11	7,713.37
.	.
.	.
.	.

This procedure uses a control break (BREAK BY prod-line) to logically separate the item file into product line categories. However, because you do not know what all the product lines are or even how many product lines there are, the procedure must make two passes through the file:

- The BREAK keyword causes the first pass through the file: PROGRESS sorts the `item` file and puts break points into the sort table.

- The second file pass occurs when PROGRESS goes through the sort file, using the recids in that file to read item records from the item file. The recid is the internal database identifier that PROGRESS associates with every database record.

Figure 10–1 shows how the procedure processes the item file.

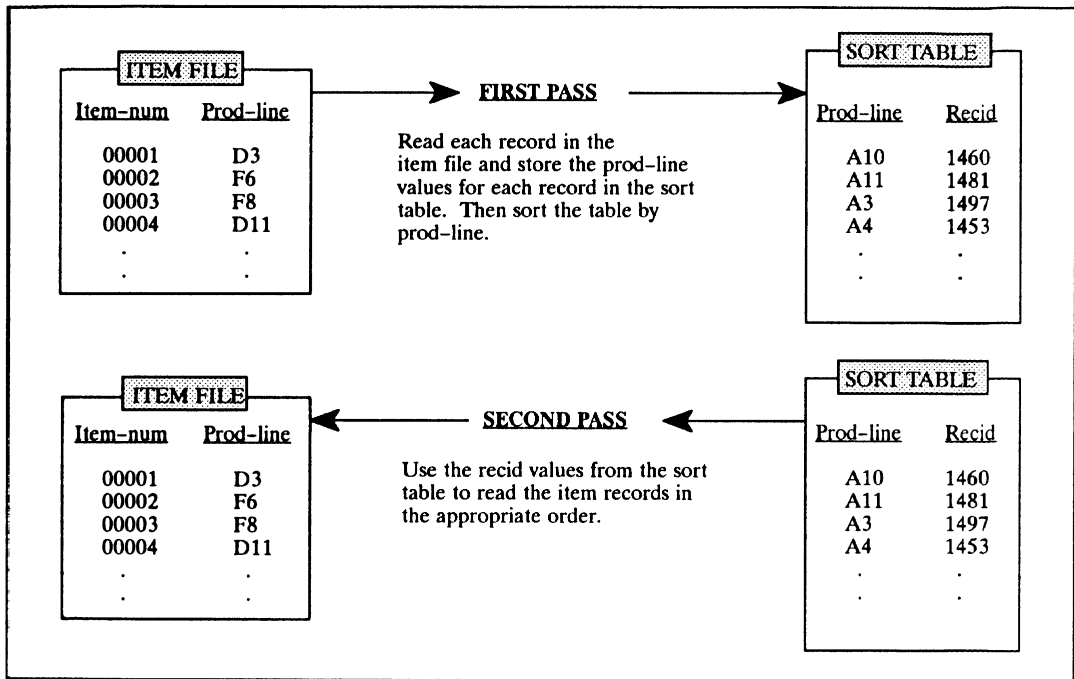


Figure 10–1: Processing a File for a Categorized Report

The following procedure uses work files to produce the same report:

p-wrk2.p

```
1. DEFINE WORKFILE pline
     FIELD w-prod-line LIKE item.prod-line
     FIELD w-inv-value AS DECIMAL FORMAT "->>>,>>>,>>9.99"
                       LABEL "Inventory Value".

   FOR EACH item:
2.   FIND FIRST pline WHERE
       pline.w-prod-line = item.prod-line NO-ERROR.
     IF NOT AVAILABLE pline
     THEN DO:
       CREATE pline.
       pline.w-prod-line = item.prod-line.
     END.
3.   pline.w-inv-value = pline.w-inv-value +
                         (item.cost * item.on-hand).
   END.
4. FOR EACH pline BY w-prod-line:
     DISPLAY w-prod-line w-inv-value.
   END.
```

This procedure does the following:

1. The DEFINE WORKFILE statement defines a temporary file, `pline`, that contains two fields, `w-prod-line` and `w-inv-value`. The `w-prod-line` field has the same definition as the `prod-line` field in the `item` file.

2. For each of the records in the `item` file, the procedure checks to see if there is a `pline` record for the product line to which the item belongs. If there isn't, the procedure creates a record and stores the product line value in the `pline` record.

3. The = (ASSIGNMENT) statement accumulates the value of the inventory for each product line by adding the inventory value of the current `pline` record to the existing inventory value.

4. The FOR EACH block sorts the `pline` file and displays the inventory value of each product line.

Figure 10–2 shows the file processing this procedure does.

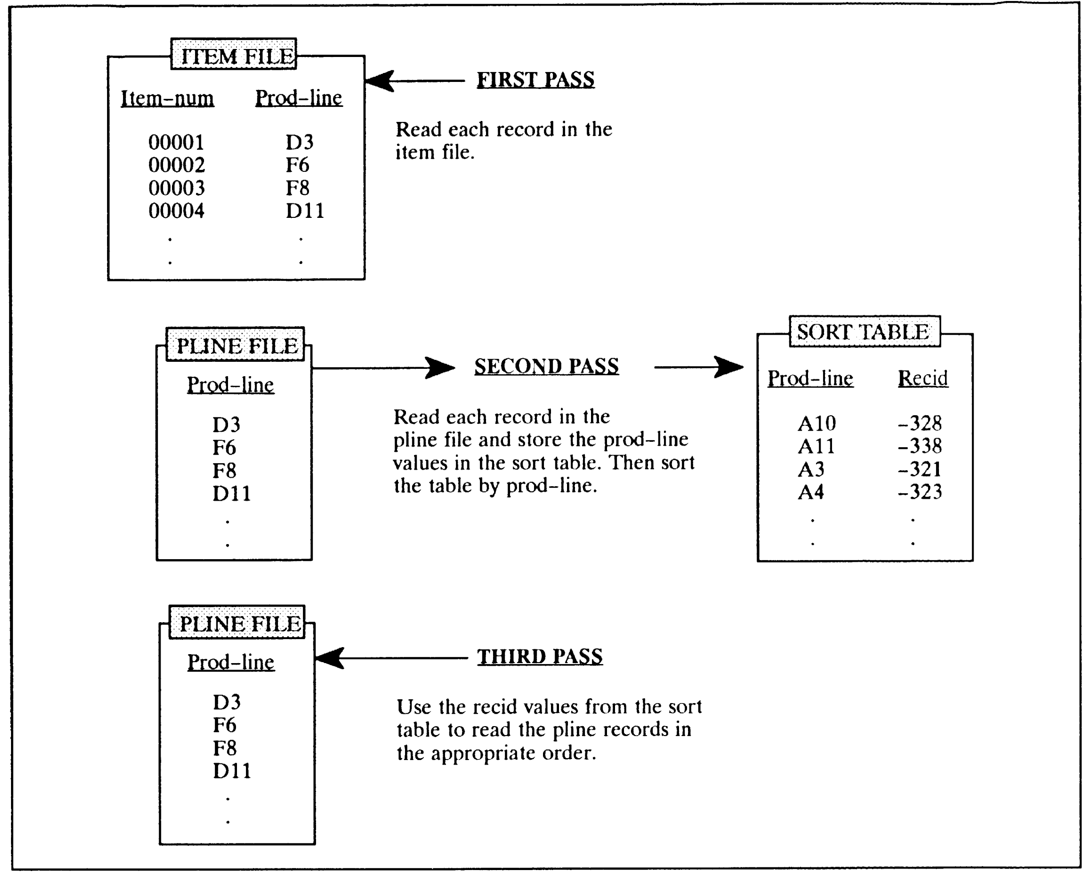


Figure 10–2: Using Work Files to Produce a Categorized Report

The work file version of the procedure also makes a pass through the item file. But it does not need to first sort the entire item file since it can create new work file records for each new product line as it encounters that product line. The procedure then makes a pass through the `pline` work file, building a sort table for that file. Finally, the procedure makes a second pass through the `pline` file, retrieving the pline records in the appropriate order.

You might be wondering: "If the `p-wrk1.p` procedure makes two file passes and the `p-wrk2.p` procedures makes three file passes, why is the work file procedure any better than the procedure that only uses database files?". There are two reasons:

- First, the `pline` file is far smaller than the `item` file (in a typical database). Therefore, passes of the `pline` file take much less time than passes of the `item` file.
- Second, all file processing for work files is done in memory and is faster than file processing for database files.

10.3 USING WORK FILES TO COLLECT AND MANIPULATE DATA

Suppose now 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 product line.
- Display, for the item with the largest inventory value in each product line, the inventory value of that item.

To prepare this report without using work files, you would 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 product lines in the database. You may not know this number when you write the procedure. When you use work files, you can store item quantities and inventory values in fields in the work file. For example:

p-wrk3.p

```
1. DEFINE WORKFILE pline
     FIELD w-prod-line LIKE item.prod-line
     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 pline
       WHERE pline.w-prod-line = item.prod-line NO-ERROR.
     IF NOT AVAILABLE pline
     THEN DO:
       CREATE pline.
       pline.w-prod-line = item.prod-line.
     END.
3.   IF cost * on-hand > w-item-value
     THEN DO:
       w-item-value = cost * on-hand.
       w-item-num = item.item-num.
     END.
4.   pline.w-inv-value = pline.w-inv-value +
                         (item.cost * item.on-hand).
   END.
5. FOR EACH pline BY w-prod-line:
     DISPLAY w-prod-line w-inv-value w-item-num w-item-value.
   END.
```

This procedure does the following:

1. The DEFINE WORKFILE statement defines a temporary file, pline, that contains four fields, `w-prod-line` (representing workfile-product-line), `w-inv-value` (representing workfile-inventory-value), `w-item-value` (representing the inventory value of a particular item), and `w-item-num`.

2. For each of the records in the item file, the procedure checks to see if there is a `pline` record for the product line to which the item belongs. If there isn't, the procedure creates a record and stores the product line value in the `pline` record.

3. If the value of the current item is greater than the value of the previous item, the = (ASSIGNMENT) statements store the value and number of that item in the corresponding work file fields.

4. The = (ASSIGNMENT) statement accumulates the inventory value for the product line.

5. The FOR EACH block displays, for each product line, the inventory value of the product line, and the number and value of the highest value item in that product line.

Also, using work files gives you the ability to sort on any value you want. For example, if you wanted to sort on the inventory value of the product lines, you would use the BY w-inv-value option on the FOR EACH block. Without work files, you would not be able to do this sort nearly as easily.

10.4 USING WORK FILES TO DO COMPLEX SORTING

As you know, PROGRESS orders database records by using indexes, and then placing entries for newly created records in order in each index. Although work files have no indexes, you can position work file records where you want because whenever you create a work file record, PROGRESS places it after the record that was found last in that work file. This flexibility lets you keep the work file records in a particular order, avoiding the necessity of sorting at a later point in the procedure. For example, here is a modified version of the `p-wrk3.p` procedure:

p-wrk4.p

```
DEFINE WORKFILE pline
  FIELD w-prod-line LIKE item.prod-line
  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 pline WHERE
    pline.w-prod-line >= item.prod-line NO-ERROR.
  IF NOT AVAILABLE pline OR pline.w-prod-line > item.prod-line
  THEN DO:
    FIND PREV pline NO-ERROR.
    CREATE pline.
    pline.w-prod-line = item.prod-line.
  END.
  IF cost * on-hand > w-item-value
  THEN DO:
    w-item-value = cost * on-hand.
    w-item-num = item.item-num.
  END.
  pline.w-inv-value = pline.w-inv-value +
                      (item.cost * item.on-hand).
END.
FOR EACH pline:
  DISPLAY w-prod-line w-inv-value w-item-num w-item-value.
END.
```

This procedure uses the following logic to maintain the pline work file in product line order:

The FIND FIRST statement tries to find a pline record whose product line is the same or larger than that of the current item.

- If it finds a `pline` record with the same product line as the item, no new `pline` record is created.
- If the product line of the `pline` record is larger than that of the current item, or if the `pline` record does not exist, PROGRESS finds the previous `pline` record and creates the new `pline` record after that record in the file.

Figure 10-3 shows how this procedure keeps the `pline` work file in order by product line.

10
Workfiles

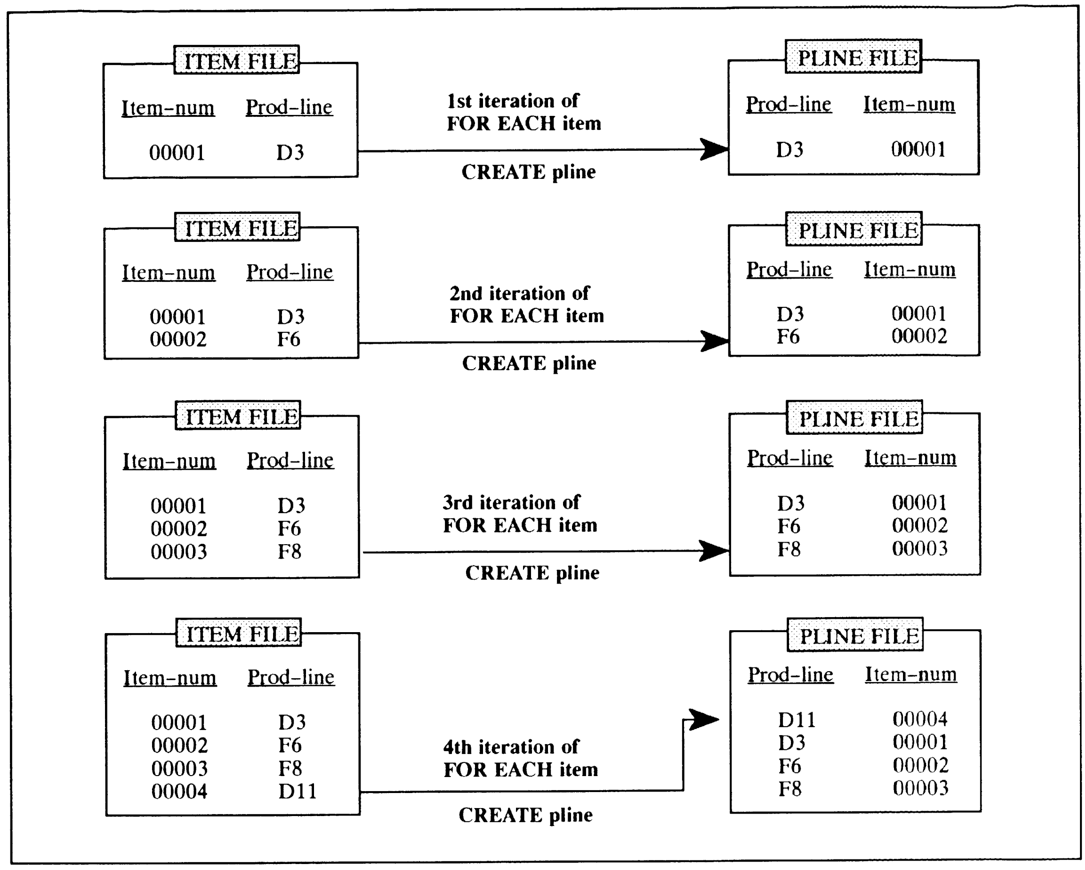


Figure 10–3: Maintaining Work Files in Sorted Order

By maintaining the `pline` file in product line order as new `pline` records are created, there is no need to sort the file in the final FOR EACH block. When that block displays the `pline` records, they are already sorted by product line.

There is another advantage to using this method of sorting as opposed to the method used in `p-wrk3.p`. Figure 10–4 illustrates this advantage.

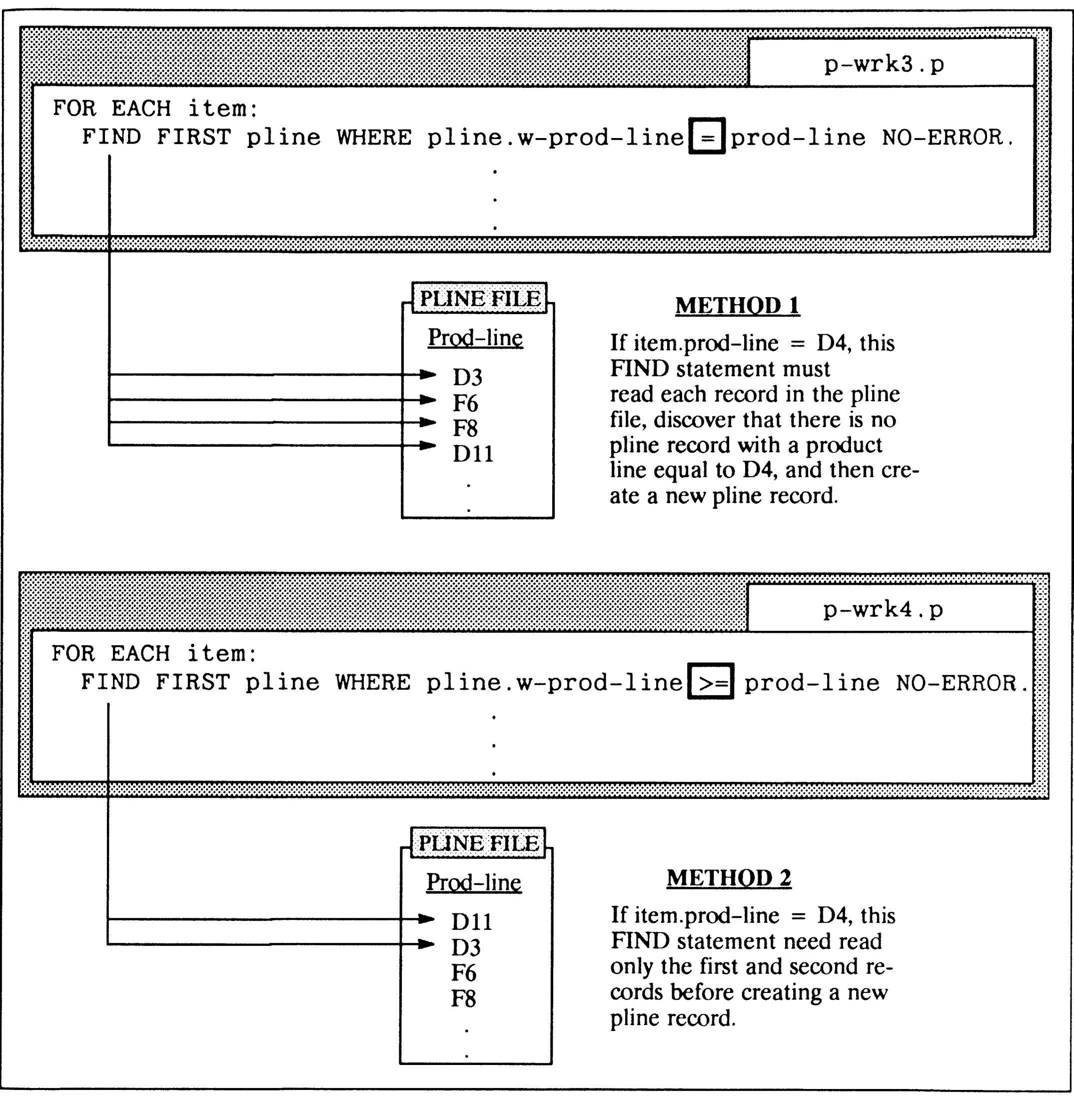


Figure 10–4: Comparing Two Sorting Methods

In Method 2, because the records in the `pline` file are already in sorted order, the find is much more efficient than in Method 1. In the case of an item with a product line that already exists in the `pline` file, both methods are equally efficient on average.

10.5 USING WORK FILES TO DO "CROSS-TAB" REPORTS

A "cross-tab" report typically lists information in tabular, or spreadsheet, format. For example, suppose you want a report that looks like the following:

Order Value by Product Line and Region

Product	East	West	Central
F8	0.00	0.00	0.00
D11	0.00	735.08	0.00
E10	408.00	0.00	0.00
A7	3,150.00	4,000.00	675.00
D7	5,625.00	0.00	1,375.00
D3	3,326.00	0.00	0.00
A5	0.00	1,060.20	725.40
D4	247.17	0.00	0.00
A4	0.00	951.00	1,397.00

.
.
.

This report lists the order value for each of the product lines in each of the three regions.

In general, if you think of this kind of report in terms of rows and columns (in the report shown above, product lines are the rows and regions are the columns), there are two ways to produce this report without using work files:

1. Usually, your report has more rows than columns or more columns than rows. For the sake of this example, assume that, as in the report shown above, the number of rows is larger than the number of columns. Sort the records represented by each row (in the report shown above, you would sort by product line) and then accumulate values acquired during that sort into an array whose elements represent each of the columns (in the report shown above, these values would be the order value for each sales region).

2. Displays results whenever PROGRESS encounters a new product line value.

Using work files simplifies the task of producing cross-tab reports: you don't have to sort the data and you need not be concerned with the dimensions of the table as you do when using arrays. Here is the procedure that produces this report using work files:

p-wrk5.p

```
DEFINE WORKFILE pline
  FIELD w-prod-line LIKE item.prod-line
  FIELD w-ord-value AS DECIMAL FORMAT "->>>,>>>,>>9.99" extent 3.

DEFINE VARIABLE reg-code AS INTEGER.
DEFINE VARIABLE reg-names AS CHARACTER
                          INITIAL "East,West,Central".

FOR EACH order, customer OF order, EACH order-line OF order,
  item OF order-line:

  FIND FIRST pline WHERE pline.w-prod-line = prod-line NO-ERROR.
  IF NOT AVAILABLE pline
  THEN DO:
    CREATE pline.
    pline.w-prod-line = prod-line.
  END.
  reg-code = LOOKUP(customer.sales-reg,reg-names).
  w-ord-value[reg-code] = w-ord-value[reg-code]
                        + (order-line.price * order-line.qty).
END.

FOR EACH pline BY pline.w-prod-line

  FORM HEADER "Product" "East" AT 20 "West" AT 40 "Central" AT 60
    WITH TITLE "Order Value by Product Line and Region".

  FORM pline.w-prod-line pline.w-ord-value[1] TO 27
                         pline.w-ord-value[2] TO 47
                         pline.w-ord-value[3] TO 67
    WITH NO-LABELS.

  DISPLAY pline.w-prod-line pline.w-ord-value.
END.
```

The size and number of records you can create in a work file depends on the amount of memory you allocate with the Total Private Database Buffers (-I) startup option. See *System Administration II: General* for more information about startup options and memory use.

Chapter 11
Providing Application Security

After you develop an application, you may want to add **application security**. Application security protects your database by allowing users to access only the data and procedures that they are authorized to access. This chapter covers the following topics:

- PROGRESS application security basics.
- Establishing userids and passwords.
- Checking userids at run-time.
- Performing activities-based security checking.
- Using file- and field-level security at compile time.
- Performing security administration.

If you are working with a multi-database application, see also Chapter 13 in this manual.

11.1 INTRODUCTION TO PROGRESS APPLICATION SECURITY

The PROGRESS Data Dictionary gives you the ability to establish user identifications (userids) and passwords. For applications on DOS and OS/2 systems, the Data Dictionary provides a level of security not provided by the operating system. For applications on UNIX and VMS systems, PROGRESS provides a method for implementing security that is portable across operating systems. PROGRESS also allows you to implement security for each database, independent of operating system security.

At startup, your application can prompt for a userid and password and check that the userid is valid. This ensures that only known users can start the application. At runtime, the application can check the userid that was established at startup to be sure that only authorized users run individual procedures.

One way to use security is to list, for each procedure, those userids that may access it. However, hard-coding userids into procedures is not always practical. If you distribute your application to many sites or if the people who run the application change frequently, you must recompile your procedures each time a new userid is added or an old one is deleted.

A more practical approach is to define an activities file, which defines the different procedures within your application as "activities" and assigns a list of users authorized to run each activity. The application can simply check that the userid established at start-up is in the list of authorized users for that activity. The system administrator at each site can maintain this list of users.

These security methods protect precompiled procedures but they do not prevent users from writing their own procedures to access the database. Therefore, the Data Dictionary also lets you name those users who can read, write, create, or delete files and individual fields within those files. These permissions are checked whenever a procedure is compiled.

All of these provisions require security measures of their own to establish who are the valid users and what access rights each user is given to the database. Typically, a security administrator performs this function.

Finally, if you are running PROGRESS on a UNIX or VMS system, you can use the operating system security features to further protect your object or database files. See *System Administration II: General* for information about operating system security.

11.2 ESTABLISHING USERIDS AND PASSWORDS

In PROGRESS, a userid is a 32-character string that is associated with a particular PROGRESS session. Like file names, userids must begin with a character from a–z or from A–Z. The name can consist of alphabetic characters, digits, and the characters #,$,%,&, –, and _. Userids are not case sensitive; they can be uppercase, lowercase, or any combination of these.

A password is a 16-character string that is known only to the user. When a user enters a password, it can be encoded with the ENCODE function. (See the *PROGRESS Language Reference* manual for a complete description of the ENCODE function.) Because ENCODE returns different values for upper and lowercase input, all PROGRESS passwords are case-sensitive.

The PROGRESS metaschema defines a file called `_User`, which contains fields to hold userids, passwords, and user names. Using the Data Dictionary, you or the system administrator at the user's site can add a record to this file for each of the users who are authorized to run your PROGRESS application.

You can establish a userid when the user starts your application by implementing a login procedure. You can check this userid at the beginning of any or all procedures to ensure that only authorized users run certain procedures.

Since the `_User` file is part of the empty database, anyone could write and compile programs to modify the security protections defined there. To prevent this, PROGRESS performs the following checks at run-time:

- When a userid is established, either at start-up or with SETUSERID, PROGRESS checks to see if the user has permission to create and delete records in the `_User` file. This information is saved.

- Whenever the user tries to create or delete a _User record, PROGRESS checks the permission information to make sure the user is allowed to do so.

11.2.1 Including Login Procedures in Your Application

You can prompt users to login to your application with a procedure called login.p, which is supplied in source form with PROGRESS 4GL/RDBMS. You can run login.p from your application to establish the userid and password of your users at start-up. Once the userid is established, individual procedures can check that the user is authorized to perform a specific task.

When you start PROGRESS, the prostart.p procedure tests to see if there are any entries in the _User file. PROGRESS always runs login.p if there are any entries in the _User file.

When PROGRESS calls login.p, login.p looks to see if records are defined in the _User file. If so, it prompts you to enter a userid and password. You must enter a userid and password that match one of the records in the _User file. You need to repeat this process for each database you want to connect to. PROGRESS gives you three tries. If you do not enter a correct userid and password after three tries, it exits from the application.

You can bypass login.p by pressing F4 even if entries exist in the _User file. However, you enter PROGRESS with a blank userid if you bypass login.p. Depending on the way security is set up on your system, having the blank userid can prevent you from performing certain activities in PROGRESS.

If no records are found in the _User file, login.p takes different steps depending on the operating system on which it is running. On UNIX and VMS systems, it takes the userid established when the user logged onto the system and assigns that userid to the user. On DOS and OS/2 systems, where no userid is maintained by the operating system, it assigns a blank userid to the user.

The login.p procedure uses the SETUSERID function to establish the userid. You can use SETUSERID to implement your own log-in procedure if you do not want to use login.p. (See the *PROGRESS Language Reference* manual for more information about the SETUSERID function.)

11.2.2 Establishing a Userid for Batch Jobs

If you are running PROGRESS or a PROGRESS application in batch or background mode, you can establish the userid with the -U and -P startup options. The -U option lets you supply a userid and the -P option lets you supply the password. The script you use to start the batch job on UNIX might look like this:

```
echo Please type the userid and password.
read user pswd
bpro dbname -p jobs.p -U $user -P $pswd
```

The script you use to start the batch job on VMS might look like this:

```
$ INQUIRE USERID "Please type the userid"
$ INQUIRE PASSWORD "Please type the password"
$ PROGRESS/BATCH/START=JOBS.P/USERID='USERID'/PASSWORD='PASSWORD'
```

See Chapter 3 in *System Administration II: General* for more information about the -U and -P startup options.

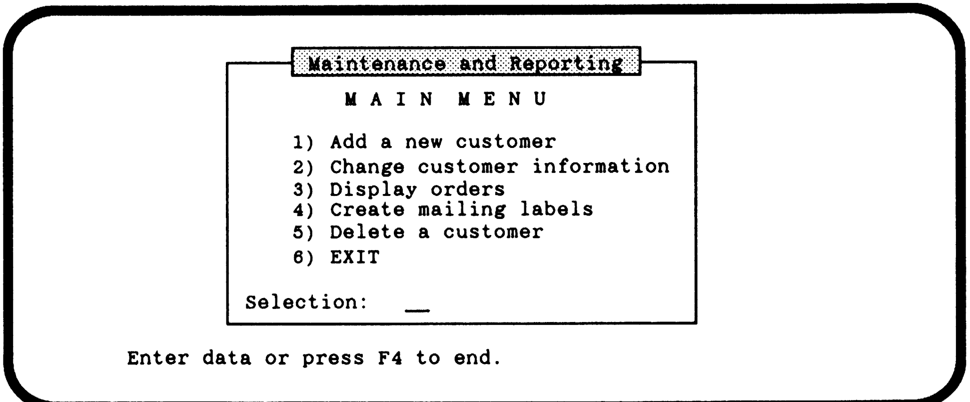
11.3 CHECKING USERIDS AT RUN-TIME

Suppose you want to define who can run certain procedures in your application. One simple way to provide this kind of security is by checking the userid of the user running the procedure. For example, run the following procedure.

p-csmnu3.p

```
DEFINE VARIABLE selection AS INTEGER FORMAT "9".
RUN login.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.
```

You see this screen.



The procedure `p-csmnu3.p` defines user access to itself by calling `login.p` before displaying the main menu. This establishes the userid of the person running the application.

Now suppose you want to define, on a per procedure basis, who can run each of the Maintenance and Reporting menu procedures. You can use the CAN-DO function to check the userid established by `login.p`. For example, you can limit the use of the `p-adcust.p` procedure to users with a userid of manager or salesrep. Here is how you can modify `p-adcust.p` 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 — annotation to the IF NOT CAN-DO … END. block)

The first part of `p-adcus2.p` handles security checking, making sure that the user is authorized to run the procedure. The CAN-DO function compares the values listed in the parentheses against the userid of the user running the procedure. If the userid does not match any of the values, the procedure displays a message and exits. If the userid does match one of the values, the procedure continues.

You can also use the USERID function to check userids in a procedure. Use this function when only one userid is allowed access to a procedure. For example:

```
p-adcus3.p

IF USERID <> "manager"
THEN DO:
  MESSAGE "You are not authorized to run this procedure.".   } Security
  RETURN.                                                      Checking
END.

REPEAT:
  INSERT customer WITH 2 COLUMNS.
END.
```

In this variation of p-adcust.p, if the userid of the user running the procedure is not manager, the procedure displays a message and exits. If the userid is manager, the procedure continues.

If you use either the CAN-DO function, or the USERID function, to compare the userid of a user with one or more userids you include in a procedure, you must modify and recompile that procedure whenever you want to change the userids allowed access to it. You can avoid having to make these changes by building a permissions file for activities in your application.

NOTE: For multiple-database users: if more than one database is connected, the USERID and CAN-DO functions require database-name qualification. For more information, see Chapter 13 in the *Programming Handbook.*

11.4 PERFORMING ACTIVITIES-BASED SECURITY CHECKING

Applications 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 of procedures that handle inventory.

You can do activities-based security checking in your application by building a permissions file for those activities. To use a permissions file, you must:

- Create a file that defines the kinds of access different users have to application activities.
- Include statements in application procedures to check the permissions file at run time.
- Write a procedure that can modify the application activity access permissions.

11.4.1 Creating an Application Activity Permissions File

You create a permissions file for application activities with the Data Dictionary. This chapter calls the file the *permissions file* but you can use any name you want. Each record in the permissions file should contain at least two fields: an `activity` field and a `can-run` field. The `activity` field contains the name of the activity and the `can-run` field contains the permissions for who can run the activity. You should also define the activity field as the primary index.

Here is the Dictionary definition for a sample permissions file:

```
  Field-Name: Activity                               Data-Type: character
      Format: x(10)                                     Extent:
       Label: Activity                                Decimals: ?
Column-Label: ?                                          Order: 10
     Initial:                                        Mandatory: no   (Not Null)
Component of-> View: no   Index: no             Case-sensitive: no
Valexp:
      :
      :
      :
Valmsg:
  Help:
  Desc:
Database: demo (PROGRESS)                File: permission
```

```
  Field-Name: Can-Run                                Data-Type: character
      Format: x(60)                                     Extent:
       Label: Can-Run                                 Decimals: ?
Column-Label: ?                                          Order: 20
     Initial:                                        Mandatory: no   (Not Null)
Component of-> View: no   Index: no             Case-sensitive: no
Valexp:
      :
      :
      :
Valmsg:
  Help:
  Desc:
Database: demo (PROGRESS)                File: permission
```

```
Index: activity                    Primary: Yes  Unique: Yes  Active: Yes

  activity                  Seq  Field Name          Type  Asc  Abbr
                            ---  ---------------     ----  ---  ----
                              1  Activity            char  Asc  No
```

After you create a permissions file, you must add a record for every application activity for which you want to provide security.

11.4.2 Adding Records to the Permissions File

Look again at the procedures run from the Maintenance and Reporting menu procedure p-csmnu3.p shown earlier in this chapter. Now, suppose you want to define security for the following activities, each of which are handled by a single procedure:

- 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.

In addition, let's assume you want to group the order report and mailing label procedures (p-itlist.p and p-rept6.p) into a functional area called print. You want manager and inventory to have permission to print.

To add these application activities as records to the permissions file (if you create a permissions file), you can write a simple PROGRESS procedure:

p-prmsn.p

```
REPEAT:
 INSERT permission.
END.
```

This procedure lets you add records (application activities) to the permissions file until you press END-ERROR (F4). For example:

```
Activity        Can-Run
-----------     -------------------
p-adcust.p      manager,salesrep
p-chcust.p      manager,salesrep
p-delcus.p      manager
print           manager,inventory
```

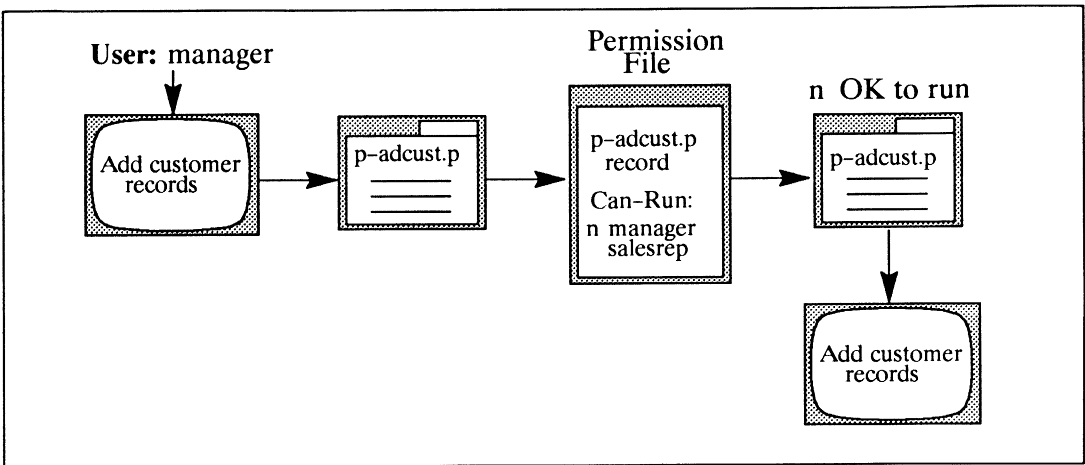
After you create records for the activities in your application, you must include statements in your procedures that check these records at run time.

11.4.3 Including Security Checking in Procedures

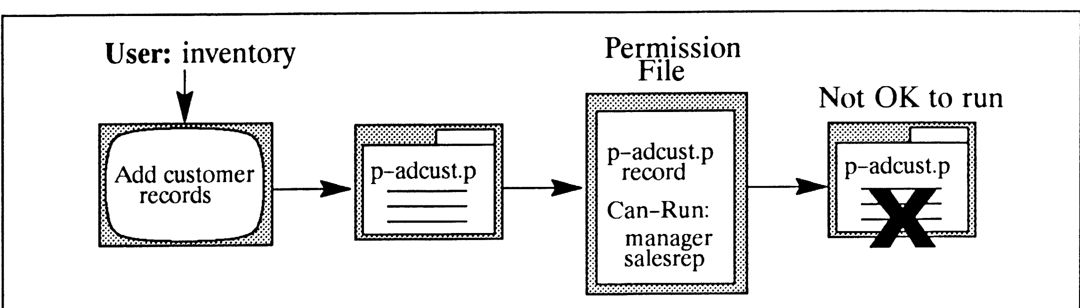
When a user runs a procedure, you want to check the permission for the activity associated with the procedure. Specifically, you want to:

- Find the record in the activity permissions file.
- Compare the permissions for the activity with the userid of the user running the procedure.
- Display a message and exit from the procedure if the permissions do not match the userid.

This is what happens when the manager runs the `p-adcust.p` procedure:



When the user named manager runs `p-adcust.p`, the userid and the permission defined in the `p-adcust.p` record in the permissions file match. Therefore, the manager can run the procedure. However, when the user with the userid inventory runs `p-adcust.p`, there is no match. That user receives a message and cannot run the procedure:



As you saw earlier, you use the CAN-DO function to do security checking in your procedure. Here is a modified version of the p-adcust.p procedure that includes security checking:

p-adcus4.p

```
DO FOR permission:
  FIND permission "p-adcust.p" NO-LOCK.
  IF NOT CAN-DO(can-run)
  THEN DO:
     MESSAGE "You are not authorized to run this procedure.".
     RETURN.
  END.
END.

REPEAT:
  INSERT customer WITH 2 COLUMNS.
END.
```

security checking

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 userid 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.

The security checking part of the procedure is done within a DO FOR block in order 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 statement ensures that other users can access or update the permissions file while this procedure is running.

The part of the p-adcust.p procedure that does security checking is fairly 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 record being read in the permissions file.

p-delcs2.p

```
DO FOR permission:
  FIND permission "p-delcus.p" NO-LOCK.
  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.
FIND customer USING cust-num.
DELETE customer.
```

security checking

Suppose you want to check security in one of the print procedures, such as `p-itlist.p`:

p-itlst2.p

```
DO FOR permission:
  FIND permission "print" NO-LOCK.
  IF NOT CAN-DO(can-run)
  THEN DO:
     MESSAGE "You are not authorized to run this procedure.".
     RETURN.
  END.
END.

FOR EACH customer WHERE max-credit > 500 BY zip:
  DISPLAY name address city st zip max-credit WITH SIDE-LABELS.
  FOR EACH order OF customer:
    DISPLAY order WITH SIDE-LABELS.
    FOR EACH order-line OF order, item OF order-line:
      DISPLAY line-num item.item-num idesc qty price
              WITH SIDE-LABELS.
    END.
  END.
END.
```

security checking

Here, the FIND statement reads the "print" record from the permissions file. The CAN-DO function compares the value of the `can-run` field with the userid 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 that there is no separate record in the `permissions` file for the `p-itlist.p` procedure. However, the record for the print activity can be used to handle security for any procedure that you specify as a print activity. You can include exactly the same security checking statements in any other procedure that you consider to be a print activity, such as `p-rept6.p`.

For application maintenance purposes, you may 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.

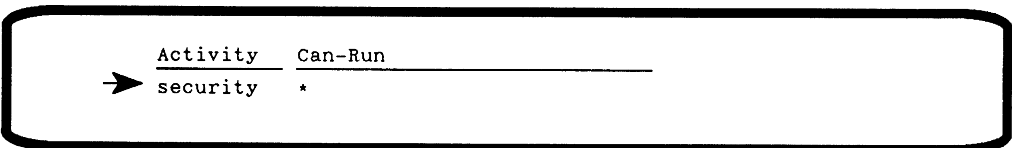
11.4.4 Protecting the Permissions File

In order to protect the security provisions you have established with the `activities` file, you must supply the following:

- A procedure that defines who can modify the `permissions` file. Once the application is ready for use, you run this procedure to assign a security administrator who is authorized to make modifications to security definitions.
- A procedure that the security administrator runs to modify security definitions for procedures and functions.

These two procedures use the same methods for defining and checking security that the other procedures in this section have used. That is, in order to do security checking, there must be a record in the `permissions` file associated with the security activity. Because both of these procedures deal with security, you can create a single record for the security activity in the `permissions` file, using the `p-prmsn.p` defined earlier in this chapter.

Here is the definition of the record for the security activity in the `permissions` file:



Activity	Can-Run
security	*

When you create the record for the security activity, initialize the `can-run` field with an asterisk. That way, any user will be able to run the security administration procedure when they first run your application.

After you define a record for the security activity, you can write a procedure that lets the user of the application assign one or more security administrators. The userids of these administrators are stored in the can-run field of the security record in the permissions file, allowing only those users to run the security procedures. Here is an example of this procedure:

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

When a user runs this procedure for the first time, there is an asterisk value in the security record's can-run field. Therefore, anyone can run this procedure. However, after the can-run field is modified, only the users with IDs matching those in the can-run field can change the security record or other records stored in the permissions file.

After a user defines one or more security administrators using the p-secadm.p procedure, those security administrators must have a way to modify definitions for activities in the application. The procedure p-secupd.p updates security for procedures and activities:

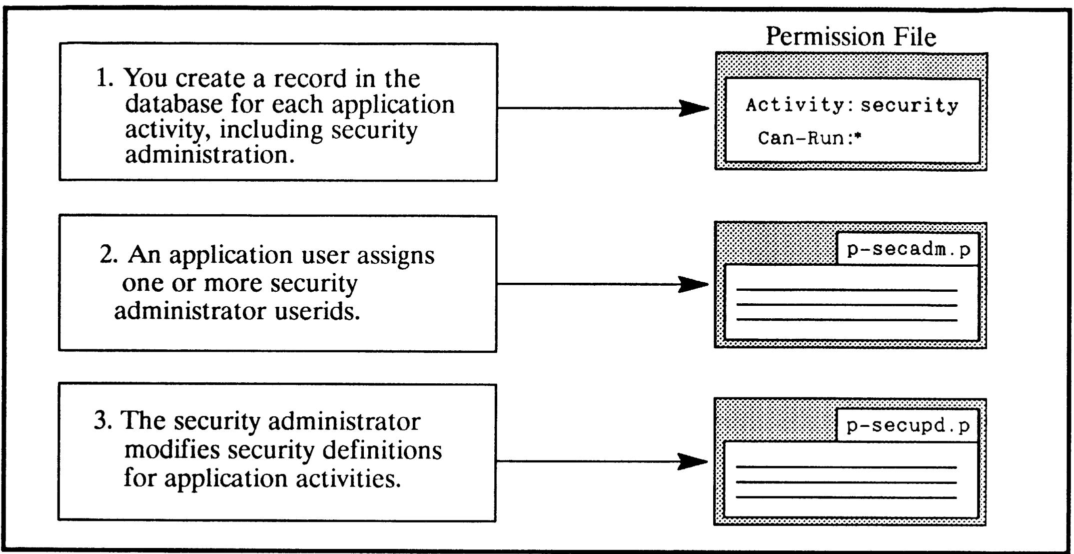
p-secupd.p

```
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 file to be sure that the user is a security administrator authorized to run the procedure. If the user is not authorized, the procedure displays a message and exits. Otherwise, the user can modify the can-run field for a specified activity.

This figure summarizes the process you use to handle security for procedures and activities in an application.



11.5 USING FILE- AND FIELD-LEVEL SECURITY AT COMPILE TIME

The security measures described in this chapter so far protect your database from unauthorized users running precompiled procedures. However, the security measures do not prevent users from writing their own procedures to access the database. In order to prevent users from accessing the database with their own procedures, you must use security checking at compile time. (To prevent users from changing field and index definitions in the database, see the subsection "Freezing and Unfreezing Files" later in this chapter.)

Compile time security checking is built into PROGRESS. PROGRESS lets you define, on a file and field basis, the level of access available to each user of an application. The permissions you define for the files and fields in your database describe the kinds of access different users can have to those files and fields. PROGRESS checks those permissions when a procedure is compiled. Therefore, if you do not use the COMPILE or COMPILE SAVE statements to precompile your procedures, permissions are checked when a procedure is run (and compiled) for the first time during each PROGRESS session. Permissions are not checked when you run the object version of a procedure.

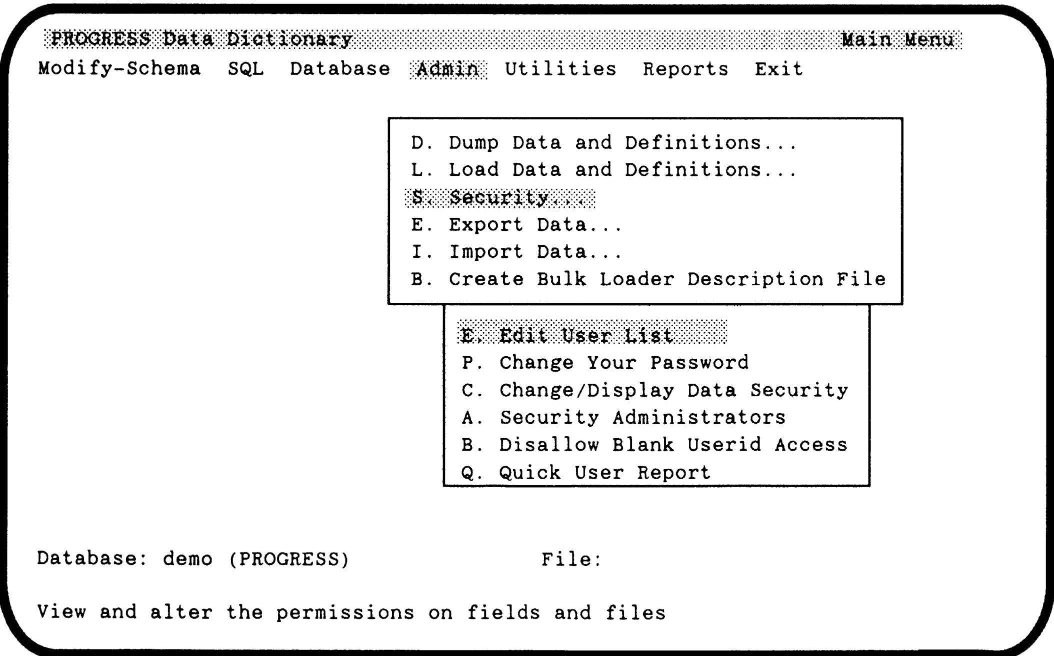
You use the Data Dictionary to define compile time security for an application database. Select the `Admin` option from the Data Dictionary horizontal menu.

```
PROGRESS Data Dictionary                                             Main Menu
Modify-Schema  SQL  Database  Admin  Utilities  Reports  Exit

    ----------- Welcome to the PROGRESS Data Dictionary -------------
    | You can use these facilities to set up or alter the structure of |
    | your database(s).  To navigate the menus, use the arrow or tab    |
    | keys to move the highlight.  Press [RETURN] or [F1] to select     |
    | an item.  You may also choose an item by typing its letter.       |
    |                                                                   |
    ---------------------------------------------------------------------


Database: demo (PROGRESS)               File:

System Administration
```

PROGRESS displays the following screen. Type **s** to select the `Security` option.

```
PROGRESS Data Dictionary                                            Main Menu
Modify-Schema  SQL  Database  Admin  Utilities  Reports  Exit

                              D. Dump Data and Definitions...
                              L. Load Data and Definitions...
                              S. Security...
                              E. Export Data...
                              I. Import Data...
                              B. Create Bulk Loader Description File

Database: demo (PROGRESS)               File:

Security menus
```

PROGRESS displays the following screen.



To see the definitions for compile time security, type **c** to select `Change/Display Data Security`.

The Data Dictionary displays a list of files for you to choose from. Select the file that has the permissions you want to change. Then you'll see the current permission lists for that file. For example, if you choose `customer`:

```
PROGRESS Data Dictionary                     Change/Display Data Security
Modify-Schema  SQL  Database  ADMIN  Utilities  Reports  Exit

                 File name:
                  customer

                     agedar
                     customer
                     monthly
                     order
                     order-line
                     salesrep
                     shipping
                     state
                     syscontrol

Database: demo (PROGRESS)              File: customer

Press the [F4] key to end.
```

PROGRESS then displays the access rights, also known as permissions, for the customer file:

```
PROGRESS Data Dictionary                          Change/Display Data Security
   Modify-Schema  SQL  Database  ADMIN  Utilities  Reports  Exit
┌──────────────────────── File-name: "customer" ─────────────────────────┐
│   Can-Read: *                                                          │
│  Can-Write: *                                                          │
│ Can-Create: *                                                          │
│ Can-Delete: *                                                          │
└────────────────────────────────────────────────────────────────────────┘
┌──────────────── Access restrictions for files ─────────────────────────┐
│  *                       -  All users are allowed access               │
│  <user>,<user>,etc       - Only these users have access                │
│  !<user>,!<user>,*       - All except these users have access.         │
│  acct*                   - Only users that begin with "acct" allowed   │
│                                                                        │
│   Users are named by their login ids, which may contain wildcards.     │
│   Exclamation mark means NOT. Commas must separate users in a list.    │
│   Do not use spaces in the string, as they will be taken literally.    │
└────────────────────────────────────────────────────────────────────────┘
NextField  PrevField  ForwardFile  BackwardFile  Modify  SwitchFile
  JumpField  CallAdmin  UserEditor  Report  Exit
Database: demo (PROGRESS)                   File: customer

Next Field.
```

The highlight bar is on NextField. Select `Modify` if you want to change the permissions for the `customer` file.

Any changes you make to file permissions take effect only after you leave and restart PROGRESS. That means, if other users are working while you change file permissions, they will not be affected.

NOTE: Do not try to bypass the Data Dictionary to modify the permissions for the files and fields in the database. You may lock yourself out of the database.

11.5.1 File Permissions

There are four levels of file security permission:

- **CAN-READ**

  Users who are permitted to read a file are listed here. An asterisk indicates that all users are authorized to read a file. If, instead of the asterisk, the permission was "sales", only users with a userid of "sales" could read the `customer` file.

- **CAN-WRITE**

  Users who are permitted to write to a file are listed here. Any user that needs to update records must be authorized to write to a file. An asterisk indicates that all users are authorized to update the `customer` file.

- **CAN-CREATE**

  Users who are permitted to create new records are listed here. A user with `can-create` privileges does not need `can-write` privileges.

- **CAN-DELETE**

  Users who are permitted to delete records from a file are listed here.

The following table shows the values you use to define the permissions for any file:

Table 11-1: Access Restrictions for Files

EXPRESSION	MEANING
*	All users are allowed access.
<user>	This *user* has access.
! *<user>*	This *user* does not have access.
<string> *	Only userids that begin with *string* have access.

11.5.2 Field Permissions

In addition to defining permissions at the file level, you can define permissions at the field level. Being able to define permissions for fields means that you do not have to deny access to an entire file because there are one or two fields that contain sensitive information. You can simply protect those fields.

You must be a security administrator, and be included in the permission list being modified, in order to change field permissions.

After you define the permissions for a file, follow the same procedure for each field. For example, select `NextField` to see the permissions for the first field in the `customer` file:

```
PROGRESS Data Dictionary                     Change/Display Data Security
Modify-Schema  SQL  Database  ADMIN  Utilities  Reports  Exit
---------------------------- File-name: "customer" ------------------------
Field-Name: Address

  Can-Read: *          } permissions
 Can-Write: *

------------------- Access restrictions for files -------------------------
 *                        -  All users are allowed access
 <user>,<user>,etc        - Only these users have access
 !<user>,!<user>,*        - All except these users have access.
 acct*                    - Only users that begin with "acct" allowed

   Users are named by their login ids, which may contain wildcards.
   Exclamation mark means NOT. Commas must separate users in a list.
   Do not use spaces in the string, as they will be taken literally.

 NextField  PrevField  ForwardFile  BackwardFile  Modify  SwitchFile
   JumpField  CallAdmin  UserEditor  Report  Exit
 Database: demo (PROGRESS)                File: customer

 Next Field.
```

There are only two permissions you can define for fields: `can-read` and `can-write`. In the `Address` field, all users have can-read and can-write privileges.

Continue pressing SPACEBAR or ↓ until you see the access lists for each field. Press ↑ or type **p**, for `PrevField`, to see the permissions for previous fields.

Use the spacebar or arrow keys to move through the following choices.

NextField Selects the next field.

PrevField Selects the previous field.

ForwardFile Selects the next file in the list.

BackwardFile Selects the previous file in the list.

Modify Allows you to modify the permissions for a file.

SwitchFile Displays the list of files again so that you can choose a new file.

JumpField Displays a list of all the fields in the current file, so you can rapidly select the field you want to see.

CallAdmin Displays the screen that allows you to set security administrators.

UserEditor Displays the Edit User List screen.

Report Calls User Review from the "Reports" menu.

Exit Exits to the Data Dictionary Main Menu.

Press END-ERROR (F4) to return to the `Data Dictionary Main Menu`.

11.5.3 Example of Personnel Security

Let's look at how you can define permissions for the mywork database. Suppose the userids defined in the `_User` file for the database are `salesrep`, `inventory` and `manager`. Here is more information about about these userids:

- All the salesreps use the `salesrep` account.
- Warehouse personnel use the `inventory` account.
- Managers use the `manager` account.

Suppose you want all users to have can-read permission for the customer file, but only users with the specified userids to have `can-write`, `can-create`, and `can-delete` privileges. You would specify this information as shown in the following screen.

```
                    .
                    .
----------------- File name: "customer" -----------------
   Can-Read: *
  Can-Write: salesrep,manager
 Can-Create: salesrep,manager
 Can-Delete: manager
                    .
                    .
                    .
```

All users have `can-read` permission for the name field. However, users with a userid of `inventory` cannot write to this field when permissions are set as they are in the following example:

```
                    .
                    .
----------------- File name: "customer" -----------------
Field name: Name
  Can-Read: *
 Can-Write: *,!inventory
                    .
                    .
                    .
```

All users except those with a userid of `inventory` have `can-read` permission for the `max-credit` field. Only managers have permission to write to this field when permissions are set as they are in this next example:

```
                                 .
                                 .
                                 .
+------------------- File name: "customer" ---------------------+
|Field name: Max-credit                                         |
|                                                               |
|  Can-Read: *,!inventory                                       |
| Can-Write: manager                                            |
+---------------------------------------------------------------+
                                 .
                                 .
                                 .
```

After you define permissions, you want to be sure that only certain users have the right to change those permissions. You can protect database security by naming a security administrator to maintain the file and field permissions.

11.6 PERFORMING SECURITY ADMINISTRATION

A security administrator is a user who is allowed to:

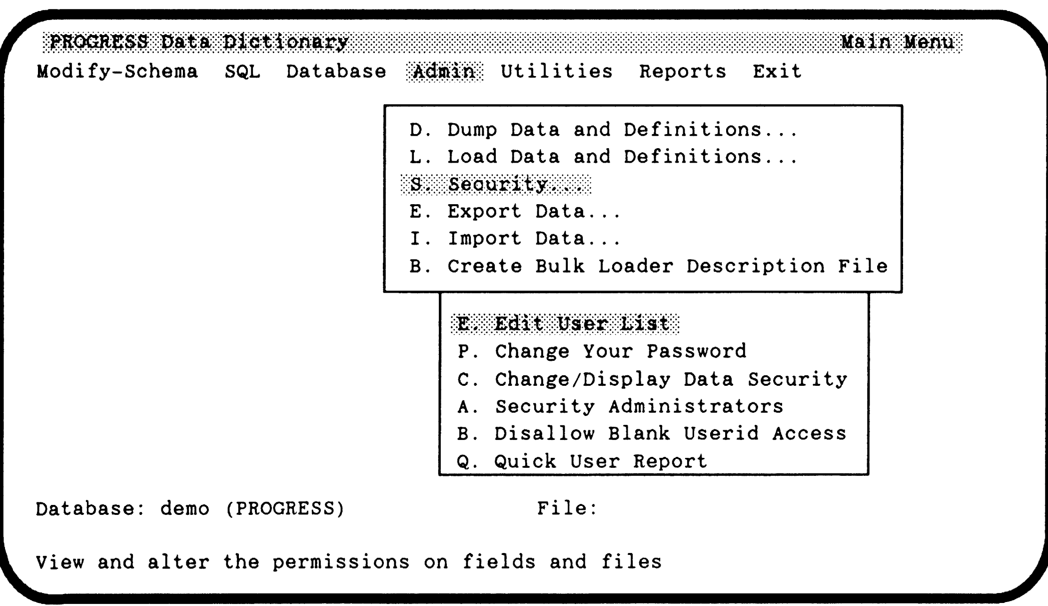
- Set up and maintain the `_User` file.
- Define the permissions for files and fields in the database.
- Control freezing and unfreezing of database files.
- Designate other security administrators.
- Determine whether the blank userid can access a database.

You perform these administrative tasks by selecting the `Security` option from the `Admin` submenu of the Data Dictionary Main Menu.

```
PROGRESS Data Dictionary                                           Main Menu
Modify-Schema  SQL  Database  Admin  Utilities  Reports  Exit

                                D. Dump Data and Definitions...
                                L. Load Data and Definitions...
                                S. Security...
                                E. Export Data...
                                I. Import Data...
                                B. Create Bulk Loader Description File

                                   E. Edit User List
                                   P. Change Your Password
                                   C. Change/Display Data Security
                                   A. Security Administrators
                                   B. Disallow Blank Userid Access
                                   Q. Quick User Report

Database: demo (PROGRESS)               File:

View and alter the permissions on fields and files
```

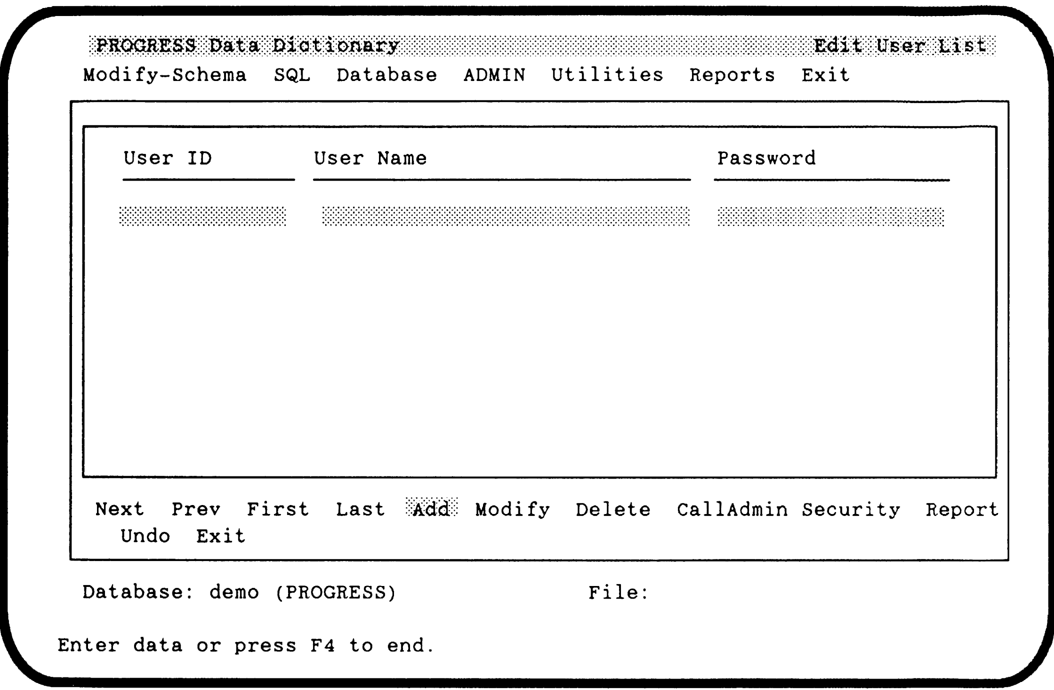
11.6.1 Setting Up the _User File

You define userids and passwords with the `Edit User List` option in the `Security` submenu of the Data Dictionary Main Menu.



The Edit User List option provides a horizontal menu of choices for maintaining the _User file.

When you choose the Edit User List option, the screen displays a list of userids and user names. You may choose to perform any of the options listed in the horizontal menu at the bottom of the screen.



When you choose to Add a user, the Data Dictionary displays the areas where you need to enter a userid, user name, and a password. You must enter the password twice before the password takes effect. Be sure to enter the exact same password both times.

```
PROGRESS Data Dictionary                                         Main Menu
Modify-Schema  SQL  Database  ADMIN  Utilities  Reports  Exit

   User ID          User Name                     Password
   ___________      ___________________________   _________________

   someuserid      For verification purposes, please type
                   the same password in again.  Remember
                   that passwords are case-sensitive.

                   Password:

  Next  Prev  First  Last  Add  Modify  Delete  CallAdmin Security  Report
    Undo  Exit

Database: demo (PROGRESS)              File:

Enter data or press F4 to end.
```

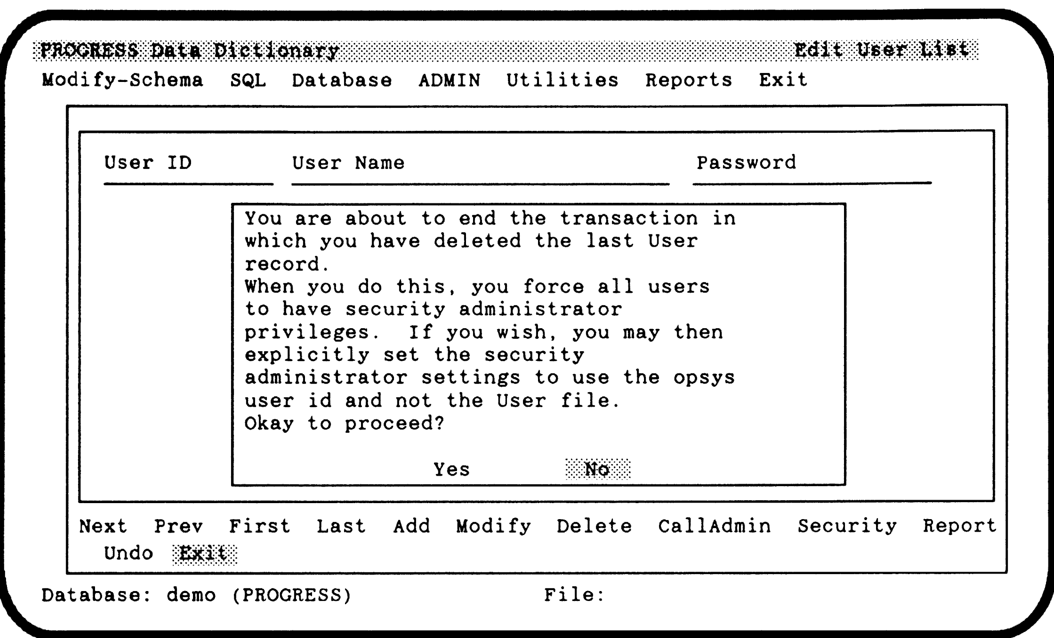
PROGRESS does not allow any change to the userid field unless the record is new. PROGRESS does not allow any change to the password field unless it is either new or the userid field matches the current userid. This means that the security administrator can assign passwords only when creating a new user. The only other way that a security administrator can change a userid or password is by deleting and then recreating the user record. In this way, users cannot be "locked out" of a database if they forget their passwords.

However, if you are the only security administrator and you forget your password, there is no way to add or delete users or change file and field permissions for data files. Depending on the permissions that exist at that time, you may be unable to access the database or recover the data contained in it. However, as long as one security administrator is able to log into the database, that security administrator can create more security administrators.

When you choose to `Delete` a user from the list of users, the Data Dictionary displays a message to verify that you want to remove that user. Use your left arrow key or type **Y** to confirm. Then press RETURN.

```
PROGRESS Data Dictionary                                              Main Menu
Modify-Schema  SQL  Database  ADMIN  Utilities  Reports  Exit

  User ID        User Name                         Password
  ___________    ______________________________    __________________

  someuserid     someusername
   auserid        ausername
                 Are you sure that you want to remove
                 this user?
                               Yes    No

Next  Prev  First  Last  Add  Modify  Delete  CallAdmin  Security  Report
  Undo  Exit

Database: demo (PROGRESS)               File:
```

Suppose there were users in this list when you came into this screen, and you delete all of them. Once you select `Exit`, PROGRESS displays the following message to let you know that the Data Dictionary will remove all security restrictions when the last record is deleted. Then PROGRESS prompts you to confirm that you want to proceed.



```
PROGRESS Data Dictionary                                    Edit User List
Modify-Schema  SQL  Database  ADMIN  Utilities  Reports  Exit

    User ID        User Name                      Password

              You are about to end the transaction in
              which you have deleted the last User
              record.
              When you do this, you force all users
              to have security administrator
              privileges.  If you wish, you may then
              explicitly set the security
              administrator settings to use the opsys
              user id and not the User file.
              Okay to proceed?

                              Yes        No

  Next  Prev  First  Last  Add  Modify  Delete  CallAdmin  Security  Report
    Undo  Exit
Database: demo (PROGRESS)              File:
```

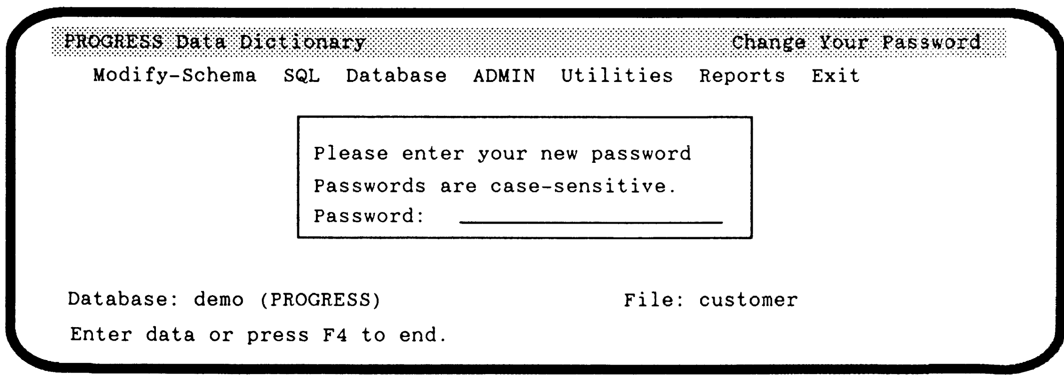
When you choose `Report`, the Data Dictionary lists the userids and user names. You do not need the privileges of a security administrator to use this option. If you want to print a copy of this list, you can do so by selecting the `Reports` option from the Data Dictionary's Main Menu. From there, select the `User Review` option.

11.6.2 Changing Your Password

PROGRESS allows users, including the System Administrator, to change their own passwords. To change your password, select `Change Your Password` from the Security submenu of the Data Dictionary Main Menu.

```
PROGRESS Data Dictionary                                          Main Menu
Modify-Schema  SQL  Database  Admin  Utilities  Reports  Exit

                              D. Dump Data and Definitions...
                              L. Load Data and Definitions...
                              S. Security...
                              E. Export Data...
                              I. Import Data...
                              B. Create Bulk Loader Description File
                                 E. Edit User List
                                 P. Change Your Password
                                 C. Change/Display Data Security
                                 A. Security Administrators
                                 B. Disallow Blank Userid Access
                                 Q. Quick User Report

Database: demo (PROGRESS)                  File:

View and alter the permissions on fields and files
```

Then PROGRESS prompts you for a new password:



You must enter the new password twice, exactly the same way both times, before the new password takes effect. Remember that passwords are case-sensitive.

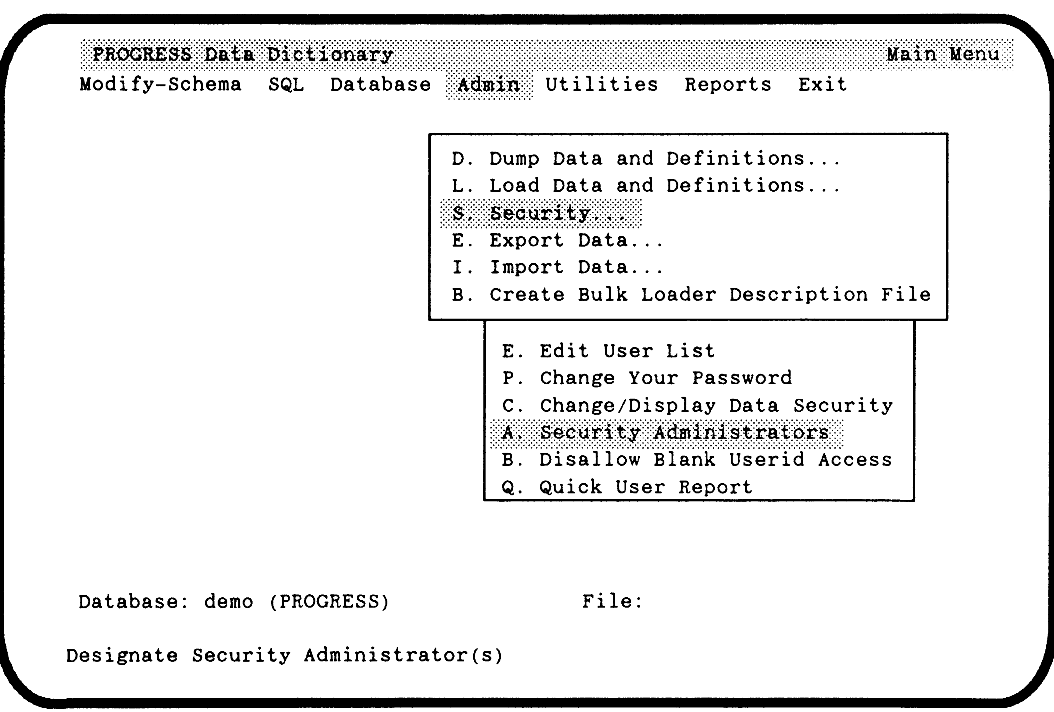
NOTE: Do not try to bypass the Data Dictionary to modify passwords. You may lock yourself out of the database.

11.6.3 Designating a Security Administrator

Before designating security administrators, you should be sure that the userid of each security administrator is included in the permissions for the files and fields for the database. If you assign userid jones as a security administrator but jones is not included in the permissions for the customer file, jones will not be able to modify permissions for the customer file. To be able to modify file and field permissions, a userid must be designated as a security administrator *and* be included in each of the individual file and field permissions.

To define one or more security administrators for your database, choose the `Security Administrator` option from the `Security` submenu of the Data Dictionary Main Menu.

When you choose option A for Security Administrators, you can enter the userids of the users that you want to have act as security administrators. The Data Dictionary then changes the permissions for the `_User` file to include only the userids of the security administrators.



```
PROGRESS Data Dictionary                              Security Administrators
Modify-Schema  SQL  Database  ADMIN  Utilities  Reports  Exit

Enter login ids to control who can change security.
*

----------------- Access restrictions for files -----------------
*                        - All users are allowed access
<user>,<user>,etc        - Only these users have access
!<user>,!<user>,*        - All except these users have access.
acct*                    - Only users that begin with "acct" allowed

  Users are named by their login ids, which may contain wildcards.
  Exclamation mark means NOT. Commas must separate users in a list.
  Do not use spaces in the string, as they will be taken literally.

Database: demo (PROGRESS)                 File:

Enter data or press F4 to end.
```

11.6.4 Determining the Privileges of the Blank Userid

A user has the blank userid under the following circumstances:

- On DOS and OS/2 systems when there are no records in the `_User` file and when neither the -U and -P startup options, nor a login procedure such as `login.p`, are used to establish the userid.
- On all systems when records are found in the `_User` file and neither a login procedure such as `login.p` nor the -U and -P startup options are used.

A user can use PROGRESS or a PROGRESS application with a blank userid and access files and fields in the database as long as the file- and field-level permissions permit it or as long as the procedure being run is precompiled. PROGRESS expects an application to run a login program to set the userid to a more meaningful value. If you, or your security administrator, have not reset the userid, you may want to deny privileges to the blank userid to ensure that unknown users do not use the database.

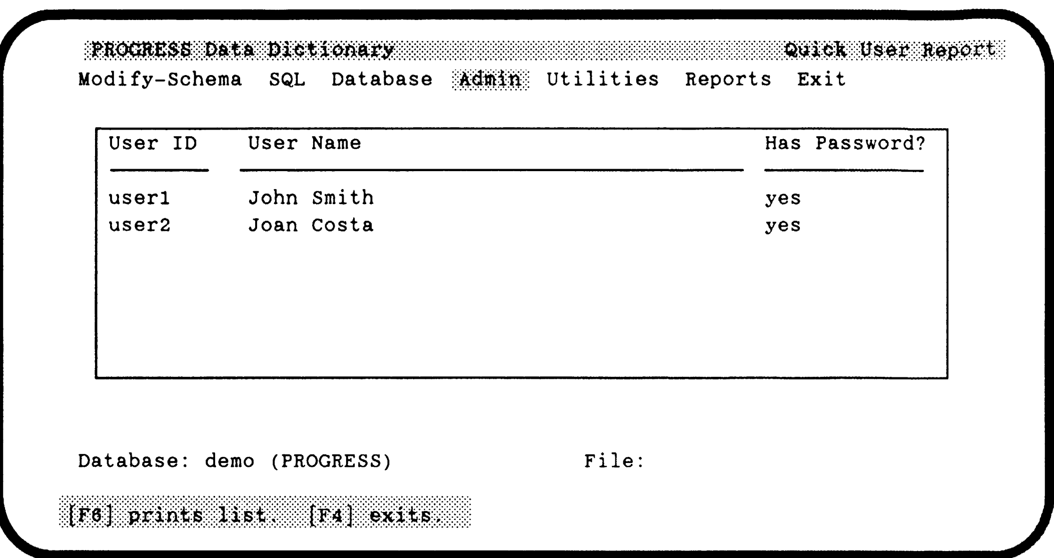
When you choose the Disallow Blank Userid Access option from the Security submenu of the Data Dictionary Main Menu, the can-read, can-write, can-create, and can-delete privileges are denied to anyone with the blank userid. Although this prevents anyone from bypassing the login procedures to gain access to the database and writing their own procedures to access the data there, your precompiled procedures should still check the current userid at run-time to make sure only authorized users are running them.

11.6.5 Generating a Quick User Report

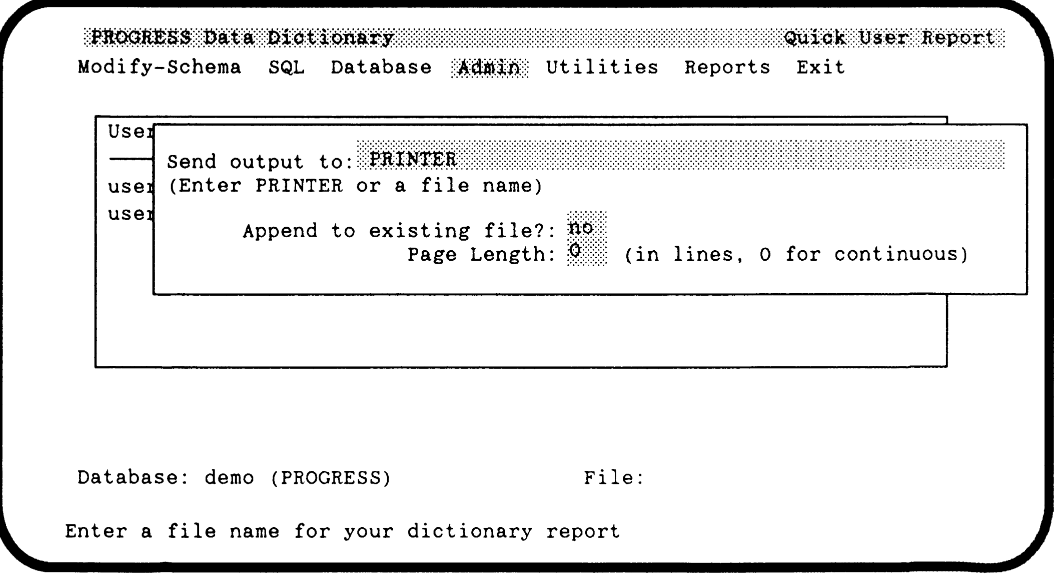
You can display or print a summary of users defined for a database by typing **q**, for Quick User Report, at the Security submenu shown in the following figure.

```
PROGRESS Data Dictionary                                          Main Menu
Modify-Schema  SQL  Database  Admin  Utilities  Reports  Exit

                           D. Dump Data and Definitions...
                           L. Load Data and Definitions...
                           S. Security...
                           E. Export Data...
                           I. Import Data...
                           B. Create Bulk Loader Description File
                              E. Edit User List
                              P. Change Your Password
                              C. Change/Display Data Security
                              A. Security Administrators
                              B. Disallow Blank Userid Access
                              Q. Quick User Report

Database: demo (PROGRESS)               File:

Display a summary of users defined in this database
```

If there are users defined in the database, the Data Dictionary displays a screen similar to the following example screen:



You can print your Quick User Report to a file, or to your printer if it is on-line, by pressing PUT (F6). The Data Dictionary displays the following screen:



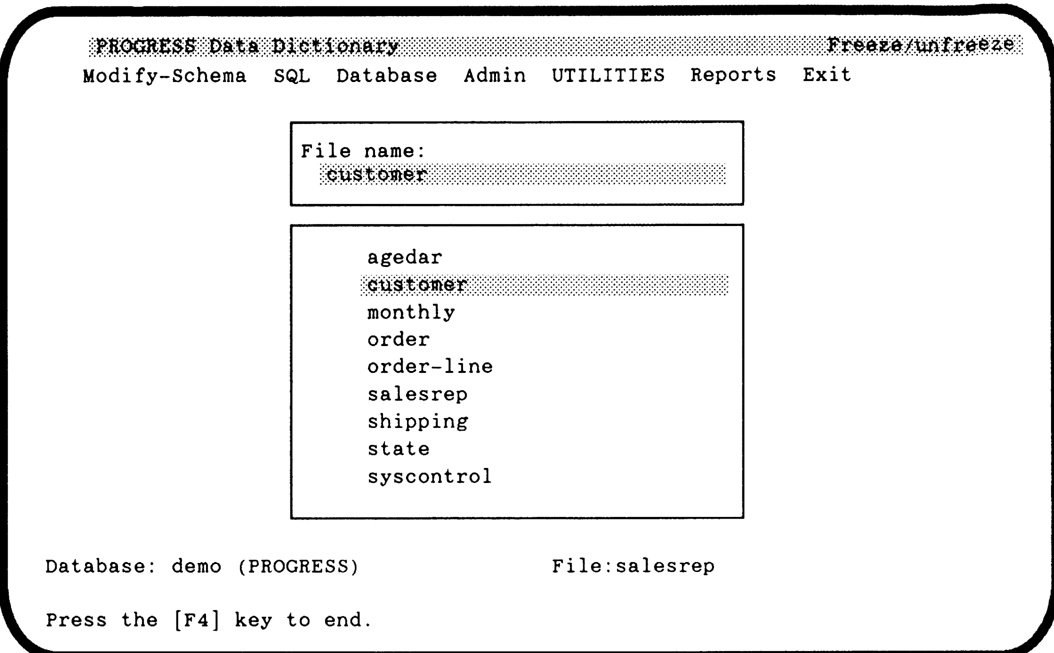
11.6.6 Freezing and Unfreezing Files

When a database file is *frozen*, its field and index definitions cannot be changed. Typically, you freeze the database files when you complete application development. Frozen files can be unfrozen when you need to make changes like adding or deleting fields or indexes, or modifying field or index definitions. (Recall that changing a file's data definitions gives the file a new time stamp, and any procedures that reference the file must be recompiled.)

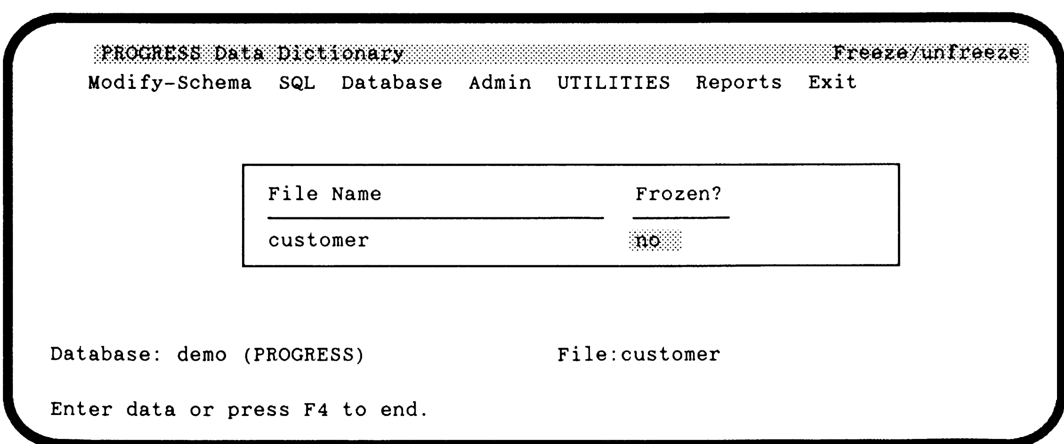
To freeze a file, type **f** to select the `Freeze/unfreeze` option from the `Utilities` submenu in the `Data Dictionary`.

```
PROGRESS Data Dictionary                                          Main Menu
Modify-Schema  SQL  Database  Admin  Utilities  Reports  Exit

                                  E. Editor for Parameter Files
                                  F. Freeze/unfreeze
                                  Q. Quoter Functions ...
                                  G. Generate Include Files ...
                                  I. Information

Database: demo (PROGRESS)                File: salesrep

Freeze or Unfreeze file definitions
```

Select the file to be frozen. The example screen has the customer file highlighted.



The following screen appears:



Set the `frozen` field to `yes` to freeze the file.

The security administrator can control which users freeze or unfreeze database files. To freeze or unfreeze files, you must have `Can-Write` access to the `_File` file and `Can-Write` access to the `_File._Frozen` field.

Chapter 12
Writing Multi–user Applications

Everything you have read so far applies to writing any kind of PROGRESS application. But if you are writing an application that will be used by many users simultaneously, there are some additional issues and features you should address during your application development cycle.

This chapter explains:

- How PROGRESS handles multi–user applications.
- What the specific multi–user issues are.
- How you can create the most efficient and effective application for your users.

The concepts explained in this chapter apply to applications running in a multi–user (UNIX, VMS, BTOS/CTOS, or LAN) environment. If you are developing an application on a single–user DOS or OS/2 system, you can use all the multi–user-related statements shown in this chapter. PROGRESS simply disregards those statements when you run the application in single–user mode.

12.1 THE MULTI–USER ENVIRONMENT

To start multi–user PROGRESS, you must:

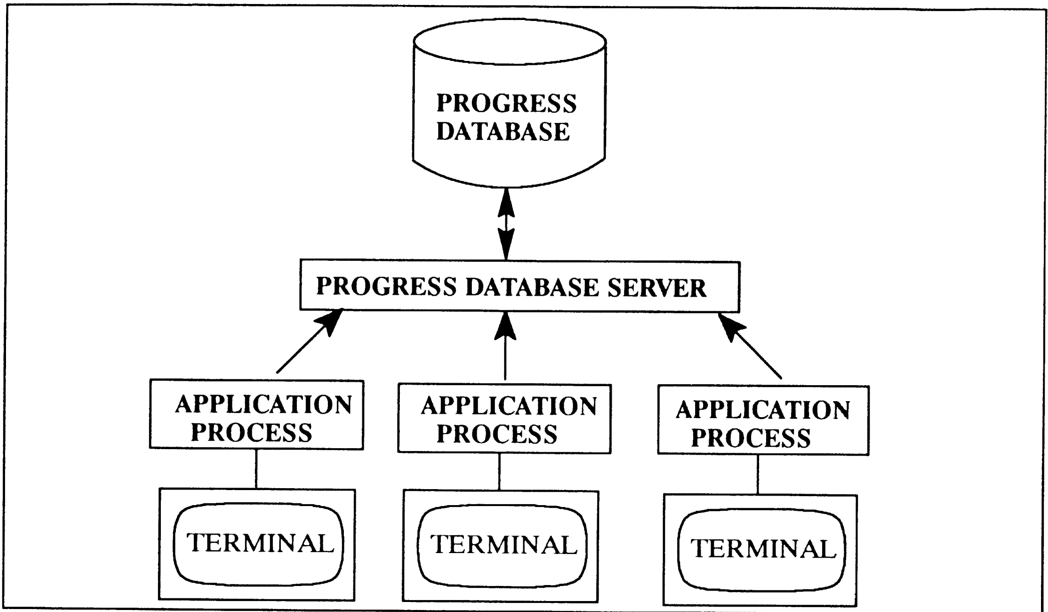
- Start the multi–user server process.
- Start multi–user PROGRESS.

12.1.1 Starting a Multi–user Server

In a single–user environment, you interact directly with the PROGRESS database manager. However, in a multi–user environment, PROGRESS uses a special mechanism, a *server*, for coordinating the database access required by each user.

NOTE: When you open a database with the Read-Only (-RO) startup option, multiple users can access the database simultaneously. However, read-only users cannot update records, do not acquire record locks, and do not require a database server. See the Read-Only (-RO) startup option in Chapter 3 of *System Administration II: General* for more information.

On UNIX systems, the main database server process is called the *broker.* The broker manages the resources shared by multiple users, and if there are **remote users**, starts *servers* to service them.



In UNIX, the multi-user broker or server runs in background mode. You can start the broker by typing this command in response to the UNIX system prompt:

SYNTAX (UNIX)

```
proserve database-name
```

Here, *database-name* is the name of the application database you want the broker to control. (If you are using DOS LAN, see *System Administration I: Environments* for more information.)

In VMS, the multi-user server runs as a detached process. You can start the server from a batch queue. Use this command to start the multi-user server in VMS:

SYNTAX (VMS)

```
PROGRESS/MULTI_USER=START_SERVER database-name
```

As with the pro command, there are a number of startup options you can use with the proserve command. For a complete description of these options, see Chapter 3 in *System Administration Guide II: General*.

You start a broker or server for a database just once in order for many users to be able to start PROGRESS using that database.

12.1.2 Starting Multi-User PROGRESS

After you use the proserve command to start PROGRESS, each user types one of these commands to start PROGRESS:

Operating System	Multi-user PROGRESS Command
UNIX, DOS, and OS/2	`mpro` *database-name*
VMS	`PROGRESS/MULTI_USER=LOGIN` *database-name*
BTOS/CTOS	`PROGRESS 4GL` ⋮ `[Options ]` *database-name*

You may want to define this VMS command in a command file, or as a system symbol called mpro.

In the examples above, *database-name* is the name of the application database you are using.

Each user must use the mpro command either directly or from within a UNIX script, VMS program, or DOS or OS/2 batch file to start their own PROGRESS session.

There are a number of startup options you can use with the mpro command. For a complete description of these options, see Chapter 3 in *System Administration Guide II: General*.

12.1.3 Stopping a Multi-User Server

After all users have left a UNIX application, you can stop the broker with the following command:

Operating System	Stopping Multi-user PROGRESS
DOS & OS/2	`SHUTDOWN`
UNIX	`proshut` *database-name*
VMS	`PROGRESS/MULTI_USER=SHUTDOWN` *database-name*

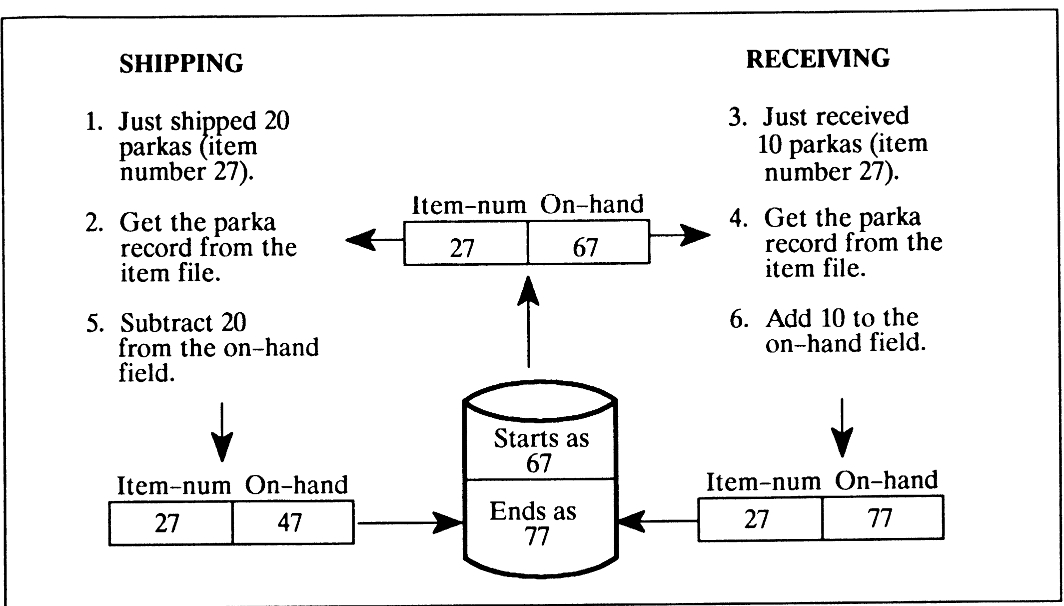
BTOS/CTOS	`PROGRESS Shutdown Server`
	`Database Name` *database-name*
	`Server Name`
	`[Options ]`

12.2 USING APPLICATIONS IN A MULTI-USER ENVIRONMENT

Suppose there are two departments in a company: shipping and receiving. When an order is shipped, the shipping department subtracts the number of items shipped from that item's on-hand value. When inventory is received 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 should now be 57 (67 on-hand - 20 shipped + 10 received = 57).

The following illustration shows how this simple situation, done with no special multi-user controls, can produce a very different result from the obvious one you expect.



Let's examine what happens here if you don't take any special steps to handle multiple users accessing the same database:

1. The shipping department ships an order for 20 parkas (item number 27).
2. The shipping department gets the parka record from the item file. 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 file. 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.

So, we ended up with 77 parkas in the database instead of the 57 we expected. It is as if the update done by the receiving department never even happened.

Naturally, this kind of situation arises all the time in a multi-user environment and, for that reason, PROGRESS supplies services to make sure that the data in your database is what you expect it to be.

12.3 SHARING DATABASE RECORDS

The situation described above has to do with multiple users needing access to the same record at the same time. In the illustration, things didn't 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.

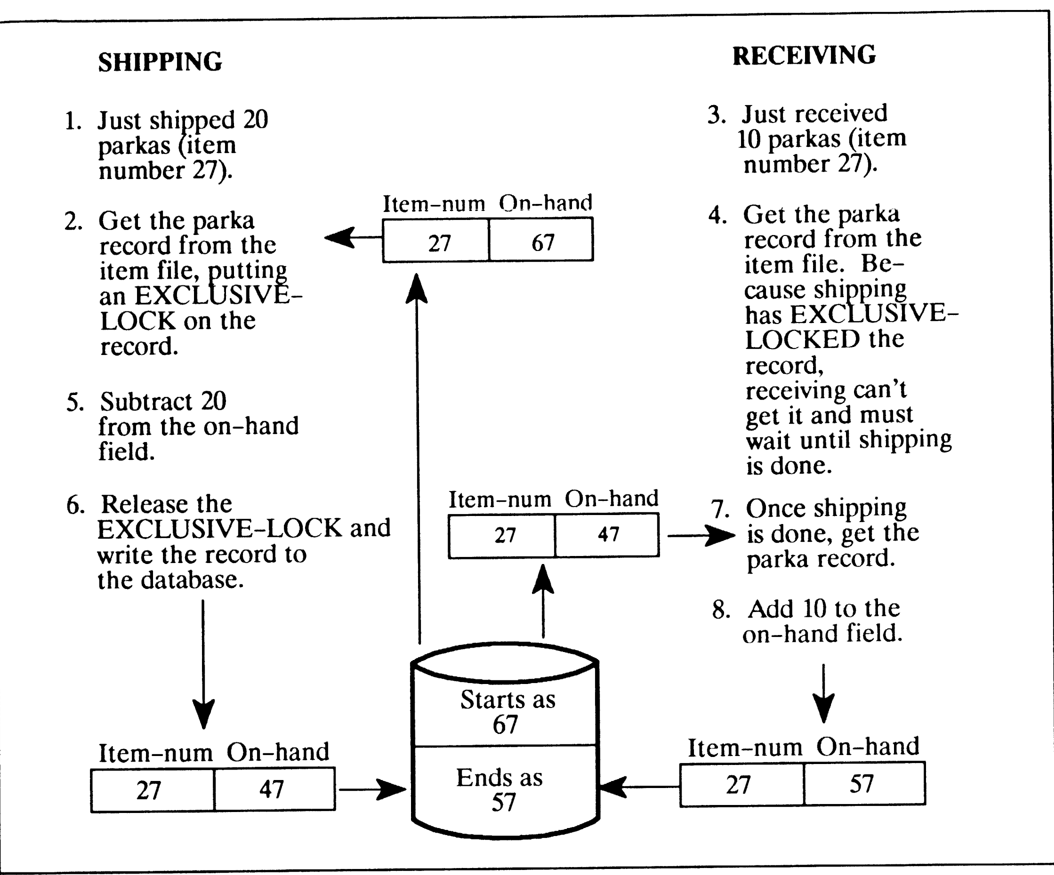
  BUT

- Just one user at a time can update, or make changes to, a record.

PROGRESS uses record locks to adhere to these rules.

12.3.1 Using Locks to Avoid Record Conflicts

Let's take another look at the shipping and receiving situation. This time we'll show how you can use locks to manage concurrent use of the same database record.



Here are the procedures that do this work. The numbers next to some of the procedure lines match up with the numbers in the list that follows the procedures.

p-lock1.p

```
                    /* SHIPPING */
   DEFINE VARIABLE qty-shipped AS INTEGER LABEL "Number Shipped".

   REPEAT:
     PROMPT-FOR item.item-num.
2.   FIND item USING item-num EXCLUSIVE-LOCK.
     DISPLAY idesc on-hand.
     SET qty-shipped.
5.   on-hand = on-hand - qty-shipped.
     DISPLAY on-hand.
6. END.
```

p-lock2.p

```
                    /* RECEIVING */
   DEFINE VARIABLE qty-recvd AS INTEGER.LABEL "Number Received".

   REPEAT:
        PROMPT-FOR item.item-num.
4. & 7. FIND item USING item-num EXCLUSIVE-LOCK.
        DISPLAY idesc on-hand.
        SET qty-recvd.
   8.   on-hand = on-hand + qty-recvd.
        DISPLAY on-hand.
      END.
```

1. The shipping department ships an order for 20 parkas (item number 27). See the diagram on the previous page.

2. The shipping department gets the parka record from the item file, placing an EXCLUSIVE-LOCK on the record. That means that no other user can even look at the record until shipping is finished with it. 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 tries to get the parka record from the item file but can't 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. Once the record is returned to the database, the EXCLUSIVE-LOCK is released and the receiving department can successfully read the record from 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 47 and returns the record to the database. Therefore, the final on-hand value for the parka record is 57 – exactly what you'd expect!

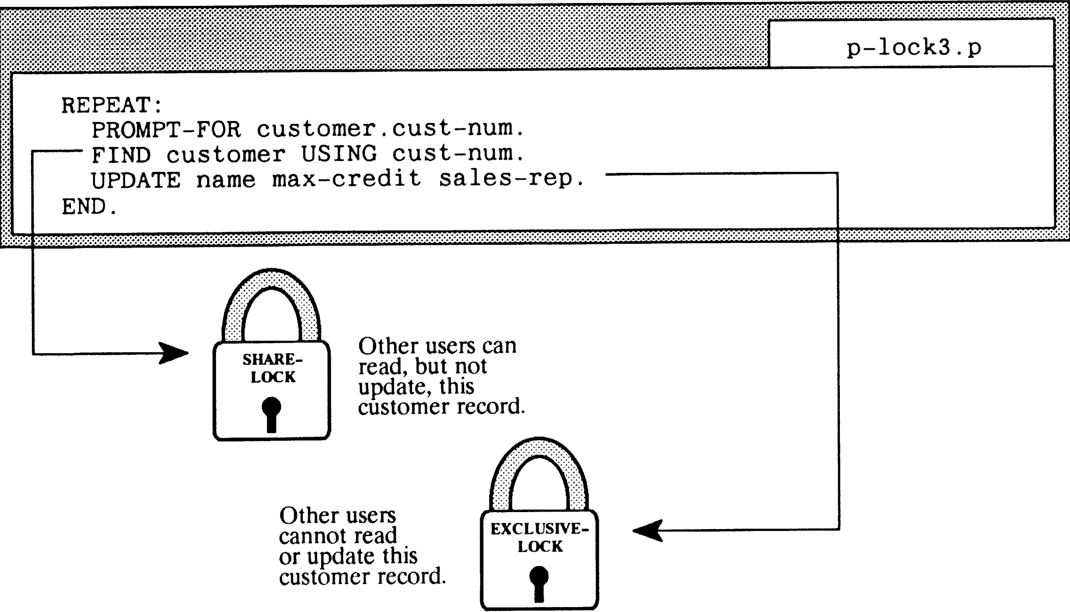
12.3.2 How PROGRESS Applies Locks

You've just seen how you can apply a lock on your own. But if you don't say anything at all about locks, PROGRESS does some default locking anyway. In particular:

- Whenever a record is read, PROGRESS puts a SHARE-LOCK on that record. This means that other users can read the record but cannot update it until the SHARE-LOCK is released. If you try to read a record SHARE-LOCK when another user has that record EXCLUSIVE-LOCKed, you receive a message telling you that the record is in use and to wait.
- Whenever a record is updated, PROGRESS puts an EXCLUSIVE-LOCK on that record. This means that other users cannot read or update that record until the EXCLUSIVE-LOCK is released. If you try to read a record EXCLUSIVE-LOCK when another user has that record SHARE-LOCKed, or EXCLUSIVE-LOCKed, you receive a message telling you that the record is in use and to wait.

NOTE: SHARE-LOCKs and EXCLUSIVE-LOCKs occupy entries in the lock table. The number of entries in the lock table defaults to 500 and is set with the -L start-up option. A user program is aborted if it attempts a record access that overflows the lock table.

Look at this example:



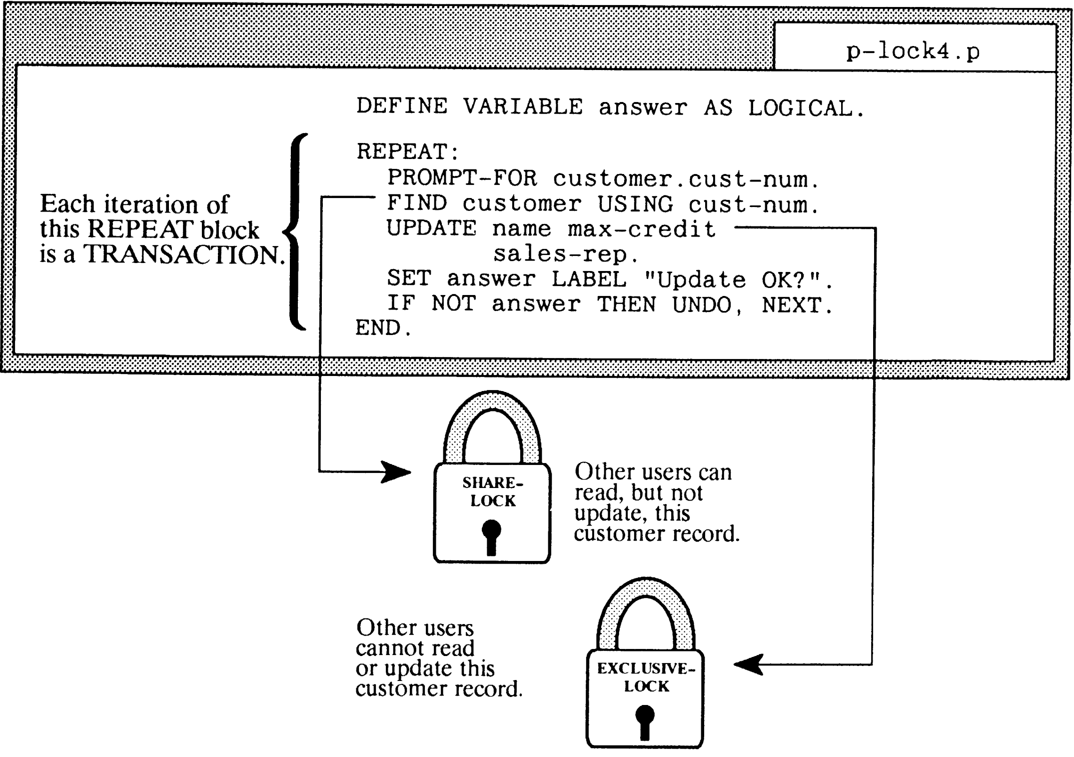
Here, when the FIND statement reads the customer record, PROGRESS puts a SHARE-LOCK on that record. Because the UPDATE statement lets you change the record, PROGRESS upgrades the SHARE-LOCK to an EXCLUSIVE-LOCK at the completion of the UPDATE statement.

12.4 HOW LONG DOES PROGRESS HOLD A LOCK?

Record locks and transactions are closely related. You learned in Chapter 8 that transactions are always related to blocks. As a 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 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.

To understand the relationship between record locks and transactions, take a look at a modified version of the last procedure.



The following series of imaginary 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, the EXCLUSIVE-LOCK is released.
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" question and the changes you made are undone.
8. The other user answers yes to the "Updates OK" question and the record is written back to the database, including the changes you made 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 12-1 lists the standards PROGRESS uses to determine when to release record locks.

Table 12-1: When PROGRESS Releases Record Locks

Type of Lock	Acquired During a Transaction (1)	Acquired Outside a Transaction (2)	Acquired Outside But Held Into a Transaction (3)
SHARE	Held until the later of transaction end and record release. (4)	Held until record release.	Held until the later of transaction end and record release. (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

The following numbered paragraphs explain the references in Table 12–1:

1. A lock is acquired during a transaction if a record is read or reread after the start and before the end of a transaction.
2. A lock is acquired outside a transaction if the record is read when a transaction is not active and is released before a transaction starts.
3. A lock is acquired outside a transaction and held into a transaction if it is read when a transaction is not active and has not been released when a transaction starts.
4. Record release from a buffer occurs at end of scope, when a RELEASE statement is executed, or when the record is replaced in the buffer by a CREATE, FIND, or FOR EACH statement.
5. This is true even if the record was read with NO–LOCK prior to the transaction and is re–read EXCLUSIVE–LOCK during the transaction. By default, the lock is converted to SHARE, not back to NO–LOCK. To avoid holding the record SHARE in this case, release it during the transaction and re–read it as NO–LOCK.

Locks acquired within a transaction are released (or changed to SHARE–LOCK if locked prior to the transaction) if a transaction is backed out. This does not occur when a subtransaction is backed out because locks acquired within a transaction are not released unless the transaction ends or the entire transaction is undone.

How do the rules in Table 12–1 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. According to the rules, this SHARE–LOCK will be held until the end of the transaction or when the record is released, whichever is 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. According to the rules, this lock is held 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 is told 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 and the transaction ends, releasing the EXCLUSIVE–LOCK on the record.

7. The other user is able to find the record and update it.

You can see that, because of the duration of the record locks, your transactions are processed consistently in a multi-user environment.

12.5 RESOLVING LOCKING CONFLICTS

You can use different options to resolve locking conflicts to describe the way you want to lock records:

- EXCLUSIVE-LOCK
- SHARE-LOCK
- NO-LOCK

See Chapter 3 in *System Administration II: General* for more information about these options.

For example, if you know you are going to be updating a record, you could use the EXCLUSIVE-LOCK option with the FIND statement. Look at these procedures:

p-lock4.p

```
/* USER 1 */
DEFINE VARIABLE answer AS LOGICAL.

REPEAT:
  PROMPT-FOR customer.cust-num.
  FIND customer USING cust-num.
  UPDATE name max-credit sales-rep.
  SET answer LABEL "Update OK?".
  IF NOT answer THEN UNDO, NEXT.
END.
```

p-lock5.p

```
/* USER 2 */
FOR EACH customer:
  DISPLAY name address city st zip.
END.
```

There are ways to avoid this situation. For example, if you know you are going to be updating a record, you can read that record with EXCLUSIVE-LOCK. Alternatively, where you are running a reporting procedure such as p-lock5.p, you may not care as much about seeing data that might be 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 name address city st zip.
END.
```

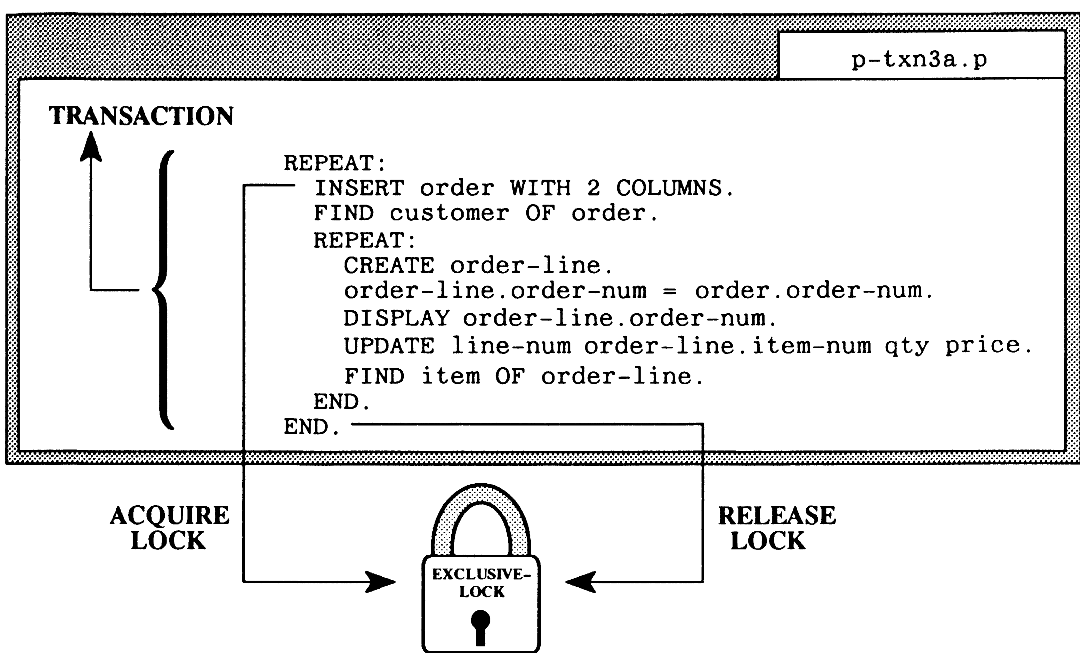
Try running the p-lock4.p procedure again. Once again, supply 1 as the customer number, make any change to the customer record, and press GO (F1). Now go to another terminal and run the p-lock6.p procedure.

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 three functions you can use when working with locking situations: NO-WAIT, AVAILABLE, and LOCKED. See the *PROGRESS Language Reference* manual for information and examples for using these functions to manage record locking conflicts.

12.5.1 Changing the Size of Transactions

Chapter 8 showed how the size of a transaction affects how much work gets undone or recovered in the event of a system failure or an error. Now you know that transactions and record locks are closely related. Let's look at how changing the size of a transaction affects record 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).

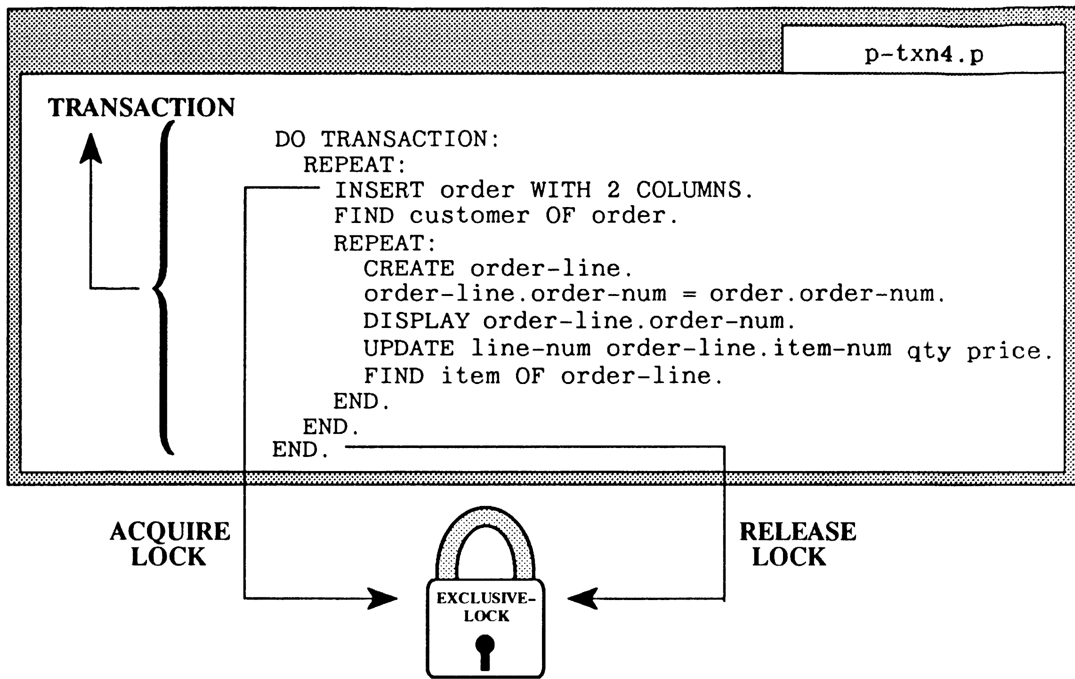
The UPDATE statement in the inner REPEAT block lets you update order-line records. While you are updating order-line records, you still have an EXCLUSIVE-LOCK on the order record you created at the start of the REPEAT block. Because of that lock, no other user can access that order record until you are finished 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.

12.5.2 Making Transactions 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:



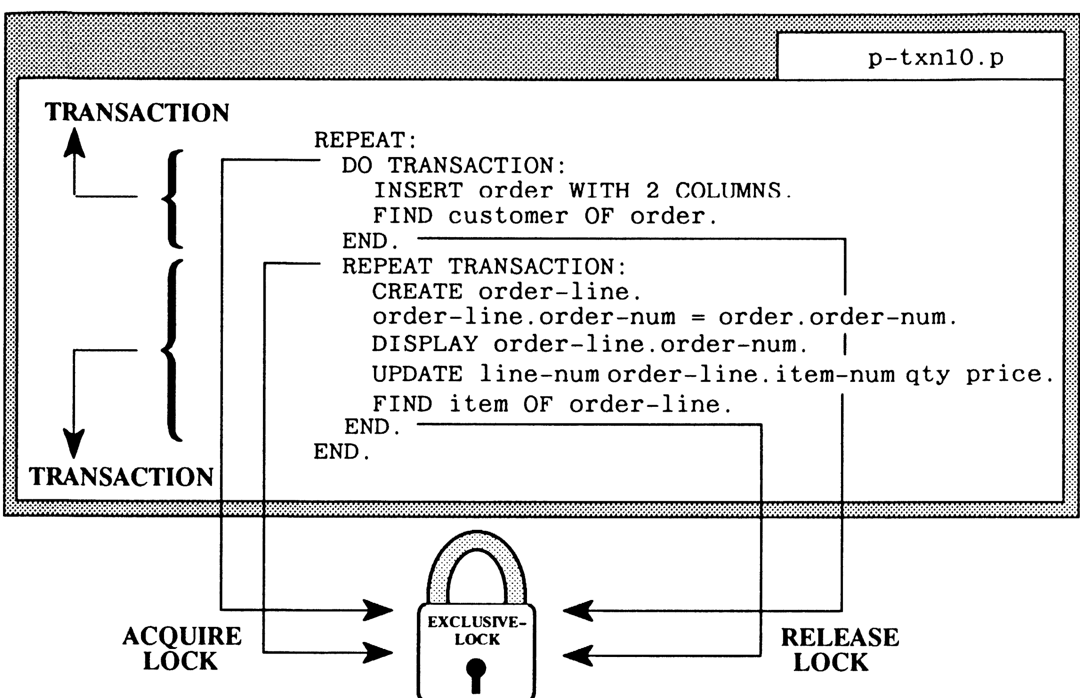
The TRANSACTION option on the DO block overrides the default transaction placement. That is, even though an outermost REPEAT, FOR EACH, or procedure block that contains direct database updates is normally automatically made the transaction block, the TRANSACTION option overrides that default.

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 see that you can have many order and order-line records EXCLUSIVE-LOCKed, preventing other users from accessing those records. And you have many item records SHARE-LOCKed, preventing other users from updating those records.

12.5.3 Making Transactions Smaller

Now imagine the reverse situation to the one in the last section. That is, 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.



Here, 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.

12.6 AUTOMATICALLY GENERATING AN INDEX VALUE

Most applications use several indexes, such as invoice numbers, customer numbers, employee identification numbers, and the like. Sometimes you don't want the user of the application to have to keep track of the last invoice, customer, or employee number used, and you may want your application to be able to generate these index values automatically.

To generate index values:

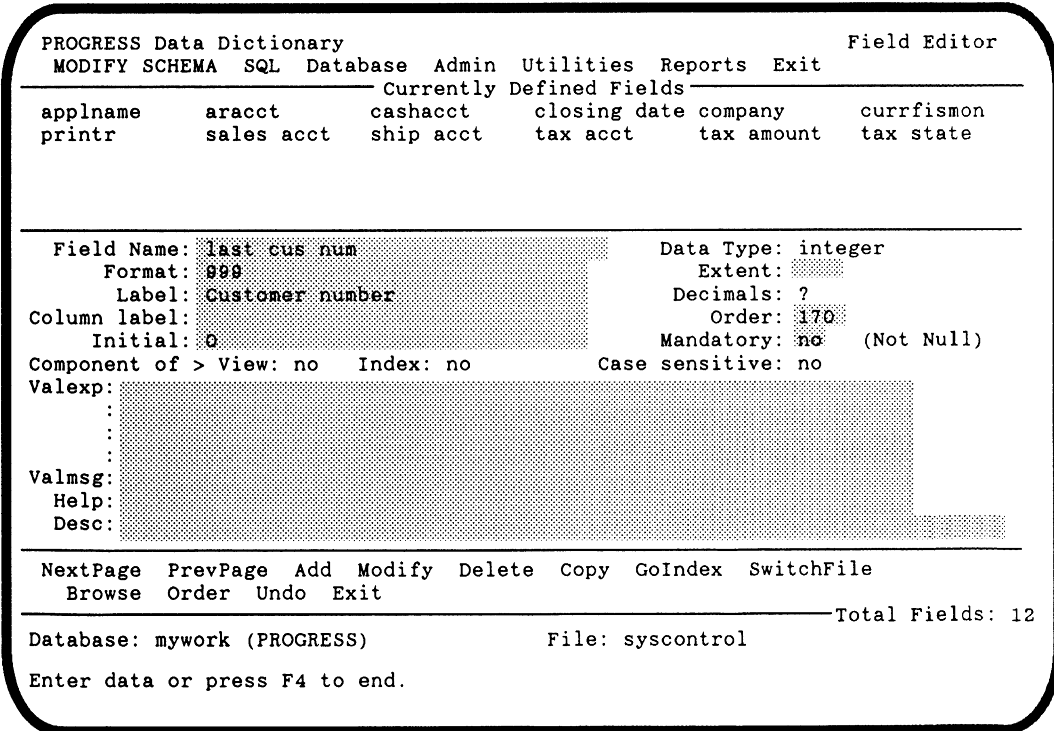
- Create a database file that contains a single field for each index value you want to generate.
- Write a procedure that produces the index value.

12.6.1 Creating a System Control File

You can use the Data Dictionary to create a "system control file" for your application. This file contains a single field for every piece of control information used by your application. For example, the demo database supplied with PROGRESS has a file called syscontrol that contains several fields:

- The company field contains the name of the company that owns the application.
- The applname field contains the name of the application.

Suppose you want to be able to automatically generate customer numbers. You might add a field called `last-cus-num` to your system control file. If you're not already in the Dictionary, go ahead and start it and choose syscontrol as the file in which you want to add and change fields.



After you define the `cust-num` field in the `syscontrol` file, you must store a value in that field. Let's store a 100 in that field as a starting customer number:

p-sctrl.p

```
FIND FIRST syscontrol.
DISPLAY last-cus-num LABEL "Old cust-num value" WITH FRAME a.
syscontrol.last-cus-num = 100.
DISPLAY last-cus-num LABEL "New cust-num value" WITH FRAME b.
```

This procedure gives the `last-cus-num` field an initial value of 100. Now you can write the procedure that will use that initial value as a starting point for generating other customer numbers.

12.6.2 Writing the Procedure to Generate an Index Value

Here is an example of a procedure that uses the syscontrol file to generate a key value, customer number. It assumes that you have defined a field called last-cus-num in your syscontrol file. (This field is not part of the distributed demo database.)

genky.p

```
DEFINE VARIABLE next-cust-num LIKE last-cus-num.
REPEAT WITH 1 COLUMN 1 DOWN ROW 5 NO-HIDE:
  VIEW FRAME a.
  CREATE customer.
  UPDATE name address city st zip max-credit sales-rep.
  FIND FIRST syscontrol EXCLUSIVE-LOCK.
  next-cust-num = syscontrol.last-cus-num + 1.
  syscontrol.last-cus-num = next-cust-num.
  customer.cust-num = next-cust-num.
  DISPLAY cust-num WITH FRAME a ROW 1 1 COLUMN.
  CLEAR FRAME a.
END.
```

This procedure creates a customer record and lets you update some fields (not including the `cust-num` field) in that record. The FIND statement reads the `syscontrol` record from the `syscontrol` file. The EXCLUSIVE-LOCK option ensures that no other user will be able to read the record before you have a chance to update it. If another user were able to read the record before you had a chance to update it, both you and the other user would be waiting to get an EXCLUSIVE-LOCK on that record until one of you pressed STOP.

After finding the `syscontrol` record, the procedure increments the `last-cus-num` field in that record by 1 and assigns that new value both to the newly created customer record and to the `last-cus-num` field. The procedure then displays the assigned customer number.

The EXCLUSIVE-LOCK acquired in the FIND statement is held until the end of the transaction, which is the REPEAT block in this procedure. This means that from the time you find the syscontrol record to the time a customer number has been assigned to the new customer record, no other user can use the value stored in the `last-cus-num` field of the `syscontrol` record. This time period will be very short.

If, in the above procedure, the customer number were determined before the record was created and the data entered, then the syscontrol record would be locked for a relatively long period, depending on the time a user takes to enter the data. During that time other users could not update the syscontrol record and could not also be entering orders.

Chapter 13 Multi–database Programming

This chapter provides an introduction to multi–database programming with the PROGRESS 4th generation language (4GL). Multi–database programming entails the use of several databases within a single PROGRESS application. The concepts and information supplied in this chapter are geared towards programmers who have already developed a PROGRESS application that accesses a single database. The chapter covers the following topics:

- Database connections.
- General connection considerations.
- Disconnecting databases.
- Development connection considerations.
- Run–time connection considerations.
- Multi–database programming techniques and considerations.
- Multi–database programming case studies.

In order for a PROGRESS session to communicate with a database, you must provide a *connection* between the session and the database. PROGRESS provides the ability to connect to several databases from a single session.

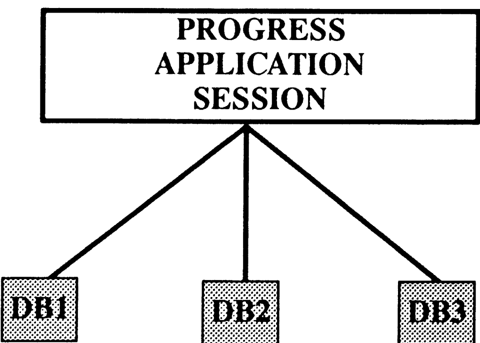


Figure 13–1: A PROGRESS Multi–database Application

With this capability, you can develop reports that join data from different databases and procedures that update several databases simultaneously. A PROGRESS multi-database application can simultaneously access PROGRESS, ORACLE, and RMS databases located on different operating systems using several different networking protocols.

Multi-database programming with the PROGRESS 4GL can be split into two general concepts: database connections and programming. The organization of this chapter reflects this conceptual split. There are two general sections in this chapter:

- Database connections - This section provides a general overview of how to connect/disconnect a PROGRESS session to/from several databases. It presents some general connection considerations. It also discusses database connections in the context of the application development process.
- Multi-database programming techniques and considerations - This section provides tips, programming techniques, and recommendations that you should consider when developing a PROGRESS application that accesses several databases.

At the end of this chapter, there are four multi-database programming case studies that put the techniques and recommendations presented in the previous sections to use in PROGRESS code examples.

13.1 DATABASE CONNECTIONS

As stated previously, a database must be connected in order to access its contents. PROGRESS allows you to connect to more than one database during a single session. There are four ways to connect to a database:

- At PROGRESS startup with a PROGRESS startup command or in a startup file with the PROGRESS executable.
- From the PROGRESS editor or within an application with the CONNECT statement.
- With the PROGRESS Data Dictionary.
- Using the Auto-Connect feature.

All of the database connection methods listed above provide for the use of connection parameters. *Connection parameters* control how you connect to databases. The following table lists the PROGRESS database connection parameters:

Table 13-1: Connection Parameters

PROGRESS Database Connection Parameters			
Single user	-1 (1)	No crash protection	-i (1)
After-image file (ai)	-a (1)	Logical database name	-ld
Buffers	-B (1)	Suppress file write	-Mf (1)
Index cursors	-c (1) (2)	Network	-N
Multi-user client	-cl (3)	Private buffers	-O (3)
Communications parm. file	-cp	Password	-P (2)
Index cursor size	-cs (1)	Parameter file	-pf
Communications timeout	-ct	UNIX non-raw I/O	-r (1)
Multi-user direct access	-da (3)	Read only	-RO (1)
Database name	-db	UNIX raw I/O	-R (1) (4)
Database type	-dt	Service	-S
Before image file (bi)	-g (1)	Userid	-U (2)
Host name	-H		

(1) Single user database connections only, otherwise use at PROGRESS server startup.
(2) Different meaning for Non-PROGRESS databases.
(3) For shared memory systems only.
(4) Limited use with CONNECT statement for PROGRESS versions on UNIX that do not use SWRITE or O_SYNC.

The most important connect parameter for multiple database programming is the Database Name (-db) parameter, which allows you to specify multiple databases in a parameter file or on the command line at PROGRESS startup. The default number of databases that can be connected during a PROGRESS session is 5. The Maximum Databases (-h) startup parameter allows you to set the number of connected databases allowed during a PROGRESS session up to a maximum of 240. For more information about PROGRESS startup and connect parameters, see Chapter 3 in *System Administration II: General.*

You can group connect parameters in an ASCII file called a *parameter file*. Use the Parameter File (-pf) connect parameter to invoke a parameter file with any of the connection methods listed above. Parameter files ordinarily have .pf file extensions. When creating a parameter file on a BTOS, CTOS, or VMS system, you can only enter UNIX-style connect parameters into the parameter file.

Parameter files are an important factor in developing database connection strategies for multi-database applications. In the sections that follow, you will see how parameter files are used in database connection strategies to allow you to freely change connection information for databases without recompiling applications. For more information on parameter files, see Chapter 3 in *System Administration II: General.*

NOTE: On UNIX machines that do not support O_SYNC and SWRITE, permissions issues limit the use of the CONNECT statement for raw I/O connections to databases in single-user and multi-user direct access mode.

At startup, the `_progres` module has super-user privileges that allow it to open raw disk devices. Any databases specified on the startup command line can therefore be opened with raw I/O. After startup, the `_progres` module relinquishes the super-user privileges that allow it to open raw disk devices. As a result, you cannot use the CONNECT statement to establish a raw I/O connection to a database in either single-user or multi-user direct access mode. For more information, see the CONNECT statement in *PROGRESS Language Reference.*

NOTE: If your application uses FAST TRACK on multiple databases, you must connect all databases either at startup or with the auto-connect method.

13.1.1 Connecting At Startup

You can connect to databases at session startup with one of the PROGRESS startup commands or in an application startup file. For example:

```
pro mydb1 -db mydb2 -db mydb3
```

In order to retain compatibility with earlier versions of PROGRESS, it is not necessary to use the Database Name (-db) connect parameter with the **first** database on the PROGRESS single and multi-user startup command line.

You can also use a parameter file to specify database connection parameters. The following example shows a UNIX startup script for a PROGRESS application:

starter

```
DLC=${DLC-/usr/dlc}; export DLC
PATH=:$PATH:$DLC; export PATH
PROPATH=:$DLC; export PROPATH
exec $DLC/_progres -pf parm1.pf -pf parm2.pf -p $APPL/start.p
```

The startup script above sets up environment variables for PROGRESS application and starts a PROGRESS application session using two parameter files called `parm1.pf` and `parm2.pf`. These parameter files contain the following entries (note that it is possible to put the contents of both of these files into one parameter file):

parm1.pf

```
-db appldb1
```

parm2.pf

```
-db appldb2
```

Note that the script `starter` is specific to a single operating system, but the parameter files are not. For more information on starting PROGRESS, see Chapter 2 in *System Administration II: General*.

13.1.2 Connecting From a Procedure with the CONNECT Statement

The CONNECT statement allows you to connect to a database from a PROGRESS procedure or from the PROGRESS editor. The CONNECT statement has the following syntax:

SYNTAX

```
CONNECT { physical-name             [options] }
        { -db physical-name         [options] }
        { -pf parameter-file        [options] }

    [ { -db physical-name   [options] } ] ... [NO-ERROR]
    [ { -pf parameter-file  [options] } ]
```

The *physical-name* represents the actual name of the database on a disk. The first *physical-name* specified in a CONNECT statement does not require the Database Name (-db) parameter. All subsequent *physical-names* must be preceded by the Database Name (-db) parameter. The *parameter-file* is the name of a parameter file containing database connection information. The *options* argument represents one or more connect options. See Chapter 3 in *System Administration II: General* for more information about PROGRESS connect parameters. The NO-ERROR option suppresses the error condition, but still displays the error message when an attempt to CONNECT to a database fails.

Connecting with the CONNECT statement is very similar to connecting at startup. For example:

```
CONNECT appldb1.
CONNECT appldb2.
```

Although it is possible to connect to several databases within one CONNECT statement, it is a good idea to connect only one database per CONNECT statement. This is because a connection failure for one database causes a termination of the current CONNECT statement, leaving any subsequent databases specified with the CONNECT statement unconnected.

You can also use parameter files with the CONNECT statement. For example:

```
CONNECT -pf parm3.pf.
```

In the example above, the CONNECT statement uses the following parm3.pf file to connect to a database (appldb1).

parm3.pf

```
-db appldb1
```

You cannot use the CONNECT statement to connect to a database and then reference a file or field from that database in the same procedure. You must connect to a given database before you run a procedure that references files and fields in that database. The following example procedure will not run because it connects and references the demo database with the same procedure.

```
/* NOTE: this code does NOT work */

CONNECT demo.
FOR EACH demo.customer:
  DISPLAY customer.
END.
```

Split the above procedure into a procedure and a subprocedure. Connect to a database in the main procedure, and then run a subprocedure that accesses the connected database.

```
/* topproc.p */

CONNECT demo.
RUN subproc.p.
```

```
/* subproc.p */

FOR EACH demo.customer:
  DISPLAY customer.
END.
```

For more information about the CONNECT statement, consult the *PROGRESS Language Reference*.

13.1.3 Connecting With the PROGRESS Data Dictionary

You can connect to a database during a PROGRESS session using the PROGRESS Data Dictionary. To connect to a database from the PROGRESS Data Dictionary, press **D** (`Database`) at the Data Dictionary Main Menu; then press **C** (`Connect a Database`). The following screen appears:

```
PROGRESS Data Dictionary                                  Connect a Database
Modify-Schema  SQL  Database  Admin  Utilities  Reports  Exit

 ------------ Enter Parameters for Database Connection ------------

  Physical name:
   Logical name:
  Database type: PROGRESS
 Parameter File:
        User id:                Password:
   Single user?: yes
   Before image:
    After image:
 (Leave blank to use PROGRESS default values.)

 Other UNIX-style parameters:
      :
      :

Database: demo (PROGRESS)                 File:

Enter data or press F4 to end.
```

Figure 13–2: Connect A Database Window

The screen prompts for nine connection parameters explicitly, and provides two lines for additional parameters. Internally, the Data Dictionary constructs (and executes) a CONNECT statement using the information you supply. Therefore, any rules that apply to the CONNECT statement also apply to database connections using the PROGRESS Data Dictionary. For more information about connection parameters, consult Chapter 3 in *System Administration II: General*. Chapter 3 in the *PROGRESS Language Tutorial* contains information about using the PROGRESS Data Dictionary.

13.1.4 Connecting With Auto-Connect

The PROGRESS auto-connect feature uses information stored in a primary application database to connect to a second application database when data from the second database is referenced in a compiled application at run-time. The primary application database, containing the database connection information for the second database, must already be connected before PROGRESS can execute the auto-connect for the second application database. PROGRESS executes an auto-connect immediately prior to running a procedure that accesses a database on the auto-connect list.

You can use the PROGRESS Data Dictionary to set up automatic database connections in a database. Select the `Database` option from the Data Dictionary Main Menu, followed by the `PROGRESS Utilities` and `Edit PROGRESS Auto-Connect List` options to create and edit an auto-connect list. The following screen appears:

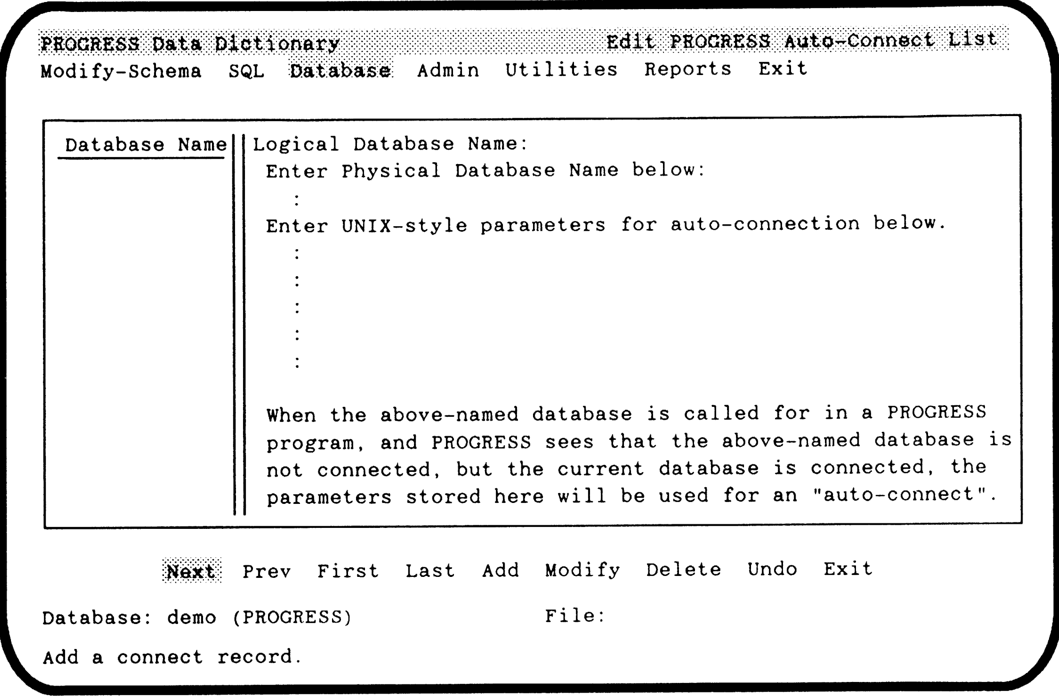


Figure 13-3: Auto-Connect List Window

This window allows you to enter and edit connect information (physical name, logical name, and any other connection parameters) for a database. Chapter 3 in the *PROGRESS Language Tutorial* contains information about using the PROGRESS Data Dictionary.

If you connect a database with the CONNECT statement and that database also has an auto-connect entry in an already connected database, the connect information from both the CONNECT statement and the auto-connect list is merged. In this situation, the connection information in the CONNECT statement takes precedence. See also CONNECT in the *PROGRESS Language Reference.*

13.2 GENERAL CONNECTION CONSIDERATIONS

When connecting to databases, there are several important connection issues you should consider:

- Logical database names
- Connection modes
- Database types
- Database location

The following sections discuss these connection considerations in relation to multi-database applications and identify important connect parameters. For more information about connect parameters, see Chapter 3 in *System Administration II: General.*

13.2.1 Logical Database Names

A *logical database name* is a database reference representing the name of a connected physical database. When you connect to a database, that database is automatically assigned 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. The logical database name mydb1 is used to resolve database references and is stored in the .r code of any procedures that reference the mydb1.db database.

The logical database name (-ld) connect parameter allows you to specify a logical database name other than the default. For example:

```
pro mydb1 -ld firstdb
```

The example above establishes the logical name firstdb for the physical database mydb1.db during the current PROGRESS session. When you develop and compile an application to run on the mydb1.db database, it is the logical name, not the physical name, that is stored 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. In order to run a compiled application on a new physical database without recompiling, the new database must have identical structure and time stamp information for the files accessed by the application and must be connected with the same logical name used to compile the application. For example:

```
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 Dictionary against a database connected with the logical name DICTDB.

A database connection fails if the logical database name of the database that you are connecting has the same logical name as an already connected database. If you try to connect a database with a logical database name that matches the logical database name of an existing, connected database of the same database type (PROGRESS, ORACLE, etc.), PROGRESS assumes that database to be already connected and ignores that request.

13.2.2 Connection Modes

All databases have a connection mode established at connection time. A database connection mode determines how many PROGRESS sessions can use a database at one time. There are three connection modes:

Single user - Only the current PROGRESS session can access the specified database. If the database is already in use, the database cannot be connected for the current PROGRESS session.

Multi-user Client - Many PROGRESS sessions can access the specified database simultaneously. With this connection mode, the current PROGRESS session uses a server process to access the database. Multi-user client is the default connection mode on non-shared memory systems and on shared memory systems when you specify both the Host Name (-H) and the Server Name (-S) connect parameters. This mode is required for remote database connections.

Multi-user Direct Access - Many PROGRESS sessions can access the specified database simultaneously. With this connection mode, the current PROGRESS session bypasses all server processes and accesses the database through shared memory. This connection mode is available only on shared memory systems and is the default connection mode on shared memory systems. You cannot connect to remote databases with this connection mode.

If you connect to a database in either of the multi-user connection modes, a server or a broker must already be running for that database. For more information about starting a server or broker, consult Chapter 2 in *System Administration II: General.*

NOTE: On some UNIX systems, permissions issues limit the use of the CONNECT statement for raw I/O connections to databases in single-user or multi-user direct access mode. Consult the CONNECT statement in the *PROGRESS Language Reference* for more information about limitations on raw I/O database connections and some suggested work arounds.

When you connect to a database with a PROGRESS multi-user startup command (`mpro` or `mbpro`), all of the databases specified on the command line are connected in the default multi-user mode for the current machine. All databases connected with the PROGRESS (`_progres`) module in a startup file or with a CONNECT statement in an application are also connected in the default multi-user mode for the current machine. To maintain compatibility with previous versions of PROGRESS, the first database specified after the PROGRESS single user (`pro`) or PROGRESS single user batch (`bpro`) command is connected in single user mode by default. All subsequent databases specified with a PROGRESS single-user startup command using the Database Name (-db) connect parameter are connected in default multi-user mode for the current machine.

PROGRESS supplies three connect parameters that allow you to override the default connection mode for a database and force another connection mode:

Table 13-2: Connection Mode Parameters

CONNECTION MODE	PARAMETER
Single User	-1
Multi-user Client	-cl
Multi-user Direct Access	-da

The following connection example uses a parameter file to connect three databases on a shared memory UNIX system using all three connection modes:

```
pro -pf modprml.pf
```

The `modprml.pf` file has the following entries:

modprml.pf

```
-db dbl -ld firstdb
-db db2 -1 -ld secondb
-db db3 -ld thirdb -H machinel -S db3sv
```

This example connects to db1 in the default multi-user direct access mode, to db2 in single user mode, and db3 in multi-user client mode. The Host (-H) and Service name (-S) parameters are required by a TCP/IP network to access a database over that network.

For more information about PROGRESS on shared memory systems, see Appendix C in *System Administration II: General.*

13.2.3 Database Types

Connections to non-PROGRESS databases (ORACLE or RMS) are handled with the appropriate database gateway. A *database gateway* is a PROGRESS add-on module that allows you to access a non-PROGRESS database from within applications written in PROGRESS 4GL.

Before you can connect to a non-PROGRESS database, you first must create a *schema holder* and then create a *schema* within the schema holder for the non-PROGRESS database. A schema holder is a PROGRESS database that contains the schema definitions for one or more non-PROGRESS databases. A schema is a description of the structure of a non-PROGRESS database (ORACLE or RMS) within a schema holder. Use the PROGRESS Data Dictionary to set up a non-PROGRESS schema within a schema holder. In addition to non-PROGRESS schemas, a schema holder can contain an ordinary PROGRESS schema. To learn more about non-PROGRESS databases, consult the appropriate chapter in the *Database Gateways Guide.*

After you have created a PROGRESS schema holder and schema for a non-PROGRESS database, connecting to a non-PROGRESS database involves two steps:

1. Connect to the PROGRESS schema holder that contains the non-PROGRESS schema.

2. Connect to the non-PROGRESS database.

You can make the two connections using any of the connection methods discussed above. Non-PROGRESS database connections may have special requirements, such as certain connect parameters. Consult the appropriate chapter in the *Database Gateways Guide* for information about connecting to non-PROGRESS databases. The following example shows a connection to a PROGRESS schema holder and an ORACLE database at session startup:

```
pro -pf oraparm.pf
```

The oraparm.pf file has the following entries:

oraparm.pf

```
-db holdb -ld holder
-db x -ld odbl -U orauid -P orapwd
```

The first line in the oraparm.pf file connects to the holdb database, which is a PROGRESS schema holder. The second line in the oraparm.pf file connects to the ORACLE database. The logical database name (-ld) odb1 is the name of the ORACLE schema within the schema holder. The logical name of a non-PROGRESS database must match the name of the non-PROGRESS schema in the PROGRESS schema holder. Use this logical name within PROGRESS procedures, to reference the ORACLE database. The Userid (-U) and Password (-P) connect parameters are required by ORACLE to establish privileges for the ORACLE database.

13.2.4 Database Location

Another important connection consideration is the location of the database in relation to the application session. The database location splits application databases into two general types:

- Local database – located on the same machine as the application session.
- Remote database – located on a remote machine that is networked to the machine containing the application session.

This section shows how to connect to multiple databases in three basic configurations:

- Federated – All of the application databases are local to the application session.
- Distributed – One or more of the application databases are located on remote machines connected to the machine containing the application session using a single networking protocol.
- Distributed/Simultaneous Networks – The application databases are located on remote machines connected to the machine containing the application session using different networking protocols.

Federated Connection Scenario. All databases in a federated configuration are local databases. The following diagram shows a PROGRESS application session on a UNIX machine accessing two local PROGRESS databases and a local non-PROGRESS database. Remember, a database gateway, a PROGRESS schema holder, and a non-PROGRESS schema for the non-PROGRESS database, must already be set up before you can connect to the non-PROGRESS database.

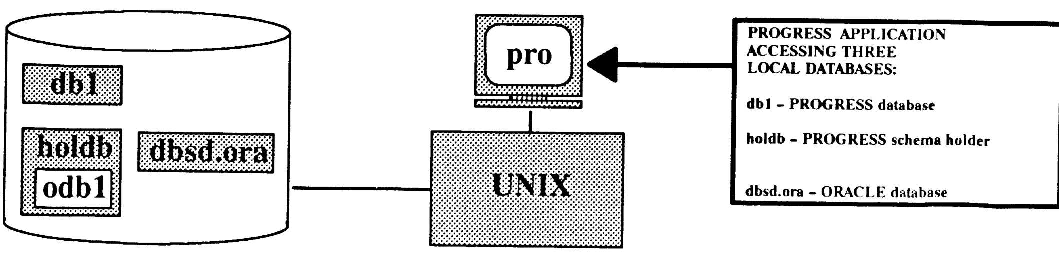


Figure 13–4: Federated Scenario

The db1 database is a PROGRESS database and the holdb is a PROGRESS schema holder holding a ORACLE schema called odb1. The ORACLE schema odb1 describes the schema in the ORACLE database dbsd.ora. The ORACLE Database Gateway handles the interface between the PROGRESS application session and the ORACLE database dbsd.ora through the ORACLE schema odb1 in the holdb database. The example below shows how to connect these local databases to a PROGRESS session on a shared memory system at session startup.

```
pro -pf parm4a.pf -pf parm4b.pf
```

The parameter files have the following entries:

parm4a.pf

```
-db db1 -ld onedb
```

parm4b.pf

```
-db holdb -1 -ld twodb
-db x -ld odb1 -1 -U orauid -P orapwd
```

In the parm4a.pf file, the first line connects to the db1 database in multi-user direct access mode. The first line in the parm4b.pf file connects to the holdb database in the single-user mode. The second line connects to the ORACLE schema odb1 associated with the ORACLE database dbsd.ora. Use the logical name odb1 within your PROGRESS procedures to reference the ORACLE database dbsd.ora.

Distributed Connection Scenario. Distributed databases are remote databases accessed by the PROGRESS application session using a single networking protocol. To connect a PROGRESS application session to a remote database over a network, you must use the connect parameters required by the networking protocol. For information about connecting to remote databases using a network, consult the appropriate networking chapter in the *System Administration I: Environments* guide.

The following diagram shows a PROGRESS application on a UNIX machine accessing a local database and two remote databases using TCP/IP.

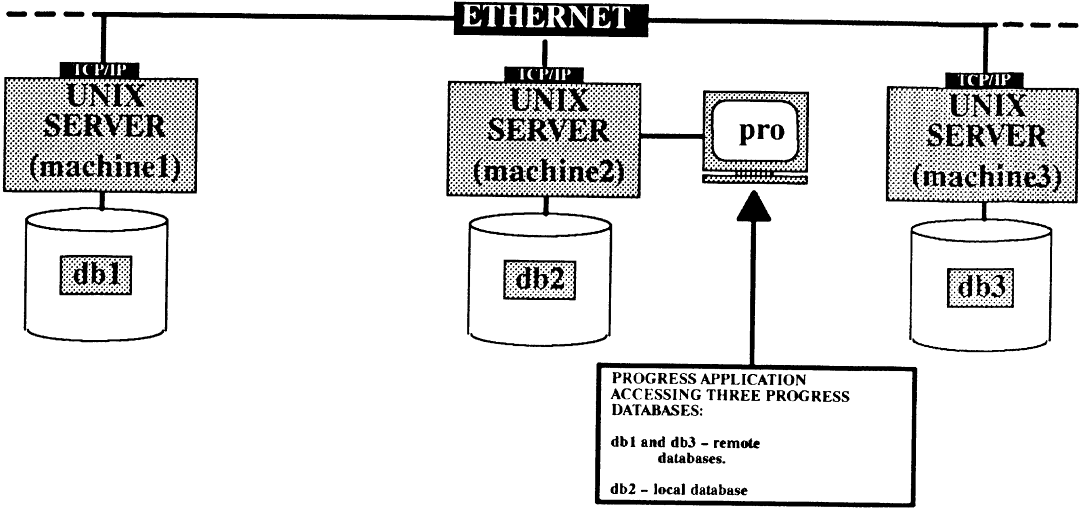
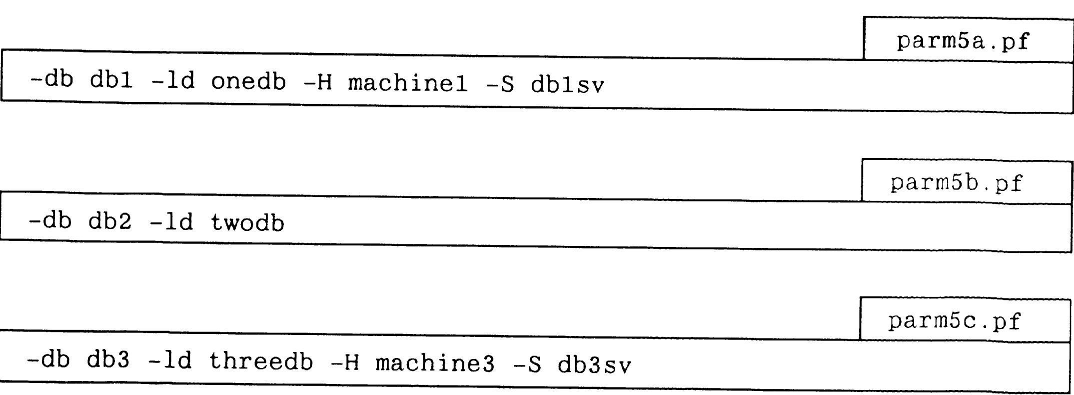


Figure 13–5: Distributed Scenario

The `db1` and `db3` databases are remote PROGRESS databases accessed using TCP/IP. The `db2` database is a local PROGRESS database. The example below shows how to connect to these three databases on a shared memory system at session startup.

```
pro -pf parm5a.pf -pf parm5b.pf -pf parm5c.pf
```

The parameter files have the following entries:



parm5a.pf

```
-db db1 -ld onedb -H machine1 -S db1sv
```

parm5b.pf

```
-db db2 -ld twodb
```

parm5c.pf

```
-db db3 -ld threedb -H machine3 -S db3sv
```

This example connects to the remote database db1 on machine machine1 and to the remote database db3 on machine machine3 in the default multi-user client mode. The TCP/IP networking protocol requires that you specify the host machine (-H) and a service name (-S) to access remote databases. It also connects to the local database db2 in multi-user direct access mode.

Distributed/Simultaneous Networks Connection Scenario. PROGRESS also allows an application session to access remote databases using different networking protocols simultaneously. The machine containing the application session must have the appropriate PROGRESS networking modules installed to access remote application databases.

In this scenario, you must use the Network (-N) connect parameter to specify the networking protocol used to access each remote database. The following table shows the networking protocols that you can use to remotely access a database and supplies the appropriate Network (-N) parameter to use for each:

Table 13-3: The Network (-N) Parameter

NETWORK PROTOCOL	NETWORK PARAMETER (DOS)
TCP/IP	-N TCP
NETBIOS	-N NETBIOS
IPX/SPX	-N SPX
DECnet-DOS	-N NETBIOS*

* PROGRESS uses the NETBIOS emulation capabilities of DECnet-DOS to facilitate communications between a PROGRESS client session on a DOS machine and a database server on a VMS machine.

NOTE: This table is DOS specific. Currently only DOS has simultaneous networks.

Along with the Network (-N) parameter, you must also use the connect parameters required by the networking protocol specified with the Network (-N) parameter to connect to remote databases.

The following diagram shows a PROGRESS application session on a DOS machine accessing remote databases using three different network protocols simultaneously.

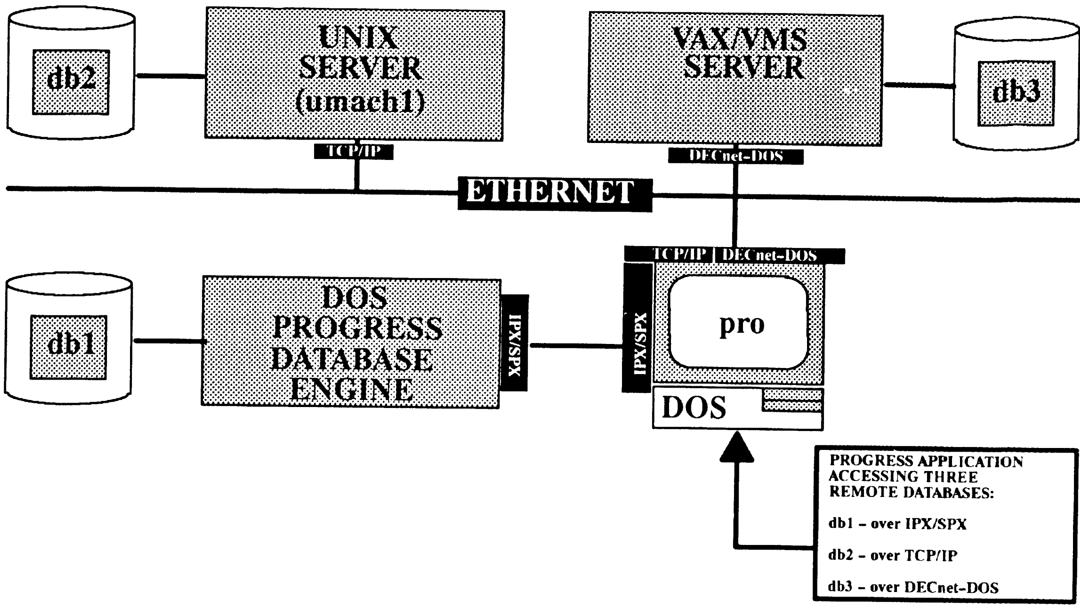


Figure 13–6: Distributed/Simultaneous Networks Scenario

The db1 database is a remote PROGRESS databases accessed using IPX/SPX. The db2 database is a remote PROGRESS database accessed using TCP/IP. The db3 database is a remote PROGRESS database accessed using DECnet-DOS. The DOS machine containing the PROGRESS application session must be directly networked to the machines that contain the remote databases. The example below shows how to connect a PROGRESS session to these databases on a shared memory system at session startup.

```
pro -pf parm6a.pf -pf parm6b.pf -pf parm6c.pf
```

The parameter files have the following entries:

parm6a.pf

```
-db db1 -ld onedb -N SPX -S db1sv
```

parm6b.pf

```
-db db2 -ld twodb -N TCP -H umach1 -S db2sv
```

parm6c.pf

```
-db db3 -ld threedb -N NETBIOS -S db3sv
```

The parm6a.pf file accesses the remote database db1 using IPX/SPX. Only the Service Name (-S) parameter is required when connecting remote databases over an IPX/SPX network. The parm6b.pf file accesses the remote database db2 on a UNIX machine called umach1 using TCP/IP. Because you are using TCP/IP to access the db1 database remotely, the Host Name (-H) and Service Name (-S) startup parameters are required. The parm6c.pf file accesses the remote database db3 located on a VMS node using DECnet-DOS. Only the Service Name (-S) parameter is required when connecting remote databases using DECnet-DOS. All three databases are connected in the multi-user client mode.

For more information about connecting to remote databases over a network, consult the appropriate networking chapter in *System Administration I: Environments*.

13.3 DISCONNECTING DATABASES

By default, all connected databases are disconnected at the end of a PROGRESS session. To explicitly disconnect a database from within a PROGRESS application, use the DISCONNECT statement. The DISCONNECT statement 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. For example:

mainproc1.p

```
CONNECT -db mydb -1.
RUN subproc1.p.
```

subproc1.p

```
RUN subproc2.p.
FOR EACH mydb.customer:
   DISPLAY name address city st zip.
END.
```

subproc2.p

```
DISCONNECT mydb.
```

In the procedures listed above, the mydb database is not disconnected until the end of the subproc1.p procedure.

If a CONNECT statement executes for an already connected database after a DISCONNECT statement for the same database, that database will remain connected despite the explicit DISCONNECT statement. For example:

mainproc2.p

```
CONNECT -db mydb -1.
RUN subproc3.p.
```

subproc3.p

```
RUN subproc4.p.
FOR EACH mydb.customer:
   DISPLAY name address city st zip.
END.
```

subproc4.p

```
DISCONNECT mydb.
CONNECT -db mydb -1.
```

In the procedures listed above, the mydb database is not disconnected and the connect parameters specified with the second CONNECT statement are ignored.

13.4 DEVELOPMENT CONNECTION CONSIDERATIONS

When preparing to develop a PROGRESS multi-database application, keep the following information about database connections in mind:

- A copy of each application database must be connected before you develop and compile your multi-database PROGRESS application. The database location and connection modes of the application databases in the development environment do not have to match the intended run-time environment.

- When you connect your application databases, establish unique logical names for each of the databases. These logical database names are used to reference databases within your application and are stored in the .r code of the application at compilation time.

- If you connect to a non-PROGRESS database, be sure to consult the appropriate chapter in the *Database Gateways Guide* for information about special connection and programming considerations.

- If any of the application databases have data security, you must connect to each database with a userid and password that allows you to compile procedures. Userids and passwords may vary across databases. When you start a PROGRESS session with the default startup procedure (`prostart.p`), PROGRESS runs `login.p` for each PROGRESS database specified on the startup command line that has records in its `_User` file. Use the Userid (`-U`) and Password (`-P`) connection parameters to connect with the appropriate userid from the PROGRESS editor. The Userid (`-U`) and Password (`-P`) connect parameters are required by some non-PROGRESS databases. For more information about data security, consult Chapter 5 in *System Administration II: General*. For more information about security considerations for non-PROGRESS databases, consult the appropriate chapter in the *Database Gateways Guide.*

Developers can connect to databases at PROGRESS startup, from the PROGRESS editor using the CONNECT statement, or using the PROGRESS Data Dictionary. Although the auto-connect feature allows run-time access of databases that are not explicitly connected within a compiled application, the same does not hold true at compilation time. To compile a procedure, you first need to connect to all of the databases that the procedure references.

13.5 RUN-TIME CONNECTION CONSIDERATIONS

One of the first and most important steps in the development process of a multi-database application is the selection of a run-time connection method. In selecting a run-time connection method, you must weigh the costs and benefits of a database connection method with the needs of your application in the end-user environment. There are three basic run-time database connection methods for multi-database applications:

- Connect application databases as needed within the PROGRESS application using the CONNECT statement and parameter files.

- Connect all application databases at PROGRESS application startup.

- Connect to a primary application database at PROGRESS startup and use the auto-connect feature to connect to secondary application databases as they are referenced in the compiled PROGRESS application.

Parameter files can be used with all three of the connection methods listed above. When you connect to application databases in the run-time environment, you must also be sure to establish the same logical database names used to compile the application. If you do not use the proper logical database names, your application will not run.

The following sections discuss a variety of run-time connection issues that you should consider when selecting one or a combination of the connection methods listed above to use in your application. The recommendations made in the following sections are general in nature and do not apply to every application development situation.

13.5.1 Creating Machine Independent Applications

Database connection requirements for a multi-database application change from machine to machine. Parameter files allow you to freely change connection information for databases without recompiling applications. This allows you to develop and compile multi-database applications that are machine independent. The following diagram shows two sessions of an application (3dbapp) running on two different machines on a network.

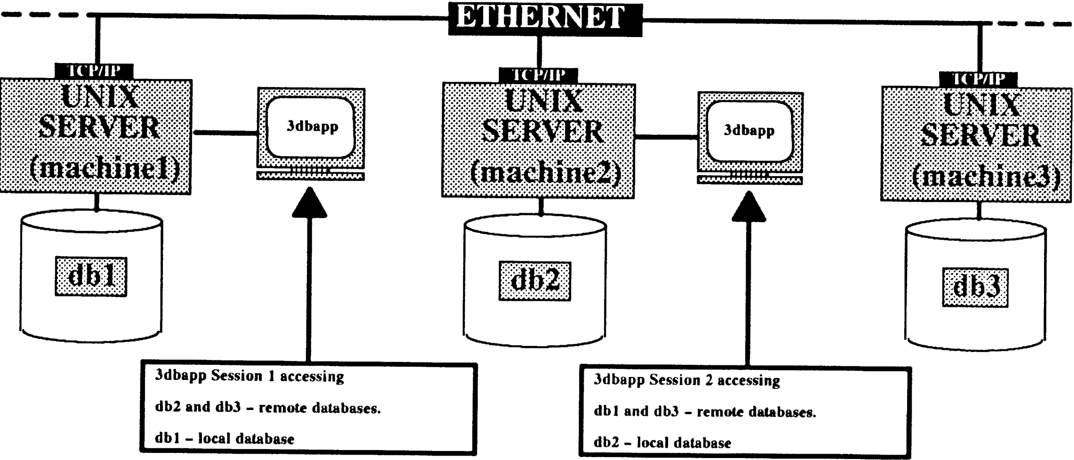


Figure 13-7: Machine Independent Applications

As illustrated above, the database connections for the 3dbapp application change from machine1 to machine2. The location of application databases changes in relation to the location of the 3dbapp application session. This has an impact on the database connection modes established for each application database.

The 3dbapp application connects to the databases at application session startup using a parameter file as follows:

```
$DLC/_progres -p 3bdapp.p -pf 3dbapp.pf
```

Each machine must have its own local copy of the 3dbapp.pf file. The local 3dbapp.pf file for 3dbapp application session running on machine1 (a shared memory system) has the following entries:

```
3dbapp.pf

-db db1 -ld onedb
-db db2 -ld twodb -H machine2 -S db1sv
-db db3 -ld threedb -H machine3 -S db3sv
```

This copy of the 3dbapp.pf parameter file connects to the local database db1 in multi-user direct access mode and connects to the remote databases db2 and db3 in multi-user client mode. You cannot access a remote database in multi-user direct access mode.

The local 3dbapp.pf file for 3dbapp application session running on machine2 (a shared memory system) has the following entries:

```
3dbapp.pf

-db db1 -ld onedb -H machine1 -S db1sv
-db db2 -ld twodb
-db db3 -ld threedb -H machine3 -S db3sv
```

This copy of the 3dbapp.pf parameter file connects to the local database db2 in multi-user direct access mode and connects to the remote databases db1 and db3 in multi-user client mode.

Despite the changes in the database connection information, both copies of the 3dbapp.pf parameter file create the same logical database names (onedb, twodb, and threedb) for use with the compiled multi-database application (3dbapp). For a full explanation of parameter files, see Chapter 2 in *System Administration II: General.*

13.5.2 Understanding and Managing Connection Failures and Disruptions

A key facet of multiple database programming is the ability to manage database connection failures and disruptions to minimize their effect on an application. The following list presents several of the reasons a database connection can fail:

- The database no longer exists or has a new physical name.
- The network used to access the application database is down.
- The machine containing the application database is down.
- There is no database server for multi-user database access.

- The logical name of the application database to be connected already exists for the current session.
- The syntax for a connection parameter is wrong.

It is important to understand how PROGRESS handles a database connection failure with each of the three run-time connection methods discussed previously. The following table lists the default connection failure behavior for each of the run-time connection methods.

Table 13-4: Connection Failure Behavior

Connection Method	Default Connection Failure Behavior
At PROGRESS startup	The PROGRESS session does not run.
CONNECT statement	The procedure executes up to the CONNECT statement in which the connection failure occurs. The database connection failure raises the error condition for the procedure. PROGRESS error processing does not undo database connections or disconnections. Any databases connected in the procedure before the failed database connection remain connected. See Chapter 8 in the this book for more information about error handling.
Auto-connect	The procedures containing a reference to the auto-connect database does not run, and a stop or break condition results in the calling procedure.

With all of the connection methods listed above, PROGRESS displays error messages at connection time if a database connection fails.

Prior to running a procedure or subprocedure, PROGRESS checks the procedure for database references and then looks for a corresponding connected database. If the database is not connected, PROGRESS checks the auto-connect list of each connected database for a corresponding database entry. If the database appears in the auto-connect list of an already connected database, the database is connected prior to running the procedure that references it. If the database is not connected and does not have an entry in an auto-connect list, the procedure that references that database does not run, and a stop or break condition is raised in the calling procedure.

Even after a successful database connection, it is possible that a server, network, or machine failure could disrupt your application's access to an application database. If program control passes through a subprocedure that references a database that was successfully connected but has since failed, the result is a stop or break condition in the calling procedure.

There are a number of ways to manage connection failures and disruptions to minimize their effect on an application. The following sections present a number of recommendations to help you manage database connection failures and disruptions in an application. Not all of these recommendations will suit the needs or design of your particular application.

Using CONNECT With NO-ERROR. The NO-ERROR option of the CONNECT statement allows the procedure containing the CONNECT statement to continue executing despite the termination of the CONNECT statement due to database connection failure. NO-ERROR suppresses the error condition; however, PROGRESS still displays a message about the failed database connection. A database connection failure during a CONNECT statement without the NO-ERROR option causes an error condition in the procedure containing the CONNECT statement. Always use the NO-ERROR option with a CONNECT statement in a PROGRESS startup procedure designated with the Startup Procedure (-p) parameter.

When you suppress the error condition at database connection time, PROGRESS does not know about the unconnected database. If program control passes to a subprocedure that references the unconnected database, PROGRESS does not run the subprocedure, and a stop or break condition results in the calling procedure. Therefore, you should test if a database is connected before executing subprocedures that reference that database.

Using the CONNECTED Function. Use the CONNECTED function to test if a database is actually connected prior to running procedures that reference a database. The CONNECTED function sends out a message from the application session to a database to confirm that the database is still connected. The CONNECTED function helps you to route program control around portions of your application affected by database connection failures and disruptions. For example:

```
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".
```

The PROGRESS code listed above tests a database connection using the CONNECTED function. If the database is not connected, the code attempts to connect the database. This example only runs the *procedure* that references the database if the database is connected. Auto-connect precludes the use of the CONNECTED function to test for database connections. For more information about the CONNECTED function, consult the *PROGRESS Language Reference*.

Positioning Database References Within Your 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. For example:

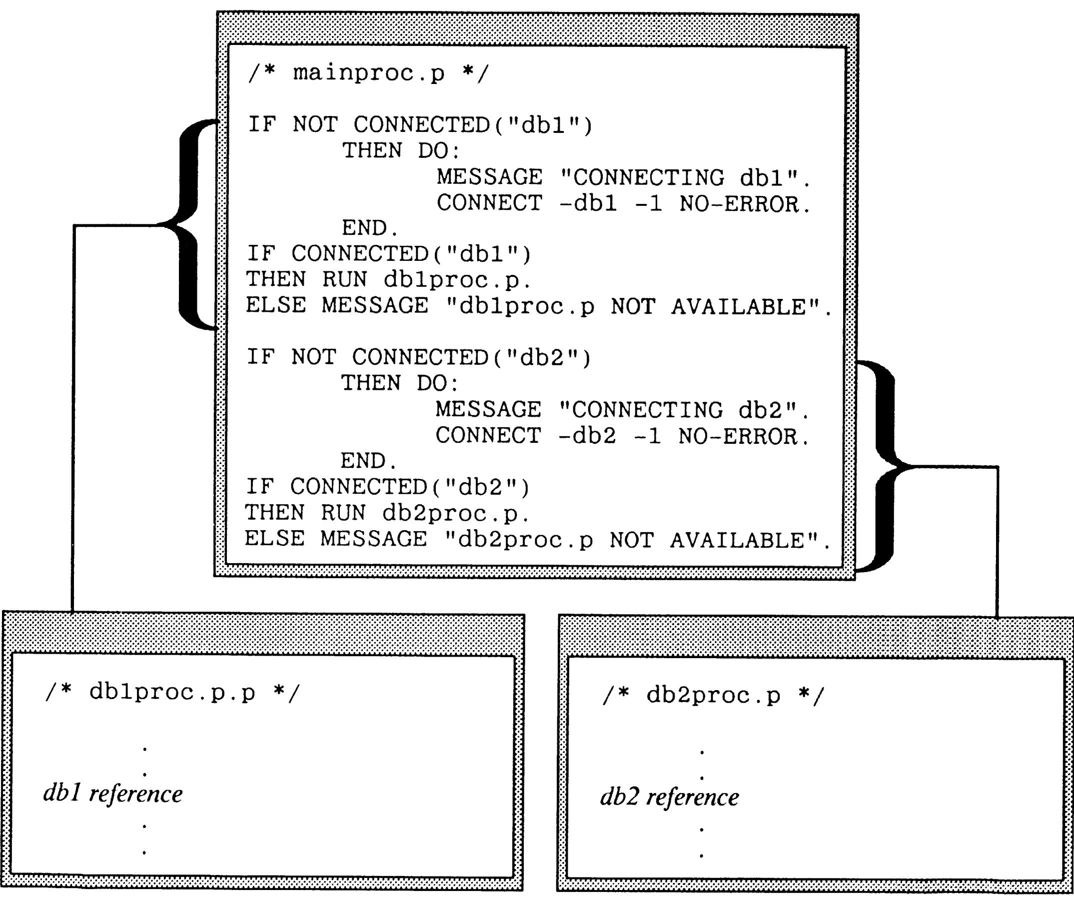


Figure 13–8: Positioning Database References

The diagram above 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 - except possibly to connect it - in your application startup procedure. Remember, you cannot connect to a database and reference a database in the same procedure, so this recommendation is for those developers who are connecting application databases with auto-connect. 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 will not run.

13.5.3 Conserving Database Connections Versus Minimizing Connection Overhead

On systems where the number of connections to a multi-user database is limited, conserving database connections may be an issue. The maximum number of connections to a multi-user database varies from machine to machine. System administrators can also limit the number of connections to database servers with the Number of users (-n) startup parameter at server startup time. When connections to a particular database nears the maximum for that database, connection conservation becomes a necessity.

When you connect to a database at startup, the database remains connected for the entire application session, or until it is explicitly disconnected with the DISCONNECT statement in the application. If demand is high for connections to a particular database, and your application accesses that database only in certain procedures, connecting that database at startup and holding the database connection for the entire length of the application session is inefficient and wasteful.

When database connection conservation is an issue, use explicit CONNECT and DISCONNECT statements in your application to connect to a database only as needed. It is important to note that as you increase the number of physical connections that take place within an application, you are increasing connection overhead for the application.

Connection overhead is the amount of time it takes an application session to connect application databases. Connection overhead varies depending upon the nature of the database connections. A connection to a database over a network generally takes longer than a connection to a local database. Within an application, an increase in the number of physical database connections increases connection overhead. When an application session establishes a connection to a database, the end-user waits.

When you use explicit CONNECT and DISCONNECT statements to access databases as needed in your application, you are increasing the number of physical database connections required by the application. You are also dispersing connection overhead throughout the application.

On systems where the number of connections is not an issue, you may want to minimize connection overhead in your application. The best way to minimize the number of connection operations is to connect all application databases only once--at session startup. Remember, databases connected at startup remain connected throughout the session, unless explicitly disconnected in the application. With this method, all connection overhead occurs at application startup.

13.6 MULTI-DATABASE PROGRAMMING TECHNIQUES AND CONSIDERATIONS

Once you've settled on a design and run-time connection method 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.

13.6.1 Referencing Files And Field Names In Multiple Database Applications

Unique file 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 file prefix to avoid ambiguous field references. Use the following syntax to reference a non-unique field in a procedure:

file-name.field-name

The *file-name* is the name of the database file containing the field *field-name*.

The ability to connect to several databases from a single PROGRESS session introduces the possibility of non-unique file names and increases the possibility of non-unique field names. References to non-unique file names within multi-database PROGRESS applications require database prefixes to avoid ambiguous file references. Ambiguous file and field references cause compilation failures. Use the following syntax to include a database prefix in a file or field reference:

database-name.file-name   *database-name.file-name.field-name*

The *database-name* is the logical name (or an alias for a logical name) representing the database containing the file *file-name*.

For example, suppose you are connected to two databases `db1` and `db2`, both of which contain a file called `customer`. The following procedure will not compile due to an ambiguous file reference:

```
FOR EACH customer: /* In db1 or db2 ? */
  DISPLAY name.
END.
```

The procedure below uses fully qualified file 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 file references.

NOTE: When PROGRESS encounters a statement such as "DISPLAY X.Y", it first attempts to process "X.Y" as a *databasename.file-name*. If that fails, PROGRESS then attempts to process "X.Y" as *file-name.field-name*.

13.6.2 Using Aliases

An *alias* is a synonym for a logical database name and is used within a PROGRESS procedure to reference a database. An alias is used as a database reference in general purpose PROGRESS procedures in place of a logical database name.

A logical database name is established when you connect a PROGRESS session to a physical database. An alias is created and assigned 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 general purpose procedure on other connected databases that have identical structure and time stamps for the files 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.

PROGRESS FAST TRACK illustrates the use of an alias. The first database connected that contains an _menu file automatically receives the alias FTDB. PROGRESS FAST TRACK uses the database with the FTDB alias or logical name to store and access FAST TRACK data files.

The PROGRESS Data Dictionary offers another 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. It is possible for the first database connected during a PROGRESS session to receive both the DICTDB and the FTDB aliases. 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: 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. The CREATE ALIAS statement has the following syntax:

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 logical name database must be connected unless NO-ERROR is used.

When you create an alias, the alias assignment is logged in a table in memory for the current PROGRESS 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, the same aliases can be used to reference that logical database name. If you create an alias that already exists in the session alias table, the existing alias is replaced by 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.

Creating Aliases In Applications. You cannot assign and reference an alias in the same procedure. An alias must be assigned 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 the alias `myalias` cannot be assigned and referenced in a single procedure.

```
/* alias1.p */
/* Note that this procedure does not work */
CREATE ALIAS myalias FOR DATABASE eastdb.
FOR EACH myalias.customer:
  DISPLAY customer.
END.
```

To solve this problem, split alias1.p into two procedures:

```
/* alias2.p */
CREATE ALIAS myalias FOR DATABASE eastdb.
RUN dispcust.p.
```

```
/* dispcust.p */
FOR EACH myalias.customer:     /* myalias.customer */
  DISPLAY customer.            /* myalias.customer */
END.
```

Compiling Procedures With Aliases. There is no difference in the representation of an alias and a logical database name in the .r code of a procedure. In the .r code, a database reference is a database reference. To precompile a procedure that uses an alias to reference a database, connect one of the databases on which the procedure is intended to run with a logical database name that is the same as the alias. When you compile the procedure, all unqualified file and field references resolve to the logical database name.

For example, suppose you have three databases called eastdb, centraldb, and westdb that contain customer files with identical structure and time stamps. Your application requires a general report procedure that can run against any of these customer files. 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 file and field references as shown in the previous procedure dispcust.p. All unqualified file and field references in the report procedure dispcust.p resolve to the myalias logical name at compilation time. When you are finished compiling your procedure, disconnect from the database represented by the myalias logical database name.

An alias must be assigned 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 shown previously.

In situations where you cannot precompile a procedure that uses an alias, it is important to understand how database references are placed in the .r code of the procedure at compilation time. Remember, the alias must be assigned to a logical database name prior to compiling the procedure. In general, use only the alias as a database prefix for all file and field references in the general purpose procedures, and always fully qualify every database reference within such a procedure. Use the following syntax:

alias.file-name *alias.file-name.field-name*

If you use this syntax for every file or field reference in your procedure, only the alias will be represented as the database reference in the procedure's .r code after compilation.

Unqualified file 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. The following code examples illustrate how logical database names are included within procedures as a result of unqualified database file and field references. All of these examples assume that a single physical database has been connected with the logical database name `demo1` and that the alias `myalias` has been assigned to `demo1`.

When the procedure below is compiled, only the alias `myalias` is included in the .r code. This is because the unqualified field reference `name` is contained within the same block as a qualified database file `customer` that contains the field.

```
/* alias3.p */
FOR EACH myalias.cust: /*  References myalias.customer */
   DISPLAY name.  /* References myalias.customer.name */
END.
```

For the following procedure, only the `myalias` alias is represented in the .r code. The unqualified file and field names reference a previously qualified file in the same block.

```
/* alias4.p */
FIND FIRST myalias.customer.  /* References myalias.customer */
FIND NEXT customer.  /* References myalias.customer */
DISPLAY name.   /* References myalias.name */
```

When the procedure below is compiled, both the alias `myalias` and the logical database name `demo1` are included in the .r code. This is because the file and field references in the second block are unqualified.

```
/* alias5.p */
FOR EACH myalias.customer: /* References myalias.customer */
   DISPLAY name. /* References myalias.customer.name */
END.
FOR EACH customer: /* References demo1.customer */
   DISPLAY name. /* References demo1.customer.name */
END.
```

For the following procedure, both the alias myalias and the logical database name demo1 are included in the .r code. This is because of the unqualified reference to the order file.

```
/* alias6.p */
FIND FIRST myalias.customer.   /* References myalias.customer */
FIND FIRST order. /* References demo1.order */
DISPLAY name.   /* References myalias.customer.name */
```

If the procedure below is compiled, the compilation will fail because both the alias myalias and the logical database name demo1 are referenced and the unqualified field reference name is ambiguous.

```
/* alias7.p */
FIND FIRST customer.   /* References demo1.customer */
FIND FIRST myalias.customer.   /* References myalias.customer */
DISPLAY name.  /* Ambiguous reference - two possible files */
```

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. For example:

```
/* main2.p */
CREATE ALIAS myalias FOR DATABASE demo1.
RUN makebuf.p.
```

```
/* makebuf.p */
DEFINE NEW SHARED BUFFER mybuf FOR myalias.customer.
CREATE ALIAS myalias FOR DATABASE demo2.
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 `demo1`. In `makebuf.p`, the shared buffer `mybuf` is defined for the file `myalias.customer`. Then, in the next line, `myalias` is changed, so that it now refers to database `demo2`. 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.`"

13.6.3 Using the Raw Datatype

The PROGRESS raw datatype allows you to retrieve and manipulate raw data from non-PROGRESS databases. You can only use the raw data type in memory, you cannot use it to define fields in the schema. PROGRESS does not perform any conversion on the data, you receive the bytes as supplied by the non-PROGRESS database.

You can use the raw data type for numerous reasons — anytime you need to manipulate data from non-PROGRESS databases without having PROGRESS give that data the characteristics of a certain data type. For example, you can use the raw datatype to bring over data that has no parallel PROGRESS datatype. 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 give you the means to define raw datatype 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.

rawdemo1.p

```
/*You must run this procedure against a non-PROGRESS
 demo 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.
END.
```


This procedure first creates the variable r1 and defines it as a raw data type. Next, it finds the first customer in the database and with the RAW function, takes the raw value of the field name and writes it into the variable r1. The PUTBYTE statement then puts the ASCII 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. Second Skin Scuba becomes second Skin Scuba. The next procedure shows how you can pull bytes from a field.

rawdemo2.p

```
/*You must run this procedure against a non-PROGRESS
 demo 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 = -1 THEN LEAVE.
      i = i + 1.
END.
```

This procedure finds the customer with the customer number 27, and then finds the ASCII value of each letter in the customer name. To do this, it retrieves the bytes from the name one by one and places them into the variable a. The GETBYTE statement returns a -1 if the byte number you try to retrieve is greater than the length of the expression you are retrieving it from. The next procedure demonstrates how you find the length of a raw value and how to change length of a raw expression.

rawdemo3.p

```
/*You must run this procedure against a non-PROGRESS
  demo 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 an `18` because the customer name `Second Skin Scuba` contains 18 bytes (the number of letters plus the null terminator). It then displays "`Se`" and a 3, because you truncated the name to two bytes and added a null terminator.

13.6.4 PROGRESS Functions For Database Connection Information

PROGRESS supplies a number of functions that allow you to test database connections and receive information about connected databases and the types of databases that the installed PROGRESS product can access. The following table lists these functions:

Table 13-5: PROGRESS Database Functions

PROGRESS Function	Description
CONNECTED	Tests whether a given database is connected.
DBTYPE	Returns the database type of a currently connected database ("PROGRESS", "RMS", "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".
DBVERSIONS	Returns a "5" if a connected database is a Version 5 database and a "6" if it is a Version 6 database.
GATEWAYS	Returns a character string containing a list of database types supported by the installed PROGRESS product. For example: "PROGRESS,ORACLE,RMS"
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.

Use these functions to perform various connection tasks in your application and report connection status for an application. For example, the following procedure displays a status report for all connected databases.

```
                                                    /* db-info.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.
```

For syntax descriptions and more information about the PROGRESS functions listed above, consult the *PROGRESS Language Reference.*

13.6.5 Using the LIKE option

The PROGRESS LIKE option in a DEFINE VARIABLE statement, DEFINE WORKFILE statement, or format phrase requires that a database be connected. Since you can start a PROGRESS session without connecting to a database, you should use the LIKE option with caution.

13.6.6 Understanding Transaction Behavior In Multiple Database Applications

A transaction is a set of changes for a database or several of databases, which should be done completely or not done at all. In a procedure, a transaction is one iteration of the outer most block that directly updates a database or reads with EXCLUSIVE-LOCK. Chapter 8 defines transactions and error handling and describes how to use and manipulate transaction blocks to ensure data integrity in your application. This section describes the behavior of transactions that update multiple databases and provides a few tips for transaction management.

During a transaction, PROGRESS writes data to the database(s) 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 database(s). A two-phase commit protocol is used to commit the changes to the databases. In the two-phase commit protocol, all the databases affected by the transaction are polled to see if they are reachable.

In the first phase of the two-phase commit, PROGRESS checks if 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, the transaction is backed out and the databases are returned to their pre-transaction states using the before-image files. If all of the databases are reachable and their validation checks succeeded, then the changes are committed to the databases.

NOTE: The two phase commit protocol does not work with ORACLE databases. If you have a transaction that updates several PROGRESS databases and an ORACLE database, PROGRESS polls and performs validations for only the PROGRESS databases in the first phase of the two-phase commit. In the second stage of the two-phase commit, changes are committed to the ORACLE database prior to committing changes to the PROGRESS databases. In this way, PROGRESS can catch any problem with the ORACLE database before committing changes to the PROGRESS databases.

If you want to test database connections prior to entering a transaction, use the CONNECTED function. For example:

```
/* runtxn.p */

IF CONNECTED("db1") AND CONNECTED("db2")
THEN RUN txnblk.p. /* transaction block */
ELSE MESSAGE "Unable to perform transaction".
```


All databases affected by a transaction should be connected 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 leave records in other databases affected by the transaction locked for considerable length of time. A database connection failure will also cause a transaction error. DISCONNECT statements in a transaction are deferred until the transaction completes or is undone.

For more information about transactions and error handling, see Chapter 8.

13.6.7 Providing Help For Multi-database Applications

In applications that access multiple databases, supplying context sensitive help can be a complex endeavor if the same file and field names occur in more than one of the application databases. The PROGRESS FRAME-DB function allows you to capture the logical database name for a current input field. This is useful when a form contains fields from several different databases. You can use the FRAME-DB function along with the FRAME-FILE, and FRAME-FIELD functions in an `applhelp.p` procedure to display help for a current input field.

The following procedure illustrates how FRAME-DB is used in an generic application help procedure to supply context-sensitive help for any input field on a form. When a user presses HELP (F2) while in any input field on a form, the following applhelp.p procedure runs:

```
/* applhelp.p */

CREATE ALIAS helpdb FOR DATABASE VALUE(SDBNAME(FRAME-DB)).
RUN dbhelp.p (FRAME-FILE, FRAME-FIELD).
RETURN.
```

The applhelp.p procedure creates an alias for the database captured from the current input field with the FRAME-DB function and then runs a help procedure (dbhelp.p). The SDBNAME function returns the name of the schema holder for a specified logical database name.

```
/* dbhelp.p */

DEFINE INPUT PARAMETER filname AS CHARACTER.
DEFINE INPUT PARAMETER fldname AS CHARACTER.

FORM helpdb._field._help SKIP(1)
  "The" helpdb._field._label "field requires"
  helpdb._field._data-type "data."SKIP
  helpdb._field._valmsg
  WITH FRAME help-frm NO-LABELS CENTERED OVERLAY.

FIND helpdb._db WHERE _db-name =
  (IF DBTYPE(FRAME-DB) = "PROGRESS" THEN ? ELSE FRAME-DB).
FIND helpdb._file OF _db WHERE _file-name = filname.
FIND helpdb._field OF helpdb._file WHERE _field-name = fldname.
DISPLAY helpdb._field._help helpdb._field._label
  helpdb._field._data-type helpdb._field._valmsg
  WITH FRAME help-frm.
```

The dbhelp.p procedure uses the FRAME-FILE and FRAME-FIELD values from the current input field as input parameters. This procedure uses the database alias established in the applhelp.p procedure to display field information from the proper database for the current input field. For more information on writing a help procedure, see Chapter 15 of the *PROGRESS Language Tutorial.*

13.6.8 Implementing Run-time Security For Multi-database Applications

This section presents information and recommendations for setting up a run-time security system for a multi-database application. You will learn about the following topics in this section:

- Database connections and userids.
- Activity permissions files and multi-database applications.

The information presented in this section builds upon security concepts presented in Chapter 11.

Understanding Database Connections and Userids. When attempting to develop a run-time security system for a multi-database application, it is important to understand the relationship of userids and application databases. Every time you connect an application session to a database, you are establishing a userid for that database connection.

The userid established for a database connection depends on the connection method and whether or not there are records in the _User file of the database. The following table explains the default userid assignments for run-time connections to PROGRESS databases:

Table 13-6: Default Userid Assignments For Run-time Connections

Database has entries in _User	Default Userid Assignment
No	The system userid. On DOS and OS/2 systems, the "blank" userid.
Yes	The "blank" userid.

NOTE: Some non-PROGRESS databases require the Userid (-U) and Password (-P) connection parameters.

Multi-database applications can use PROGRESS userid/password combinations for all database connections rather than relying on operating system userid/password combinations. Operating system userids may not always be transportable to other operating systems. This could present a problem for multi-database applications that access databases on several different operating systems in a distributed/simultaneous network scenario.

The _User file in the database must contain valid userid records in order to use PROGRESS userid/password combinations for the database connection. System administrators can enter userids into the _User file using the PROGRESS Data Dictionary. For information about entering userids into an _User file, consult Chapter 11.

There are several ways to assign a PROGRESS userid for a connection. The Userid (-U) and Password (-P) connection parameters allow you to assign a PROGRESS userid/password combination for a database connection from the command line or within a CONNECT statement. For example:

```
CONNECT db2 -1 -U userid -P password.
```

Another way to assign a PROGRESS userid to a database connection is the SETUSERID function. The SETUSERID function has the following syntax:

SYNTAX

```
SETUSERID (userid, password [ , logical-name ])
```

If the *userid* and *password* supplied to the SETUSERID function are in the `_User` file of the database represented by *logical-name*, SETUSERID returns TRUE and assigns the userid to the user. If the userid is not in the `_User` file or the password is incorrect, SETUSERID returns FALSE and does not assign the userid to the user. If you omit the *logical-name* argument, and more than one database is connected, a compiler error results.

The database must already be connected before you can use the SETUSERID function to change the userid/password combination for the database connection. For example:

```
CONNECT db2 -1.
IF NOT SETUSERID(userid,password,"db2") THEN
   DISCONNECT db2.
ELSE RUN procedure.
```

The example above connects to a database and then uses the SETUSERID function to assign a userid/password combination for the connection. If the userid/password combination specified with the SETUSERID function does not exist in the `_User` file of the `db2` database, the `db2` database is disconnected. If the userid/password does exist in the `_User` file, you can then run a procedure.

The PROGRESS `login.p` procedure uses the SETUSERID function to allow a user to interactively establish a userid/password combination for a database connected with the `DICTDB` alias or logical name. Remember, the first database connected during a PROGRESS session automatically receives the `DICTDB` alias. When you start a PROGRESS session with the default startup procedure (`prostart.p`), PROGRESS runs `login.p` for each PROGRESS database specified on the startup command line (assuming each database has entries in its `_User` file). To run the `login.p` procedure for a database connected later, reassign the `DICTDB` alias to that database before executing `login.p`.

Once you've connected to a database, you can use the USERID function to return the userid for that database connection. The USERID function has the following syntax:

SYNTAX

```
USERID [ ( logical-name ) ]
```

logical-name is the name of the database that is connected when a procedure is compiled, or when the CONNECT statement is executed. If you do not specify this argument, the compiler inserts the name of the database that is connected when the procedure is compiled. If you omit this argument and more than one database is connected, a compiler error results.

For more information about the SETUSERID and USERID functions, consult the *PROGRESS Language Reference.*

Using Activity Permissions Files. Just as with a single database application, you can use userids and an activities permission file to control end-user access to modules within a multi-database application. For multi-database applications, this requires a primary application database to be connected at application session startup and remain connected throughout the entire session.

If you are using PROGRESS userid/password combinations for your security system, your application should use a login procedure at startup to establish a valid PROGRESS userid/password for the primary application database. All PROGRESS userid/password combinations must be checked against the `_User` file in that database. Remember, the PROGRESS `login.p` procedure runs on any PROGRESS database that has `DICTDB` as an alias or logical name.

The primary application database should also contain the activity permissions file for your application. The activity permissions file controls who can run application modules by checking the current userid against a list of valid userids for an application procedure. For information about setting up an activity permissions file, see Chapter 11.

Prior to running a procedure in your application, use the PROGRESS CAN-DO function to check the userid assigned to the primary application database at login with a list of valid userids associated with the procedure in the activity permissions file. The CAN-DO function has the following syntax:

SYNTAX

```
CAN-DO (idlist,[userid])
```

idlist is an expression whose value is a list of one or more userids. If the expression contains multiple userids, you must separate the userids with commas only. *userid* is a character expression used to specify a userid to compare against the userid(s) in *idlist*. If there is more than one database connected, you must supply a logical database name for the *userid*. If you do not specify a logical database for a *userid* entry and there is more than one database connected, a compiler error results. The USERID function can be used for the *userid* argument to specify a current userid for a particular database connection to be checked against the userids listed in the *idlist* argument.

The following code fragment shows how you can use the userid established for one database connection and an activity permissions file to control end-user access to procedures and other database connections within your application. In this is example, `db1` is the primary application database containing an activity permissions file called `actperm`. The CAN-DO function checks the userid for `db1` against a list of valid userids (`can-run`) for a procedure (`db2proc.p`) listed in the activity permissions file.

```
DO FOR db1.actperm:
   FIND db1.actperm "db2proc.p" NO-LOCK.
   IF CAN-DO(db1.actperm.can-run,USERID("db1"))
   THEN DO:
      IF NOT CONNECTED("db2")
      THEN DO:
         MESSAGE "Connecting db2".
         CONNECT db2 -1 NO-ERROR.
      END.
      IF CONNECTED("db2")
      THEN RUN db2proc2.p.
      ELSE MESSAGE "db2proc.p not available - db2 not connected".
   END.
   ELSE MESSAGE "You are not authorized to run this procedure".
END.
```

In the example above, the `db2` database is connected without a userid/password designation. If the `_User` file in `db2` has userid entries, the `db2` database connection receives the "blank" userid. Running an application procedure against a database connected with the "blank" userid is not recommended. Your application may perform a run-time compile and require a userid/password combination with compile privileges for a particular database.

Any run-time security system that you develop for a multi-database application must be able to establish and coordinate userid/password combinations for all of the application's databases. There is no guarantee that _User files in each of the application databases has identical userid/password entries. To get around this problem, use a single *connect userid* within your application for database connections. The userid used to login to the primary application database is still used with the activity permissions file to govern end-user access application procedures. However, the connect userid is assigned to each database connection within the application. This security scheme requires entering the connect userid into the _User file of each database accessed by the application and coding the connect userid into the application or a parameter file. For example:

```
CONNECT db2 -1 -U conuid -P conpwd.
CONNECT db3 -1 -U conuid -P conpwd.
```

NOTE: You should disguise all userid and password strings in your code using a unique encryption method. For example, use PROGRESS functions to build your userid and password strings.

In the example above, both the databases are connected using the same connect userid/password combination. This userid/password combination should exist in the _User file of all application databases and has the database privileges required to run the application.

13.7 MULTI-DATABASE PROGRAMMING CASE STUDIES

The following sections contain four multi-database programming case studies. The case studies introduce you to some of the issues involved in programming multi-database applications and also illustrate how to use certain PROGRESS language statements and functions that support multi-database programming.

Each of the four case studies contains a main procedure and one or more subprocedures called from the main procedure. The names of the main procedures and subprocedures are listed in the following table:

Table 13–7: Case Study Procedure Names

Case Study	Main Procedure	Subprocedure(s)
1	case1.p	qucas1.p
2	case2.p	ecnct.p ccnct.p wcnct.p eastrn.p centrl.p westrn.p
3	case3.p	regrpt.p
4	case4.p	regupd.p updsub.p

The procedures shown above are available in the PROGRESS Procedure Library. The Preface contains instructions on extracting the procedures so you can run them.

In order to run the procedures, you also need to create three copies (db1, db2, and db3) of the PROGRESS demo database in the same directory where you placed the procedures.

To run a case study, start a PROGRESS session with no databases connected using the PRO or PROGRESS command.

Then run the main procedure for the case study (see procedure table above) from within the PROGRESS session. For example, to run Case Study 1, execute the following statement from the PROGRESS editor:

```
RUN case1.p.
```

NOTE: Because the case study databases db1, db2, and db3 are identical, the case studies are technically not true to life; you are never going to see a real world situation with three databases that have both identical structure and identical data. Also, in many cases the databases in a multi–database application do not reside in the same directory on the same machine, as do the case study databases. However, the case studies are designed so that these differences do not effect the databases' usefulness in illustrating various important multi–database programming principles.

13.7.1 Case Study 1

The objective of multi-database programming is to access more than one database from the same PROGRESS session. Case Study 1 shows a simple example of multi-database access: a join between two different databases.

The programming problem is to write a report that displays the salespeople of the eastern region, one by one, and lists any customer accounts in the western region held by the current eastern region salesperson.

Case Study 1 illustrates the following multi-database programming language elements:

- CONNECT statement
- Parameter file
- The Logical Name (-ld) parameter
- DISCONNECT statement
- LDBNAME function
- NUM-DBS function

Below are the two procedures, `case1.p` and `qucas1.p`, and the parameter file, `case1.pf`, which comprise Case Study 1:

case1.p

```
CONNECT -pf case1.pf.

RUN qucas1.p.

DO WHILE NUM-DBS > 0:
  DISCONNECT VALUE(LDBNAME(1)).
END.
```

case1.pf

```
-db db1 -ld eastern -1
-db db2 -ld central -1
-db db3 -ld western -1
```

qucasl.p

```
FOR EACH eastern.salesrep:
  DISPLAY slsname LABEL "SALESPERSON".
  FOR EACH western.customer OF eastern.salesrep:
    DISPLAY name LABEL "CUSTOMER" WITH FRAME new-frame 10 DOWN.
  END.
END.
```

When case1.p is run, a connection attempt is made using the connection parameters in case1.pf. The successful result of the connection attempt is that databases db1, db2, and db3 are connected with logical names eastern, central, and western, respectively, in single-user mode.

After the connection, the procedure qucasl.p is called. This procedure contains two FOR EACH statements, one nested within the other. The outer FOR EACH statement cycles through eastern region salespeople, while the inner FOR EACH statement,

```
FOR EACH western.customer OF eastern.salesrep:
```

performs a join between the eastern and western databases (db1 and db3).

After procedure qucasl.p completes, execution returns to case1.p, and the following DO WHILE loop disconnects all currently connected databases:

```
DO WHILE NUM-DBS > 0:
  DISCONNECT VALUE(LDBNAME(1)).
END.
```

This WHILE loop is a good way to ensure that no databases are connected.

Some comments on this case study:

- In general, all databases referenced in a procedure must be connected prior to the start of the procedure. That is why the code in qucasl.p (which references databases db1 and db3) is contained in a subprocedure called from case1.p (where the database connections take place).

- The file references in qucasl.p are qualified with the logical names of the appropriate databases (eastern for db1 and western for db3).

- The case study does not trap possible connection errors. Case studies 2 and 4 show how to trap connection errors.

13.7.2 Case Study 2

One possible use of PROGRESS's multi-programming features is to bring together a set of unrelated applications under a common menu-driven access system. Case Study 2 shows such a system.

The programming problem for Case Study 2 is to set up a menu that allows you to access the Eastern, Central, or Western application and its corresponding database.

Case Study 2 illustrates the following additional multi-database programming language elements:

- CONNECTED function.
- CONNECT statement with NO-ERROR option.

Seven procedures comprise Case Study 2: the main procedure `case2.p`, and three subprocedure pairs: `ecnct.p` and `eastrn.p`, `ccnct.p` and `centrl.p`, and `ecnct.p` and `eastrn.p`. The subprocedure pairs all work the same way: each accesses its own application and database. Here are the listings for the main procedure `case2.p`, and the subprocedure pair `ecnct.p` and `eastrn.p` (listings of the other two pairs are not provided because of their similarity to `ecnct.p` and `eastrn.p`):

case2.p

```
DEFINE VARIABLE menu AS CHARACTER EXTENT 4 FORMAT "x(20)"
      INITIAL
      ["Eastern Region",
       "Central Region",
       "Western Region",
       "Quit"].

DO WHILE TRUE:

  DISPLAY menu WITH NO-LABELS ROW 10 ATTR-SPACE
      FRAME f-menu CENTERED 1 COLUMN
      TITLE "MENU".
  CHOOSE FIELD menu AUTO-RETURN WITH FRAME choices.
  IF FRAME-INDEX = 1 THEN RUN ecnct.p.
  IF FRAME-INDEX = 2 THEN RUN ccnct.p.
  IF FRAME-INDEX = 3 THEN RUN wcnct.p.
  IF FRAME-INDEX = 4 THEN DO:
     DISPLAY "Leaving Application" WITH CENTERED.
     PAUSE 1 NO-MESSAGE.
     LEAVE.
  END.
  HIDE ALL.

END.
```

ecnct.p

```
IF NOT CONNECTED("eastern") THEN DO:
  MESSAGE "Connecting to db1...".
  CONNECT db1 -1 -ld eastern NO-ERROR.
END.

IF CONNECTED("eastern") THEN DO:
  RUN eastrn.p.
  DISCONNECT eastern.
END.
ELSE
  DISPLAY "Connect to database db1 failed." WITH CENTERED.
```

eastrn.p

```
DISPLAY " You have reached eastrn.p " WITH CENTERED.
```

The following diagram shows how Case Study 2 is implemented:

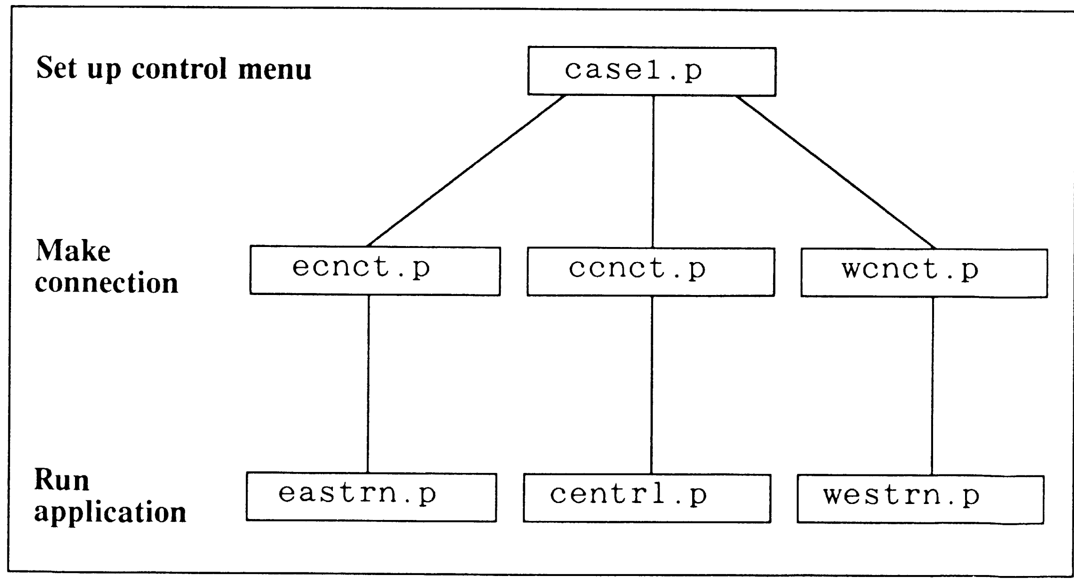


Figure 13–9: Case Study 2 Flow of Execution

First, main procedure case1.p runs. This procedure sets up a menu that looks like the following:

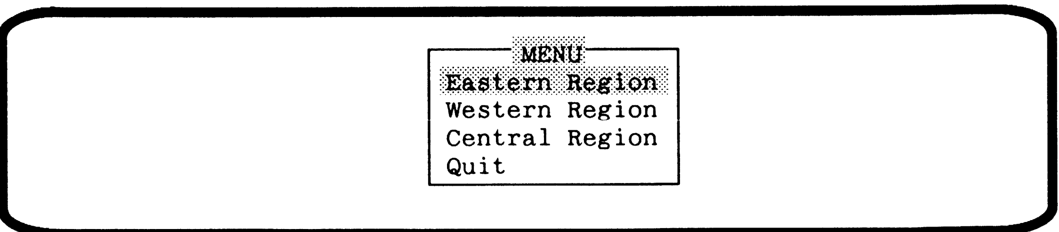


Figure 13–10: Main Menu

Suppose the user then selects Eastern Region from this menu. This causes subprocedure ecnct.p to run. Using the CONNECTED function, ecnct.p checks whether there is a currently connected database with logical name eastern. If not, then an attempt is made to connect to database db1, with logical name eastern. The NO-ERROR option stops a run-time error condition from occurring if the connection fails.

The CONNECTED function is then used again. This is necessary to determine if the CONNECT was successful (although redundant in the case where eastern was already connected going into ecnct.p). If eastern is connected, then the application eastrn.p is run. In this case, the application eastrn.p contains only a simple DISPLAY statement.

When eastrn.p is exited, database eastern (or db1, to use its physical name) is disconnected.

Some comments on this case study:

- The code in subprocedures ecnct.p, ccnct.p, and wcnct.p is split out from case2.p for better modularity. The code could be included in case2.p with no change in program operation.

- Although ecnct.p checks for the presence of a connected database with logical name eastern, it does not check the physical name of this database. In some situations, this could cause access to the wrong physical database. The following augmented version of ecnct.p uses the PDBNAME function to check the physical name of the connected database with logical name eastern:

```
IF CONNECTED ("eastern") THEN
  IF PDBNAME("eastern") <> "dbl" THEN
    DISCONNECT eastern.

IF NOT CONNECTED ("eastern") THEN DO:
  MESSAGE "Connecting to dbl...".
  CONNECT dbl -1 -ld eastern NO-ERROR.
END.

IF CONNECTED("eastern") THEN DO:
  RUN eastrn.p.
  DISCONNECT eastern.
END.
ELSE
  DISPLAY "Connect to database dbl failed." WITH CENTERED.
```

Code that checks the physical name of the database with logical name "eastern", if present.

In the above code, if there is a connected database with logical name eastern, but with physical name other than dbl, the database is disconnected. The remaining code is identical to ecnct.p, discussed above.

13.7.3 Case Study 3

Case Study 3 shows how to use the Logical Name (-ld) connection option to run the same procedure against a series of databases. The only additional multi-database programming language element that Case Study 3 uses is the PDBNAME function.

The case study procedures, case3.p and regrpt.p, are shown below:

case3.p

```
CONNECT db1 -1 -ld region NO-ERROR.
IF CONNECTED("region") THEN DO:
  DISPLAY PDBNAME("region") @ database-name AS CHAR
    WITH FRAME pdb-disp.
  RUN regrpt.p.
  DISCONNECT region.
END.

CONNECT db2 -1 -ld region NO-ERROR.
IF CONNECTED("region") THEN DO:
  DISPLAY PDBNAME("region") @ database-name
    WITH FRAME pdb-disp.
  RUN regrpt.p.
  DISCONNECT region.
END.

CONNECT db3 -1 -ld region NO-ERROR.
IF CONNECTED("region") THEN DO:
  DISPLAY PDBNAME("region") @ database-name
    WITH FRAME pdb-disp.
  RUN regrpt.p.
  DISCONNECT region.
END.
```

regrpt.p

```
FOR EACH region.customer:
  DISPLAY region.customer.name
          region.customer.address.
END.
```

As you can see, procedure case3.p connects to and disconnects from databases db1, db2, and db3 in turn, each time running the report procedure regrpt.p.

Some comments on this case study:

- The connections in case3.p are made without checking to see if the databases concerned are previously connected. This type of "naked connect" is appropriate only in cases where you are certain there are no previously connected databases that could cause a conflict.
- One motivation for using the Logical Name (-ld) connection option to run the same procedure against a series of databases is to reduce the size of your code; in Case Study 3,

only one procedure (`regrpt.p`) is used instead of three. Also, it is easier to support a single procedure than to support a separate one for each database.

In order to run the `regrpt.p` on each of the three databases, you had to connect and disconnect each database. The following case study shows how you can use aliases to run a single procedure on different connected databases.

13.7.4 Case Study 4

Case Study 4 shows the following multiple database programming concepts:

- The recommended way to code a transaction involving multiple databases.
- The use of arrays to store connect options (reduces code size and promotes maintainability).
- How to use the aliases with the CREATE ALIAS and DELETE ALIAS statements.

The case study procedures case4.p, regupd.p, and updsub.p are shown below:

case4.p

```
DEFINE VARIABLE physical-name AS CHAR EXTENT 3
  INITIAL ["db1", "db2", "db3"].
DEFINE NEW SHARED VARIABLE logical-name AS CHAR EXTENT 3
  INITIAL ["eastern", "central", "western"].
DEFINE VARIABLE options AS CHAR EXTENT 3
  INITIAL ["-1", "-1", "-1"].
DEFINE VARIABLE i AS INTEGER.

DO i = 1 TO 3:
  IF NOT CONNECTED(logical-name[i]) THEN DO:
    MESSAGE "Connecting to " logical-name[i].
    CONNECT VALUE(physical-name[i])
            -ld VALUE(logical-name[i])
            VALUE(options[i])
            NO-ERROR.
  END.
END.

IF CONNECTED(logical-name[1]) AND
   CONNECTED(logical-name[2]) AND
   CONNECTED(logical-name[3])
   THEN DO:
      RUN regupd.p.
      DISPLAY "Transaction successfully completed."
        WITH FRAME new-frame.
END.
ELSE
  DISPLAY "Unable to make necessary connections.".
```

regupd.p

```
DEFINE SHARED VARIABLE logical-name AS CHAR EXTENT 3.
DEFINE VARIABLE i AS INTEGER.

DO TRANSACTION:
  DO i = 1 TO 3:
    CREATE ALIAS region FOR DATABASE VALUE(logical-name[i]).
    DISPLAY "Updating database" logical-name[i].
    RUN updsub.p.
  END.
END.
```

updsub.p

```
FOR EACH region.customer:
  region.customer.max-credit = region.customer.max-credit * 1.1.
END.
```

The goal in Case Study 4 is to perform a transaction which updates each of the three databases db1, db2, and db3. The scope of the transaction should include all three of the updates. For example, if the updates to databases db1 and db2 are successful, but then the transaction fails for some reason before database db3 is updated, then databases db1 and db2 should be rolled back to the state they were in before the transaction started.

As you can see, case4.p starts off by defining connect option variable to be used subsequently. Note that although the options variable contains only "-1" in this example, it could contain a longer series of options, and even the name of a parameter file.

The next section of code in case4.p checks database connections and is very similar to the code which accomplishes the same task in ecnct.p (shown above in Case Study 2). If any of the three databases is not connected, a connect attempt is made.

Next, the connections to the three databases are tested again. This is necessary because a connect attempted in the previous section of code might have failed. If the three databases are all connected, then procedure regupd.p is run.

Procedure regupd.p contains a transaction block defined by a DO TRANSACTION statement. The CREATE ALIAS and DELETE ALIAS statements make it possible for the same procedure, updsub.p, to run against three simultaneously connected databases.

Procedure updsub.p is where the update is actually performed. The update shown adds an "x" to the address field of the customer file in each of the three databases.

After running case4.p, you can view the results by running case3.p.

Some comments on this case study:

- The code in subprocedure regupd.p is split out from case4.p for better modularity. The code could be included in case4.p with no change in program operation.

- In general, if a procedure uses an alias to reference a database, that alias must have been created before the procedure is run. That is why the code in updsub.p (which references databases db1, db2, and db3 using alias region) is contained in a subprocedure called from regupd.p (where alias creation takes place).

Chapter 14
PROGRESS Programming Tips

This chapter tells you how to design your PROGRESS applications so that are portable across the various operating systems that PROGRESS runs on. The chapter covers the following topics:

- Differences between operating systems running PROGRESS.
- Writing transportable applications.
- Writing UNIX transportable applications.
- Writing VMS transportable applications.
- Transporting a database.
- Testing and debugging.
- Accessing the PROGRESS procedure library.

14.1 DIFFERENCES BETWEEN OPERATING SYSTEMS RUNNING PROGRESS

The major differences between the DOS, UNIX, BTOS/CTOS, and VMS versions of PROGRESS are the following:

- To use the operating system commands from within PROGRESS, you use the DOS statement when on a DOS system, the OS/2 statement when on an OS/2 system, the UNIX statement when on a UNIX system, and the VMS statement when on a VMS system. If you include the operating system statement in a procedure on a system other than the one named, the procedure will compile but will not run if the flow of control passes through that operating system statement. PROGRESS raises the error condition when it tries to process a DOS, OS/2, UNIX, or VMS statement on a system other than the one named.
- A procedure containing the INPUT THROUGH, OUTPUT THROUGH, or INPUT-OUTPUT THROUGH statements will compile on a DOS, OS/2 or BTOS/CTOS system but will not run on that system if flow of control passes through the

statement. PROGRESS raises an error when trying to execute those statements on a DOS system.

- If you are using UNIX or VMS, your terminal can be either spacetaking or nonspacetaking. If you are using DOS, OS/2 or BTOS/CTOS, your terminal is automatically nonspacetaking.

- On UNIX and VMS systems, the PROGRESS function USERID returns the UNIX or VMS userid of the user executing the function. The USERID function works the same way on DOS and OS/2 systems where security has been implemented. If security has not been implemented on a DOS or OS/2 system, USERID returns the value of "".

- ON BTOS/CTOS systems, the userid returns the username.

- PROGRESS security is based on the concept of a userid. If you are using UNIX, BTOS/CTOS, or VMS (both of which assign a userid to each user), you can use PROGRESS security checking in your application. Although DOS and OS/2 do not assign ids to users, and you cannot automatically use PROGRESS file and field security checking in applications on DOS and OS/2 systems, you can use PROGRESS security with DOS and OS/2 by prompting users for a userid and password as part of the login process. See Chapter 11 for information on providing application security for all types of applications.

- Under DOS and OS/2, the number of lines on the terminal is 25. Under UNIX and VMS, the number of lines varies depending on the terminal model, but is usually 24. Under BTOS/CTOS, the number of lines varies with the terminal, but it is usually 29. Procedures that rely on 25 lines on the screen (e.g. 22 line frames) will not operate on 24 line terminals.

- Under DOS and OS/2, the Data Dictionary Dump/Reload routines will not process database files where the first eight characters of the name are the same as another file in the same database.

- DOS, OS/2, and VMS file names are not case sensitive; UNIX file names are case sensitive. Under UNIX, files and commands spelled with uppercase letters are different from those spelled with lowercase letters.

- On DOS and OS/2 systems, you can use the tilde (~) as an escape character. On UNIX and VMS systems, you can use either the backslash (\) or the tilde as an escape character. If you are planning to move from one operating system to the other, it is a good idea to use the tilde as an escape character in your application.

- You can use UNIX directory and file specifications in PROGRESS procedures on VMS systems. However, you cannot use VMS directory or file specifications in PROGRESS procedures on DOS, OS/2 or UNIX systems.

- You can use VMS directory and file specifications with the PROGRESS VMS statement.

14.2 WRITING TRANSPORTABLE APPLICATIONS

If you are developing applications on a DOS system that will be used on another operating system, or or developing applications on another operating system that will be used on a DOS system, keep these guidelines in mind:

- Avoid using the OUTPUT THROUGH statement. In place of OUTPUT THROUGH, you can use the OUTPUT TO statement together with an escape to the operating system.

  For example:

```
OUTPUT TO file1.
      .
      .
      .
OUTPUT CLOSE.
IF OPSYS = "UNIX"
  THEN UNIX program-name < file1.
ELSE IF OPSYS = "VMS"
  THEN VMS program-name/INPUT=file1.
ELSE IF OPSYS = "OS2"
  THEN OS2 program-name < file1.
ELSE IF OPSYS = "BTOS"
  THEN BTOS [volumne]<directory> programn-name.
ELSE IF OPSYS = "DOS"
  THEN DOS program-name < file1.
ELSE
   MESSAGE "Operating system not supported".
```

  First, you output the data to a file (`file1` in this example). Then you use either the appropriate operating system statement to escape to the operating system. Once at the operating system level, you run the program against the data in `file1`.

  Most BTOS/CTOS commands do not support redirection of output, therefore, to send the output of a command to a file you need to know the specific command parameters.

  Alternatively, you could use OUTPUT THROUGH when OPSYS is UNIX or VMS and OUTPUT TO otherwise.

- Avoid using the INPUT THROUGH statement. In place of INPUT THROUGH, you can use an escape to the operating system together with the INPUT FROM statement. For example:

```
IF OPSYS = "UNIX"
THEN UNIX program-name > file1.
ELSE IF OPSYS = "VMS"
  THEN VMS program-name/OUTPUT=file1.
ELSE DOS program-name > file1.
INPUT FROM file1.
      .
      .
      .
```

  First, you use either the operating system statement to escape to the operating system. Once at the operating system level, you run a program to create the data in `file1`. Then you use the INPUT FROM statement to retrieve the data in `file1`.

  Alternatively, you could use INPUT THROUGH when OPSYS is UNIX and VMS.

- If you plan to use either the operating system statement in your procedures to escape to the operating system, be sure to use the OPSYS function in those procedures. That way, a procedure can test to see which operating system it is running under and use the appropriate operating system statement to escape to that system.

- Use file names that have a maximum of 8 characters. Under DOS and OS/2, if your PROGRESS file names are more than 8 characters and if the first 8 characters of some of the file names are the same, then you cannot use the standard Data Dictionary Dump/Reload procedures. Those procedures dump data to an ASCII file whose name is the file name plus ".d". Since the maximum file name in DOS and OS/2 is 8 characters, these dump files would overwrite one another. The Dictionary will print an error message in this case.

- If you plan to precompile procedures, avoid using procedure names that are longer than 8 characters or that contain periods. A filename extension of .p is allowed and recommended.

- Use forward slashes (/) as separators in specifying file paths (when using the RUN, INPUT FROM, or OUTPUT TO statements). Both / and \ are acceptable in DOS, OS/2, and BTOS/CTOS but only / is allowed in UNIX. For VMS, use path specifiers.

- Use lowercase when specifying a procedure name in a RUN statement and be sure your procedure files have lowercase names in UNIX.

- Design your use of the terminal based on a 24 line screen. Remember that the PC screen is always considered to have 25 lines and you must exercise care to ensure that your procedures do not depend on that extra screen line.

- If you are writing a multi-user application on a DOS machine, be sure to test that application in multi-user mode on a UNIX or VMS machine, or on a LAN.

- There are two different ways a terminal can handle screen formatting. It can either:

  - Reserve a character position, on both sides of every field, for special screen field attributes, such as underlining or highlighting. These are called "spacetaking" terminals.

  - Not reserve a character position for special field attributes. These are called "nonspacetaking" terminals. All PC terminals are nonspacetaking.

  Before designing your frames, you should know whether your procedures will be running on spacetaking, nonspacetaking, or both kinds of terminals. Frames you design to run on spacetaking terminals will run on all terminals. However, if you assume that your procedures will always run on nonspacetaking terminals, frames you design for those procedures may not fit or operate properly on spacetaking terminals.

14.3 WRITING UNIX TRANSPORTABLE APPLICATIONS

If you are developing applications on one UNIX system that will be used on other UNIX systems, keep these guidelines in mind:

- Avoid using UNIX commands that are not generally available and compatible on all UNIX systems.

- Although the PROGRESS source code is fully transportable from one UNIX system to another, the PROGRESS object code produced on one brand of machine is not compatible with another machine of a different brand. However, code produced on a machine based on the Motorola 68000 CPU chip may be compatible with other 68000-based machines.

- The size of the object code produced on different machines differs. If you have a very large procedure whose object code is at or very near the limit of 63K, then that procedure may not compile on another machine. This is caused by compiler and hardware alignment differences. For example, .r code produced on an AT&T 3B series or Pyramid machine is usually 10 percent larger than that produced on 68000-based machines.

14.4 WRITING VMS TRANSPORTABLE APPLICATIONS

If you are developing applications on one VMS system that will be used on other VMS systems, keep these guidelines in mind:

- VMS works with many layered products (such as ALL-IN-ONE, DECNET, and VAX Information Architecture products). Avoid referencing layered products that your users may not have.
- VMS has byte ordering and alignment requirements similar to those for DOS. Therefore, .r files you generate on a PC may run on VMS. However, .r code that you generate on 68000 machines will not run on VMS. Alignment requirements for the Ultrix and VMS C compilers also differ, so you cannot use .r code that you generate under Ultrix on VMS, and vice versa.

14.5 TRANSPORTING A DATABASE

After you dump your database, the database consists of two parts:

- One or more .df files which contain the data definitions from your database.
- One .d file for each file in the database. These files contain the actual data from your database.

To move your database from one machine to another machine, you must move the .df and .d files from your source machine to your target machine. The procedure is the same whether you are transporting a database between machines that have the same operating system or between machines that have different operating systems.

To move your PROGRESS database from one machine to another, follow these steps:

1. Dump your data definitions (.df files).

   Start PROGRESS and get into the Data Dictionary on your source machine. Select Load/Dump Data Files from the Dictionary main menu. Select Dump Data Definitions from the Load/Dump Data menu. PROGRESS prompts you to select either a file from the list of definition files in your database to dump, or to select ALL to dump all the files in the database. Then PROGRESS prompts you to name the file where you want to store the data definitions. When you press return, PROGRESS dumps your data definitions. When it is finished, PROGRESS displays a message that tells you where the data definitions are stored and returns you to the Load/Dump Data menu.

2. Dump your data (.d files).

   Select Dump File Contents from the Load/Dump Data Files menu. Again PROGRESS prompts you to select either a file from the list of files in the database, or to select ALL to

dump all the file contents in the database. PROGRESS dumps the data for each file into a file with the .d extension. When it is finished, PROGRESS returns you to the Load/Dump Data menu. Exit from the menu and from PROGRESS.

3. Move your .df and .d files to the target system.

   Use diskettes or tape to copy the .df and .d files from your source system to your target system. This process works when the two systems have compatible media devices. However, different vendors often do not have compatible devices.

   You may need a communications link through a program such as KERMIT or CROSSTALK to move files between machines which are not compatible. On many systems, the PROGRESS Developer's Toolkit includes KERMIT (free of charge) as a convenience to you. See the *Developer's Toolkit Manual* for complete directions on transporting applications between incompatible machines.

4. Make a copy of the empty database on your target machine and start PROGRESS.

   Make a copy of the empty database on the target machine. For example, if newdb is the name of the database on the source machine, first create a database named newdb. Then start PROGRESS against the new database by typing:

Operating System	To Copy The Empty Database
UNIX, DOS & OS/2	`prodb newdb empty` `pro empty`
VMS	`PROGRESS/CREATE NEWDB empty` `PROGRESS NEWDB`

BTOS/CTOS	`PROGRESS Create Database`
	`New Database Name` newdb
	`Copy From Database Name` empty
	
	`PROGRESS 4GL`
	`[Options] -1` newdb

5. Load your data definitions (.df files).

   Select Load/Dump Data Files from the Data Dictionary main menu. Then select Load Data Definitions. PROGRESS prompts you to supply the name of the file that contains

the data definitions. You see a message for each file that PROGRESS loads. When PROGRESS is through, you see the message Load Data Definitions Complete.

6. Load your data (.d files).

   Select Load File Contents from the Load/Dump menu. PROGRESS prompts you to either select a file from the list of available files or to load all the files. Then you see the following message:

```
PROGRESS Data Dictionary                          File Contents (.d files)
MODIFY-SCHEMA  SQL  Database  Admin  Utilities  Reports  Exit

 ---------------- Load Data Contents for All Files ----------------
   Read from directory:
 Acceptable error rate:  10

  As PROGRESS loads records it may encounter load files with malformed
  or corrupted records. In such a case, PROGRESS does not load that
  record, generates an error message, and continues on to the next
  record. Please specify an acceptable percentage of unreadable
  records. If this percentage is exceeded the data load for that file
  will be stopped.
  Enter 0 if any error should stop the load.
  Enter 100 if the load should not stop for any error.

Database: newdb                       File: ALL

Enter data or press F4 to end.
```

Follow the directions in the message. PROGRESS uses 10% as the default rate of acceptable error and no as the default for whether you are running with the No Crash Protection (-i) option. Then you see this screen:

```
PROGRESS Data Dictionary                                  File Contents (.d files)
MODIFY-SCHEMA  SQL  Database  Admin  Utilities  Reports  Exit

                                                 Records  Total      Expected
Database file  Opsys load file                   Loaded   Errors     Records

agedar         ./agedar.d                         26       0          26
customer       ./customer.d                       33       0          33
item           ./item.d                           55       0          55
                                 .
                                 .
                                 .

Database: newdb (PROGRESS)                                 File: ALL
Load of database contents completed
Errors listed in .e files placed into same directory as .d files
Press space bar to continue
```

Exit from the Load/Dump Data menu and use PROGRESS and your database on the new machine.

It is a good idea to use the No Crash Protection (–i) startup option (/NORECOVERY in VMS) when you load a large database because this option helps PROGRESS to run efficiently. If a load fails while it is running with the –i option, you must restart the load at the beginning. See Chapter 3 in *System Administration II: General* for more information about this startup option.

14.6 TESTING AND DEBUGGING

There are several parts of the PROGRESS language that you can use specifically for testing and debugging your application procedures:

- Use PAUSE, MESSAGE, MESSAGE SET and MESSAGE UPDATE to help debug procedures.
- Use comments (/* */) to temporarily bypass certain procedure lines.

- Use PROGRESS to automatically generate large databases. For example:

p-test.p

```
DEFINE VARIABLE ctr AS INTEGER.

REPEAT ctr = 1 TO 999:
  CREATE customer.
  cust-num = ctr.
  name = "custname" + string(ctr,"999").
END.
```

This procedure creates 999 customer records. Each customer record has a different customer number and a name that is a combination of the string "custname" and the customer number. Because PROGRESS stores data in variable-length records, be sure to put reasonable data values into the database. Otherwise the size of each record will be unrealistically small.

- Use the RANDOM function to help build databases. For example, you might want to create between 1 and 10 orders for each customer and between 1 and 5 lines per order.
- Use the PAUSE *n* statement to simulate terminal input delays when testing multi-user response using input from a file or data generated within a procedure.
- Use the PAUSE 0 BEFORE-HIDE statement to run large outputs on-line without pauses.

14.7 ACCESSING THE PROGRESS PROCEDURE LIBRARY

As part of your PROGRESS software, you receive the PROGRESS Procedure Library, which is actually several libraries of procedures that you can use to enhance your applications. The Procedure Library includes routines for creating menus, scrolling, performing mathematical and statistical functions, and pop-up procedures that include calculators and a perpetual calendar — to name just a few.

Many of the procedures are ready to use. That is, they can be run against any database without modification of the database or the procedures. Other procedures in the library are include files or procedures that need to be customized for your application. The number procedures in this directory may change with every release.

If you plan to use the Procedure Library, you must use the Local Buffer Size (-l) startup option to increase the local buffer size when you start PROGRESS. For example:

```
pro database-name -l 30
```

You access the PROGRESS Procedure Library from the PROGRESS Help menu, as follows:

1. Press the HELP (F2) key to display the PROGRESS Help menu.
2. From the Help menu, select option **p**, Access the Procedure Library. The Main Menu of the PROGRESS Procedure Library appears on the screen. It looks similar to this:

```
                P R O G R E S S Library Utility

                          Main Menu

          1. Extract Procedure(s)
          2. Search Procedure(s)
          3. Test Bench
          4. Add/Update Libraries
          5. Delete Procedure(s) from Library
          6. Change .lp storage directory
          h. Help
          r. Run a PROGRESS program
          u. Unpack Utility (unpack.p)
          x. Exit
```

3. From the Library Main Menu, select option **h**, Help, for an overview of the Library contents and instructions on how to use the Library. There is also a tutorial selection that gives you a description of all the menu options.

From the Main Menu of the PROGRESS Procedure Library, you can also access the procedures used in this book, the *PROGRESS Language Tutorial*, and the *PROGRESS Language Reference* by choosing option **u**, Unpack Utility.

Chapter 15 PROGRESS/SQL

This chapter describes PROGRESS/Structured Query Language (SQL). PROGRESS/SQL is a relational database language based on the SQL86 database access standard of the American National Standards Institute (ANSI). It meets the Level 1 requirements of the standard and supports a majority of Level 2 features. In addition, PROGRESS/SQL supports the use of many PROGRESS features. To understand the material covered in this chapter, you should already be familiar with the basic concepts and use of PROGRESS.

PROGRESS/SQL supports the following features:

- Extensions to the SQL86 standard, including ALTER TABLE, DROP TABLE, DROP VIEW, CREATE INDEX, DROP INDEX, and REVOKE statements.
- Database creation using the PROGRESS Data Dictionary, PROGRESS/SQL data definition statements, or both.
- PROGRESS 4GL (Fourth-Generation Language) features within PROGRESS/SQL statements, such as the PROGRESS DATE and LOGICAL data types and PROGRESS format and frame phrases.

Included in this chapter are descriptions of:

- PROGRESS/SQL Language Components.
- SQL Data Manipulation Language.
- SQL Data Definition Language.

Because there are several PROGRESS features that you can use in SQL, this chapter contains references to other PROGRESS documentation. While using the other manuals, it is important to note that SQL and PROGRESS use different terms to refer to the same database objects. Table 15-1 lists SQL terms and their equivalent PROGRESS terms.

Table 15-1: PROGRESS/SQL and PROGRESS Terms

PROGRESS/SQL Term	PROGRESS Term
table	file
column	field
row	record

The examples in this chapter are based on the PROGRESS demonstration database. Appendix A in the *PROGRESS Language Tutorial* explains the files, fields, and indexes of the demonstration database.

15.1 The PROGRESS/SQL Language

To define the structure of a database, you can use the PROGRESS/SQL Data Definition Language (DDL), the PROGRESS Data Dictionary, or both. A single table or file must be defined with a single language (PROGRESS/SQL or the Data Dictionary).

Modifications to the structure of an existing table or file must be made using the same language with which it was created. This means that if you created a table with PROGRESS/SQL data definition statements, you must use PROGRESS/SQL statements to change the structure of that table and the privileges granted on that table. You cannot use the PROGRESS Data Dictionary. (You can, however, use the Data Dictionary to set or alter dictionary fields not maintained by SQL DDL, such as the description and help fields.) In the same way, you must use the Data Dictionary to make changes to the structure of a file created with the Data Dictionary.

You are encouraged to use the PROGRESS Data Dictionary to define files, even if you expect to use SQL against them. By doing this, you establish a consistent way to maintain the data for all the files for your application. The only time you *must* use PROGRESS/SQL to define a table is when you want to be able to modify the privilege/change controls provided by PROGRESS/SQL using the GRANT and REVOKE statements (see section, "Data Definition Language"). If you use SQL DDL to define tables, some features of the PROGRESS Data Dictionary are not available to you. For example, you will not be able to deny access to the blank userid for SQL tables, since the notion of the blank userid is not defined in SQL.

You use the PROGRESS editor to enter procedures that contain PROGRESS/SQL statements. SQL statements in PROGRESS procedures are considered *interactive SQL* as opposed to SQL statements embedded in a host language program. With interactive SQL, your procedures can contain SQL statements only, or both SQL and PROGRESS statements.

PROGRESS also supports the use of embedded SQL in host language programs. See the *3GL Interface Guide* for information about using embedded SQL in PROGRESS.

15.1.1 PROGRESS Extensions

With PROGRESS/SQL, you can use several PROGRESS features, namely:

- The PROGRESS format phrases FORMAT, LABEL, and COLUMN-LABEL in PROGRESS/SQL CREATE TABLE, ALTER TABLE, and SELECT statements. These phrases control the display format and labels used when data is displayed on the screen or in printed reports.
- The DEFAULT format phrase to set an initial value for the column.
- The WITH frame phrase in PROGRESS/SQL queries. This phrase determines frame attributes.
- The PROGRESS DATE and LOGICAL data types.
- PROGRESS procedure variables and field values (retrieved by previous queries) in PROGRESS/SQL statements.
- Subscripts for access to existing array fields created with PROGRESS. (You cannot create array fields with PROGRESS/SQL.)
- The PROGRESS RECID function, used mainly for access to the schema files.
- The plus sign (+) for character string concatenation.
- The PROGRESS editor to enter, compile, and run PROGRESS/SQL statements.

15.1.2 Using Foreign Databases with SQL

You can use SQL to retrieve and update data in the foreign databases PROGRESS supports. For Version 6, PROGRESS supports the use of DEC's Record Management Services (RMS) for VAX/VMS, and ORACLE's SQL-based DBMS. For each foreign database, PROGRESS provides a database gateway that processes the database requests from a client application. The interface processing is transparent to the application. To use SQL with a foreign database, you only need to specify the appropriate PROGRESS database parameters when you use the CONNECT statement to connect the databases. You cannot use SQL to modify the schema of foreign databases.

15.2 PROGRESS/SQL LANGUAGE COMPONENTS

This section describes the basic components you use to create SQL procedures. It lists the PROGRESS/SQL reserved words, explains SQL naming conventions, and explains how SQL handles null values.

15.2.1 PROGRESS/SQL Reserved Words

Table 15-2 contains a list of SQL keywords. They are *reserved words* and have special meaning in PROGRESS/SQL.

Table 15-2: PROGRESS/SQL Reserved Words

ADD	CURSOR	INSERT	REVOKE
ALL	DEC	INT	ROLLBACK
ALTER	DECIMAL	INTEGER	SCHEMA *
AND	DECLARE	INTO	SELECT
ANY	DELETE	IS	SET
AS	DESC	LIKE	SMALLINT
ASC	DISTINCT	MAX	SOME
AUTHORIZATION *	DOUBLE *	MIN	SUM
AVG	DROP	NOT	TABLE
BETWEEN	ESCAPE	NULL	TO
BY	EXISTS	NUMERIC	UNION *
CASE-SENSITIVE	FETCH	OF	UNIQUE
CHAR	FLOAT	ON	UPDATE
CHARACTER	FOR	OPEN	USER
CHECK	FROM	OPTION	VALUES
CLOSE	GRANT	OR	VIEW
COLUMN	GROUP	ORDER **	WHERE
COMMIT	HAVING	PRECISION *	WITH
COUNT	IN	PRIVILEGES	WORK
CREATE	INDEX	PUBLIC	
CURRENT	INDICATOR	REAL	

* These keywords are reserved for future PROGRESS/SQL development.
** ORDER is a keyword but is not reserved.

Some PROGRESS/SQL reserved words are also PROGRESS reserved words. These words are context-sensitive, which means that the compiler can detect the difference between, for example, a PROGRESS/SQL UPDATE statement and a PROGRESS UPDATE statement by the language elements that follow the reserved word UPDATE. For a list of PROGRESS reserved words, see the end of the *PROGRESS Language Reference* manual. Reserved words are not case-sensitive.

15.2.2 SQL Statements

When you enter SQL statements, the sequence of keywords and values is determined by the syntax of the statement you are entering. However, the arrangement of the statements on the screen is not significant to SQL. Any number of spaces, tabs, and newline characters are treated as a single space. This feature allows you to arrange the statements in any way that is clear and convenient for you.

The SQL character set consists of the uppercase letters A through Z, the lowercase letters a through z, the digits 0 through 9, and the following special characters:

% ( ) _ < > . + - = * , ' " /

Each PROGRESS/SQL statement ends with a period. To include comments in your procedures, begin the comment with /* and end it with */. SQL ignores any information between these symbols.

When entering lists of items, such as a series of column names, separate the items with commas. For example, in the following SELECT statement, the column names in the select list are separated with commas.

```
SELECT name, address, city, st FROM customer.
```

Identifiers. *Identifiers* are names that you give language elements. All of the following elements are identifiers:

- Table and view names
- Column names
- Index names
- Range variables

Identifiers can be up to 32 characters in length. The first character must be an uppercase or lowercase letter. The remainder of the identifier can be any combination of letters, digits, and underscore (_) characters. You cannot use any of the SQL special characters (except for the underscore character) as a part of an identifier. Identifiers are never case sensitive.

You cannot use SQL reserved words or PROGRESS reserved words as identifiers. One exception is ORDER. To allow use of the `order` table in the demonstration database, ORDER is not a reserved word.

A table or view name must be unique within the database. A column name must be unique within the table; however, you can have duplicate column names within the database. An index name must be unique within the database, not just within the table.

PROGRESS/SQL allows you to access existing PROGRESS files, fields, and variables even if their names contain characters that are not valid in PROGRESS/SQL identifiers, such as hyphens and percent signs. You cannot, however, create new SQL tables and columns with these characters. Similarly, you cannot create new views and view columns with characters that are not valid in SQL identifiers. However, if the view columns are created implicitly (that is, using the column names from the base table), PROGRESS/SQL allows the retention of the invalid characters.

Name Qualifiers. Name qualifiers are necessary to avoid ambiguous references. When the same column name is used in more than one table and those columns are referenced in the same query, you must qualify the column names with their table names (for example, `customer.cust_num` and `order.cust_num`).

When accessing multiple or foreign databases, you should also qualify table names and view names. If you do not qualify your reference, PROGRESS/SQL takes the name from the default database, if it exists there. This is different from PROGRESS. In PROGRESS, you receive an error message if the table or view name exists in more than one database.

Table 15–3 shows the general syntax for qualifying column, table, and view names.

Table 15–3: Column, Table, and View Name Qualifiers

Identifier	Syntax for Name Qualifier
Column	[[*db.*]*table-name.*]*column-name*
Table	[*db.*]*table-name*
View	[*db.*]*view-name*

Literals. *Literals* are constants. Literals can be character string, numeric, date, or logical data.

A character string literal can consist of non-numeric or character data, or a combination of numeric and non-numeric data. Character string literals must be enclosed in either single quotation marks (′ ′) or double quotation marks (" "). For example:

```
'Jane Smith'    "$125.00"    '01754'    "15 Oak St."
```

To include a single or double quotation mark within a character string literal, either enter the mark twice or use the opposite mark from the one you want to include. For example:

```
'Brown''s Plumbing Supply'   or "Brown's Plumbing Supply"
```

Date literals require the format *mm/dd/yy.* For example: 09/13/57.

Logical literals are represented by `yes/no`, or `true/false`.

PROGRESS/SQL provides two *wildcard* characters that you can use in LIKE clauses to match character strings. The percent sign character (%) matches zero or more characters. The underscore character (_) matches any single character.

Operators. *Operators* are the symbols you use to perform calculations and comparisons. SQL operators are the same as PROGRESS operators. For detailed information about operators, refer to the *PROGRESS Language Reference* manual.

NOTE: When you use the arithmetic operators +, -, * and / in PROGRESS procedures, at least one blank space must precede and follow an operator. If you omit the space(s), you will get an error message.

15.2.3 Null Values

A *null value* is used to indicate that the value for a particular column is "unknown" or "not available". The PROGRESS/SQL null value is equivalent to the PROGRESS unknown value (?).

Any arithmetic expression that contains a null value evaluates to null. Any comparison expression that contains a null value (in WHERE and HAVING clauses) evaluates to unknown, neither true nor false. When a WHERE or HAVING clause evaluates to unknown for a particular row, PROGRESS does not select or modify that row.

The following table shows the resulting truth values when you combine comparison expressions with AND, OR, and NOT. The unknown value is represented by "?".

Table 15-4: Truth Values in Combined Comparison Expressions

	AND			OR			NOT
	T	?	F	T	?	F	
True	T	?	F	T	T	T	F
Unknown	?	?	F	T	?	?	?
False	F	F	F	F	?	F	T

15.3 SQL DATA MANIPULATION LANGUAGE

You use the PROGRESS/SQL data manipulation language to retrieve and update data in SQL tables, PROGRESS records, and foreign databases. The following section describes the basic SQL data manipulation statements which include: SELECT, INSERT, UPDATE, and DELETE. It also describes the use of cursors, transaction processing, and privilege checking. All of the examples in this section can be used with the PROGRESS demo database.

15.3.1 The SELECT Statement

The SELECT statement retrieves data from the database. Because the SELECT statement is the most complex SQL statement, the following explanation is divided into sections. The sections describe the basic SELECT statement, search conditions, joins, grouping data, sorts, subqueries, and the use of aggregate functions.

The basic SELECT statement retrieves and displays as many rows of data as satisfy the selection criteria you specify. Used in this way, the basic SELECT statement is equivalent to the PROGRESS statements FOR EACH...DISPLAY.

The SELECT statement has the following syntax:

SYNTAX

SELECT [ALL | DISTINCT] { * | *column-list* } FROM *clause*
[WHERE *clause*] [GROUP BY *clause*] [HAVING *clause*] [ORDER BY *clause*]
[WITH [STREAM *stream*] EXPORT]

To specify the columns to retrieve (the column-list), you can either use an asterisk (*) to indicate all columns, or you can enter one or more column names separated by commas. The keyword ALL, the default, selects all of the values in the specified columns. The optional keyword DISTINCT retrieves only the unique values in a column or the unique combinations of values in two or more columns. The asterisk (*) selects all columns from a table.

The following example selects all values from all columns in the table named `customer` and displays it in a readable format.

```
SELECT * FROM customer WITH 2 COLUMNS.
```

The WHERE clause of a SELECT statement specifies search conditions for the data you want to select. If the WHERE clause is omitted, all rows of the table qualify for selection.

This next example selects the `name, address, city,` and `st` columns from the `customer` table. The WHERE clause specifies only customers from Massachusetts.

```
SELECT name, address, city, st
  FROM customer
     WHERE st = 'MA'.
```

A WHERE clause consists of one or more search conditions connected by the logical operators AND, OR, and NOT. All of the following are search conditions:

- *expression relational-operator expression*

  In the following example, the WHERE clause specifies customers whose customer number is greater than 10.

```
SELECT cust-num, name
 FROM customer
  WHERE cust-num > 10.
```

- *expression1* [NOT] BETWEEN *expression2* AND *expression3*

  The BETWEEN search condition is equivalent to:
  expression1 > = *expression2* AND *expression1* < = *expression3*

  The following example uses BETWEEN and NOT BETWEEN to select item numbers.

```
SELECT item-num, idesc
 FROM item
     WHERE item-num BETWEEN 10 AND 20
        AND item-num NOT BETWEEN 15 AND 17.
```

- *column-name* IS [NOT] NULL

  This search condition tests whether a column contains a null value or not. See the section "Null Values" in this chapter for information.

  The following SELECT statement uses the IS NULL phrase to find which customer's orders have not been shipped.

```
SELECT name, city, st, odate
  FROM order
    WHERE shipped IS NULL.
```

- *column-name* [NOT] LIKE *"string"* [ESCAPE *"character"*]

  The LIKE search condition compares character values in a column to a character string. The string must be enclosed in either single or double quotation marks. The string can be a pattern created with any number of characters, the wildcard characters percent sign (%) and underscore (_). The percent sign character matches zero or more characters. The underscore character matches any single character.

The following example selects all items with locations that begin with "Bin 2" followed by a single character.

```
SELECT item-num, idesc, loc
  FROM item
   WHERE loc LIKE 'Bin 2_'.
```

If you want to search for the percent sign or underscore character, you must declare an escape character to disable the use of the character as a wildcard. To declare an escape character, use the keyword ESCAPE followed by an escape character of your choosing. Enclose the escape character in either single or double quotation marks. In the search string, precede the percent sign or underscore character with the declared escape character.

In the following example, the vertical bar (|) is used as the escape character to search for the percent sign character in the last position of `terms`.

```
terms LIKE '%|%' ESCAPE '|'.
```

If you want to use for a backslash (\) as the escape character, you need to declare it as "\\", since "\" is already defined as an escape character in PROGRESS.

NOTE: See Chapter 14 for special information on using the tilde (~) or backslash (\) as the escape character.

- *expression* [NOT] IN {*value-list* | *SELECT-statement*}

This search condition tests for the expression being equal or not equal to any item in the *value-list* or any value returned by the SELECT statement (the SELECT statement is a subquery and is explained below). If you enter a list of values, the values must be separated by commas and enclosed in parentheses.

The following query uses the IN syntax to select customers from New England.

```
SELECT name, address, city, st
  FROM customer
    WHERE st IN ('MA','NH', 'VT', 'ME', 'CT', 'RI')
      WITH FRAME x CENTERED.
```

Specifying Frame Properties. To modify a frame's properties with the SELECT statement, use a Frame Phrase. To modify the frame properties of an individual field, variable, or expression, use a Format Phrase. See the *PROGRESS Language Reference* for more information on Frame Phrases and Format Phrases.

Selecting Data from Multiple Tables (Joins). The examples in the preceding section selected data from one table by specifying a single table name in the FROM clause of the SELECT statement. You can also select data from multiple tables and compare rows within the same table by using the following syntax in the FROM clause:

SYNTAX

FROM { *table-name* [*range-variable*] } [,...]

To select data from multiple tables, specify a list of tables names separated by commas. To compare rows within the same table specify a *range variable*.

Selecting data from multiple tables by means of a single SELECT statement is referred to as a *join* operation. A join operation effectively creates a temporary table in which the rows from the specified tables that satisfy the WHERE clause are joined to form a single row.

The following example joins the `order` table and the `order-line` table.

```
SELECT name, order.order-num, line-num, item-num
  FROM order, order-line
    WHERE order.order-num = order-line.order-num.
```

Since both the tables `order` and `order-line` have a column named `order-num`, you must use name qualifiers to avoid ambiguity.

The SELECT statement also performs *self-joins*. A self-join joins a table with itself. A self-join operation requires the specification of a *range variable* in the FROM clause. A range variable is an alias for a table name. In the following example, `alt` is a range variable used to make a second reference to the `customer` table.

```
SELECT item.idesc, item.item-num, alt.idesc alt.item-num
  FROM item, item alt
    WHERE alt.item-num = item.subs-item.
```

The preceding query lists every item that has a designated substitute (based on the field `subs-item`, the item number of the substitute) with the number and description of the substitute. This example requires the use of a range variable to compare rows within the same table.

Grouping Data. A *group* is a set of rows that has the same value for a specified column or columns. The optional GROUP BY clause of a SELECT statement results in a single row in the result table for each group of rows. This clause is useful for arranging a table into conceptual groups so that you can apply an operation or function, such as COUNT or SUM, to each group.

The GROUP BY clause has the following syntax:

SYNTAX

```
GROUP BY column-list [HAVING search-condition]
```

The *column-list* determines the group. It consists of a column name or a list of column names separated by commas. The optional HAVING clause is used to exclude particular groups from the query results. In practice, the *search-condition* in the HAVING clause usually involves an aggregate function (see "Using Aggregate Functions" for more information). HAVING is rarely used without GROUP BY. If you specify a HAVING clause without a GROUP BY clause, the entire table is treated as a single group.

The following example displays the total number of items in a product line and the cost of the most expensive maximum cost of an item in that product line.

```
SELECT prod-line, count(*), max(cost)
  FROM item
    GROUP BY prod-line.
```

Sorting Data. The ORDER BY clause of a SELECT statement sorts query results by the values in one or more columns. It has the following syntax:

SYNTAX

```
ORDER BY { { column-name | n } [ ASC | DESC ]} [,...]
```

The following example of the ORDER BY clause sorts the data alphabetically by state and, within each state, alphabetically by city.

```
SELECT name, address, city, st
  FROM customer
   ORDER BY st, city.
```

As an alternative to column names, you can specify integers in the ORDER BY clause. The integer refers to the ordinal position of the column in the SELECT list. You must specify an integer to sort on an expression in the SELECT list. In the following example, the 2 refers to the SUM column.

```
SELECT loc, sum(cost)
  FROM item
   GROUP BY loc
    ORDER BY 2 desc.
```

You can optionally specify the keyword DESC to sort the results in descending order. Ascending is the default sort order so you do not have to specify the keyword ASC.

Two null values are considered equal within the context of GROUP BY, ORDER BY, and DISTINCT clauses. In ORDER BY clauses, null values are also considered greater than all non-null values. That is, when the sort order is ascending (the default), null values are sorted last. When the sort order is descending, null values are first.

Exporting Data. In PROGRESS/SQL, you can convert data to standard character format and display it to the current output destination or to a named output stream with the WITH EXPORT clause of the SELECT statement. This clause works exactly like the PROGRESS EXPORT statement.

The following example converts all of the values in the table `customer` into a standard character format and sends it the file `customer.d`.

```
OUTPUT TO customer.d
SELECT * FROM customer WITH EXPORT.
```

The code in the above example is equivalent to the following PROGRESS code.

```
OUTPUT TO customer.d.
FOR EACH customer:
   EXPORT customer.
END.
```

Using Subqueries. A *subquery* is a SELECT statement that is nested within a WHERE clause. The result of the subquery is used as a value in the WHERE clause. A subquery must be enclosed in parentheses ( ), and it can normally have only one column in its SELECT list (EXISTS is the exception).

Subqueries are of two types: singleton subqueries and set subqueries. A *singleton subquery* is preceded by a comparison operator and returns, at most, a single row.

You use a singleton subquery in the following context:

SYNTAX

```
comparison operator (SELECT-statement)
```

The following example contains a singleton subquery in the WHERE clause. The subquery returns a single value, the maximum number of item 51 ordered.

```
SELECT order-num
  FROM order-line
 WHERE item-num = 51 AND qty =
      (SELECT MAX(qty) FROM order-line
        WHERE item-num = 51).
```

A *set subquery* returns a set of one or more rows. You use set subqueries in the following contexts:

SYNTAX

```
{ expression operator { ALL | ANY | SOME } (SELECT-statement) }
{ expression [NOT] IN (SELECT-statement)                      }
{ [NOT] EXISTS (SELECT-statement)                             }
```

The set subquery options are search conditions that:

- Compare the result of a subquery with an expression.
- Determine if the result of a subquery includes an expression.
- Determine if the subquery selected any rows.

When you specify ALL, the search condition is true if the comparison is true for each of the values returned by the subquery. With ANY, the search condition is true if the comparison is true for at least one of the values returned by the subquery. The keyword SOME is equivalent to the keyword ANY. The subquery can return zero, one, or more values. If it returns no value, the search condition has a truth value of UNKNOWN (?).

The [NOT] IN search condition tests for the expression being equal or not equal to any value returned by the SELECT statement.

The EXISTS search condition determines whether the subquery returns any rows. The search condition is true if the subquery returns one or more rows. The EXISTS search condition currently works only when an indexed search can satisfy the request. Therefore, you must use an indexed field in the WHERE clause when you specify the EXISTS search condition. (This is equivalent to the PROGRESS CAN-FIND function.)

The following example lists information about customers who have orders in the order file.

```
SELECT name, cust-num, city, st
  FROM customer
 WHERE EXISTS
     (SELECT * FROM order
       WHERE order.cust-num = customer.cust-num).
```

The use of SELECT * is permitted in subqueries with the EXISTS search condition only, even if the result has more than one column. In fact, you would normally specify SELECT * for an EXISTS search condition, since no data is actually returned.

Using Aggregate Functions. *Aggregate functions* are a set of tools that allow you to evaluate groups of values. In general, each function operates on the set of values in one column or on an expression. The set of column values is determined by the WHERE clause of a SELECT statement. SQL provides the following aggregate functions:

AVG([DISTINCT] *x*) — Calculates the average of values in column or expression *x*. If you use DISTINCT, the calculation is the average of distinct values only. You can use AVG only with numeric columns or expressions.

COUNT(*) — Counts all rows in a table without eliminating duplicates.

COUNT(DISTINCT *x*) — Counts the number of different values in column *x*.

MAX([DISTINCT] *x*) — Returns the maximum value contained in column *x*.

MIN([DISTINCT] *x*) — Returns the minimum value contained in column *x*.

SUM([DISTINCT] *x*) — Calculates the sum of values in column or expression *x*. If you use DISTINCT, the calculation is the sum of distinct values only. You can use SUM only with numeric columns and expressions.

The following example counts all rows in the customer table.

```
SELECT COUNT(*) FROM customer.
```

You can include multiple aggregate functions in a single SELECT list. The next example returns the average, maximum, and minimum cost of items in the `item` table.

```
SELECT AVG(cost), MAX(cost), MIN(cost)
    FROM item.
```

When null values are encountered by aggregate functions, they are not included in the aggregate total. The exception is COUNT(*). COUNT(*) counts all rows, including rows that have null values.

Aggregation of columns or expressions containing zero (0) values yield NULL (the PROGRESS unknown value, ?), not 0 as in Progress. In the following example, the aggregate function AVG yields a value of ? for the `max-credit` field:

```
SELECT AVG(max-credit) FROM customer
     WHERE cust-num < 0.
```

The next example counts the number of different locations and the total number of rows in the `item` table.

```
SELECT COUNT(DISTINCT loc),
       COUNT(*)
        FROM item.
```

Aggregate functions can include arithmetic expressions, and the results of aggregate functions can be used in other operations. For example:

```
SELECT loc, SUM(cost * on-hand)
  FROM item
  GROUP BY loc
HAVING SUM(cost * on-hand) > 0
  ORDER BY 2 DESC.
```

15.3.2 Inserting Rows

The INSERT INTO statement inserts one or more rows into an existing table. The data you insert can be a list of values that you supply or values from another table.

The INSERT INTO statement has the following syntax:

SYNTAX

INSERT INTO { *table-name* | *view-name* } [(*column-list*)]
{ { VALUES (*value-list*)} | *SELECT-statement* }

The VALUES clause contains a list of values that you specify in your SQL statement. If you use the VALUES clause, you can insert only one row at a time into the table. The *value-list* can be one value or a series of values separated by commas. Each value can be a literal, the keyword NULL, a procedure variable, or a field defined and assigned a value elsewhere in the procedure. Each character string literal must be enclosed in either single or double quotation marks.

To insert values from another table, enter a SELECT statement.

This first example inserts a new customer by specifying a list of values.

```
INSERT INTO customer (name, address, city, st, zip, cust-num)
     VALUES ('Golf Pro', '5 First Street',
             'Goffstown', 'NH', 01234, 101).
```

SQL inserts the first value into the first column in the list, the second value into the second column, and so on. Therefore, the number of columns must be the same as the number of values. If you specify a column list that does not include all columns of the table, SQL assigns the null or default value to each column omitted from the list. If you omit the column list entirely, SQL inserts the values into the columns in the order in which the columns were created. In that case, there must be a value for every column in the table.

If you use the SELECT statement with the INSERT INTO statement, all the rows that satisfy the SELECT statement are inserted into the table. The number and order of columns in the SELECT statement must match the implicit or explicit column list in the INSERT INTO statement. If you specify SELECT *, the asterisk is expanded into a list of all columns in the FROM-clause table(s).

The following example inserts rows into a preferred customer table by selecting data from the `customer` table. It inserts a row for each customer whose purchases exceed $100,000.

```
INSERT INTO prefer_cust
       (name, address, city, st, zip, cust-num)
       SELECT name, address, city, st, zip, cust-num
         FROM customer
        WHERE ytd-sales > 100000.
```

When you use the INSERT INTO statement to add rows through a view, you can only insert values into columns that are defined in the view definition. (See section, "Creating Views", for more information.) Any other columns in the underlying table for the view are set to their default value, if there is one, or to null.

15.3.3 Updating Rows

The searched UPDATE statement changes column values in one or more rows of a table. It has the following syntax:

SYNTAX

UPDATE { *table-name* | *view-name* }
SET *column-name* = *expression* [, ...]
[WHERE *search-condition*]

The SET clause evaluates the *expression* and assigns it to the *column-name*. The expression can be the keyword NULL, or it can be a column name, a literal, an arithmetic operation, a procedure variable or field name, or any combination of these.

The WHERE clause determines the rows to be updated. If you omit the WHERE clause, all rows of the target table are updated. The section, "The SELECT Statement", explains the search conditions used in WHERE clauses.

The following example changes the zip code for customers in Loudon, New Hampshire.

```
UPDATE customer SET zip = 03045
     WHERE city = 'Loudon' AND st ='NH'.
```

15.3.4 Deleting Rows

The searched DELETE statement deletes one or more rows from a table. It has the following syntax:

SYNTAX

```
DELETE FROM { table-name | view-name } [ WHERE search-condition ]
```

The WHERE clause identifies the rows to be deleted. If you omit the WHERE clause, all rows of the target table will be deleted. The section, "The SELECT Statement", explains the search conditions used in WHERE clauses.

The following example deletes item number 51 from the `item` table.

```
DELETE FROM item
     WHERE item-num = 51.
```

This next example uses the LIKE search condition to specify the rows for deletion.

```
DELETE FROM item
     WHERE idesc LIKE 'Croquet%'.
```

15.3.5 Working with Cursors

In interactive SQL, the SELECT statement retrieves multiple rows of data and displays the results directly to the terminal. However, the basic SELECT statement cannot store and process multiple rows in a retrieval set. The way around this limitation is to step through the rows chosen by the SELECT statement and manipulate them one by one. This is accomplished with a *cursor*.

A cursor is a pointer used to search through a retrieval set, pointing to each row in the set one at a time. When a cursor is pointing to a row, it is said to be positioned on that row. The positioned row can then be updated or deleted using the positioned forms of the UPDATE and DELETE statements.

Cursors enable you to retrieve column values from a row using a SELECT statement and to assign the column values to procedure variables. A cursor must always be used with each SELECT statement (except the singleton SELECT) if you want manipulate and update row values in the retrieval set. Without cursors, there is no way to process individual rows.

Defining a cursor for a SELECT operation is a two-step process in which you declare the cursor with the DECLARE CURSOR statement and then open the cursor with the OPEN statement. Once the cursor is opened, you use the FETCH statement to retrieve the data. When you are finished with a cursor, you close it with the CLOSE statement. You do not need to declare the cursor again if you reselect the data associated with it by reopening the cursor.

You can declare and open more than one cursor at a time to use with a single table and there is no limit on the number of cursors you can declare in your procedure. However, you can only use a cursor in the procedure in which it is declared.

15.3.6 Cursor Statements

SQL cursor statements are used to define, open, and close cursors, as well as to retrieve individual rows and move the column values into procedure variables.

There are six PROGRESS cursor statements:

DECLARE CURSOR	Associates a cursor name with a SELECT statement.
OPEN	Selects all rows that satisfy the DECLARE CURSOR statement and positions the cursor before the first row.
FETCH	Retrieves the next row and assigns the column values to procedure variables. Positions the cursor on the next row, or if there is no next row, after the last row.
Positioned UPDATE	Modifies the data in the row to which a cursor is pointing.
Positioned DELETE	Deletes the row to which a cursor is pointing.
CLOSE	Closes the cursor when the rows have been processed.

Figure 15-1 shows the sequence of the cursor statements when the following procedure is executed.

```
    DEFINE VARIABLE namevar LIKE customer.name.
    DEFINE VARIABLE maxvar LIKE customer.max-credit.
[1] DECLARE c1 CURSOR FOR
    SELECT name, max-credit
          FROM customer
                WHERE max-credit < 1000.
[2] OPEN c1.
[3] REPEAT:
    FETCH c1 INTO namevar, maxvar.
[4] UPDATE customer
          SET max-credit = maxvar + 1000.
                WHERE CURRENT OF c1.
    END.
    CLOSE c1.
```

[1] DECLARE c1 CURSOR
FOR SELECT statement

[2] OPEN c1

SELECT statement is evaluated and yields a retrieval set. The cursor is positioned before the first row.

Customer Table	
Name	Max-Credit
Thundering Surf Inc.	3,290
Hard Knocks Skating	2,253
Buffalo Shuffleboard	223
Fast Flipper Pinball	472
Sticky Wicket Cricket	317

[3] REPEAT:
FETCH c1 INTO *variable-list*

There is one procedure variable for each column listed in the SELECT statement. The FETCH statement retrieves the next row from the retrieval set and moves the column values into the corresponding procedure variables.

[4] UPDATE customer
SET column-name = expression
WHERE CURRENT OF c1.

The SET clause sets the column to the value derived from the expression. The WHERE CURRENT OF clause uses the cursor name to identify the row to be updated.

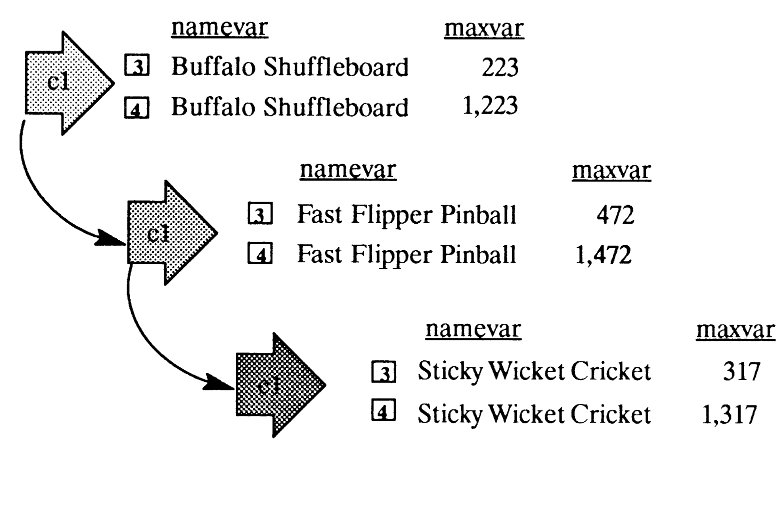


Figure 15–1: Cursor Execution Sequence

Defining a Cursor. The DECLARE CURSOR statement has the following general syntax:

SYNTAX

DECLARE *cursor-name* CURSOR FOR *SELECT-statement* [FOR READ [ONLY]]

The *cursor-name* can be any valid SQL identifier. The *SELECT-statement* is any valid SQL SELECT statement, including WHERE, GROUP BY, HAVING, and ORDER BY clauses, if desired, as well as subqueries, aggregates, joins, and all other valid SELECT statement syntax. The *SELECT-statement* specifies a retrieval set of rows that is accessible when the cursor is opened.

FOR READ ONLY allows you to read the selected rows but not update or delete them. FOR READ ONLY is performed with NO_LOCK to prevent lock table overflow. To coordinate concurrent access to records in multi-user applications, SQL statements use a default record locking scheme based on the PROGRESS record locking phrases. You do not insert PROGRESS record locking phrases directly into embedded SQL programs. Table 15-5 shows the PROGRESS record locks and explains their effects. Table 15-6 shows the locks associated with SQL statements. Record locking is explained in Chapter 12 of this handbook.

Table 15–5: PROGRESS Record Locking Phrases

Record Lock	Effect of Using Record Lock
NO-LOCK	Rows are not locked. Rows are read even if they are EXCLUSIVE-LOCKed by another application.
SHARE-LOCK	Rows are read in anticipation of a possible update. Rows are not read if they are EXCLUSIVE-LOCKed by another application.
EXCLUSIVE-LOCK	Rows are locked until the end of the transaction. Rows cannot be read by another application using SHARE-LOCK or EXCLUSIVE-LOCK until then.

Table 15–6: SQL Statements and Associated Record Locks

Embedded SQL Statement	Associated Record Lock
Non-cursor SELECT and singleton SELECT statements	NO-LOCK. You may see any uncommitted changes to the database.
OPEN cursor FOR SELECT	SHARE-LOCK. Records retrieved through a cursor use SHARE-LOCK while being read and are converted to EXCLUSIVE-LOCK if updated or deleted.
UPDATE WHERE *condition* UPDATE WHERE CURRENT DELETE WHERE *condition* DELETE WHERE CURRENT	EXCLUSIVE-LOCK.

To define a cursor, you must have the SELECT privilege on all tables referred to in the DECLARE CURSOR statement (see section, "Access Privileges", for more information). The DECLARE CURSOR statement associates the cursor with the SELECT statement and assigns a PROGRESS buffer with the same name as *cursor-name*. Later, PROGRESS/SQL will fetch rows into the buffer. If the cursor definition involves a join, PROGRESS/SQL defines a buffer for each table in the join. The first buffer is named *cursor-name*. The other buffer names start with *cursor-name* followed by a hyphen and a letter, as in *cursor-name-a* and *cursor-name-b*.

You must use the DECLARE CURSOR statement in your PROGRESS/SQL procedure before the cursor is opened or referenced. You declare a cursor only once, even if you intend to close and reopen it several times in a procedure.

In the following example, the DECLARE CURSOR statement uses a cursor named c1 to retrieve the name and customer number for all Massachusetts customers.

```
DECLARE c1 CURSOR FOR
        SELECT name, cust-num
          FROM customer
         WHERE st = 'MA'.
```

The SELECT statement can refer to a procedure variable in its WHERE clause. The variable is evaluated when the cursor is opened, not when it is declared. The following example illustrates how a procedure variable is used in the WHERE clause:

```
DEFINE VARIABLE state-var AS CHARACTER INITIAL "MA".

DECLARE c1 CURSOR FOR
      SELECT name, address, city, st
        FROM customer
       WHERE st = state-var ORDER BY st, city.
```

NOTE: The SQL standard says that a cursor is not updatable if an ORDER BY clause appears in the DECLARE CURSOR statement. PROGRESS/SQL does not enforce this restriction. However, updatable cursors must obey the same restrictions as updatable views. For a list of these restrictions, see section, "Updating Views."

Opening a Cursor. The OPEN statement has the following general syntax:

SYNTAX

OPEN *cursor-name*

The OPEN statement preselects the retrieval set of rows resulting from the execution of the SELECT statement in the DECLARE CURSOR statement. PROGRESS/SQL uses current values for any procedure variables referenced in the SELECT statement to yield the retrieval set. The cursor is positioned before the first row in the retrieval set.

The following example opens the cursor declared in the previous section.

```
OPEN c1.
```

A cursor can be opened and closed multiple times after it has been defined. Each time the cursor is opened, the SELECT statement is reexecuted (therefore, you do not need to declare the cursor more than once). With each execution, the retrieval set may contain different data if the input parameters or database values have changed.

Retrieving Values from the Retrieval Set. The FETCH statement has the following general syntax:

SYNTAX

FETCH *cursor-name* INTO *variable-list*

The *variable-list* is a list of procedure variables separated by commas. There must be one variable for each column listed in the SELECT statement that is associated with the DECLARE CURSOR statement. The data types of the variables and columns must be compatible. You should not use the same name to specify procedure variables and SQL columns.

When referring to the values fetched into the list of variables, use the procedure variable name into which it was fetched. Do not use the column name from the DECLARE CURSOR statement.

The FETCH statement retrieves the next row from the retrieval set and moves each column value to the corresponding variable. There can be more than one FETCH statement for an open cursor, each with its own list of variables, as long as the number and data types of the variables correspond to the cursor definition.

In the following example, the customer name and number are fetched into the procedure variables, `var1` and `var2`:

```
FETCH c1 INTO var1, var2.
```

If the DECLARE CURSOR statement specifies SELECT *, the number of variables in the list of variables must match the column count represented by the asterisk.

Each time it is executed, the FETCH statement retrieves a single row in the retrieval set. To fetch all the rows in a retrieval set, you must include the FETCH statement in a PROGRESS looping block. For example, you could place the FETCH statement within a REPEAT block to fetch multiple rows, one row at a time.

Positioned UPDATE. The positioned form of the UPDATE statement is used with an open cursor. Once you have used the cursor statement FETCH to retrieve a row, you can individually update one or more columns in that row.

The positioned UPDATE statement has the syntax:

SYNTAX

```
UPDATE table-name
    SET column-name = expression [, ... ]
        WHERE CURRENT OF cursor-name
```

The table in the positioned UPDATE statement is the same table associated with the cursor in the DECLARE CURSOR statement. The SET clause sets the specified columns to their new values, as derived from the expressions. The WHERE CURRENT OF clause uses the *cursor-name* to identify the cursor whose current row is to be updated.

A cursor cannot be updated if it is defined for a SELECT statement that contains any of the following clauses or expressions:

- A join.
- A GROUP BY or HAVING clause.
- An expression or constant in the SELECT list.
- The DISTINCT keyword.
- A subquery.
- A reference to a view that cannot be updated

Positioned DELETE. The positioned form of the DELETE statement is used with an open cursor. Once you have used the cursor statement FETCH to retrieve a row, you can delete that row.

The positioned DELETE statement has the syntax:

SYNTAX

```
DELETE FROM table-name WHERE CURRENT OF cursor-name.
```

The table in the positioned DELETE statement is the same table associated with the cursor in the DECLARE CURSOR statement. The WHERE CURRENT OF clause uses the *cursor-name* to identify the cursor. When the current row is deleted, the cursor is positioned before the next row, or after the last row if there is no next row.

Closing a Cursor. The CLOSE statement has the following general syntax:

SYNTAX

```
CLOSE cursor-name
```

This statement closes the cursor. You cannot refer to the cursor again unless you open it again. If it is reopened, PROGRESS/SQL reselects the retrieval set and repositions the cursor to the first row. All open cursors are closed on exit from the procedure.

The following statement closes cursor `c1`:

```
CLOSE c1.
```

15.3.7 Selecting Single Rows (Singleton SELECT)

The SELECT INTO statement retrieves only one row of the table. If you are certain that only one row of data will be retrieved, you can use the SELECT INTO statement instead of declaring a cursor and fetching one row through it.

Here is the general syntax for the SELECT INTO statement:

SYNTAX

```
SELECT [ALL | DISTINCT] { * | column-list } INTO variable-list
    FROM clause
   [WHERE clause] [GROUP BY clause] [HAVING clause] [ORDER BY clause]
```

The SELECT INTO statement assumes that the WHERE clause evaluates to only one row. If you know that multiple rows exist with the same values, specify DISTINCT to prevent a cardinality error. ALL is the default. The *variable-list* is a list of procedure variables separated by commas. There must be one variable for each column in the *column-list*. The data types of the variables and columns must be compatible. You can use the same name to specify procedure variables and SQL columns. All other elements of the syntax are the same as for the basic SELECT statement.

In the following example, the SELECT INTO statement retrieves the name and state for customer number 10.

```
DEFINE VARIABLE namevar LIKE customer.name.
DEFINE VARIABLE statevar LIKE customer.st.

SELECT name, st INTO namevar, statevar
  FROM customer
 WHERE cust-num = 10.
```

A SELECT INTO statement may be used instead of a cursor for retrieving aggregate values that are not grouped, because such queries return only a single row.

15.3.8 Transaction Processing

Because PROGRESS has its own transaction management facilities, it is not necessary to terminate a transaction explicitly. In fact, the COMMIT WORK and ROLLBACK WORK statements are disabled by default in interactive SQL. Since PROGRESS offers you the capability to terminate transactions automatically, it is not advisable to enable COMMIT WORK and ROLLBACK WORK unless you have very specific reasons for doing so. To enable the use of these statements, enter the COMMIT ON command in the editor, and press F1.

There are two commands associated with COMMIT ON:

COMMIT OFF — Disables COMMIT WORK and ROLLBACK WORK and enables PROGRESS transaction management.

COMMIT STATUS — Displays a message regarding the status of the COMMIT WORK statement.

If you enter the COMMIT ON command, you can perform database queries and make ad hoc changes, then enter COMMIT WORK or ROLLBACK WORK as needed. To execute the COMMIT WORK and ROLLBACK WORK statements, you must enter them in the editor. You cannot include these statements in PROGRESS/SQL procedures.

When the COMMIT WORK statement is executed, all open cursors are closed and all updates made by the transaction are applied to the database. When the ROLLBACK WORK statement is executed, all open cursors are closed and all database updates are cancelled. In either case, all updated rows are locked until a COMMIT WORK statement or a ROLLBACK WORK statement is issued.

15.3.9 Privilege Checking at Compile-time and Runtime

Privilege checking for procedures containing PROGRESS/SQL statements is done automatically by PROGRESS at compile-time, in the same way it is done for PROGRESS 4GL procedures. You can view the table- and column-level compile-time security privileges by choosing the Change/Display Data Security option from the Admin menu in the Data Dictionary. However, you cannot use this menu to change privileges if the file you are viewing was created using the CREATE TABLE statement.

In addition to providing compile-time security, you may also want to check privileges at run-time. To prevent unauthorized users from running procedures that contain PROGRESS/SQL statements, you can use the CAN-DO function to check the userid established during login, or you can check the contents of the userid directly with the PROGRESS/SQL USER keyword, or the PROGRESS USERID function. The USER keyword is equivalent to the value of the USERID function for the current default (working) database.

If you use the CAN-DO or USERID function in a PROGRESS/SQL procedure, you must modify and recompile the procedure whenever you want to change the userids that are allowed to execute it. Alternatively, you can set up an activities-based file to define the users who are permitted to run a particular procedure. You read the file in your procedure to check the permissions for the current userid.

See the *PROGRESS Language Reference* manual for descriptions of the CAN-DO and USERID functions. For an explanation of compile-time security and activities-based security checking, see Chapter 14.

15.4 DATA DEFINITION LANGUAGE

With PROGRESS, you can use SQL data definition statements, the PROGRESS Data Dictionary, or both to create SQL tables and PROGRESS schema. You can use SQL Data Definition Language in either a new or an existing database. A single table or file, however, must be defined with a single mechanism (either SQL or the Data Dictionary).

As stated in the introduction, you are encouraged to use the PROGRESS Data Dictionary to define files, even if you expect to use SQL data manipulation statements against them. By doing this, you establish a consistent way to maintain the data for all the files for your application. The only time you must use SQL to define a table is when you want to be able to modify the privilege/change controls provided by SQL using the GRANT and REVOKE statements.

To access the data in a database, you can use SQL or PROGRESS 4GL data manipulation statements, or both.

A PROGRESS procedure containing SQL data definition statements must be compiled and run by the same userid. This is checked at run-time, and you will get an error message if the procedure is compiled and run by different userids. A check is also made to ensure that data definition statements are not compiled or run when the userid is blank. Therefore, on a PC or in other environments where a userid is not automatically assigned, a user must log in before running a procedure that contains SQL data definition statements.

This section explains how to define a database structure using SQL data definition statements to create tables, views, indexes, and access privileges. The section concludes with information about the schema records PROGRESS creates for tables, columns, and views.

15.4.1 Tables

A *table* is a collection of information organized into named columns. The following SQL statements define, modify, and delete tables:

CREATE TABLE	Creates a new, empty table and defines the columns and their data types.
ALTER TABLE	Adds new columns, deletes existing columns, or changes the characteristics of existing columns.
DROP TABLE	Deletes a table from the database.

When you create a new database, the security administrator in the Data Dictionary is automatically set to an asterisk (*), which means that all users can create tables. You can limit the users who can create tables by replacing the asterisk with a list of user names. Then, only those users can create tables. See Chapter 11 for additional information about designating security administrators.

When you create a table, you become the *owner* of that table. The owner is the first name in the privilege fields of the schema files and, by default, is the only user who can access the table. The owner cannot be changed.

The owner of a table can grant and revoke access privileges by means of the GRANT and REVOKE statements. In addition to controlling access privileges, the owner of a table is the only user who can alter or delete the table, and create or delete associated indexes.

Creating Tables. The CREATE TABLE statement creates a new, empty table with the columns that you specify. It has the following syntax:

SYNTAX

```
CREATE TABLE table-name ({{ column-name datatype [ NOT NULL [UNIQUE] ]
[[NOT]CASE-SENSITIVE][FORMAT string] [LABEL string] [DEFAULT value]
[COLUMN-LABEL string [! string]...] } | { UNIQUE ( column-name [,...] ) } } [,...] )
```

With the CREATE TABLE statement, you enter a name for the table and define one or more columns. No two columns can have the same name. For each column you name, you must enter a data type for that column. You can enter the CASE-SENSITIVE, FORMAT, LABEL, DEFAULT, and COLUMN-LABEL keywords in any order. You cannot create tables for foreign databases.

You can use the optional keywords NOT NULL to indicate that the column must have a value and UNIQUE to indicate that all values in the column must be unique.

The optional keyword CASE-SENSITIVE ensures the column is case sensitive, in accordance with the ANSI standard. If you indicate that the column is case sensitive, any comparisons with the column are case sensitive as well. For example, the comparison "IF CONTACT = NAME" is case sensitive if either the column CONTACT or NAME is case sensitive. If you attempt to use the CASE-SENSITIVE keyword with non-character columns you receive a warning error and the keyword is ignored. You use NOT CASE-SENSITIVE when you want to disable the case sensitive default created by using the ANSI SQL (-Q) startup option.

Including the UNIQUE keyword in an individual column definition is equivalent to including that column in a separate UNIQUE clause. When you use the keyword UNIQUE to specify that a column or combination of columns must be unique, PROGRESS automatically creates an index on the column or columns and adds records to the _Index and _Index-Field schema files. The index is named `sql-uniq`*n*, where *n* is a number assigned by PROGRESS. You should consider the following points when creating a unique index:

- The first unique column you specify in the CREATE TABLE statement becomes the primary index. If you do not specify a unique constraint on any column, PROGRESS creates a default index named `sql-default`, which orders rows by their internal RECID.
- If you do not specify any unique columns in the CREATE TABLE statement, you can create indexes later on.
- You cannot use the DROP INDEX statement to remove an index created with the UNIQUE option in a CREATE TABLE statement. This is because such an index is part of the basic table definition and its deletion removes the unique constraint check.
- If you include a column in a separate UNIQUE clause, you must also specify that the column is NOT NULL in the individual column definition.

The following example creates a table named `cust_table`. The `cust_num` field is indexed because of the UNIQUE option, and its column definition is specified as NOT NULL.

```
CREATE TABLE cust_table
    (cust_num INTEGER NOT NULL,
     name CHARACTER (30),
     address CHARACTER (30),
     city CHARACTER (15),
     state CHARACTER (2),
     phone CHARACTER (10),
     max_credit DECIMAL (6),
     curr_bal DECIMAL (6),
     UNIQUE (cust_num) ).
```

You can also optionally describe how the data appears on the screen or in a printed report by using the PROGRESS syntax for display formats and labels.

Altering Tables. The ALTER TABLE statement allows you to add new columns to a table, delete columns from a table, or change the format and labels associated with an existing column. The ALTER TABLE statement uses the following syntax:

SYNTAX

```
ALTER TABLE table-name  { ADD COLUMN column-name datatype
                            [NOT NULL [UNIQUE]] [[NOT] CASE-SENSITIVE]
                            [ FORMAT string ][ LABEL string ]
                            [DEFAULT value]
                            [ COLUMN-LABEL string [! string] ... ]
                          DROP COLUMN column-name
                          ALTER COLUMN column-name
                            [[NOT] CASE-SENSITIVE]
                            [ FORMAT string ] [ LABEL string ]
                            [ COLUMN-LABEL string [! string] ... ]
                            [DEFAULT value]                              }
```

You can enter the CASE-SENSITIVE, FORMAT, LABEL, DEFAULT, and COLUMN-LABEL keywords in any order. You cannot alter tables in foreign databases.

The following example adds a new column to a table named `cust_table`:

```
ALTER TABLE cust_table
    ADD COLUMN sales_rep CHARACTER (3)
        LABEL 'Sales Rep'.
```

When you add a new column to an existing table, PROGRESS inserts null or default values in all existing rows for the new column. Therefore, you cannot use the NOT NULL clause in the ALTER TABLE statement unless a default is specified. You can, however, replace the column's null values by using an UPDATE statement following the ALTER TABLE statement. In addition, you cannot use the UNIQUE qualifier if the table already contains data rows, because all rows receive the same value for the new column.

If you have altered a table using the ALTER TABLE statement, do not use SQL data manipulation statements on that table within the same procedure. If you do, PROGRESS displays the message: `DICTIONARY CHANGES COMPLETE, RESTARTING SESSION.` The rest of the procedure may not execute correctly or completely.

As with the CREATE TABLE statement, you can use the PROGRESS FORMAT, LABEL, and COLUMN-LABEL format phrases to specify display formats and labels for column data. Refer to the *PROGRESS Language Reference* manual for information about the FORMAT, LABEL, and COLUMN-LABEL format phrases.

Deleting Tables. The DROP TABLE statement deletes a table from the database. It also deletes all indexes defined on that table, all access privileges, and all data associated with the table.

SYNTAX

DROP TABLE *table-name*

The following example deletes a table named `cust_table`:

```
DROP TABLE cust_table.
```

Only the owner of a table can delete that table. SQL does not allow you to delete a table that is referenced in a view definition.

15.4.2 Data Types

When you define columns in a CREATE TABLE or ALTER TABLE statement, you must specify a data type for each new column. The data type determines the kind of data (such as characters, digits, or a date) that a column can store. The following are SQL data types:

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

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

SMALLINT — Currently maps to the INTEGER data type.

DECIMAL[(*m*[,*n*])] — A decimal number up to 50 digits in length. The precision (*m*) specifies the number of significant digits. The scale (*n*) determines the number of digits to the right of the decimal point. The scale can be up to 10 digits. PROGRESS/SQL truncates the value to the specified scale before storing it in the column. You can abbreviate DECIMAL to DEC.

FLOAT [(*m*)] — Currently maps to the DECIMAL data type. By default, the FLOAT data type receives a sliding point format.

DATE	A date from 1/1/32768 BC through 12/31/32767 AD. DATE values must be enclosed within either single or double quotation marks. You can specify dates in this century with the format *mm/dd/yy* or with *mm/dd/yyyy*. Dates in other centuries require the format *mm/dd/yyyy*.
LOGICAL	Yes/no or true/false values.
REAL	Currently maps to the DECIMAL data type.
NUMERIC[(m [,n])]	Equivalent to DECIMAL.

PROGRESS/SQL rejects any attempt to violate data type specifications. For example, a character string value cannot be inserted into a column defined as DECIMAL.

15.4.3 Views

A *view* is a "window" into one or more tables. To the user, a view looks like a table, but it does not exist as such. A view does not represent physical data of its own, instead it represents the data in the table or tables used to define it.

The following statements define and delete views:

CREATE VIEW	Creates a view, which is a "window" into columns and rows of one or more existing tables or other views.
DROP VIEW	Deletes a view from the database.

To create a view you must have at least the SELECT privilege on the tables referenced by the view (see section, "Access Privileges", for more information). When you create a view, you become its owner. If the view is updatable, the owner's initial privileges are the same as the privileges held on the underlying table. If the view is not updatable, the owner's initial privilege is SELECT only. The section, "Updating Views", describes the conditions that determine whether a view is updatable.

The owner of a view has the GRANT OPTION on these privileges only if the owner also has the GRANT OPTION on the corresponding privileges on the underlying tables. Privileges inherited from the underlying tables are those in effect at the time the view is created. Changes to privileges in the underlying tables do not affect the privileges in existing views.

Once you have created a view, you can use it as if it were a table. You can update the underlying table through the view as long as the view is updatable.

You cannot create an index on a view or alter a view. In addition, only the owner of a view can delete that view.

If you want to see information that is stored for a particular view, you can select Browse Views from the SQL menu in the Data Dictionary, or use SELECT statements to extract the fields from the _View schema file. Or, you can run a procedure called `viewer.p` to display the view definition with user-friendly menus. To access these menus do the following:

1. If this is the first time you are running your PROGRESS application, you must unpack the SQL procedures in the Procedure Library before you can access the view definition menus. For more information about using the unpack utility, refer to the *PROGRESS Installation Notes*, "A Note About PROGRESS" section.

2. Enter `RUN viewer.p` in the PROGRESS editor, and press F1 to run the procedure. The Dictionary View Display menu appears, and you can enter the name of the view definition you want to see.

Creating Views. The CREATE VIEW statement creates a view based on one or more tables, views or both. It has the following syntax:

SYNTAX

CREATE VIEW *view-name* [(*column-list*)] AS *SELECT-statement*
[WITH CHECK OPTION]

The view name you specify must be unique among all existing table names and view names in the database. A view of a foreign database created with CREATE VIEW is stored in the schema-holder, not in the foreign database.

The SELECT statement in the CREATE VIEW statement determines the tables and/or other views used to create the view and the columns that appear in the view. The view column names can be the same as the table column names, or they can be different. To use the table column names as the view column names, omit the *column-list*. To use different column names in the view, enter the column names separated by commas. If you are using different column names, the names must not contain any characters invalid in SQL. Note, however, that you must supply a *column-list* if there are any expressions or duplicate column names in the select list.

The following example creates a view that includes only customers from New York.

```
CREATE VIEW ny_view
    AS SELECT * FROM cust_table
        WHERE st = 'NY'.
```

The keywords WITH CHECK OPTION ensure that all updates to the view (for example, inserting a new row or updating an existing row) satisfy the view-defining condition. The view-defining condition is found in the WHERE clause of the SELECT statement in the view definition. The error message "VIEW INSERT OR UPDATE VIOLATES THE VIEW DEFINITION" occurs at run-time if a view update is rejected.

The following example shows how the WITH CHECK OPTION clause is used to select only those customers from Massachusetts.

```
CREATE VIEW View_Mass
    AS SELECT Name,City, st FROM cust_table
        WHERE st = 'MA' WITH CHECK OPTION.
```

Without the WITH CHECK OPTION clause, a row could be inserted or an existing row updated through this view with a value for the ST column other than MA. Those rows would not appear in the view (although they would exist in the database). If the CHECK OPTION is specified, such inserts or updates will be rejected.

The CHECK OPTION is especially useful in situations where an administrator creates a view as a security feature. For example, assume a security administrator creates a view for clerical use. All new customers must be entered through this view:

```
CREATE VIEW View_Num
    AS SELECT * FROM CUSTOMER
        WHERE CUST-NUM BETWEEN 1000 AND 9999
        WITH CHECK OPTION.
```

The CHECK OPTION enforces the rule that customer numbers must fall in the range 1000-9999. In addition, the CHECK OPTION prohibits an existing number to be changed or a new number to be inserted if it is outside the range.

You cannot specify the CHECK OPTION on a non-updatable view. If you try, the error message "CHECK OPTION NOT PERMITTED ON NON-UPDATABLE VIEWS" appears for the CREATE VIEW statement.

You can nest views in a view definition, that is, a view can be defined in terms of another view. If the top-level view contains the CHECK OPTION, then the WHERE clauses for all the nested views are enforced for INSERT and UPDATE. If no CHECK OPTION is specified for the top-level view, then the WHERE clauses for any underlying view with the CHECK OPTION are enforced instead.

Deleting Views. The DROP VIEW statement deletes one or more views from the database. The DROP VIEW statement has the following syntax:

SYNTAX

```
DROP VIEW view-name
```

The following example deletes a view named ny_view:

```
DROP VIEW ny_view.
```

If other views have been defined in relation to the view you are deleting, PROGRESS/SQL does not allow the deletion. The underlying tables are unaffected by dropping the view.

Updating Views. When you update a view, you update the underlying table for the view. The same SQL data manipulation statements update rows in views and tables. For example, to add new rows through a view you use the INSERT INTO statement. You substitute the view name for the table name to manipulate the rows.

Updates to views may be limited by the view definitions used to create them. Your choice of keywords, expressions, constants, and aggregate functions in the SELECT statement determines whether a view can be updated. You can specifically impose restrictions on updating views by using the CHECK OPTION keywords as a runtime security feature.

You can update the underlying database through the view as long as the view is updatable. A view is updatable only if the following restrictions are observed:

- The view definition must not include the keyword DISTINCT, a GROUP BY clause, a HAVING clause, or contain expressions, constants, or aggregate functions in the SELECT list.
- The FROM clause of the SELECT statement in the view definition must specify only one table.
- The WHERE clause, if any, must not include a subquery.
- If the FROM clause references a view, that view must be updatable.
- If a view is defined with the CHECK OPTION, you should know the criteria for the view-defining condition before attempting to update and insert rows. If you try to update a view that does not satisfy the view-defining condition, the update (or insert) is rejected, and you receive the error message, "VIEW INSERT OR UPDATE VIOLATES THE VIEW DEFINITION".

NOTE: With ORACLE, you cannot update a view whose definition contains a join operation.

15.4.4 Indexes

Indexes provide for more efficient sorting of rows and optimize the performance of queries. In addition to using the UNIQUE option in a CREATE TABLE statement to create indexes, the following SQL statements define and delete indexes:

CREATE INDEX — Creates an index on one or more columns.

DROP INDEX — Deletes an index.

Only the owner of a table can create and delete indexes on that table.

Creating Indexes. The CREATE INDEX statement creates an index on one or more columns. It has the following syntax:

SYNTAX

CREATE [UNIQUE] INDEX *index-name* ON *table-name* (*column-list*)

The following example creates a unique index on the `order-num` and `order-line` columns of the `order-line` table.

```
CREATE UNIQUE INDEX po
    ON order-line (order-num, line-num).
```

The UNIQUE option places a restriction on the column or columns of the index. The restriction requires that the values in the column or combination of columns be unique; no duplicate entries are allowed. In the preceding example, the keyword UNIQUE ensures that each set of `order-num` and `line-num` values will be unique within the table.

The index name must be unique within the database.

Deleting Indexes. The DROP INDEX statement deletes one or more indexes from the database. It has the following syntax:

SYNTAX

DROP INDEX *index-name*

The following example deletes an index named po.

```
DROP INDEX po.
```

You cannot use the DROP INDEX statement to delete an index created with the UNIQUE option in a CREATE TABLE statement.

15.4.5 Access Privileges

To perform operations on tables and views, you must have the appropriate *privileges*. Privileges consist of a combination of object (table, view, or column) and one or more operations. The operations are SELECT, INSERT, UPDATE, DELETE, and ALL PRIVILEGES.

The owner (creator) of a table automatically holds all privileges on that table. The privileges held by the owner of a view depend on the privileges the view owner holds on the underlying tables and views. If the owner holds INSERT, UPDATE, and/or DELETE privileges on the underlying tables and views and the view is updatable, then the owner holds those same privileges on the view. Otherwise, the owner holds only the SELECT privilege on the view.

As the owner of a table or view, you can grant privileges to other users by means of the GRANT statement. With the WITH GRANT OPTION clause, you can also allow the recipients of the privileges to grant those same privileges to other users. The owner and any user with the GRANT OPTION can revoke privileges by means of the REVOKE statement. No privileges can be revoked from the owner.

Granting Privileges. The GRANT statement grants access privileges to users of a table or view, and optionally grants the recipient the right to grant those privileges to other users. The GRANT statement has the following syntax:

SYNTAX

```
GRANT
   { ALL [PRIVILEGES] |
    { SELECT |
      INSERT |
     DELETE |
     { UPDATE [(column-list)] }       } [,...] }
   ON table-name TO { grantee-list | PUBLIC } [ WITH GRANT OPTION ]
```

The *grantee-list* is either a list of user names separated by commas, or the keyword PUBLIC. The keyword PUBLIC grants the specified privileges to all users. It is equivalent to the use of the asterisk (*) in PROGRESS privilege fields.

Table 15-5 lists the PROGRESS/SQL statements and their associated PROGRESS privilege fields.

Table 15-7: SQL Statements and Associated PROGRESS Privileges

SQL Statements	PROGRESS Privilege Fields
SELECT	_Can-read
INSERT	_Can-create
DELETE	_Can-delete
UPDATE	_Can-write

To allow the recipients of the privileges the right to, in turn, grant those privileges to other users, use the WITH GRANT OPTION keywords. In the privilege lists of the schema files, PROGRESS uses a pound sign (#) preceding the user name to indicate that the user *does not* have the GRANT option. The absence of the pound sign means that the user holds the GRANT option and can grant the privilege to other users.

The following statement grants all privileges on the `cust_table` to users `freligh` and `wilson`. It also allows them to grant any of the privileges to other users.

```
GRANT ALL PRIVILEGES
    ON cust_table
    TO freligh, wilson
    WITH GRANT OPTION.
```

This next example grants user `garcia` the SELECT privilege on all columns and the UPDATE privilege on only the `max_credit` column of the `cust_table`.

```
GRANT SELECT, UPDATE (max_credit)
    ON cust_table
    TO garcia.
```

Revoking Privileges. The REVOKE statement revokes either particular privileges or all privileges previously granted to one or more users. The owner and any user with the GRANT OPTION can revoke privileges. The REVOKE statement has the following syntax:

SYNTAX

```
REVOKE
  { ALL [PRIVILEGES] |
   { SELECT |
     INSERT |
    DELETE |
    { UPDATE [(column-list)] }       } [,...] }
  ON table-name FROM {grantee-list | PUBLIC}
```

The *grantee-list* is either a list of user names separated by commas, or the keyword PUBLIC. The keyword PUBLIC revokes the specified privileges from all users, except those users to whom privileges have been individually granted. No privileges can be revoked from the owner.

The following example revokes DELETE and UPDATE privileges on the `cust_table` from users `jones` and `brown`.

```
REVOKE DELETE, UPDATE
    ON cust_table
    FROM jones, brown.
```

15.4.6 PROGRESS Schema Files

PROGRESS schema files contain important information about tables, columns, and views. The information includes indication of table ownership, privileges, and indexes, and the status of various flags. When SQL data definition statements are processed, syntax is generated to create the appropriate schema records and give values to certain schema fields.

The schema records created for SQL tables have the same format as the schema records that PROGRESS uses although some fields in the _File and _Field schema files are not used with SQL-created tables. Views, however, have their own associated schema files: _View, _View-Col, and _View-Ref.

Schema file names all begin with the underscore character (_). All fields within the schema files also begin with the underscore character.

Schema file privilege fields (such as _Can-create) contain a pound sign (#) in front of the user name if the user has that privilege but cannot grant it to or revoke it from other users. In all other respects, the form of the privilege fields is the same as in PROGRESS.

You can update certain schema fields in the PROGRESS Data Dictionary even if they are created using the SQL CREATE TABLE statement. These include the description and help fields and other fields not used by SQL. The Data Dictionary screens are automatically adjusted for SQL tables to indicate which schema fields can be updated through the Data Dictionary.

Schema Files for Tables and Columns. The fields in the schema files, _File, _Field, _Index, and _Index-Field are updated directly by the SQL data definition statements specified in your procedures. Tables 15-6, and 15-7 list the information contained in the _File, and _Field schema files. The _Index and _Index-Field schema files contain primarily system information so they are not documented here.

Table 15-6 lists and describes the fields in the the _File schema file.

Table 15-8: Fields in the _File Schema File

Field	Description
_File-Name	The table name, as specified in the CREATE TABLE statement. This value cannot be changed.
_Can-Create	Initially set to the userid of the user who first compiles and runs the CREATE TABLE statement for this table. This first userid cannot be removed or changed. The rest of the field can be modified through the use of GRANT and REVOKE statements.
_Can-Read	Initially set to the owner. Can be modified by the GRANT and REVOKE statements.
_Can-Write	Initially set to * (all users). Write privilege (through the UPDATE statement) is checked at the column level. This value cannot be modified.
_Can-Delete	Initially set to the owner. Can be modified by the GRANT and REVOKE statements.
_DB-Lang	Set to 0 for tables created using the Data Dictionary, and 1 for tables created with SQL. This value cannot be modified.

Table 15-7 lists and describes the fields in the the _Field schema file. The _Field file contains one record for each column in a table.

Table 15-9: Fields in the _Field Schema File

Field	Description
_Field-Name	The column name, as specified in the CREATE TABLE statement. This value cannot be changed.
_Data-Type	The data type of the column, as specified in the CREATE TABLE statement. This value cannot be changed.
_Label	How the column is labeled horizontally. The default value is ? (question mark), which means that the _Field-Name will be used as the label. Can be modified through the ALTER TABLE statement.
_Decimals	Set to the number of digits to the right of the decimal point for columns declared as DECIMAL or NUMERIC in the CREATE TABLE statement. Also used for character columns to store the declared length of the column. This field is set to the maximum for the FLOAT and REAL data types.
_Order	Set to correspond to the order of the column in the CREATE TABLE statement. This indicates the default order, a left-to-right display of the columns in the table. The lowest number is the leftmost column and highest number is the rightmost. This value cannot be modified
_Can-Read	Initialized to all users by the CREATE TABLE statement because the read privilege (through the SELECT statement) is checked at the table level. This value cannot be modified.
_Can-Write	Initialized to the owner in CREATE TABLE statement. Can be modified by the GRANT and REVOKE statements.
_Col-Label	How the column is labeled vertically. Column labels that appear at the top of a column can contain exclamation points (!) to indicate line breaks. Can be modified through the ALTER TABLE statement.
_Initial	Contains default value if there is one. Can be modified through the ALTER TABLE statement.

(Continued on the next page)

Table 15-7: Fields in the _Field Schema File (Continued)

Field	Description
_Fld-Case	Indicates if the field is case sensitive. For tables created with the CREATE TABLE statement, default is NO unless the ANSI SQL (-Q) startup option is used. Field cannot be changed if record is in an index. Can be modified through the ALTER TABLE statement.
_Extent	Always set to 0 from the CREATE TABLE statement. PROGRESS array fields can be referred to but not created in SQL statements.
_Format	Set through the CREATE TABLE or ALTER TABLE statement. If not set explicitly, an appropriate default is used, depending on the declared size of the column. Can be modified through the ALTER TABLE statement.
_Mandatory	Set to YES for a column declared as NOT NULL in the CREATE TABLE statement. Otherwise, set to NO. This value cannot be modified.

Schema Files for Views. Three schema files store information on views: _View, _View-Col, and _View-Ref. The view schema files translate all references into fully-qualified base table terms.

The _View schema file contains view definitions, including view name, the names of base tables referenced by the view, base table translations of the WHERE and GROUP BY clauses (if included), and the actual text of the view definition from the CREATE VIEW statement. Four privilege fields contain read, write, create, and delete privileges. The _Updatable field tells whether the view is updatable.

The _View-Col schema file is the equivalent of the _Field schema file. There is a record for each column in the view definition. Each record contains the view name, column name, equivalent base table column or expression, and two privilege fields.

The _View-Ref schema file is a cross-reference table used to connect views with their underlying tables. A record is created in this file for each base table column referred to by the view. This table provides information on which views are defined in terms of other views, as well as which base tables are used. This schema file prevents you from deleting tables, columns, or views that are referred to in view definitions. You can use this file to look up view dependencies to see, for example, all the view definitions that refer to a particular view, base table, or column. When operating in a multi-database environment, only the view relationships within a single database are stored in that database's _View-Ref file. Dependencies on other databases are not maintained.

Tables 15-8, 15-9, and 15-10 list and describe fields created by the CREATE VIEW statement. Fields in the view schema files should never be modified.

Table 15-10: Fields in the _View Schema File

Field	Description
_Auth-id	The owner (creator) of the view
_View-Name	The name of the view.
_Base-Tables	A list of the base tables referred to directly or indirectly in the FROM clause of the view definition. If the view is defined in terms of other views, these view definitions are resolved to base tables when the CREATE VIEW statement is compiled.
_Where-Cls	The WHERE clause of the view definition, again in terms of base tables. All column references are fully qualified and references to other views are resolved when the CREATE VIEW statement is compiled.
_Group-By	The GROUP BY clause of the view definition, also in base table terms. If the view definition contains the CHECK OPTION, the CHECK OPTION WHERE clause is stored here.
_View-Def	The original view definition as entered in the CREATE VIEW statement. This field is included primarily for documentation purposes so you can see the original view definition. You should never attempt to alter this field. Any quotation marks in the view definition are not included in the _View-Def field. Otherwise, the original text is unchanged, but may be truncated when displayed on the screen.
_Can-Read _Can-Write _Can-Create _Can-Delete	These fields are initialized as for base tables and are modified through the use of the GRANT and REVOKE statements.
_Updatable	When the view is created, SQL determines whether it is updatable according to the ANSI standard. Set to YES or NO.

Table 15-11: Fields in the _View-Col Schema File

Field	Description
_Auth-id	The owner (creator) of the view
_View-Name	The name of the view.
_Col-Name	The name of the column. May be specified explicitly in the view definition, or, if no column list is included, will be the same as the name of the corresponding column in the SELECT list.
_Base-Col	The fully-qualified base table column name or expression to which this view column refers. Fully resolved when the CREATE VIEW statement is compiled.
_Can-Write	Initialized to the owner in the CREATE VIEW statement. Can be modified by the GRANT and REVOKE statements.
_Can-Create	Reserved for future use.
_Vcol-Order	Provides the default, left-to-right display order of the columns. The lowest number represents the leftmost column; the highest number represents the rightmost column.

Table 15-12: Fields in the _View-Ref Schema File

Field	Description
_Auth-id	The owner (creator) of the view
_View-Name	The name of the view whose definition contains the reference.
_Ref-Table	The referenced table or view. This field is not resolved to the base table if the immediate reference is to a view.
_Base-Col	The referenced base table column. Note that this field is resolved to the base table, even if the immediate reference is to a view.

15.5 Performance Notes

For SQL UPDATE, DELETE, and INSERT statements that effect large numbers of records, you can get better performance by using the -Q startup option. When -Q is used, the whole statement (all effected records) are treated as a single transaction.

Chapter 16
PROGRESS and X Windows

This chapter provides information about using PROGRESS to develop applications for the X Window System. In order to understand the information presented in this chapter, you should be familiar with X Window concepts and the X Window environment. This chapter describes:

- How PROGRESS interacts with the X Window System.
- PROGRESS Installation considerations.
- PROGRESS supported character sets for the X Window System.
- How to run PROGRESS with X Windows to:
  - Start PROGRESS.
  - Use windows with PROGRESS applications.
  - Use icons to close PROGRESS applications.
  - Use a mouse with PROGRESS applications.

16.1 INTRODUCTION

The information outlined in this chapter represents Progress Software Corporation's initial step toward providing a graphic user interface for PROGRESS applications. PROGRESS developers can create PROGRESS applications that work in windows and that utilize many of the user interface (icons, mouse, etc.) tools supplied in window systems.

A *window system* is software responsible for allocating rectangular areas of a display to window applications. It also manages the location and size of each window, the window border, the mouse and cursor interface, the operations for moving and resizing windows, and clipping information for each window. With a window system, you can run several PROGRESS applications in different windows on a single screen display using a mouse and keyboard simultaneously.

The X Window System has emerged as a standard windowing system. This window system is designed to be device-independent, portable across operating systems, and provide extensibility. The X Window System is a network-based windowing system. A *display server* provides the display capabilities and manages user input on a particular bit-mapped screen device for clients. *Clients* are applications, perhaps PROGRESS applications, that interface with the X system library (libXll.a), which interacts with the display server. The display server can be on the same machine as the client or on a remote machine. Thus, you can run your PROGRESS application on a remote system with all the graphics displayed on your local machine.

Applications can interface with the X system library directly (low level interface) or through a toolkit interface. A *toolkit* is a software layer on top of the X Window System that consists of a predefined set of user interface objects (such as buttons, menus, scroll bars, etc.) that typically enforce a consistent "look and feel" across all the applications that use it.

Presently, PROGRESS Version 6 products interface directly with the X system library and do not support the use of toolkits. However, you can use any X window manager (Motif, OpenLook, uwm, etc.) to enforce general window display characteristics for PROGRESS applications. (For example, Figure 16-1 shows PROGRESS running in a Motif window, while Figure 16-2 shows an OpenLook window.)

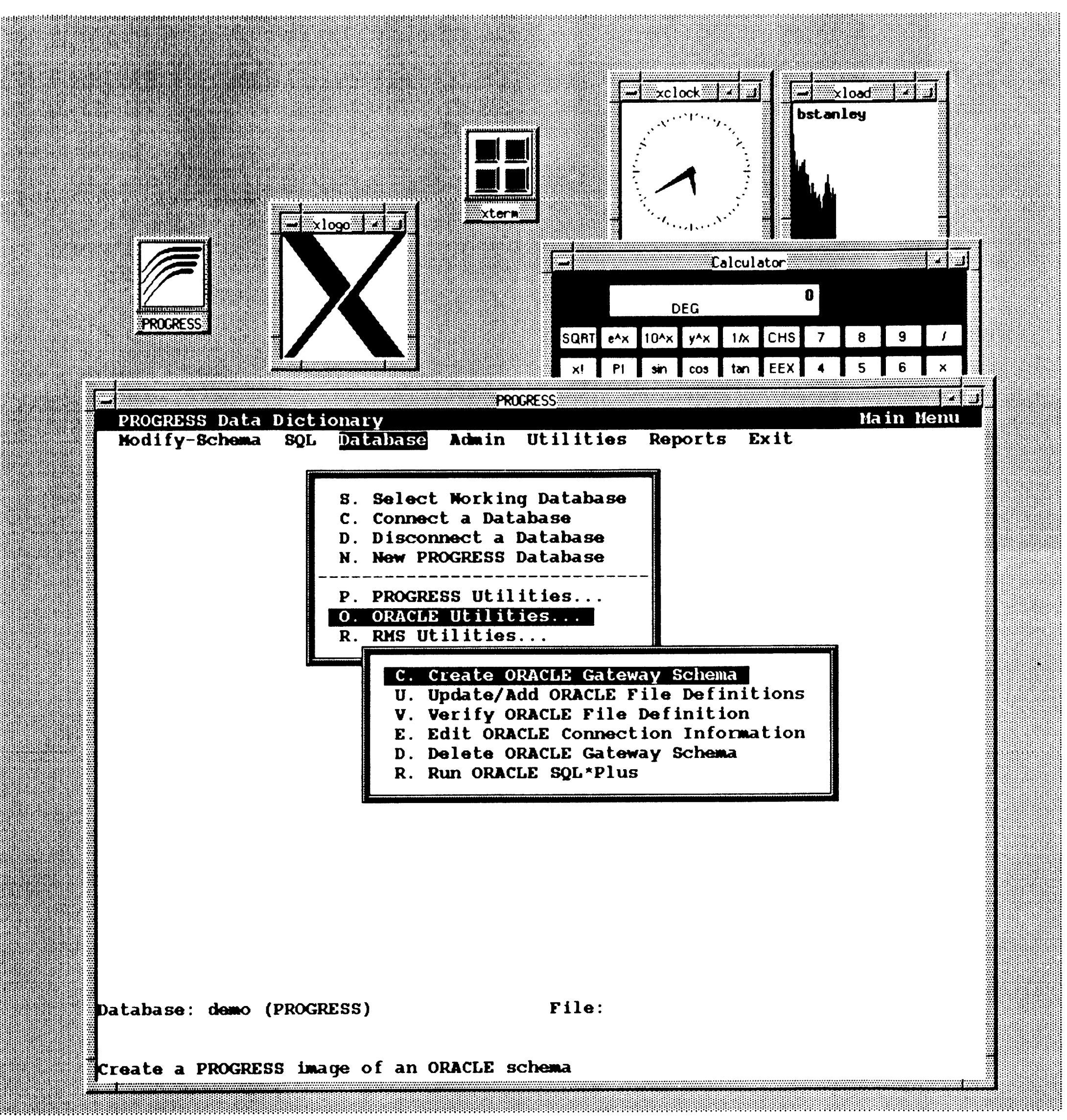


Figure 16–1: Sample Motif Window

16
X Windows

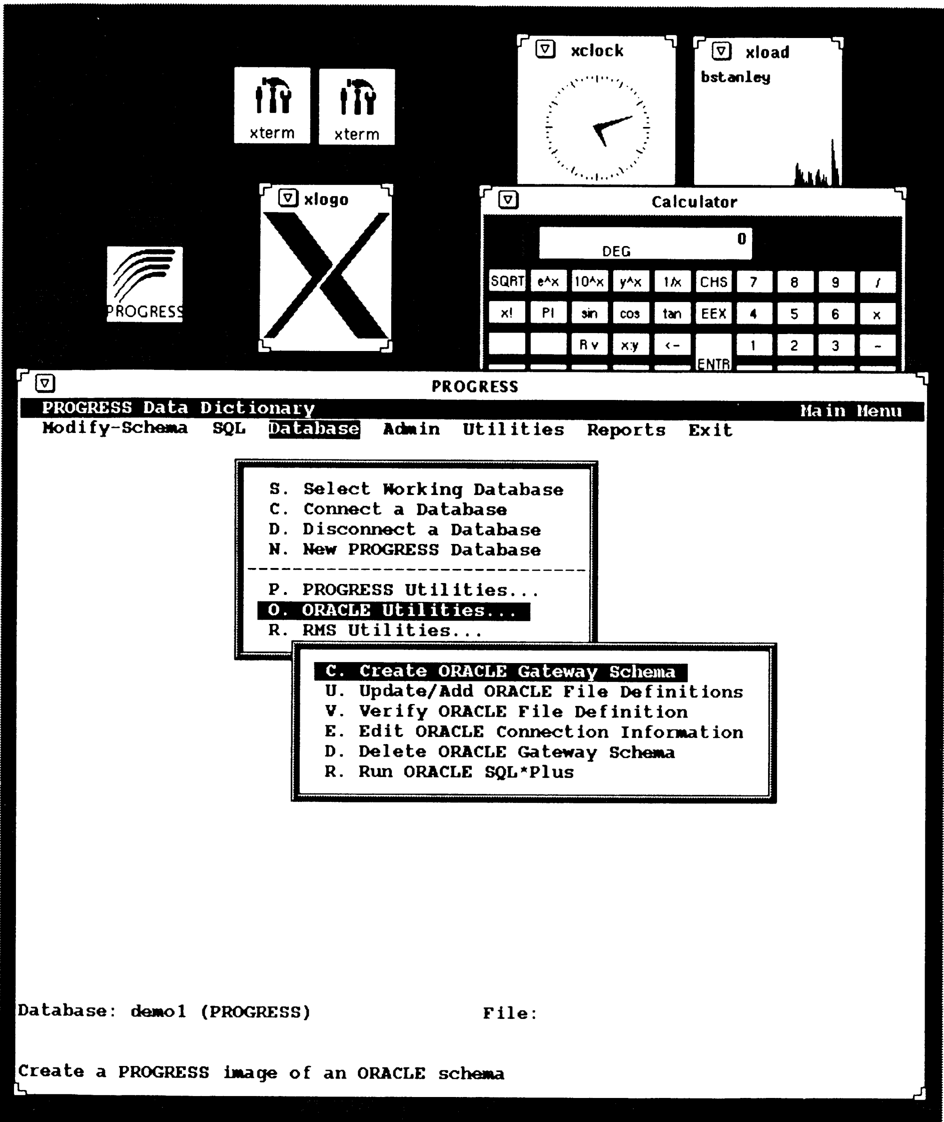


Figure 16–2: Sample OpenLook Window

A *window manager* is an X client that runs simultaneously with other X clients and determines the general characteristics of all windows on the display. The window manager determines the appearance of window borders and title bars, and if windows are overlapping or tiled on the display. It also supplies popup or pushdown menus to allow you to move, resize, or iconify a window and control the display.

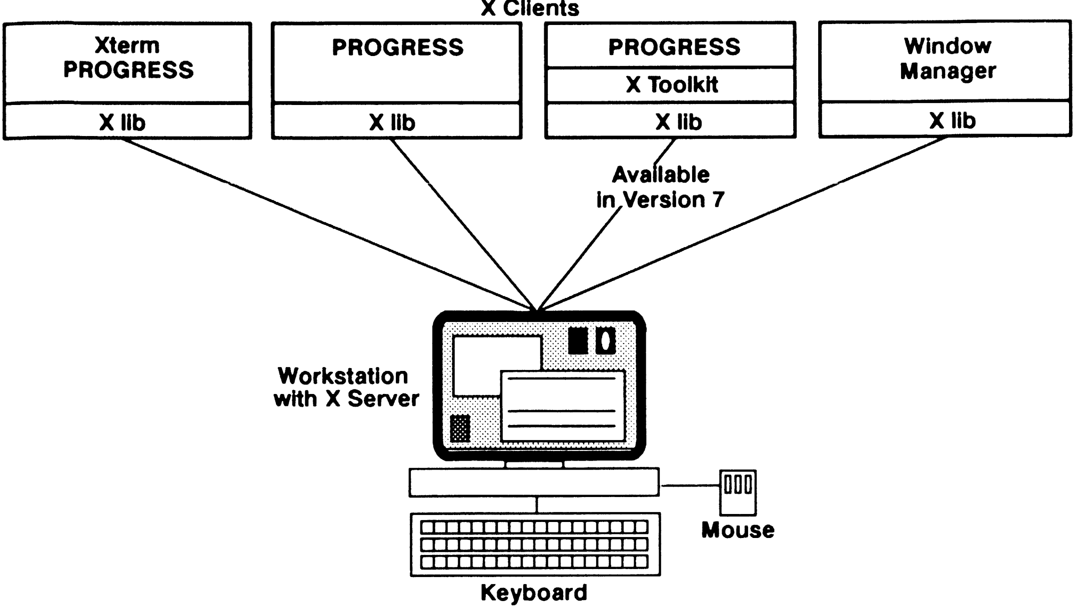


Figure 16–3: PROGRESS and X Windows

16.2 INSTALLATION CONSIDERATIONS

The PROGRESS installation and configuration procedure requires that you properly install your run-time X environment (the X server, library, fonts, color name list, etc.) in order to build the proper PROGRESS executables for the X Window environment. See your X documentation for information about installing and using your X Window System. For more information about the PROGRESS installation and configuration procedure, see Chapter 1 in the *3GL Interface Guide*.

16.3 RUNNING PROGRESS IN X WINDOWS

This section contains the following information about running PROGRESS in an X Window environment:

- Starting PROGRESS.
- Using windows with PROGRESS applications.
- Using icons to close PROGRESS applications.
- Using a mouse with PROGRESS applications.

16.3.1 Starting PROGRESS With Windows

Chapter 2 of *System Administration II: General* provides general information about how to start PROGRESS. This section serves as an addendum to that chapter and describes several window specific startup parameters that you should be aware of before attempting to start a PROGRESS session in an X Window environment. The following table lists these window specific startup parameters:

Table 16–1: PROGRESS Startup Parameters for X Windows

Startup Parameter	Parameter Syntax
Background Color	-bg *color* -background *color*
Border Color	-bd *color* -bordercolor *color*
Border Width	-bw *number* -borderwidth *number*
Display Host	-display *hostname*:*server*[.*screen*]
Font	-fn *fontname* -font *fontname*
Foreground Color	-fg *color* -foreground *color*
Geometry	-geometry *widthxheight ±xoff±yoff*
Iconic	-iconic
Icon	-icon *filename*
Put Screen Compatibility	-psc
Title	-title *string*
X Windows	-ws

Many of these startup parameters are derived from standard X options which are supported by many X applications. The following section provides a description of each of these options. For more information about these parameters, consult your X Window documentation.

Background Color. The Background Color (-bg) startup parameter specifies the color of the window background. The window background consists of space surrounding the application output within the window.

SYNTAX	UNIX	-background *color* -bg *color*			
	Use With	**Max Value**	**Min Value**	**Single-User Default**	**Multi-User Default**
	P,M				

- *color* – A string or hexadecimal number representing a color.

The default background color is white. The *color* can be any of the color names in the `rgb.txt` file that are assigned to standard predefined colors, or you can use a hexadecimal number representing rgb values to specify custom colors not defined in the `rgb.txt` file. The following example sets background color to blue using a string:

```
-background blue
```

The following example sets background color to blue using a hexadecimal number:

```
-bg #0000FF
```

The first two 0's are for the red component, the next two 0's are for the green component, and the last "FF" is for the blue component.

For more information about hexadecimal color specifications and the RGB color model, consult your X documentation.

If you specify a color other than black or white for a monochrome display, the specified color maps to black or white, depending on whether the *color* specified is closer in lightness to black or white. For example, if you specify the color navy for the background color on a monochrome display, the color maps to black.

The background and foreground colors cannot be the same color. If the foreground and background color settings are set to the same color, or default to the same color, the foreground color defaults to the black setting and the background color defaults to the white setting.

Border Color. The Border Color (-bd) startup parameter specifies the *color* of the window border. Some window managers do not support window border specification.

SYNTAX	UNIX	-bordercolor *color* -bd *color*			
	Use With	**Max Value**	**Min Value**	**Single-User Default**	**Multi-User Default**
	P,M				

- *color* – A string or hexadecimal number representing a color. The *color* can be any of the color names in the `rgb.txt` file that are assigned to standard predefined colors, or you can

use a hexadecimal number to specify custom colors not defined in the `rgb.txt` file. See the Background Color (-bg) parameter for examples of color specifications. For more information about hexadecimal color specifications and the RGB color model, consult your X documentation.

If you use -bd to start a PROGRESS session on a system using a window manager that does not support window border specification, the system will ignore the -bd parameter.

Border Width. The Border Width (-bw) startup parameter specifies the width of the window border. Some window managers do not support window border specification.

SYNTAX	UNIX	`-borderwidth` *number* `-bw` *number*			
	Use With	**Max Value**	**Min Value**	**Single-User Default**	**Multi-User Default**
	P,M				

- *number* – A integer number representing width of the window border in pixel units.

If you use the -bw parameter to start a PROGRESS session on a system using a window manager that does not support window border specification, the -bw parameter is ignored.

Display Host. The Display Host (-display) startup parameter allows you to run a window application on a machine remote from the client. It allows you to designate the machine, X server number, and screen device that you want to display the window application.

SYNTAX	UNIX	`-display` *hostname*: *server* [. *screen*]			
	Use With	**Max Value**	**Min Value**	**Single-User Default**	**Multi-User Default**
	P,M				

- *hostname* – The name of the host that displays the graphical output.

- *server* – An integer number representing a server program on *hostname*. The default is 0.

- *screen* – An integer number representing a bit-mapped screen device on *hostname*. This argument is optional. Use it to specify a screen device if there is more than one screen device available to the *hostname* machine. If you do not specify *screen*, the system uses the default screen device.

Here is an example of this startup parameter.

```
-display oz:0
```

Font. The Font (-fn) startup parameter specifies the name of a font used. You can list the available fonts on your X Window System with the `xlsfonts` command.

SYNTAX	UNIX	-font *fontname* -fn *fontname*			
	Use With	**Max Value**	**Min Value**	**Single-User Default**	**Multi-User Default**
	P,M				

- *fontname* – A string representing the default font for output in a window. Currently, PROGRESS only supports fixed width fonts. If you do not specify a font, PROGRESS automatically uses:

  `-adobe-courier-bold-r-normal--14-140-75-75-m-90-i508859-1`

  as the primary default font. If you do not have the primary default font on your machine, the secondary default font is fixed.

Foreground Color. The Foreground Color (-fg) startup parameter specifies the color of the window foreground. The window foreground consists of the application output in the window.

SYNTAX	UNIX	-foreground *color* -fg *color*			
	Use With	**Max Value**	**Min Value**	**Single-User Default**	**Multi-User Default**
	P,M				

- *color* – A string or hexadecimal number representing a color. The default foreground color is black. The *color* can be any of the color names in the `rgb.txt` file that are assigned to standard predefined colors, or you can use a hexadecimal number to specify custom colors

not defined in the `rgb.txt` file. See the Background Color parameter for examples of color specifications. For more information about hexadecimal color specifications and the RGB color model, consult your X documentation.

If you specify a color other than black or white for a monochrome display, the specified color maps to black or white, depending on whether the *color* specified is closer in lightness to black or white. For example, if you specify the color navy for the background color on a monochrome display, the color maps to black.

The background and foreground colors cannot be the same color. If the foreground and background color settings are set to the same color, or default to the same color, the foreground color defaults to the black setting and the background color defaults to the white setting.

Geometry. The Geometry (-geometry) startup parameter specifies the size and placement of the window.

SYNTAX	UNIX	`-geometry` *width* x *height* ± *xoff* ± *yoff*			
	Use With	Max Value	Min Value	Single-User Default	Multi-User Default
	P,M				

- *width* - An integer number representing the width of the window in pixel units.
- *height* - An integer number representing the height of the window in pixel units.
- *xoff* - An integer number representing the horizontal placement (x offset) in pixel units of the left or right window edge from the left or right side of the screen display. When *xoff* is preceded by a plus sign (+), the system measures the horizontal placement of the left window edge from the left side of the screen display. When *xoff* is preceded by a minus sign (-), the system measures the horizontal placement of the right window edge from the right side of the screen display.
- *yoff* - An integer number representing the vertical placement (y offset) in pixel units of the top or bottom window edge from the top or bottom of the screen display. When *yoff* is preceded by a plus sign (+), the system measures the vertical placement of the top window edge from the top of the screen display. When *yoff* is preceded by a minus sign (-), the system measures the vertical placement of the bottom window edge from the bottom of the screen display.

Iconic. The Iconic (-iconic) startup parameter specifies that the window first appears as an icon.

SYNTAX	UNIX	-iconic			
	Use With	Max Value	Min Value	Single-User Default	Multi-User Default
	P,M				

Icon. The Icon (-icon) startup parameter specifies the name of an icon file.

SYNTAX	UNIX	-icon *filename*			
	Use With	Max Value	Min Value	Single-User Default	Multi-User Default
	P,M				

- *filename* - The name of a file that contains an icon definition. The icon must be in X BITMAP format. Use the BITMAP icon editor to design the icon.

If you specify the Icon startup parameter without specifying the Iconic startup parameter, the icon specified with the Icon startup parameter appears when you close the window.

Put Screen Compatibility. The Put Screen Compatibility (-psc) startup option enables window refresh for PROGRESS applications that use PUT screens.

SYNTAX	UNIX	-psc			
	Use With	Max Value	Min Value	Single-User Default	Multi-User Default
	P,M				

When you overlay an X window over another X window that contains a PUT screen, by default the PUT screen does not redraw after the overlay window disappears. The -psc option causes PROGRESS to automatically maintain a raster image of the window. When the overlay window disappears, the overlayed window is restored from the raster image. Be careful, this option gives you greater compatibility but also has an impact on memory and CPU resources, especially on color machines..

Title. The Title (-title) startup parameter specifies the title that appears at the top of a window or at the bottom of an iconic window.

SYNTAX	UNIX	-title *string*			
	Use With	**Max Value**	**Min Value**	**Single-User Default**	**Multi-User Default**
	P,M				

- *string* - The window title. The default title for a PROGRESS session is "PROGRESS".

X Windows. The X Windows (-ws) startup option creates a new X window for a PROGRESS session.

SYNTAX	UNIX	-ws			
	Use With	**Max Value**	**Min Value**	**Single-User Default**	**Multi-User Default**
	P,M				

If you do not use -ws, your PROGRESS session uses the terminal emulator window (xterm) that invoked the PROGRESS startup command.

Starting a PROGRESS Session in X Windows. To start PROGRESS on an X window display, enter a PROGRESS startup command with startup options at the operating system prompt of an `xterm`. For example:

```
pro -1 demo -ws -title "PRODEV" -bg #23238E -fg "light blue"
   -bd "light grey" -geometry  750x400+20+20
```

When you execute the PROGRESS startup command shown above from an xterm, the following occurs:

- A PROGRESS single user session starts for the demo database. (`pro -1 demo`)
- The session creates in a new X window on the display. (-ws)
- The new X window has navy blue as a background color, light blue as a foreground color, and light grey as a border color. (`-bg #23238E -fg "light blue" -bd "light grey"`)
- The new X window is 750 pixels wide, 400 pixels long, and is located in the upper left portion of the display. (`-geometry 750x400+20+20`)

For more information about how to start PROGRESS, see Chapter 2 of *System Administration II: General*.

16.3.2 Windows

When you start an interactive PROGRESS session with the Window startup parameter (-ws), all interaction with the PROGRESS session takes place through a single session window on the display. The display characteristics (location and appearance) of a window are determined by the window manager and can be manipulated with the startup parameters previously described.

Operating system escapes from a PROGRESS session create an `xterm` terminal emulator window to handle all input and output for the escape. To suppress this behavior, use the SILENT option on your operating system escape. For more information about the SILENT option, consult the *PROGRESS Language Reference*.

For general information about using and manipulating windows on your display, consult the documentation supplied with your X Window system.

16.3.3 Icons

The Iconic startup parameter (-iconic) allows you to initially display a window as an icon. PROGRESS supplies a default icon in a file called `logo.bit` in the `dlc` product directory. The Icon startup parameter (-icon) allows you to use an icon other than the default PROGRESS icon. To create or edit an icon, use the BITMAP icon editor.

16.3.4 Using A Mouse With PROGRESS

When you design a PROGRESS application to run in a window environment, there is virtually no window-specific coding required. This means any PROGRESS program that is designed for a non-window version is 100% compatible with the X Windows version of PROGRESS. However, it is important to understand that the mouse becomes available as a source of user input and interaction with your window application. You can use a mouse in a PROGRESS application to position a cursor in an active field on a form or to choose a menu option. You can also define actions to take place when you press a mouse key in your application.

PROGRESS supports the left mouse key (`LEFT-MOUSE-UP`). Run the following procedure and press and release the left mouse key to obtain the key code, key label, and key function for that key.

p-keys1.p

```
REPEAT:
  DISPLAY "Press any key".
  READKEY.
  DISPLAY LASTKEY LABEL "Key Code"
      KEYLABEL(LASTKEY) LABEL "Key Label" FORMAT "x(20)"
      KEYFUNCTION(LASTKEY) LABEL "Key Function" FORMAT "x(20)".
  IF KEYFUNCTION(LASTKEY) = "END-ERROR" THEN LEAVE.
END.
```

The above procedure also displays the use of the READKEY statement and the LASTKEY, KEYLABEL, and KEYFUNCTION functions. When you press and release the left mouse key during a READKEY statement, the READKEY terminates with the LASTKEY set to the left mouse (LEFT-MOUSE-UP) key. For more information about these statements and functions, consult the *PROGRESS Language Reference*.

Users can also use the left mouse key to position the cursor in active fields on a form. For example, run the following procedure to display a form and move the mouse to position the pointer in one of the input fields. When you press and release the left mouse key, the PROGRESS cursor jumps to the field where the pointer is located. You can also press and release the left mouse key in the `sales-rep` field to cycle through available sales representatives.

p-kystk1.p

```
REPEAT:
  PROMPT-FOR customer.cust-num.
  FIND customer USING cust-num.
  UPDATE name SKIP address SKIP city SKIP st SKIP sales-rep
   HELP "Use the left mouse key to select a sales-rep" sales-region
   WITH SIDE-LABELS
      EDITING:
         READKEY.
         IF FRAME-FIELD < > "sales-rep" THEN DO:
             APPLY LASTKEY.
             IF GO-PENDING
             THEN LEAVE.
             NEXT.
         END.
         IF KEYFUNCTION(LASTKEY) = "LEFT-MOUSE-UP" THEN DO:
             FIND NEXT salesrep NO-ERROR.
             IF NOT AVAILABLE salesrep
             THEN FIND FIRST salesrep.
             DISPLAY salesrep.sales-rep @ customer.sales-rep
                     slsrgn @ sales-region.
             NEXT.
         END.
         IF LOOKUP(KEYFUNCTION(LASTKEY),
              "TAB,BACK-TAB,GO,RETURN,END-ERROR") > 0
             THEN APPLY LASTKEY.
             ELSE BELL.
         END.
END.
```

Users can also use the mouse to select items in a CHOOSE statement. For example, the following procedure displays a strip menu with four choices. The CHOOSE statement in this procedure allows the user to select an item from the strip menu. Run this procedure and move the mouse to position the pointer on one of the strip menu options. When you press and release the left mouse button, the strip menu option is selected and PROGRESS attempts to run the program associated with that menu option.

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 10.

REPEAT:
  VIEW FRAME instruc.
  DISPLAY menu WITH NO-LABELS ROW 21 NO-BOX ATTR-SPACE
    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.
```

For more information about the CHOOSE statement, see the *PROGRESS Language Reference.*

You can use the left mouse key within an ON statement in a PROGRESS application. For example, the following procedure attempts to display help information when you press and release the left mouse key:

```
p-onstmt.p

ON LEFT-MOUSE-UP HELP.

FOR EACH customer:
   DISPLAY cust-num.
   UPDATE name max-credit sales-rep.
END.
```

For more information about the ON statement, see the *PROGRESS Language Reference*.

16.4 SUPPORTED CHARACTER SETS FOR THE X WINDOW ENVIRONMENT

In the X window environment, PROGRESS supports the ISO Latin-1 character set for all display functions. PROGRESS remaps the ISO Latin-1 character set to the IBM PC Multi-lingual Code Page 850 (see IBM publication GE19-5356-0) for internal sorting and other purposes. For more information about the PROGRESS character set and extended alphabet support, see Chapter 2.

Appendix A
Using Named Pipes

Named pipes make interprocess communication possible between a non-PROGRESS application (such as a user program or a commercial software package) and a PROGRESS session. Named pipes are also known as FIFOs (First In/First Out) and are available on UNIX systems. When you use named pipes, you can issue SQL requests to a PROGRESS session from within a spreadsheet program and receive data back. Named pipes are an alternative to some of the capabilities provided by PROGRESS Host Language Call Interface (HLC) or Embedded SQL (HLI).

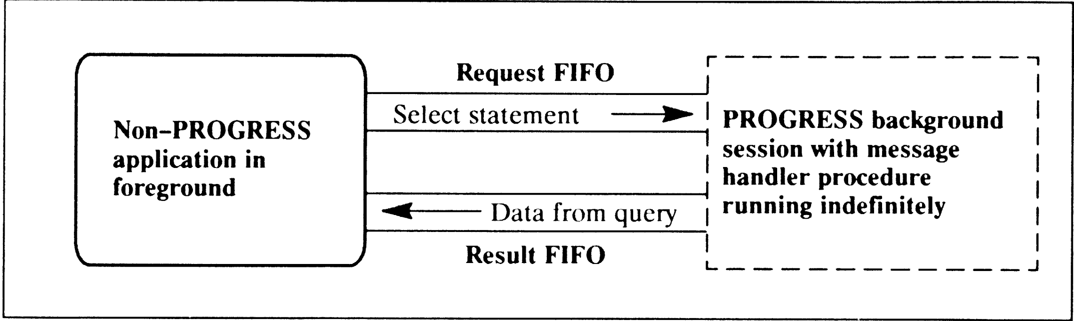


Figure A-1: Typical Named Pipe Scenario

Figure A-1 shows a typical named pipe scenario. To use named pipes, perform the following steps:

1. Create the necessary FIFOs with the UNIX `mknod` command (or the `mknod` system call from within C).

2. Start a PROGRESS session in the background (PROGRESS batch mode) with a message handler procedure running. The message handler procedure runs indefinitely, searching for input, running requests, and shipping off output. (You must supply the message handler procedure. See section A.2 in this chapter for a sample message handler procedure.)

3. Run your non-PROGRESS application. From within the application, issue messages through a named pipe to the background PROGRESS session.

The message handler procedure reads each incoming request, processes it, and returns the results through a second named pipe (called the result FIFO). You can issue any messages you want. The messages can contain SQL statements, PROGRESS statements, procedure names, or anything that your message handler procedure can manage.

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

To PROGRESS, a named pipe looks just like a file. The PROGRESS 4GL statements INPUT FROM and OUTPUT TO access named pipes and files identically.

The following sections, "AN INTRODUCTION TO NAMED PIPES" and "NAMED PIPE EXAMPLES," explain in detail how to perform the previously described steps.

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

A.1 AN INTRODUCTION TO NAMED PIPES (FIFOS)

Named pipes are not documented in detail in UNIX manuals. This chapter provides enough information to get you started using named pipes. You can find more information on named pipes in *Advanced Unix Programming*, by Marc Rochkind (Prentice-Hall,1985). The following sections describe how to:

- Create a named pipe
- Delete a named pipe
- Read from and write to a named pipe

NOTE: Named pipes may implemented differently on your system than as presented below.

A.1.1 Creating a Named Pipe

To create a named pipe, use the `mknod` command from the command line or the `mknod()` system call from a C program. The two methods produce the same results. The examples shown in the "NAMED PIPE EXAMPLES" section below, uses the command line method.

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

The mknod command has more than one version. The version that creates a named pipe has the following syntax:

SYNTAX

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

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

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

SYNTAX

```
mknod mypipe p
```

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

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

A.1.2 Deleting a Named Pipe

Deleting a named pipe uses the same command as deleting a file.

From the command line, use the rm command. For example, to delete the named pipe mypipe, type the following command:

```
rm mypipe
```

From within a C program, use the unlink() system call. Refer to your system documentation for more information on unlink().

Named pipes combine features of pipes and files. Like files, named pipes have names, and any process with appropriate permissions may open it for reading or writing. Because of this, unrelated processes may communicate over a named pipe; access to named pipes is not dependent on inheritance as it is with ordinary pipes. When a named pipe is opened for reading or writing, the syntax used is exactly the same as that used for opening a file. For example, the following line of PROGRESS code opens the previously created named pipe "inpipe" for input:

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

The "NAMED PIPE EXAMPLES" in section A.2 provides additional information on opening name pipes for input and output.

Once opened, named pipes act more like pipes than files. Data written to the named pipe is read in first-in-first-out order (FIFO). So once data written to a named pipe is read, it is removed from the named pipe. Also, the operating system regards individual reads and writes as unbreakable units and issues them one at a time, unless the amount read or written exceeds the capacity of the named pipe. The capacity of a named pipe is the same the capacity of an ordinary pipe. (The capacity of an ordinary pipe depends on the implementation; however, the amount will always be 4096 bytes or greater).

Named pipes are synchronous. When a process opens a named pipe for input, it blocks (waits) until another process opens the same named pipe for output. The reverse is also true; when a process opens a named pipe for output, it blocks until another process opens the same named pipe for input.

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

If multiple processes write messages to the same named pipe, the messages may be interleaved (mixed up). However, as previously mentioned, individual message reads and writes are atomic. (Atomic means that each read and write is an unbreakable unit, issued one at a time.)

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

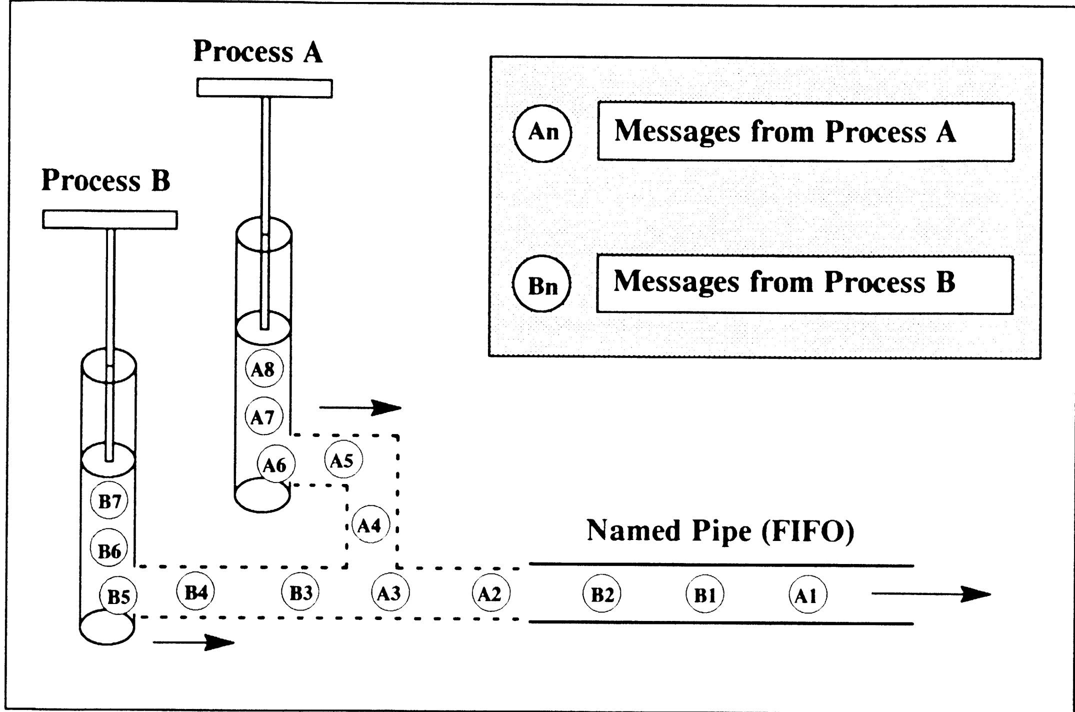


Figure A-2: Writing Messages to a Named Pipe

If two or more processes are simultaneously reading from a named pipe, they may receive alternating messages from the named pipe, as shown in Figure A-2.

A.1.3 Advantages and Disadvantages of Named Pipes

A major advantage of using named pipes is that they provide a useful way to send one-line requests to a PROGRESS background session running a message handler procedure. Multiple users can send requests through the same named pipe, and each request disappears automatically once read. In addition, the message handler procedure can loop indefinitely looking for input, because it blocks (waits) until there is something to read. Finally, output through named pipes saves time in writing a complete response to an ordinary file, closing the file, and then informing the user that the results are available.

A disadvantage with named pipes is that multiple processes cannot use single named pipe to send or receive multi-line messages, unless you define a more complex protocol to control message interleaving. Also, the synchronous nature of named pipes can be helpful in some situations and a problem in others. For example, if the message handler procedure running in the background PROGRESS session starts sending results down the result named pipe, and for some reason the requestor is not there to read them or is slow, the query server cannot move on to read the next request.

You should keep the advantages and disadvantages in mind when you decide on the arrangement most appropriate for your particular application.

A.2 NAMED PIPE EXAMPLES

The following two examples show different uses of named pipes. The first example shows how to create a named pipe, send a message to it, and read the message back all at the shell. The second example shows how to use named pipes with PROGRESS.

Before attempting to perform the examples, follow the instructions in the PREFACE for copying example files. Also, create a copy of the demo database with prodb as shown:

```
prodb demo demo
```

A.2.1 Example 1 – Creating and Using a Named Pipe at the Shell

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

pipex1

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

To see this example, run shell script pipex1.

The contents of pipex1 are shown above. First, the mknod command creates a named pipe called trypipe. Then the cat command opens trypipe for reading. It blocks because trypipe has not yet been opened for writing. Notice that an ampersand (&) is present at the end of the cat command; this runs the cat command as a background process. Next, the echo command opens trypipe for writing and writes a message. The cat command, blocked until now, kicks in, and the message appears on the screen. Finally, the process deletes trypipe.

A.2.2 Example 2 – Using a Named Pipe with PROGRESS

This example shows a simple user program that sends one line requests to a PROGRESS message handler routine running in the background, and displays the results. The example consists of four files: a script called pipex2 that runs the example, the message handler procedure pipex2.p, a sub-procedure do-sql.p, and a C source file called asksql.c for the requestor. Listings of these files are shown below:

pipex2

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

pipex2.p

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

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

do-sql.p

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

{1}
```

asksql.c

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

main()

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

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

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

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

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

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

Before you can run the example, use cc to compile and link the requestor source asksql.c to produce the executable asksql as shown:

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

Once the requestor executable is ready, execute the pipex2 script to run the example:

```
pipex2
```

The pipex2 script first uses mknod to create two named pipes: inpipe and outpipe. Named pipe inpipe carries requests from the requester to the message handler routine. Named pipe outpipe carries results back in the opposite direction, from the message handler to the requestor. Figure A–3 illustrates this process:

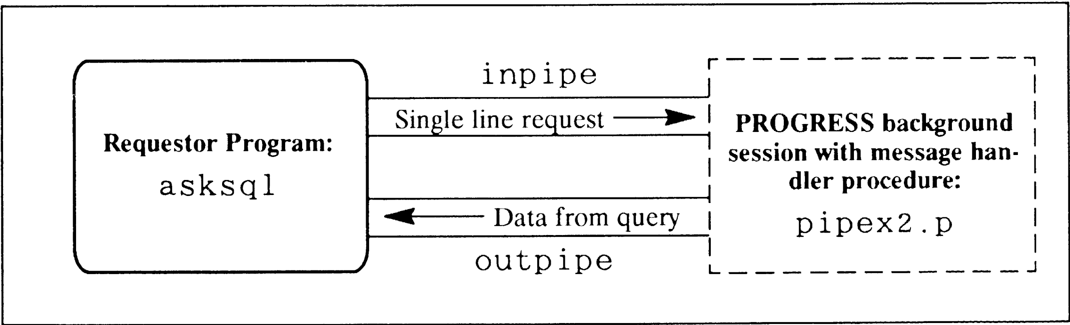


Figure A–3: Scenario Using Two Named Pipes

Next, a background PROGRESS session is run against the demo database, with message handler procedure pipex2.p invoked at startup. The following line sets up input from named pipe inpipe:

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

Notice that PROGRESS does not require a special command to access a named pipe as opposed to a file. In fact, to PROGRESS, named pipes and files look the same.

Message handler routine pipex2.p expects single line requests from requestors, and can handle more than one requestor. Each request contains the name of the output named pipe that the requestor expects to receive results from and a SQL statement set off by quotes. Requests have the form:

output-pipe-name "SQL statement "

In pipex2.p, the line that reads messages from the request pipe is:

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

In general, each requestor must specify a different output named pipe name, or results may be intermixed. (Using the requestor's PID number as part of the name ensures uniqueness. However, for that to work, the requestor would probably need to create its own named pipe using the mknod() system call.)

Once a message has been read from the request pipe, the operating system deletes it from the pipe. Message handler routine pipex2 depends on this feature to work correctly.

After the message is read, output is directed to the output named pipe with the statement,

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

and the SQL statement is compiled and run in the statement:

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

Note that although do-sql.p consists just {1}, it could be extended with formatting statements to avoid having to include these in each query. For example:

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

or

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

After the PROGRESS background process is started up, the user program asksql is run. This program sends requests through named pipe inpipe to the message handler running in the background, and displays the results, returned through the designated output pipe.

When asksql is run, the following message appears:

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

To check whether the example is working properly, enter a simple SQL statement. The following SQL statement accesses the demo database:

```
SELECT name FROM customer.
```

The correct result is a list of sporting goods retailers. If there was no output, you may have entered your SQL statement incorrectly. This causes pipex2.p to terminate. To trap this type of error, write the SQL statement to a file instead of to a named pipe, and compile the file. If the compilation is successful, run it.

To exit from asksql, press RETURN without entering an SQL statement.

The next two commands in script pipex2 terminate the PROGRESS background process:

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

The first command sends "QUIT" to be compiled (instead of an SQL statement). The second command is necessary because, as you may recall from the "INTRODUCTION TO NAMED PIPES (FIFOs)" in section A.1, a process blocks until a named pipe it opened for writing is opened for reading. In this case, the message handler opens named pipe outpipe for writing, and cannot execute the "QUIT" until the cat command opens outpipe for reading.

The code terminating the PROGRESS background process is not part of the asksql requestor program. Multiple copies of asksql can run without affecting the message handler.

Finally, pipex2 uses the rm command to remove the named pipes.

Index

Symbols

.bi file, 1-3

.db file, 1-2

.ld directory, 1-3

.lg file, 1-3, 1-4

.lk file, 1-3, 1-4

(pound sign), in GRANT option, 15-41

% (percent sign), wildcard character, in PROGRESS/SQL, 15-6

+ (concatenation character), in PROGRESS/SQL, 15-3

-db connection parameter, multiple database name, 13-3

_ (underscore character), wildcard character, in PROGRESS/SQL, 15-6

_Field schema file, table of field descriptions, 15-43, 15-44, 15-45

_User file, 11-2, 11-3
  See also Application security
  deleting a user, 11-26
  displaying all users, 11-27
  maintaining, 11-23

_View schema file, 15-45
  table of field descriptions, 15-46

_View-Col schema file, 15-45

_View-Ref schema file, 15-45

A

Activities file, 11-2, 11-6
  See also Application security

Aggregate functions, PROGRESS/SQL, 15-15

Alias(es)
  compiling procedures with, 13-31
  creating in applications, 13-30
  logical database name, 13-29
  shared record buffers with, 13-33

ALL, privilege operation, 15-40

ALL keyword, SELECT statement, 15-8

All-or-nothing processing, 8-3

ALTER TABLE statement, 15-31, 15-33

American National Standards Institute. *See* ANSI

ANSI SQl86 standard, and PROGRESS/ SQL, 15-1

applhelp.p procedure, 3-34

Application databases, and userids, 13-41

Application databases
  See also Database(s)
  changing, 3-1, 3-18
  designing, 3-1
  fields, 3-1
  files, 3-1
    creating, 3-5
    relating to one another, 3-5
  indexing, 3-11

Application security, 11-1
  activities file, 11-2, 11-6
  activities-based checking, 11-6
    See also Activities file
  blank userid, 11-30
  changing your password, 11-27
  checking userids at run-time, 11-4
  creating
    permissions file, 11-7
    the _User file, 11-23
    userids and passwords, 11-24
  deleting a user, 11-26
  designating a security administrator, 11-29
  displaying all users, 11-27

example, 11-21
field validation, 3-26
for fields, 11-14
for files, 11-14
for procedures and activities, 11-14
freezing and unfreezing files, 11-33
introduction, 11-1
login procedures, 11-3
security administration, 11-22
SETUSERID function, 11-3
using the CAN-DO function, 11-10
using the COMPILE statement, 11-14

Applications
debugging, 14-9
multi-user, 12-1, 12-4
testing, 14-9
writing for transport to other operating systems, 14-3

APPLY statement, 6-13

Arrays, changing the definition for, 3-23

AS format phrase option, 7-6

Ascending sort order, 15-13

ASSIGN statement, 1-5

AT format phrase option, 7-6

ATTR-SPACE option
format phrase, 7-6
frame phrase, 7-8
using to reserve spaces, 7-28

Audit trails. *See* Transactions

Auto-Connect, connecting databases with, 13-8

AUTO-RETURN format phrase option, 7-6

AVAILABLE function, 12-16

AVG, function in PROGRESS/SQL, 15-15

B

Batch jobs, establishing a userid for, 11-3

Before-image file (.bi), 1-3, 8-41

BELL statement, 6-13

BLANK format phrase option, 7-6

Blank userid, privileges of user with, 11-30

Block properties, 5-1, 5-2
See also Frames, designing

Blocks, 5-1
and transactions, 8-7
default frames, 5-4, 7-4
default processing, 5-1
DO, 5-1
DO ON ENDKEY, 8-15
DO ON ERROR, 8-15
DO TRANSACTION, 8-15
FOR EACH, 5-1, 5-3
frame scopes, 5-5, 7-24
looping, 5-2
Procedure blocks, 5-1
properties of, 5-1
record scoping, 5-6
REPEAT, 5-1
services for frames, 5-5, 7-24
that read records, 5-3
TRANSACTION option, 8-7, 8-17
transactions and, 5-7
undo processing, 5-8

BREAK keyword, 10-3
See also Control-break reports

Broker, main UNIX server process, 12-2

BTOS/CTOS
See also the PROGRESS System Administration Guide
color monitor support, 2-36
concepts. *See* the PROGRESS Environments Guide
differences from other operating systems, 14-1
removing log file entries, 1-4
starting multi-user PROGRESS, 12-3
stopping a multi-user server, 12-4

Buffers
See also the PROGRESS Tutorial manual
record, 1-5
screen, 1-5

C

CAN-CREATE permission, 11-19

CAN-DELETE permission, 11-19

CAN-DO function
  in PROGRESS/SQL, 15-30
  syntax, 13-43

CAN-DO function
  using for application security, 11-10
  using to check userids, 11-5

CAN-READ permission, 11-18

CAN-WRITE permission, 11-19

CAPS function, 2-37

Case studies, multi-database programming table, 13-46

Case-sensitivity, and indexes, 3-17

CENTERED frame phrase option, 7-8

Changing
  application databases, 3-1
  border characteristics, 2-30
  key definitions, 2-29
  number of lines per screen, 2-29

CHARACTER data type, PROGRESS/SQL, 15-34

Character display formats, 4-4

Character set, 2-36
  allowed characters, 2-37
  collating sequence, 2-37
    for extended characters, 2-42
  extended alphabet support, 2-38
  uppercase-lowercase conversions, 2-37
    for extended characters, 2-42
    for user-defined lanuage, 2-45, 2-46

Character string literal, PROGRESS/SQL, 15-6

Checking data entry for keys, 6-9

CHOOSE statement, 7-15, 7-21
  See also SCROLL statement

CLOSE CURSOR statement, syntax, 15-28

COLON format phrase option, 7-6

COLOR frame phrase option, 7-8

Column
  adding to a table, 15-33
  name qualification, 15-6
  ordinal position designation, 15-12

COLUMN frame phrase option, 7-8

Column name
  qualifiers, 15-6
  view, 15-36

Column names, PROGRESS/SQL. *See* see Identifiers

COLUMN-LABEL format phrase option, 7-6

COLUMNS frame phrase option, 7-8

Commands
  See also the PROGRESS System Administration Guide
  for starting a multi-user server, 12-2
  for starting multi-user PROGRESS, 12-3
  for stopping a multi-user server, 12-4

Comments, in PROGRESS/SQL, 15-5

COMMIT OFF command, 15-29

COMMIT ON command, 15-29

COMMIT STATUS command, 15-29

Commit unit. *See* Transactions

COMPILE statement, using for application security, 11-14

Compiler
  compile-time security, 11-14
  messages, 1-6

Connect parameter(s). *See System Administation II: General*

CONNECT statement, 13-5
  after a DISCONNECT statement, 13-20
  assigning userid/password, 13-42
  NO-ERROR option, 13-25
  syntax, 13-5

CONNECTED function, 13-25

Connecting to databases
  See also Multi-database connection(s)

considerations, 13–10

Connection. *See* Multi-database connection(s)

Connection failure behavior, table, 13–24

Connection failure(s)
  See also Multi-database connection(s)
  reasons for, 13–23

Connection mode(s)
  list of, 13–11
  parameters table, 13–12

Connection overhead, 13–27

Connection parameter(s)
  grouping, 13–3
  list of, 13–3
  multi-database, 13–3

Connections, default userid, for run-time, 13–41

Continuation lines in the editor, 2–9

Control key sequences, default definitions, 2–2, 6–2

Control-break reports, using work files to produce, 10–3

Conventions, typographical, xix

Converting
  between uppercase and lowercase, 2–37
    extended characters, 2–42
    for user-defined lanuage, 2–45, 2–46
  data, 9–10
    using Quoter, 9–11

Copying, data definitions, 3–24

COUNT, function in PROGRESS/SQL, 15–15

CREATE ALIAS statement, 13–29

CREATE INDEX statement, syntax, 15–39

CREATE statement, 1–5

CREATE TABLE statement, 15–31

CREATE VIEW statement, 15–35
  syntax, 15–36
  WITH CHECK OPTION keywords, 15–37

Cross-tab reports, using work files to produce, 10–12

CROSSTALK, for moving files between machines, 14–7
  See also Using the Developer's Toolkit

Cursor, 15–19
  closing, 15–28
  defining, 15–23
  opening, 15–25

Cursor restriction, 15–25

Cursor statements, list of, 15–20

D

Data
  definitions, designing, 3–1
  entry
    controlling, 6–1
    monitoring, 6–1, 6–9
  formats. *See* Display formats
  grouping, 15–12
  handling statements, 1–4
  maintaining, 15–30
  movement, 1–4
  redundancy, 3–4
  selection from multiple tables. *See* Join operation.
  sorting query results, 15–12
  streamlining or normalizing, 3–4
  truncating, 3–23
  types. *See* Display formats

Data definition, PROGRESS/SQL, 15–2

Data Dictionary, 15–2
  changing
    definitions, 3–18
    your password, 11–27
  connecting database with, 13–7
  creating
    permissions file, 11–7
    the _User file, 11–23
    userids and passwords, 11–24
  defining
    field validation, 3–26
    file validation, 3–33

Help, 3-34
deleting users, 11-26
designating a security administrator, 11-29
determining permissions of the blank userid, 11-31
freezing and unfreezing files, 11-33
overview, 1-2
performing security administration, 11-22
See also the PROGRESS System Administration Guide
security, 11-1, 11-2
security administration, 11-22
showing all users, 11-27

Data Dictionary. *See* the PROGRESS Tutorial manual

Data Interchange Format (DIF) files, 9-18

Data retrieval, using cursors, 15-19

Data types, list of PROGRESS/SQL, 15-34

Database definitions, changing, 3-18
array extent of a field, 3-23
data type of a field, 3-23
format of a field, 3-22
index definitions, 3-24
initial value of field, 3-20

Database definitons, changing, copying to another file, 3-24

Database gateway, 13-13

Database location, connection consideration, 13-14

Database Name, connection parameter, 13-3

Database structure, 15-2

Database(s)
changing definitions, 3-18
connect parameter(s), limits, 13-3
distributed. *See* the PROGRESS System Administration Guide
dumping and reloading, for transport, 14-6
field validation, 3-26
file validation, 3-33
files, 1-2
creating, 3-5
Help for fields, 3-34
indexing, 3-11
integrity, 8-1
See also Transactions
multi-user, 12-1
multi-volume. *See* the PROGRESS System Administration Guide
records, sharing, 12-6
security, 11-2
server, 12-1
transporting, 14-6

Datatypes, raw, 13-34

DATE data type, PROGRESS/SQL, 15-35

Date display formats, 4-9

DEBLANK format phrase option, 7-6

Debugging applications, 14-9

DECIMAL data type, PROGRESS/SQL, 15-34

DECLARE CURSOR statement, syntax, 15-23

DEFINE SHARED FRAME statement, 7-11

DEFINE SHARED STREAM statement, 9-26
See also Streams, shared

DEFINE SHARED VARIABLE statement, 7-14

DEFINE STREAM statement, 9-24

DEFINE WORKFILE statement, 10-5, 10-8
See also Work files

Defining
Help for a field, 3-34
large records, 3-11
validation, 3-26
See also Validation
for files, 3-33

DELETE ALIAS statement, 13-30

DELETE statement, 1-5
positioned form, 15-27
privilege operation, 15-40
syntax, 15-19

DESC keyword. *See* Descending sort order.

Index

Descending sort order, 15–13

Designing
  application databases, 3–1
  application files and fields, 3–1

Dictionary. *See* Data Dictionary

Dictionary View Display menu, 15–36

DIF files, converting, 9–19

DISCONNECT statement, 13–19
  affect on aliases, 13–30

Display formats, 4–1
  changing, 3–22
  character, 4–4
  date, 4–9
  defaults, 4–2
    for data types, 4–2
    for expressions, 4–2
    overriding, 4–12
  logical, 4–9
  numeric, 4–6
  testing, 4–12
  time, 4–10

DISPLAY statement, 1–5

DISTINCT keyword, SELECT statement, 15–8

Distributed connection scenario, 13–15
  figure of, 13–16

Distributed databases. *See* the PROGRESS System Administration Guide

Distributed/simultaneous network connection scenario, 13–17

Distributed/simultaneous network scenario, figure of, 13–18

DLC environment variable, 2–20

Documentation, list of publications, xxii

DOS
  See also the PROGRESS System Administration Guide
  concepts. *See* the PROGRESS Environments Guide
  differences from other operating systems, 14–1
  key codes and labels, 2–10
  removing log file entries, 1–4
  starting multi-user PROGRESS, 12–3
  stopping a multi-user server, 12–4
  writing transportable applications, 14–3

DOWN frame phrase option, 7–8, 7–9

DROP INDEX statement, 15–39

DROP TABLE statement, 15–31
  syntax, 15–34

DROP VIEW statement, 15–35
  syntax, 15–38

Dumping databases, for transport to other machines, 14–6
  See also the PROGRESS System Administration Guide and Using the Developer's Toolkit

E

EDITING phrase, 6–10

Editor
  See also the PROGRESS Tutorial
  entering extended characters, 2–39
  using continuation lines, 2–9

ENCODE function, to encode passwords, 11–2

END-ERROR key, 6–8

END-ERROR key, how PROGRESS handles, 8–34

ENDKEY, processing, 6–8
  See also UNDO statement

Endkey
  how PROGRESS handles, 8–29
  how you handle, 8–33

Environment variables
  DLC, 2–20
  PROTERMCAP, 2–21
  TERM, 2–20

Error key
  how PROGRESS handles, 8–26
  how you handle, 8–29

Errors
how PROGRESS handles, 8-23, 8-25
how to handle in procedures, 8-25
processing, 8-1

Example procedures, xix

EXCLUSIVE-LOCK, 13-38

EXCLUSIVE-LOCK record phrase option, 8-7, 12-8, 12-11, 12-13

EXISTS search condition, 15-15

EXPORT statement, 9-21
See also Output

Expressions
See also Literals; Operators
display formats for, 4-2

Extended alphabet support, 2-38

extensions, PROGRESS with PROGRESS/SQL, 15-3

F

Federated connection scenario, figure of, 13-14

FETCH statement, 15-26
using local variables, 15-40

Field(s)
changing, 3-18
array extent of, 3-23
data type, 3-23
format of, 3-22
initial value of, 3-20
designing for applications, 3-1
freezing and unfreezing definitions, 11-33
permissions, 11-19
security, 11-14
validation, 3-26
See also Validation

File(s)
_User, 11-2, 11-23
activities, for application security, 11-2, 11-6
before-image (.bi), 1-3, 8-41
copying data definitions, 3-24
creating, 3-5
database, 1-2
designing for applications, 3-1
DIF, converting, 9-19
for multi-user (.ld), 1-3
freezing and unfreezing definitions, 11-33
input, 9-10
local-before-image (.lbi), 8-42
lock control (.lk), 1-3, 1-4
log (.lg), 1-3, 1-4
permissions, 11-18
PROTERM.DAT (VMS), 2-20
protermcap, 2-18, 2-20
relating to one another, 3-5
See also the Tutorial Manual
security, 11-14
server log (VMS), 1-3
shared memory (VMS), 1-3
SYLK, converting, 9-19
system control, 12-20
temporary database, 1-3
validation, 3-33

Files
schema, 15-42
table of access restrictions, 11-19

Fill characters, in display formats, 4-5

FIND FIRST statement, using to sort, 10-9

FIND statement, 1-5

FLOAT data type, PROGRESS/SQL, 15-34

FOR EACH statement, 1-5

FORM statement
in shared frames, 7-11, 7-14
using to describe frames, 7-10

FORMAT format phrase option, 4-12, 7-7

Format phrase
options, 7-6
using to describe frames, 7-6
VALIDATE options, 3-26

Format phrases, PROGRESS/SQL, 15-3

Formats, display, 4-1
changing, 3-22
character, 4-4

date, 4-9
defaults, 4-2
for data types, 4-2
for expressions, 4-2
overriding, 4-12
logical, 4-9
numeric, 4-6
testing, 4-12
time, 4-10

FRAME frame phrase option, 5-5, 7-9

Frame phrase
options, 7-8
using to describe frames, 7-8
WITH phrase, extension to PROGRESS/SQL, 15-3

Frame(s)
advancing, 7-26
clearing, 7-26
default design, 7-2
default for blocks, 5-4
default services, 7-24
designing, 7-1
down, 7-26
field- and variable-level design, 7-5
for non-terminal devices, 7-27
hiding, 7-26
how statements use, 7-22
overlay, 7-26
See also the PROGRESS Tutorial manual
overriding default design, 7-1, 7-5
repainting, 7-27
scope of, 7-24
scope of shared frames, 7-14
scoping properties, 5-5
scrolling, 7-17
See also SCROLL statement
services for blocks, 5-5
shared, 7-11
single, 7-26
unnamed, 5-4
viewing, 7-26
See also TOP-ONLY, OVERLAY, NO-HIDE frame phrase options

FRAME-DB function, help procedure, 13-39

FRAME-FIELD function, 6-13

Freezing files, permissions needed to, 11-34

FROM clause, SELECT statement, 15-11

Function keys
See also Key(s), functions of
changing definitions, 6-3
default definitions, 2-2, 6-2
monitoring the use of, 6-9

Functions in PROGRESS/SQL, aggregate, 15-15

Functions, PROGRESS
AVAILABLE, 12-16
CAN-DO, 11-5, 11-10
CAPS, 2-37
FRAME -FIELD, 6-13
GO-PENDING, 6-13
IF...THEN...ELSE, 3-30
KBLABEL, 2-17
KEYCODE, 2-9, 2-17, 6-7
KEYFUNCTION, 2-17, 6-7, 6-13
KEYLABEL, 2-9, 2-17, 6-7
LASTKEY, 6-7, 6-11
LC, 2-37
LOCKED, 12-16
LOOKUP, 3-27, 6-13
NO-WAIT, 12-16
SEARCH, 9-4, 9-12
SETUSERID, 11-3
STRING, 4-10
SUBSTRING, 9-14
TERMINAL, 2-19
TIME, 4-10
USERID, 11-6

G

GETBYTE, 13-36

GO-ON phrase, 6-6

GO-PENDING function, 6-13

GRANT statement, 15-40

GROUP BY clause, SELECT statement, 15–12

H

Help
  defining for a field, 3–34
  overriding defaults, 3–35
  PROGRESS, 1–7

HELP format phrase option, 7–7

HIDE statement, 7–26

I

Identifiers, PROGRESS/SQL, 15–5

IF...THEN...ELSE function, 3–30

IMPORT, 9–22

Import. *See* Input

Include files, for field validation, 3–29

Index
  created by UNIQUE, 15–32
  deleting, 15–39
  in PROGRESS/SQL, 15–39
  naming in PROGRESS/SQL. *See* Identifiers

Indexes
  and unknown values, 3–17
  case-sensitive indexing, 3–17
  changing definitions, 3–24
  creating, 3–11
  freezing and unfreezing definitions, 11–33
  generating values for, 12–19
  maintaining, 3–17
  procedure to generate values for, 12–21
  reasons not to define, 3–16
  reasons to define, 3–11, 3–14

Input
  changing the source, 9–6
  file preparation, 9–10
    using Quoter, 9–11
  from DIF files, 9–18
  from multiple sources, 9–8, 9–23
  from SYLK files, 9–18
  streams, 9–2
  working with in procedures, 9–1

INPUT CLOSE statement, 9–10
  using with Quoter, 9–14

INPUT FROM statement
  using with Quoter, 9–14
  using to change the input source, 9–6
  using to redirect input, 9–9

INPUT THROUGH statement, 9–29

INPUT-OUTPUT THROUGH statement, 9–29

Input/Output limits. *See* the PROGRESS System Administration Guide

INSERT, privilege operation, 15–40

INSERT INTO statement, 15–16

INSERT statement, 1–5

INTEGER data type, PROGRESS/SQL, 15–34

istrans.p procedure, 8–40

Iteration, 5–2, 8–7
  See also Looping
  of an outer REPEAT block, 8–10

J

Join operation, 15–11
  self-join, 15–11

Journalling. *See* Transactions

K

KBLABEL function, 2–17

KERMIT, for moving files between machines, 14–7
  See also Using the Developer's Toolkit

Key definitions, redefining, 2–29

Key(s)
  assigning actions to, 6–5

Index

changing the function of, 6-3
codes, 2-9
default definitions, 2-2
described, 2-5
defining special keys, 2-31
END-ERROR, 6-8, 8-34
Endkey, 8-29
Error, 8-26
functions of, 6-2
how PROGRESS handles, 2-1, 6-3
labels, 2-9
alternate, 2-15
use in procedures, 6-2

Keyboard, 2-2
defining special keys, 2-31

KEYCODE function, 2-9, 2-17

KEYCODE function, 6-7

KEYFUNCTION function, 2-17

KEYFUNCTION function, 6-7, 6-13

KEYLABEL function, 2-9, 2-17

KEYLABEL function, 6-7

Keystrokes, monitoring, 6-1, 6-9

L

LABEL format phrase option, 7-7

LASTKEY function, 6-7, 6-11

LC function, 2-37

LIKE clause, in PROGRESS/SQL, 15-6

LIKE format phrase option, 7-7

LIKE option, connected database, 13-38

Literals, PROGRESS/SQL, 15-6

Loading
data, DIF and SYLK files, 9-19
data definitions, 14-7

Local database(s), 13-14

local-before-image file (.lbi), 8-42

Lock control file (.lk), 1-3

LOCKED function, 12-16

Locking records, how PROGRESS handles, 8-7

Locks. *See* Record locks

Log file (.lg), 1-3

Logging. *See* Transactions

LOGICAL data type, PROGRESS/SQL, 15-35

Logical display formats, 4-9

Login procedures, 11-3

login.p procedure, 11-3

LOOKUP function, 3-27

LOOKUP function, 6-13

Looping, 5-2, 8-7
of an outer REPEAT block, 8-10

M

Machine independent applications
creating, 13-22
figure of, 13-22

Matching, character strings in PROGRESS/SQL, 15-6

Menu, to display view definition, 15-36

Menus
See also the PROGRESS Tutorial manual
scrolling, 7-17
strip, 7-15

MESSAGE statement, 7-1

Messages in PROGRESS, 1-6

Monitoring keystrokes, 6-1, 6-9

Multi-database, logical names, 13-10

Multi-database application(s)
activity permissions files, 13-43
help for, 13-39
referencing files and field names, 13-28
run-time security, 13-41
transaction behavior, 13-38

Multi-database connection(s)
application session, 13-1

at startup, 13-4
CONNECTED function, 13-25
connection overhead, 13-27
connection parameters, 13-3
conserving vs. overhead, 13-27
considerations at development, 13-20
database location considerations, 13-14
disconnecting, 13-19
distributed connection scenario, 13-15
distributed/simultaneous networks scenario, 13-17
failure behavior, 13-24
failures and disruptions, 13-23
federated configuration, 13-14
general considerations, 13-10
logical database name, 13-10
methods list, 13-2
modes, 13-11
parameter(s)
Database Name, 13-3
table, 13-3
PROGRESS Functions for, 13-37
run-time considerations, 13-21
to non-PROGRESS databases, 13-13
types of non-PROGRESS, 13-13

Multi-database disconnecting, 13-19

Multi-database programming, 13-1
case studies, 13-45

Multi-database(s)
federated connection scenario, 13-14
logical data base name, 13-19
physical name, 13-5
positioning references in applications, 13-26
figure, 13-26

Multi-user
applications, 12-4
directory, 1-3
record locks, 12-6
sharing records, 12-6
starting a server, 12-1
starting PROGRESS, 12-3
stopping the server, 12-4
the environment, 12-4

Multi-volume databases. *See* the PROGRESS System Administration Guide

Multiple
input sources, 9-8
output destinations, 9-3

N

Named Pipe(s), blocking, A-4

Named pipe(s)
advantages of, A-5
capacity of, A-4
creating, A-2
with the mknod command, A-3
with the mknod() system call, A-3
deleting, A-3
with the rm command, A-3
with the unlink() system call, A-3
disadvantages of, A-5
examples of, A-6
at the Unix Shell, A-6
using PROGRESS, A-7
interleaving messages, A-4
introduction to, A-2
message handler procedure
funtions of, A-2
sample, A-8
requests, form of, A-10
similarity to text files, A-2, A-4
typical scenario using, A-1
using more than one, A-10
writing messages to, A-5

Network (-N) connect parameter, 13-17
protocol table, 13-17

Networks. *See* the PROGRESS Environments Guide

NEXT statement, 6-13

NO-ATTR-SPACE option
format phrase, 7-7
frame phrase, 7-9
using to keep space open on terminals, 7-28

NO-BOX frame phrase option, 7-9, 7-27

NO-HIDE frame phrase option, 7-9, 7-27

NO-LABEL format phrase option, 7-7

NO-LABELS frame phrase option, 7-9

NO-LOCK record phrase option, 12-12, 12-13

NO-UNDERLINE frame phrase option, 7-9

NO-UNDO keyword, variables defined with, 8-37

NO-VALIDATE frame phrase option, 7-9

NO-WAIT function, 12-16

Normalization of data, 3-4

NOT NULL keywords, in CREATE TABLE statement, 15-31

Null values, 15-7
  resulting truth values table, 15-7

NUMERIC data type, PROGRESS/SQL, 15-35

Numeric display formats, 4-6

O

ON statement, using to change key function, 8-27, 8-29

ON statement, using to change key function, 6-4

OPEN CURSOR statement, syntax, 15-25

Operating system, security, 11-2

Operating systems
  applications
    writing for tranport to UNIX, 14-5
    writing for transport, 14-3
    writing for transport to VMS, 14-6
  differences between, 14-1
  security, 11-2
    See also the PROGRESS System Administration Guide
  transporting databases, 14-6
    See also Using the Developer's Toolkit

Operators
  PROGRESS/SQL, 15-6
  spacing with, 15-7

ORDER BY clause
  integer in, 15-12
  SELECT statement, 15-12
  treatment of null values, 15-13

Output
  changing the destination, 9-3, 9-21
  streams, 9-2
  to DIF format, 9-20
  to multiple destinations, 9-3, 9-23
  to SYLK format, 9-20
  working with in procedures, 9-1

OUTPUT CLOSE statement, 9-5, 9-6, 9-26

OUTPUT STREAM statement, 9-25

OUTPUT THROUGH statement, 9-29

OUTPUT TO statement, 9-3, 9-5
  See also Streams
  PAGED option, 9-3

OVERLAY frame phrase option, 7-9, 7-26

Owner, restriction on revoking privileges from, 15-42

P

PAGE-BOTTOM frame phrase option, 7-9

PAGE-TOP frame phrase option, 7-10

PAGED option, OUTPUT statement, 9-3

Parameter file(s)
  See also System Administration II: General
  connect parameter (-pf), 13-3
  invoking, 13-3
  on different systems, 13-3

Parameter(s)
  connection(s), database name (-db), 13-3
  multi-database connection, 13-3
  Network connect, distributed/ simultaneous scenario, 13-17

Password, encoding using the ENCODE function, 11-2

Passwords, 11-1
  See also Application security
  changing, 11-27
  establishing, 11-2, 11-24

Permissions
  CAN-CREATE, 11-19
  CAN-DELETE, 11-19
  CAN-READ, 11-18
  CAN-WRITE, 11-19
  field, 11-19
  file, 11-18
  values used to define, 11-19

Permissions file
  See also File(s), permissions
  adding records, 11-8
  security for, 11-12
  using the Data Dictionary to create, 11-7

Physical name, database, 13-5

Positioned DELETE, 15-27

Positioned UPDATE, 15-27

Preparing input files, 9-10

Privilege checking, in PROGRESS/SQL, 15-29

Privileges
  for PROGRESS and PROGRESS/SQL, 15-41
  GRANT statement, 15-40
  granting to other users, 15-40
  initial, 15-35
  objects and operations, 15-40
  REVOKE statement, 15-42
  revoking from other users, 15-42

Privileges of the blank userid, 11-30
  See also Application security

Procedure(s)
  applhelp.p, 3-34
  changing the input source, 9-6
  changing output destination, 9-3
  continuing processing in, 6-5
  debugging, 14-9
  errors, how PROGRESS handles, 8-24
  for generating an index value, 12-21
  Handbook examples, xix
  istrans.p, 8-40
  login, 11-3
  monitoring keystrokes in, 6-1
  programming tips, 14-1
  PROGRESS Library of, 14-10
  prostart.p, 11-3
  sharing streams between, 9-26
  testing, 14-9
  using for security checking, 11-9
  using keys in, 6-2

Processes, as input and output streams, 9-29

Processing
  all-or-nothing, 8-3
  undo, 5-8
    See also Undo processing

Programming, multi-database, 13-1
  techniques and considerations, 13-28

PROGRESS
  and X Windows, 16-1
    figure of, 16-5
    installation considerations, 16-5
    mouse keys with, 16-13
    running PROGRESS in, 16-5
    starting a session in, 16-12
    startup parameters table, 16-6
    supported character set for, 16-17
  character set, 2-36
  components, 1-1
  Data Dictionary, 15-2
  Help, 1-7
  keyboard, 2-2
  keys, 2-1
    See also Key(s)
  messages, 1-6
  overview, 1-1
  Procedure Library, 14-10
  security, 11-1

PROGRESS database functions, table, 13-37

PROGRESS procedure, compiling if containing SQL statements, 15-30

PROGRESS/SQL
  aggregate functions, 15-15
  associated PROGRESS privileges table, 15-41

character set, 15-5
column names, 15-5
comments in, 15-5
compiling and running procedures, 15-30
cursor statements, 15-20
data definition, 15-2
data types, 15-34
identifiers, 15-5
null values in, 15-7
PROGRESS editor with, 15-3
PROGRESS features, 15-1
reserved words, 15-4
as identifiers, 15-5
reserved words list, 15-4
schema records, 15-42
statement syntax, 15-4
statements. *See* SQL statements
table of equivalent PROGRESS terms, 15-2
transaction processing in, 15-29
wildcard characters, 15-6

PROMPT-FOR statement, 1-5

prostart.p procedure, 11-3

Protection for your application. *See* Application security

PROTERM logical, 2-19

PROTERM.DAT file (VMS), 2-20

protermcap
additional entries, 2-25
basic entries, 2-22
defining video display attributes, 2-35
entry for non-English languages, 2-40
example entry, 2-33
file, 2-18, 2-20
making entries, 2-21
variable, 2-20
variable-size window control, 2-29

PUBLIC keyword, GRANT statement, 15-41

PUTBYTE, 13-35

Q

Queries. *See* the PROGRESS Tutorial manual

Query, sorting results using ORDER BY clause, 15-12

Quick User Report, generating, 11-31

Quoter, using interactively, 9-17

Quoter utility, 9-11
example procedure using, 9-13
options
columns in fields, 9-16
field delimiter, 9-15
using with various file formats, 9-15

R

Range variable, 15-11

Range variables, PROGRESS/SQL. *See* Identifiers

RAW, 13-35

Raw Datatype, 13-34

READKEY statement, 6-10, 6-11

REAL data type, PROGRESS/SQL, 15-35

RECID function, with PROGRESS/SQL, 15-3

Record locks, 12-6
AVAILABLE function, 12-16
duration, 12-10, 12-11
example of a common problem, 12-11
how PROGRESS uses, 12-9
in multi-user applications, 12-6
LOCKED function, 12-16
NO-WAIT function, 12-16
releasing, 12-11
resolving conflicts, 12-13
by changing size of transactions, 12-16
types, 12-12
using to avoid record conflicts, 12-6

Record phrase
EXCLUSIVE-LOCK option, 12-8, 12-11, 12-13

NO-LOCK option, 12-13
SHARE-LOCK option, 12-9, 12-11, 12-13

Record(s)
adding to a permissions file, 11-8
checking the existence of, 3-28
defining, 3-11
locks, 8-7, 8-10
retrieving, 5-3, 12-6
See also the Tutorial Manual
scoping, 5-6
sharing, 12-6

Redirecting output streams, 9-3

RELEASE statement, 1-5

Remote database(s), 13-14

Repainting frames, 7-27

Reports
control-break, 10-3
See also the PROGRESS Tutorial
cross-tab, 10-12
spreadsheet, 10-12
tabular, 10-12
using work files to write, 10-3

Reserved words
in PROGRESS/SQL, 15-4
PROGRESS/SQL, 15-4

RETAIN frame phrase option, 7-10, 7-26

Retaining data during RETRY, 7-27
See also RETAIN statement

Retrieval, of records, 12-6
See also the PROGRESS Tutorial manual

Retrieval set, 15-26

Retrieving values, PROGRESS/SQL, 15-26

REVOKE statement, syntax, 15-42

ROW frame phrase option, 7-10, 7-27

Rows
comparing, 15-11
deleting, 15-19
inserting, 15-16
updating, 15-18

Run-time connection, methods list, 13-21

S

Sample procedures, xix

Schema files
for tables and columns, 15-43
for views, 15-45

Schema holder, 13-13

Scoping
for frames, 5-5, 7-24
for records, 5-6

Screen(s)
See also Frame(s)
designing frames for, 7-1
length in lines, 7-1
on spacetaking and nonspacetaking terminals, 7-28
width, 7-1

SCROLL frame phrase option, 7-10, 7-17

SCROLL statement, 7-17, 7-21
See also CHOOSE statement

Search conditions
for set subqueries, 15-14
SELECT statement, WHERE clause, 15-8

SEARCH function, 9-4, 9-12

Security
administration, 11-22
See also the PROGRESS System Administration Guide
changing your password, 11-27
checking
activities-based, 11-6
with procedures, 11-9
checking userids at run-time, 11-4
creating
permissions file, 11-7
the _User file, 11-23
userids and passwords, 11-24
defining field validation, 3-26
defining file validation, 3-33
deleting a user, 11-26
designating a security administrator, 11-29
displaying all users, 11-27
example, 11-21
for applications, 11-1

for fields, 11-14
for files, 11-14
for permissions files, 11-12
for procedures containing SQL statements, 15-29
freezing and unfreezing files, 11-33
operating system, 11-2
See also the PROGRESS System Administration Guide

SELECT, privilege operation, 15-40

SELECT INTO statement. *See* Singleton SELECT.

SELECT statement, 15-7
ALL keyword, 15-8
ASC keyword. *See* Ascending sort order.
DESC keyword. *See* Descending sort order.
DISTINCT keyword, 15-8
FROM clause, 15-11
GROUP BY clause, 15-12
join operations, 15-11
ORDER BY clause, 15-12
WHERE clause, 15-8

Self-join, 15-11

Servers
starting multi-user, 12-1
stopping multi-user, 12-4

SET clause, UPDATE statement, 15-18

SET statement, 1-5

Set subquery, 15-14
search conditions, 15-14

SETUSERID function, syntax, 13-42

SETUSERID function, 11-3

SHARE-LOCK record phrase option, 12-11, 12-13
See also Record locking

Shared frames, 7-11

SIDE-LABELS frame phrase option, 7-10

Singleton SELECT, syntax, 15-28

Singleton subquery, 15-13

SMALLINT data type, PROGRESS/SQL, 15-34

Software failure, how PROGRESS handles, 8-37

Sorting
comparing two methods, 10-11
in ascending order, 15-13
in descending order, 15-13
using FIND FIRST statement, 10-9
using work files, 10-8

Spreadsheet reports. *See* Cross-tab reports

Spreadsheets, converting DIF and SYLK files, 9-18

SQL. *See* PROGRESS/SQL

SQL statements
ALTER TABLE, 15-33
CREATE INDEX, 15-39
CREATE TABLE, 15-31
CREATE VIEW, 15-36
DROP INDEX, 15-39
DROP TABLE, 15-34
DROP VIEW, 15-38
GRANT, 15-40
REVOKE, 15-42
syntax, 15-4

Starting PROGRESS
multi-user, 12-1
See also the PROGRESS System Administration Guide
single-user. *See* the PROGRESS System Administration Guide

Startup options
See also the PROGRESS System Administration Guide
European Numeric Format (-E), 4-8
for establishing userids and passwords, 11-3, 11-30
for work file size (-l), 10-2
Put Screen Compatibility (-psc), 16-11
X Window (-ws), 16-12

Startup parameter(s)
table for X Windows, 16-6
Window (-ws), 16-13
window Background Color (-bg), 16-6

window Border Color (-bd), 16-7
window Display Host (-display), 16-8
window Font (-fn), 16-9
window Foreground Color (-fg), 16-9
window Geometry (-geometry), 16-10
window Iconic (-iconic), 16-11, 16-13
window Title (-title), 16-12

Statements, data handling, 1-4

Streamlining data, 3-4

Streams, 9-2, 9-28
See also OUTPUT statements
closing, 9-5, 9-28
default, 9-2
defining additional, 9-23
establishing, 9-24, 9-28
for input, 9-2, 9-23
for output, 9-2, 9-23
how PROGRESS assigns to procedures, 9-2
opening, 9-25, 9-28
shared, 9-26
summary table, 9-28
two-way, 9-29
unnamed, 9-2

STRING function, 4-10

Strip menus, 7-15
See also the CHOOSE statement

Sub-totals, collecting from data. *See* Control-break reports

Subprocedures, using with transactions, 8-22

Subquery
in SELECT statement, 15-13
set, context for, 15-14
singleton, context for, 15-13

Subscripts, array field with PROGRESS/SQL, 15-3

SUBSTRING function, using with Quoter, 9-14

Subtransactions, 8-14
mechanics of, 8-42
starting, 8-16

SYLK files, converting, 9-19

SYmbolic LinK (SYLK) files, 9-18

System administration. *See* the PROGRESS System Administration Guide

System control file, creating, 12-20

System failures, how PROGRESS handles, 8-13, 8-37

T

Table, selecting from multiple. *See* Join operation.

Table names, PROGRESS/SQL. *See* Identifiers

Tables
See also Views
altering, 15-33
creating, 15-31
deleting, 15-34
list of SQL statements affecting, 15-31
owner, 15-31

Temporary files. *See* Work files

TERM environment variable, 2-20

TERM shell variable, 2-19

TERMINAL function, 2-19

Terminals
BTOS/CTOS color monitor support, 2-36
configuration of, 2-22
creating a protermcap entry, 2-21
defining for UNIX and VMS, 2-18
defining type, 2-19, 2-20
defining video display attributes, 2-35
describing characteristics, 2-20
nonspacetaking, 7-28
reserving space on, 7-28
screen formatting, 7-1
spacetaking, 7-28
UNIX stty control functions, 2-30

Testing
applications, 14-9
display formats, 4-12

Tilde (~)
escape character, 2-37, 14-2

Index

using to continue lines in the editor, 2-9

Time display formats, 4-10

TIME function, 4-10

Time stamp, for file data definition, 11-33

TITLE frame phrase option, 7-10

TO format phrase option, 7-7

TOP-ONLY frame phrase option, 7-10

TOP-ONLY option, Frame phrase, 7-26

Totals, collecting from data. *See* Control-break reports

TRANSACTION option, on blocks, 8-7, 8-17

Transaction processing, 15-29

Transactions, 8-1
affect on variables, 8-37
and blocks, 5-7, 8-7
and subtransactions, 8-14
changing size, to resolve record locks, 12-16—12-19
controlling where they begin and end, 8-16
definition, 8-2
determining when they are active, 8-40
how PROGRESS handles, 8-2
improving efficiency of, 8-42
making larger, 8-17, 12-17
making smaller, 8-20, 12-18
mechanics of, 8-41
processing, 8-3
relationship to record locks, 12-10
specifying how much to undo, 8-13
starting, 8-16
using with input from a file, 8-23
using with subprocedures, 8-22
where they begin and end, 5-8, 8-7

Transporting
See also Using the Developer's Toolkit
applications, 14-3
databases, 14-6

Two-way streams, 9-29
See also INPUT-OUTPUT THROUGH statement

Typographical conventions, xix

U

UNDO keyword, rules about using, 8-33

Undo processing, 8-13
for blocks, 5-8

UNDO statement, 5-8

UNDO statement, 8-15

Unique index, 15-32

UNIQUE keyword
creating an index with, 15-32
in CREATE TABLE statement, 15-31

UNIX
See also the PROGRESS System Administration Guide
broker process, 12-2
concepts. *See* the PROGRESS Environments Guide
defining terminals, 2-18
differences from other operating systems, 14-1
entering extended characters, 2-39
key codes and labels, 2-10
processes as input and output streams, 9-29
removing log file entries, 1-4
starting a multi-user server, 12-2
starting multi-user PROGRESS, 12-3
stopping a multi-user server, 12-4
stty control functions, 2-30
TERM shell variable, 2-19
termcap file, 2-20
terminal definitions for non-English languages, 2-40
using two-way streams, 9-29
writing transportable applications, 14-5

Unknown value
See also Null value
and indexes, 3-17

Unnamed frame, 5-4

Unnamed streams, 9-2

UPDATE, privilege operation, 15-40

UPDATE statement
positioned form, 15-27
searched form, 15-18
SET clause, 15-18
WHERE clause, 15-18

UPDATE statement, 1-5

Uppercase and lowercase conversions, 2-37
extended characters, 2-42

User-Defined Language Rules, 2-44

User-Defined Language Tables
user-cs-weight, 2-46
user-lowercase, 2-46
user-uppercase, 2-45
user-weight, 2-46

USERID function, 11-6
in PROGRESS/SQL, 15-30
syntax, 13-43

Userids, 11-1
See also Application security
checking at run-time, 11-4
establishing, 11-2, 11-24
for batch jobs, 11-3

V

VALIDATE format phrase option, 3-26, 7-7

Validation
for fields, 3-26, 3-31
checking a range of values, 3-27
checking against list of values, 3-27
checking existence of related record, 3-28
long expressions, 3-29
overriding defaults, 3-32
using IF...THEN...ELSE, 3-30
for files, 3-33

VALUE keyword
OUTPUT TO statement, 9-5
SEARCH function, 9-12

Variable, range type, 15-11

Variables, how transactions affect, 8-37

Video display attributes, 2-35

View, initial privileges for owner, 15-35

View names, PROGRESS/SQL. *See* Identifiers

viewer.p, to display view definitions, 15-36

Views
CREATE VIEW statement, 15-36
creating, 15-35
defined, 15-35
displaying, 15-36
DROP VIEW statement, 15-38
updating, 15-35, 15-38
updating WITH CHECK OPTION, 15-37

VMS
See also the PROGRESS System Administration Guide
concepts. *See* the PROGRESS Environments Guide
defining terminals, 2-18
differences from other operating systems, 14-1
key codes and labels, 2-10
login.com file, 2-19
PROTERM logical, 2-19
PROTERM.DAT, 2-20
removing log file entries, 1-4
server log file (sv.lg), 1-3
shared memory file (.shm), 1-3
SMGTERMS.TXT file, 2-20
starting a multi-user server, 12-2
starting multi-user PROGRESS, 12-3
stopping a multi-user server, 12-4
writing tranportable applications, 14-6

W

WHERE clause
SELECT statement, 15-8
UPDATE statement, 15-18

WHERE clause search conditions
[NOT] BETWEEN...AND, 15-9
[NOT] IN, 15-10
[NOT] LIKE, 15-9
[NOT] NULL, 15-9

with a relational operator, 15-8

WIDTH frame phrase option, 7-10

Wildcard characters, in PROGRESS/SQL, 15-6

Window(s)
See also Startup parameters
startup parameter (-ws), 16-13

Windows. *See* Frame(s), overlay

WITH CHECK OPTION keywords, CREATE VIEW statement, 15-37

WITH GRANT OPTION keywords, GRANT statement, 15-41

WITH phrase, extension to PROGRESS/ SQL, 15-3

Work files, 10-1
differences from database files, 10-2
number allowed, 10-2, 10-13
using to collect data, 10-7
using to create cross-tab reports, 10-12
using to manipulate data, 10-7
using to sort, 10-8
using to write reports, 10-3
See also Control-break reports

X Window(s)
and PROGRESS, 16-1
See also PROGRESS
figure of, 16-5
running, 16-5
starting a session in, 16-12
supported character sets for, 16-17
PROGRESS startup option (-ws), 16-12
sample Motif window, 16-3
sample OpenLook window, 16-4