

Progress Portability Guide

© 2001 Progress Software Corporation. All rights reserved.

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

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

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

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

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

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

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

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

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

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

.

May 2001



Product Code: 4532
Item Number: 81100;9.1C

Contents

Preface . . . . . **vii**
Purpose vii
Audience vii
Organization of This Manual vii
Typographical Conventions viii
Syntax Notation ix
Progress Messages xii
Other Useful Documentation xiv
Getting Started xiv
Development Tools xv
Reporting Tools xvi
4GL xvii
Database xviii
DataServers xviii
SQL-89/Open Access xix
SQL-92 xix
Deployment xx
WebSpeed xx
Reference xxi

1. About Progress Portability **1–1**
1.1 The Dimensions of Progress Portability 1–2
1.2 Progress Portability With and Without Recompiling 1–2
1.2.1 Display Architectures 1–2
1.2.2 Database Types 1–2
1.2.3 Platform Classes 1–3
1.2.4 R-Code Version 1–3
1.2.5 Examples of Portable and Nonportable R-Code 1–3
1.3 Designing Progress Applications 1–4

1.3.1 Designing the Database 1–4
1.3.2 Designing the User Interface. 1–4
1.3.3 Designing the Trigger Code 1–4
1.4 Designing for Portability 1–5
1.4.1 Database Portability Concerns 1–5
1.4.2 User Interface Portability Concerns 1–5
1.4.3 Code Portability Concerns 1–5

2. Database Design 2–1
2.1 Introduction 2–2
2.2 Database Objects 2–3
2.3 Code Pages 2–3
2.4 Data Types 2–3
2.5 Arrays 2–4
2.6 R-code Portability and DataServers 2–4
2.7 Progress 4GL Data Manipulation Language 2–4

3. User Interface Design 3–1
3.1 Design Overview 3–2
3.1.1 Defining Your Design Goals 3–2
3.1.2 Identifying Competing Goals. 3–3
3.1.3 Resolving Competing Portability Goals. 3–4
3.2 Creating a Window Procedure File 3–4
3.2.1 Structure of the Window Procedure File 3–4
3.2.2 Executing the Window Procedure File at Runtime 3–5
3.3 Creating Multiple Layouts 3–7
3.3.1 Designing the Master Layout 3–7
3.3.2 Choosing a Standard Layout. 3–8
3.3.3 Designing Alternate Layouts. 3–9
3.4 Testing and Adjusting Layouts 3–10
3.4.1 Absolute Layout 3–11
3.4.2 Character Units and Pixels 3–12
3.4.3 SESSION Handle Attributes 3–13
3.5 Widgets 3–14
3.5.1 Attributes. 3–14
3.5.2 Field-level Widget Portability. 3–16
3.5.3 Frame Widget Portability. 3–19
3.5.4 Window Widget Portability 3–19
3.6 Screen Formatting 3–20
3.6.1 Dimensions of Terminals. 3–20
3.6.2 Dimensions of Graphical Displays 3–21
3.7 Colors and Fonts 3–22
3.7.1 Colors 3–23
3.7.2 Fonts. 3–24

3.7.3 System Dialog Statements . . . 3–24
3.8 User-interface Environments . . . 3–25
3.8.1 Windows Environment Settings . . . 3–25
3.8.2 UNIX Environment Settings . . . 3–32
3.9 Events . . . 3–34
3.9.1 Keyboard Events . . . 3–34
3.9.2 Mouse Events . . . 3–44

4. Programming Considerations . . . 4–1
4.1 Naming Conventions for Files, Tables, and Variables . . . 4–2
4.1.1 Characters to Avoid . . . 4–2
4.1.2 Name Length . . . 4–2
4.1.3 Case Sensitivity . . . 4–3
4.1.4 File Path Specifications . . . 4–3
4.2 Terminal Key Definitions . . . 4–3
4.3 Operating System Statements . . . 4–4
4.3.1 Standard System Commands . . . 4–5
4.4 R-code . . . 4–8
4.4.1 R-code Portability and Operating System-specific Statements 4–8
4.4.2 R-code-imposed Limits on Program Size . . . 4–8
4.4.3 Using CRCs Instead of Time Stamps . . . 4–8
4.4.4 Choosing R-code or Encrypted Source Code to Ship to Customers 4–9
4.4.5 Byte Order of Integers in R-code . . . 4–9
4.5 Coding Style . . . 4–9
4.5.1 Using Preprocessor Directives . . . 4–9
4.5.2 Using the VIEW-AS Phrase . . . 4–11
4.6 Storing Data . . . 4–11
4.6.1 Storing Database Triggers . . . 4–11
4.7 Retrieving Data . . . 4–12
4.7.1 Using Logical Database Names . . . 4–12
4.7.2 Referencing Database Records . . . 4–12
4.7.3 Referencing Database Fields . . . 4–13
4.7.4 Using the FIND and GET Statements . . . 4–13
4.7.5 Using the Browse Widget . . . 4–13
4.7.6 Record Locking . . . 4–14
4.8 Security . . . 4–14
4.9 Accessing C Functions . . . 4–15
4.10 Interprocess Communications . . . 4–16
4.10.1 I/O with Other Processes . . . 4–17
4.10.2 Named Pipes . . . 4–18
4.10.3 System Clipboard . . . 4–18

Index . . . Index–1

Figures

Figure 3–1: Character-based Window 3–21

Tables

Table 3–1: User-interface Features 3–3
Table 3–2: Standard Layouts 3–8
Table 3–3: Attributes in Alternate Layouts 3–10
Table 3–4: AT Phrase Positioning and Specifications 3–11
Table 3–5: Attributes Specific to User Interfaces 3–15
Table 3–6: Size of Default Buttons 3–16
Table 3–7: Maximized Window Attributes 3–22
Table 3–8: SYSTEM-DIALOG Statements 3–25
Table 3–9: Static Key Bindings on Windows (Graphical and Character) 3–35
Table 3–10: User-definable Key Bindings on Windows (Graphical and Character) . 3–36
Table 3–11: Progress PROTERMCAP Key Bindings 3–37
Table 3–12: Mouse Buttons for Windows 3–45
Table 3–13: Portable Mouse Events 3–46
Table 3–14: Physical Mouse Events 3–47
Table 4–1: OS Statements 4–5
Table 4–2: Clipboard Operations 4–20

Preface

Purpose

Progress lets you build one application that runs in multiple environments. This guide tells you how to develop Progress applications that are portable across:

- Windows and UNIX operating systems
- Graphical and character interfaces
- Progress databases and non-Progress data sources

Audience

This book is for programmers who develop Progress applications intended to run across multiple operating systems, interfaces, or data sources.

Organization of This Manual

Chapter 1, "About Progress Portability"

Summarizes the dimensions of Progress portability, how to know when you must recompile, how to design Progress applications, and how to design portability into Progress applications.

Chapter 2, "Database Design"

Gives guidelines for making your application portable across different database systems.

Chapter 3, “User Interface Design”

Offers guidelines for making your application portable across different user interfaces and across different screen resolutions of the same interface.

Chapter 4, “Programming Considerations”

Presents guidelines for making your application code, especially trigger code, portable across different database systems and operating systems.

Typographical Conventions

This manual uses the following typographical conventions:

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

The following typographical conventions are used to represent keystrokes:

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

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

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

  CTRL-X

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

  ESCAPE H
  ESCAPE CURSOR-LEFT

Syntax Notation

The syntax for each component follows a set of conventions:

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

  In this example, ACCUM is a keyword:

  SYNTAX

  ```
  ACCUM aggregate expression
  ```

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

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

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

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

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

SYNTAX

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

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

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

SYNTAX

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

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

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

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

  SYNTAX

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

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

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

  SYNTAX

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

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

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

SYNTAX

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

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

SYNTAX

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

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

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

  SYNTAX

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

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

  SYNTAX

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

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

  SYNTAX

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

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

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

  SYNTAX

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

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

  SYNTAX

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

Progress Messages

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

- Execution messages inform you of errors encountered while Progress is running a procedure (for example, if Progress cannot find a record with a specified index field value).
- Compile messages inform you of errors found while Progress is reading and analyzing a procedure prior to running it (for example, if a procedure references a table name that is not defined in the database).

- Startup messages inform you of unusual conditions detected while Progress is getting ready to execute (for example, if you entered an invalid startup parameter).

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

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

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

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

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

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

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

1 ♦ Start the Progress Procedure Editor:

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

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

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

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

Other Useful Documentation

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

Getting Started

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

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

Progress Installation and Configuration Guide Version 9 for UNIX

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

Progress Installation and Configuration Guide Version 9 for Windows

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

Progress Version 9 Product Update Bulletin

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

Progress Application Development Environment — Getting Started (Windows only)

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

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

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

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

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

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

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

Progress Startup Command and Parameter Reference

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

Welcome to Progress (Hard copy only)

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

Development Tools

Progress ADM 2 Guide

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

Progress ADM 2 Reference

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

Progress AppBuilder Developer's Guide (Windows only)

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

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

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

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

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

Progress Debugger Guide

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

Progress Help Development Guide (Windows only)

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

Progress Translation Manager Guide (Windows only)

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

Progress Visual Translator Guide (Windows only)

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

Reporting Tools

Progress Report Builder Deployment Guide (Windows only)

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

Progress Report Builder Tutorial (Windows only)

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

Progress Report Builder User's Guide (Windows only)

A guide for generating reports with the Progress Report Builder.

Progress Results Administration and Development Guide (Windows only)

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

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

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

4GL

Building Distributed Applications Using the Progress AppServer

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

Progress External Program Interfaces

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

Progress Internationalization Guide

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

Progress Language Reference

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

Progress Programming Handbook

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

Database

Progress Database Design Guide

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

Progress Database Administration Guide and Reference

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

DataServers

Progress DataServer Guides

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

MERANT ODBC Branded Driver Reference

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

SQL-89/Open Access

Progress Embedded SQL-89 Guide and Reference

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

Progress Open Client Developer's Guide

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

Progress SQL-89 Guide and Reference

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

SQL-92

Progress Embedded SQL-92 Guide and Reference

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

Progress JDBC Driver Guide

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

Progress ODBC Driver Guide

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

Progress SQL-92 Guide and Reference

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

Deployment

Progress Client Deployment Guide

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

Progress Developer's Toolkit

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

WebSpeed

Getting Started with WebSpeed

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

WebSpeed Installation and Configuration Guide

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

WebSpeed Developer's Guide

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

WebSpeed Product Update Bulletin

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

Welcome to WebSpeed (Hard copy only)

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

Reference

Pocket Progress (Hard copy only)

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

Pocket WebSpeed (Hard copy only)

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

1

About Progress Portability

Progress applications are portable across various architectures. This chapter presents an overview of Progress portability.

This chapter covers the following information:

- The dimensions of Progress portability
- Progress portability with and without recompiling
- Designing Progress applications
- Designing for portability

1.1 The Dimensions of Progress Portability

Progress applications are portable. If you follow certain guidelines and strategies, you can port Progress applications across:

- Windows (32-bit platforms), UNIX, and AS/400 operating systems
- Graphical and character display architectures
- Progress databases and non-Progress data sources

The rest of this chapter introduces Progress portability guidelines and strategies.

1.2 Progress Portability With and Without Recompiling

When you compile a Progress application, the compiler generates r-code. You can often port the r-code without recompiling. You must recompile only if one of the following changes;

- The display architecture
- The database type
- The platform class
- R-code version

1.2.1 Display Architectures

The two display architectures supported by Progress are graphical and character.

Code that does not contain user-interface statements (specifically, code that does not create frames, either explicitly or implicitly) does not belong to any display architecture, and so does not involve a change of display architectures. An example of such code is a batch program.

1.2.2 Database Types

Database types are Progress, ORACLE, ODBC-compliant, and DB2/400. Applications access non-Progress databases through the appropriate Progress DataServer products.

Code that does not access any Progress or non-Progress database does not belong to any database type, and so does not involve a change of database types. An example of such code is a date-conversion procedure.

1.2.3 Platform Classes

Platform classes are the AS/400, and all other platforms that Progress supports.

The class of a platform depends on its machine word size and byte alignment.

1.2.4 R-Code Version

Porting across r-code versions sometimes requires recompiling. The r-code version always changes between major releases. It may or may not change between minor point releases depending on the added functionality.

If you port to:

- a lower or higher major release, you must recompile.
- a higher point release within a major release, you do not need to recompile.
- a lower point release within a major release, you may or may not need to recompile. However, it is highly recommended that you recompile.

1.2.5 Examples of Portable and Nonportable R-Code

Some r-code is portable, and some is not. You can usually port r-code between clients as long as you do not change display architectures, database types, platform classes or major r-code versions.

Here are some examples:

- If you port a Windows application to character-mode, you must recompile, because the display architecture has changed.
- If you port a character application compiled against an ORACLE DataServer on a SUN workstation to a character application accessing an ORACLE DataServer on a Hewlett-Packard (HP) workstation, you do not have to recompile (assuming that the ORACLE release level does not change).
- If you port a character application compiled against the DataServer for ORACLE on a SUN workstation to a character application accessing the DataServer for ODBC on an HP workstation, you must recompile, because the database type has changed.
- If you port an application from Version 9.0 to Version 9.1, you do not have to recompile. If you port the application from Version 8.1 to Version 9.0, you do have to recompile. If you port the application from Version 8.3 to Version 8.2, you may not have to recompile, but it is highly recommended that you do so.

For more information on r-code portability, see the "R-code Portability and DataServers" section in Chapter 2, "Database Design," and the "R-code" section in Chapter 4, "Programming Considerations." For detailed information on r-code features and functions, see the "R-code Features and Functions" appendix in the *Progress Programming Handbook*.

1.3 Designing Progress Applications

To design a Progress application, follow the same basic steps whether you are designing for a single platform or for portability. Generally, the steps are:

- Designing the database
- Designing the user interface
- Designing the trigger code

1.3.1 Designing the Database

Designing the database involves designing the database tables, records, fields, indexes, sequences, primary keys, and foreign keys that the application interacts with.

For more information on database design, see the *Progress Database Design Guide* and the *Progress DataServer Guides*.

1.3.2 Designing the User Interface

Designing the user interface involves creating the visual objects that application users interact with. The visual objects might include any of the following: database fields, queries, browsers, frames, rectangles, images, radio sets, toggle boxes, sliders, buttons, selection lists, editors, combo boxes, fill-ins, text, ActiveX Controls, SmartFolders, SmartPanels, SmartDataBrowsers, SmartDataViewers, SmartQueries, and SmartContainers.

For more information on user-interface design, see the *Progress AppBuilder Developer's Guide*, *Progress Application Development Environment — Getting Started* manual and the *Progress Programming Handbook*.

1.3.3 Designing the Trigger Code

Designing the trigger code consists of writing the code (generally Progress 4GL code) that defines what the application does when the user interacts with the user-interface elements. For example, the application might write a particular database record when the user clicks on a particular button. In this case, the trigger code is the code that writes the record when a user clicks on that button.

For more information on using the Progress 4GL to write trigger code, see the *Progress Language Tutorial* (Windows or Character), the *Progress Programming Handbook*, and the *Progress Language Reference*.

1.4 Designing for Portability

When you design a Progress application for portability, you follow the same basic steps and consider the portability concerns of each step. The steps are:

- Designing the database
- Designing the user interface
- Designing the trigger code

The portability concerns of each step are discussed in the following sections.

1.4.1 Database Portability Concerns

Database portability concerns include database-object naming conventions, multiple code pages, Progress and non-Progress data types, arrays, r-code portability across DataServers, and the behavior of certain Progress 4GL statements across DataServers.

For more information on database portability concerns, see Chapter 2, "Database Design."

1.4.2 User Interface Portability Concerns

User interface portability concerns include using multiple layouts for different interfaces, specifying layout in character units or pixels, avoiding widget techniques that work only in a particular interface, screen formatting, colors and fonts, environment files, and events.

For more information on user interface portability concerns, see Chapter 3, "User Interface Design."

1.4.3 Code Portability Concerns

Some code portability concerns include: file naming conventions, terminal key definitions, operating system statements, preprocessor directives, database trigger storage, database names, record locking, and security.

For more information on code portability concerns, see Chapter 4, "Programming Considerations."

2

Database Design

Progress applications are portable across database types. This chapter discusses how to design your applications for maximum portability.

This chapter covers the following topics:

- Database objects
- Code pages
- Data types
- Arrays
- R-code
- Progress 4GL data manipulation language

2.1 Introduction

The DataServers for ORACLE, DB2/400, and ODBC let you work within the Progress Application Development Environment (ADE) while accessing your non-Progress databases. In addition, the DataServers are compatible with other Progress database features and Progress extensions. Some of these features are:

- **Data Dictionary** — You use the Data Dictionary to modify database schemas, create indexes, and define database triggers, validation expressions, and help messages.
- **Application Builder (AppBuilder)** — You use the AppBuilder to design and generate code for your graphical interfaces.
- **Progress programming model** — You use this programming model to develop applications that respond to choices that your end users make.
- **Database triggers** — You use database triggers to define a block of Progress 4GL code that executes whenever a specific database event occurs, such as creating a record, deleting a record, or assigning a value to a field.

If you want your applications to use these features, you must make minor modifications to your non-Progress tables. The following sections describe these modifications. For a more detailed information on modifying non-Progress tables, see the appropriate *Progress DataServer Guide*.

The *Progress DataServer Guides* describe how to use the DataServers to access non-Progress databases. They tell how to build DataServer modules and how to program applications that access Progress and non-Progress database management systems. They also provide tutorials. Each DataServer has its own guide—for example, the *Progress DataServer for ORACLE Guide*, the *Progress/400 Product Guide*, and the *Progress DataServer for ODBC Guide*. These guides also include portability solutions specific to each DataServer.

The *Progress Database Design Guide* introduces the fundamental principles of relational database design using a sample database and the Progress Data Dictionary. As the *Progress Database Design Guide* stresses, designing a database is an iterative process. It consists of gathering the information and processing requirements of your business, and developing and refining a corresponding database structure. This chapter, by contrast, focuses on issues to keep in mind as you design an application for portability across database systems.

2.2 Database Objects

Non-Progress databases and Progress databases share the structural elements common to relational databases, but each system has its own elements. These structural elements are called database objects or schema objects. Database objects are components of the database's logical structure. Each database management system has its own terminology for referring to these objects.

To design for database portability, you must consider each database system's restrictions on database object names. See the *Progress DataServer Guides* for a listing of these objects and their equivalent Progress names, a description of the naming conventions and restrictions, and suggestions on resolving naming conflicts.

Progress Software Corporation recommends that you name database objects in a way that respects the restrictions of each database system you want to deploy your application against.

2.3 Code Pages

When you access a non-Progress database, the DataServer retrieves character data as determined by the code page. See the *Progress DataServer Guides* for information on how each DataServer architecture manages code pages. For more information on how Progress handles code pages, see the chapter on code pages and character sets in the *Progress Internationalization Guide*.

2.4 Data Types

Non-Progress data types differ from Progress data types. However, each non-Progress data type that the DataServer supports has at least one Progress equivalent.

The DataServer translates non-Progress data types into Progress equivalents. The schema holder is a Progress database that contains the Progress data definitions for the non-Progress columns, which you can modify using the Data Dictionary.

See the *Progress DataServer Guides* for a list of the specific data types that the DataServer supports and their Progress equivalents.

2.5 Arrays

A Progress database allows you to define fields as arrays.

The DataServers for ORACLE, DB2/400, and ODBC support arrays, also called field extents. The DataServer for ORACLE lets you extend Progress's ability to support database arrays to your ORACLE database. For information on using arrays, see the *Progress DataServer Guides*.

NOTE: DataServers for ODBC and ORACLE do not support native arrays. Progress extends its array-handling ability to these databases by implementing an array as multiple fields.

2.6 R-code Portability and DataServers

R-code is Progress compiled code. R-code is not portable across database systems. For example, code compiled against the Progress DataServer for ODBC must be recompiled to run against the Progress DataServer for ORACLE.

For more information on r-code portability, see Chapter 1, "About Progress Portability" and Chapter 4, "Programming Considerations," in this guide. Also, see the "R-code Features and Functions" appendix in the *Progress Programming Handbook*.

2.7 Progress 4GL Data Manipulation Language

See the *Progress DataServer Guides* for information on using the following 4GL statements or functions with your DataServer:

- FIND PREV/LAST statement
- ROWID function
- COMPILE statement

There are some differences and areas of incompatibility between Progress and non-Progress DataServers. To determine which Progress features your non-Progress databases do not support, use the DBRESTRICTIONS function. For more information on the DBRESTRICTIONS function, see its reference entry in the *Progress Language Reference*. For a list of 4GL statements and functions that a particular DataServer does not support, see the appropriate *Progress DataServer Guide*. For information on 4GL statements you can use to retrieve data, see Chapter 4, "Programming Considerations," in this guide.

3

User Interface Design

This chapter discusses how to design Progress applications that are portable across Windows and character interfaces, as well as across different screen resolutions of the same interface.

This chapter covers the following topics:

- Design overview
- Creating a window procedure file
- Creating multiple layouts
- Testing and adjusting layouts
- Widgets
- Screen formatting
- Colors and fonts
- User-interface environment files
- Events

3.1 Design Overview

Follow these basic steps to design a portable user interface:

1. Define your design goals.
2. Identify competing goals.
3. Resolve competing goals.
4. Customize the interface to the platform at run time.

3.1.1 Defining Your Design Goals

Each platform requires a different design to be most effective. Therefore, it is important to define strong design goals, then resolve any competition between the goals.

The same application might have to run on different platforms. For example, a factory might need to run the same application on each of the various kinds of workstations on the factory floor. In this case, the priority is consistency across platforms. By contrast, if you sell your application to different customers, and each customer uses only one kind of workstation, the priority is the application's native look and feel.

When defining your design goals, consider the following:

- Whether your application must be fully functional across displays
- Whether the same interface design is essential across all platforms
- Whether interface features are easy to incorporate and are efficient to use across all platforms
- The needs and expectations of your customers
- The technology your application uses. Know the capabilities and standards of your platforms. Sometimes they contradict each other. For example, character applications, by their nature, require a different approach than graphical applications. Consider these platform restrictions and standards, and whether your application must conform to them.
- Language and cultural differences and how they might affect your application. For each internationalization issue, determine the best solution on each platform.

3.1.2 Identifying Competing Goals

Once you define the design goals for your application, you can then determine the goals that are competing.

As an example of competing design goals, Table 3–1 lists common user-interface features and how different interfaces support them.

Table 3–1: User-interface Features *(1 of 2)*

Feature	Character	Windows
Application interface	A single screen with multiple frames.	Multiple windows and dialog boxes.
Multiple windows	Only one pseudo-window.	Multiple windows are useful and desirable.
Buttons	Clumsy without equivalent keystroke.	Useful and desirable.
Images	Not available.	Desirable (`.BMP`, `.ICO`).
Color	Not always available; colors that are desirable on one terminal often are not on another.	All colors might not be available on all machines, and the same RGB numeric color might appear differently on different terminals.
Multiple fonts	Not available.	Achievable, but desired fonts might not exist on all machines.
Widget sizes	Inner and outer sizes are the same.	Inner and outer sizes are different; borders vary.
Inner size (Format Characters) (Font)	One-to-one with format.	Varies when you use a proportional font.
Outer size (Decoration)	No decoration.	Platform dependent.
Default keys	**GO = F1** **ENDKEY = F4**	**GO = F2** **ENDKEY = ESC**

Table 3–1: User-interface Features *(2 of 2)*

Feature	Character	Windows
Default buttons	One character high; horizontal layout.	Approximately 1.2 character units high; small extra margin.
Standard dialog boxes	GET-FILE dialog available in adecomm.	Open, Save As, Color, Font, Print.
Field-level mnemonics	Not available.	Uses "&" syntax to denote **ALT** key combination.

3.1.3 Resolving Competing Portability Goals

You can resolve competing portability goals by:

- Trading off one design goal against another
- Using the 4GL to adjust the user interface to the platform at run time
- Using the multiple layouts feature of the Application Builder (AppBuilder) to customize your application for each platform at design time

3.2 Creating a Window Procedure File

Window procedure (`.w`) files are Progress procedure files generated when you create and save a Progress window or dialog box with the AppBuilder. The AppBuilder supports multiple layouts that let you represent multiple interfaces within a single window procedure file. This functionality allows you to maintain one frame definition in your code, with multiple run-time adjustment sections. Depending on the environment, the appropriate run-time adjustment section modifies the appearance of the frame when you run the application. The window procedure files adjust themselves at run time according to user-defined run-time logical expressions.

For more information on multiple layouts, see the *Progress AppBuilder Developer's Guide*.

3.2.1 Structure of the Window Procedure File

The AppBuilder lets you maintain one window procedure file for multiple display environments. This lets you build one application and customize it for each environment. For example, you might want to display one group of widgets in a Windows environment, and a

slightly different group of widgets in a character environment. You can also make adjustments to the same widgets in the same environment, but for different screen resolutions. Instead of maintaining a separate file for each layout, you can accomplish all of these within a single window procedure file.

These are the components of the window procedure file:

- User definitions section
- Control definitions
- Window creation
- Run-time attribute and AppBuilder settings
- Control triggers
- Main block (WAIT-FOR) (if not a persistent procedure)
- Disable_ui internal procedure
- Enable_ui internal procedure
- Internal procedures

In the Control Definitions Section of the window procedure file, the AppBuilder defines all controls using the master layout. In the Run-time Attributes and AppBuilder Settings section, Progress evaluates the multi-layout expressions and calls the appropriate internal procedure for the specified layout.

Applications you create using the AppBuilder are also portable from graphical to character interfaces when you use a fixed font and full character cell grid without fractional positions.

3.2.2 Executing the Window Procedure File at Runtime

When you execute the multi-layout window procedure file, the master layout definition is realized with the window's HIDDEN attribute specified. The standard run-time attributes are executed, followed by the CASE statement causing one of the customized sets of changes to execute. The window is visible if you do not specify the HIDDEN attribute for the window.

To support multiple layouts at run time, the AppBuilder creates a special internal procedure named *window-name*-Layouts (where *window-name* is the name of the window you are designing). This procedure contains a CASE statement whose key values are the names of all the layouts. The procedure is called with the name of the desired layout. As part of a window procedure file's startup code, the logical expressions of all the defined layouts are evaluated

until one is found to be TRUE. The *window-name*-Layouts procedure is then called with the name of that layout.

This is an example of run-time adjustment code:

```
IF SESSION:WINDOW-SYSTEM = 'WINDOWS':U THEN
   RUN Cust-Window-Layouts  (INPUT "Standard Windows").

ELSE IF TERMINAL = 'wy370'
   THEN RUN Cust-Window-Layouts (INPUT "WYSE 370").

ELSE IF SYSTEM:DISPLAY-TYPE = 'TTY':U
   THEN RUN Cust-Window-Layouts (INPUT "STANDARD TTY").
```

In the following example, the AppBuilder generates an internal procedure that uses the CASE statement for adjustments:

```
PROCEDURE Cust-window-layouts.
   DEFINE INPUT PARAMETER layout AS CHARACTER NO-UNDO.

CASE layout:
   WHEN "Standard Windows" THEN DO:

ASSIGN
   Btn_Done:COL IN FRAME FRAME-A = 34
   Btn_Done:ROW IN FRAME FRAME-A = 15.5.

ASSIGN
   Btn_Next:COL IN FRAME FRAME-A = 75
   Btn_Next:HEIGHT IN FRAME FRAME-A = 1.5
   Btn_Next:ROW IN FRAME FRAME-A = 13
   Btn_Next:WIDTH IN FRAME FRAME-A = 5.

END. /* Standard Windows Layout Case */

WHEN "WYSE 370" THEN DO:

   ASSIGN
     FRAME FRAME-A:BGCOLOR  = 7
           .
           .
           .
```

3.3 Creating Multiple Layouts

The AppBuilder supports multiple layouts within a single window procedure file. For more information on the window procedure file, see the "Creating a Window Procedure File" section.

The AppBuilder lets you create as many alternate layouts as necessary. When you manipulate the widgets in an alternate layout, the AppBuilder applies your changes only to that layout. The original, or master layout, and any other alternate layouts, remain unchanged. Each custom layout has an associated logical run-time expression that determines at run time which layout the user sees.

For more information on multiple layouts, see the *Progress AppBuilder Developer's Guide*.

3.3.1 Designing the Master Layout

When you create a window procedure file, you automatically create a master layout. The master layout consists of the widgets as defined or created in the window procedure file (in the Control Definitions, Frame Definitions, and Create Window sections). The master layout is the layout as it appears to the Compiler/Analyzer.

Progress Software Corporation recommends that you design the master layout first, since it offers the most flexibility in defining and adjusting layouts. The alternate layouts are all synchronized to the master, until you make adjustments to them. Therefore, complete the master layout before defining any alternate layouts.

The master layout is the basis of each alternate layout. As you modify the alternate layout, the master layout remains unchanged. The AppBuilder implements your changes to the alternate layout by adjusting the master layout at run time.

When you insert a new widget into an alternate layout, the AppBuilder defines the widget in the Control Definitions section of the window procedure file. Therefore, the widget becomes part of the master layout. However, when the user runs the application, both layouts appear as intended. In the master layout, the AppBuilder sets the widget's HIDDEN attribute to TRUE. In the alternate layout, the AppBuilder sets the HIDDEN attribute to FALSE. The user sees only the widget when the application uses the alternate layout. To hide widgets in a particular layout, activate the Remove From Layout toggle box in the widget's property sheet.

Different platforms can enable and display different sets of widgets. To specify different views of the same data, you lay out multiple widgets and hide those not used on a particular platform.

NOTE: You must define all possible widgets in the master layout so that the Analyzer and other AppBuilder internal mechanisms recognize and understand these widgets.

For this system to work for both graphical and character deployment platforms, the window procedure file is structured so that the CREATE WINDOW statement is not executed while running in character mode. The following code accomplishes this:

```
IF SESSION:DISPLAY-TYPE = "GUI" THEN
  CREATE WINDOW WINDOW-1 ASSIGN
       HIDDEN    =  YES
       TITLE     =  "Window 1"
         .
         .
         .
       SENSITIVE =  YES.
ELSE {&WINDOW-NAME} = DEFAULT-WINDOW.
```

The master layout is graphical by default. Design the master layout for the graphical platform that most of your customers use, then design alternate layouts. To change the default setting, choose Options→ User Preferences. Then choose the Character Mode radio button.

When you are developing an application to run on Windows and in character environments, Progress Software Corporation recommends that you first design the Windows version, and then the character version. Windows screens tend to contain fewer objects than character screens, so it is easier to port from Windows to character than from character to Windows.

3.3.2 Choosing a Standard Layout

Standard layouts are alternate layouts with predefined run-time expressions. The AppBuilder predefines these expressions because there are two main Progress run-time environments, Windows and character. Table 3–2 lists the two standard layouts.

Table 3–2: Standard Layouts

Layout Name	Run-time Expression
Standard character	SESSION:DISPLAY-TYPE=’TTY’:U
Standard Windows	SESSION:WINDOW-SYSTEM=’WINDOWS’:U

If you use a standard layout, you cannot change the layout’s name or run-time expression.

Follow these steps to select a standard layout:

1 ♦ Choose Layout→ Alternative Layout from the AppBuilder main window. The Custom Layout dialog box appears.

2 ♦ Choose the standard layout, then choose OK.

3.3.3 Designing Alternate Layouts

After designing and testing the master layout, you can create alternate layouts. You can create virtually any number of alternate layouts. Use alternate layouts to customize an interface for a specific display platform.

Anything you add to the master layout is propagated to the alternate layouts until you hide, delete, or adjust them in the alternate layouts. Alternate layout widgets are linked and synchronized to the master layout until you customize them for a layout.

Use the REMOVE-FROM-LAYOUT toggle box to remove a widget from a particular layout. The AppBuilder selectively hides and displays objects, which ensures that your code compiles on all platforms. If you delete objects in one layout and not in another, a trigger might rely on the code that you deleted, and might not compile. The AppBuilder uses the HIDDEN attribute instead of deletion to compile the procedure. This process of displaying certain widgets (while hiding others) allows you to customize each layout to your specifications.

For each alternate layout that you create, you supply a name and run-time expression that specifies under what conditions to use the layout. At run time, your application evaluates the expression, and if TRUE, displays that layout. This expression becomes the selection expression for a CASE statement, indicating what set of run-time attributes to execute. For more information on the CASE statement, see the "Executing the Window Procedure File at Runtime" section.

Creating Alternate Layouts

Follow these steps to create an alternate layout:

1 ♦ Choose Layout→ Alternative Layout from the AppBuilder main window. The Custom Layout dialog box appears.

2 ♦ Choose the New button. The New Layout Name dialog box appears.

3 ♦ Type the name of the layout, then choose OK.

4 ♦ In the Custom Layout dialog box, enter the run-time expression that you want tested in the Expression field, then choose OK.

After you create the alternate layout, the AppBuilder redisplays all of the widgets from the previous layout in your design window. The AppBuilder also displays the name of the alternate layout at the top of the design window.

Any subsequent changes that you make to the layout (except for inserting a new widget) affect that layout only. However, you can force the AppBuilder to synchronize an alternate layout with the master layout on a widget-by-widget basis. That is, you can force a widget in the alternate layout to adopt the same attributes as the corresponding widget in the master layout, and vice versa. For more information, see the *Progress AppBuilder Developer's Guide*.

Alternate Layout Attributes

Within an alternate layout, you can change some, but not all, of the attributes that can be changed in the master layout. Table 3–3 lists the attributes that you can change:

Table 3–3: Attributes in Alternate Layouts

BGCOLOR	HEIGHT	TITLE-BGCOLOR
COLUMN	NO-BOX	TITLE-FGCOLOR
EDGE-PIXELS	NO-LABELS	VIRTUAL-HEIGHT
FGCOLOR	NO-UNDERLINE	VIRTUAL-WIDTH
FILLED	ROW	VISIBLE
FONT	SEPARATORS	WIDTH
GRAPHIC-EDGE	THREE-D	–

NOTE: Character-interface layouts do not support color.

3.4 Testing and Adjusting Layouts

You must test your layouts on their target platforms. Character layout in the AppBuilder is actually a simulation. This does not replace testing on actual character terminals.

To test a character-mode layout, start database servers for all necessary databases. Next, start the Progress session and connect to the application database in multi-user mode. In the AppBuilder, use the Alternate Layout feature to select the character layout, then run it. The character application runs in the Windows 95/NT console window.

For more information on running character layouts in the AppBuilder, see the chapter on Multiple Layouts in the *Progress AppBuilder Developer's Guide*.

3.4.1 Absolute Layout

Absolute layout defines explicit locations for widgets. The AppBuilder generates code that uses absolute layout.

Absolute positioning ensures that the dimensions of objects are consistent on each platform; however, the decoration of each object differs.

To perform absolute positioning of field-level widgets, you can use the AT phrase with row and column specifications; however, use row and column specifications for portability. The following example demonstrates row and column specifications with the AT phrase, which is portable across graphical displays:

```
DISPLAY i AT ROW 10 COLUMN 10.
```

You can also specify the LEFT-ALIGNED, RIGHT-ALIGNED, or COLON-ALIGNED options with the AT phrase. These options specify whether to align the left edge of the field, right edge of the field, or the colon of the field label with the specified position. The syntax of the AT phrase is as follows:

SYNTAX

```
AT ROW row COLUMN column
  [ LEFT-ALIGNED | COLON-ALIGNED | RIGHT-ALIGNED ]
```

Table 3–4 lists the specifications you can use with the AT phrase.

Table 3–4: AT Phrase Positioning and Specifications *(1 of 2)*

Specification	Result
AT ROW *row* COLUMN *column*	Defaults to LEFT-ALIGNED
AT ROW *row* COLUMN *column* RIGHT-ALIGNED	Aligns the right edge of the widget with the specified position
AT ROW row COLUMN column COLON-ALIGNED	Aligns the colon of the field label with the specified position
AT X *x* Y *y*	Defaults to LEFT-ALIGNED

Table 3–4: AT Phrase Positioning and Specifications *(2 of 2)*

Specification	Result
AT X x Y y RIGHT-ALIGNED	Aligns the right edge of the widget at the X pixel
AT X x Y y COLON-ALIGNED	Aligns the colon of the field label with the X pixel

The following example demonstrates the COLON-ALIGNED and RIGHT-ALIGNED specifications:

```
DEFINE VARIABLE i AS CHAR.
FORM i AT ROW 10 COLUMN 30 WITH FRAME x SIDE LABELS.
FORM i AT ROW 10 COL 30 COLON-ALIGNED WITH FRAME y SIDE-LABELS.
FORM i AT ROW 10 COL 30 RIGHT-ALIGNED WITH FRAME z SIDE-LABELS.
```

3.4.2 Character Units and Pixels

A character unit is a basic unit of measure equal to the height of a fill-in widget, and the width of an average character in the default system font. This unit of measure is used wherever you use ROW, COLUMN, HEIGHT, WIDTH, SPACE, and SKIP syntax, regardless of the font setting for the frame you are defining. You can express a character unit as a decimal value, up to two decimal places. Precision beyond that is ignored. In character environments, Progress rounds decimals to integer values.

CAUTION: In character interfaces, Progress rounds the decimal character unit values up or down to the nearer integer, which might cause widgets to overlap.

Progress lets you plan your window layout either in pixels or in character units. For better portability between displays, use character units to specify size and location values. Using pixels is the least portable way to define widgets because specifying pixels ties your code to one resolution on one platform and makes the code unusable on character-mode platforms. Character units assist in porting graphical applications to character mode.

The default size of a fill-in widget is one character unit in height. The default width is eight characters. Character-based platforms work with fixed character units only. On Windows, the default system font is proportional. This can lead to truncation of text in some fill-in fields. Therefore, if you do not specify a font for a fill-in field or its parent frame on Windows, Progress uses the following guidelines to determine the size of the widget when you use a display format that does not contain fill characters (such as dashes in an identification number):

- If a string is three characters or less, the width of the widget is equal to the length of the string, multiplied by the widest character in the default system font.
- If a string is between 4 and 12 characters, the width of the widget is equal to three times the widest character in the default system font, plus the length of the remaining characters in the string, multiplied by the average width of the default system font.
- If a string is greater than 12 characters, the width of the widget is equal to the length of the string, multiplied by the average width of the default system font.

If you specify a display format that contains fill characters (such as the parentheses surrounding the area code of a telephone number), the font defaults to fixed:

```
DEFINE VARIABLE mychar AS CHARACTER FORMAT "XXX-XX-XXXX".
```

The size of a character unit is based on the default font. Recompiling is not necessary for different displays on the same platform (for example, with Windows on monitors with different resolutions).

3.4.3 SESSION Handle Attributes

Your application can convert pixels to character units and character units to pixels using the PIXELS-PER-ROW and PIXELS-PER-COLUMN attributes of the SESSION handle. For example, use this formula to determine at what pixel offset a certain row starts:

(ROW - 1) × SESSION:PIXELS-PER-ROW

Each widget's dimensions and position are readable in either character units or pixels. For example, the WIDTH-CHAR and WIDTH-PIXELS attributes both return the width of a widget.

You can also specify the FRAME-SPACING attribute to the SESSION handle. This integer value indicates the number of display units (pixels or character units) between frames. The default value is one row in height. In the following example on Windows, the space between frame X and frame Y is 2 pixels:

```
SESSION:FRAME-SPACING = 2.

DISPLAY "Customer" with frame X.
DISPLAY "Number" with frame Y.
```

For more information on the SESSION handle and its attributes, see the *Progress Language Reference*.

3.5 Widgets

Avoid multiple windows, images, sliders, and widgets geared toward mouse-only interfaces for the best portability between graphical and character interfaces.

Some widgets function differently on different platforms. There are also differences among widget attributes in the supported user interfaces. Consider the following issues if you plan to port your applications:

- Avoid using overlapping widgets. If two field-level widgets overlap, one of the widgets might appear, both might appear, or some part of each might appear. In character displays, overlapping field-level widgets often results in a mixture of different characters for each widget.
- In character interfaces, you do not control widget display order, and widget stacking order is random. On Windows, you can control widget display order, and widget stacking order is not random. To control widget stacking order on Windows, use the MOVE-TO-TOP and MOVE-TO-BOTTOM methods.

  These methods apply to windows, frames, and all field-level widgets. Text strings, images, and rectangles display behind other field-level widgets; you cannot move them on top. For more information on the MOVE-TO-TOP and MOVE-TO-BOTTOM methods, see the *Progress Language Reference*.

3.5.1 Attributes

Some attributes of user-interface widgets apply only to particular user interfaces. For the best portability, avoid using these attributes. For more information on these attributes and a complete listing of all attributes Progress supports, see the *Progress Language Reference*.

Table 3–5 lists the attributes that apply to specific widgets and lists the user interfaces they are valid in.

Table 3–5: Attributes Specific to User Interfaces *(1 of 2)*

Attribute	Widget	User Interface
DCOLOR PFCOLOR	All widgets except field groups, images, and text	Character
HWND	All widgets	Windows
MOVABLE RESIZABLE SELECTABLE	Field-level widgets and frames	Windows
TITLE-DCOLOR	Menus and frames	Character
GRID-FACTOR-HORIZONTAL GRID-FACTOR-VERTICAL GRID-SNAP GRID-UNIT-HEIGHT-CHARS GRID-UNIT-HEIGHT-PIXELS GRID-UNIT-WIDTH-CHARS GRID-UNIT-WIDTH-PIXELS GRID-VISIBLE	Frames	Windows
GRAPHIC EDGE	Rectangles	Character
BUFFER-CHARS BUFFER-LINES PROGRESS-SOURCE	Editor	Character
LARGE RETURN-INSERTED	Editor	Windows
RESIZE	Windows	Windows

Table 3–5: Attributes Specific to User Interfaces *(2 of 2)*

Attribute	Widget	User Interface
TIC-MARKS FREQUENCY	Sliders	Windows
NO-FOCUS	Buttons	Windows
TOOLTIP	Field-level widgets	Windows
FLAT-BUTTON	Buttons	Windows

3.5.2 Field-level Widget Portability

Field-level widgets are any widgets that you can place in a frame or dialog box.

Button Size

Buttons are action widgets that allow the user to direct the operation of your application.

The size of buttons on Windows and character displays is different. Character displays create buttons that are one character unit high. Windows creates buttons that are about 1.2 times the size of a character unit. Since the size of a button differs on each platform, buttons might overlap in one environment, but position correctly in another. You can correct this difference by explicitly defining the size of the buttons and then laying them out. Buttons defined with the SIZE phrase are portable across all supported graphical environments.

Default Button Size

Table 3–6 lists the size of default buttons in each user interface.

Table 3–6: Size of Default Buttons

Windows	Character
1.2 times the size of a character unit	One character unit high

A default button has an extra border on Windows. When defining a button, you can specify the DEFAULT option to change to the default style.

To comply with user-interface design standards, on Windows, place default buttons on the bottom or right side of the display.

Progress supplies prepackaged images for the up, down, left, and right arrows. The images are named BTN-UP-ARROW, BTN-DOWN-ARROW, BTN-LEFT-ARROW, and BTN-RIGHT-ARROW, respectively. These images are called system images and are portable across all graphical environments (but not usable in character-mode).

On Windows, you use the Windows Control Panel to assign colors to buttons and window titles. Therefore, any color assignments that you make from within a Progress application are ignored.

In environments without a mouse (for example, most character environments) buttons are sometimes awkward to use. Consider defining key sequences as equivalents to buttons for these environments. See the "Events" section for more information on key mappings. Also consider using pull-down menus with accelerator keys that have the same functions as the buttons. Then, for character layouts, you can hide the buttons and use the pull-down menus instead.

Using Buttons

You use buttons differently from the keyboard depending on your display environment. For example, on Windows, to choose a button you position to it and press **RETURN** or **SPACEBAR**. In character displays, choose a button, you position to it and press **SPACEBAR**.

On Windows, if you define a default button in a frame (using the DEFAULT-BUTTON option of the Frame phrase), then position to any button in that frame, you press **RETURN** to choose the current button. When you position to anything other than a button, pressing **RETURN** chooses the default button.

If you do not define a default button in a frame, you cannot press **RETURN** to choose a button unless the focus is on the button.

Rectangles and Images

Rectangles and images are graphic widgets that decorate your interface. Use rectangles to create different kinds of borders. An image widget is a container where you display an image from a bitmap file. Images allow you to include high-resolution graphics in your interface.

Character interfaces do not support images, since you cannot put bitmapped pictures on a terminal. Windows lets you create bitmaps (`.BMP`), icons (`.ICO`), and cursors (`.CUR`) for your applications. Bitmap files define monochrome or color bitmaps. Icon and cursor files define a family of images. For the best portability, omit file extensions when referencing an image file to allow Progress to search for a file with the appropriate file extension:

```
DEFINE IMAGE myimage.
```

Progress first searches the current working directory and then searches the PROPATH for matching filenames and extensions that a platform supports. Windows first searches for `.BMP`,

.ICO, then .CUR. Omitting file extensions makes your application more flexible, although the additional searching might make your application run slower.

Images are generally not portable across different display types. For example, a Windows icon is 32 x 32 pixels. When compiled at 800 x 600 resolution (with small fonts), Progress converts this dimension to approximately 1.5 x 1.2 character units and stores it in the r-code. If you move that r-code to a system with 1024 x 768 resolution (with large fonts), the character units (1.5 x 1.2) are interpreted as 45 x 40 pixels. Since the icon remains the same size, you might have empty screen space. To avoid this, do not use explicit size information; instead let Progress adjust the images correctly or specify the absolute size in pixels for consistency on all resolutions.

If you are using an image on a button, assign both an image and a label to the button to make your code portable between graphical and character interfaces. In graphical interfaces, Progress uses an image; in character interfaces, Progress uses a label.

Editor and Selection List Widgets

There are two ways you can specify size for an editor widget or a selection list widget. You can use the standard SIZE phrase or the WIDTH and HEIGHT attributes to specify the outside dimensions of the widget. You can also specify inner dimensions for these two widgets using the INNER-CHARS and INNER-LINES options, or attributes of the same name.

Using SIZE to specify outer dimensions in character units ensures that the object is the same relative size on all platforms. However, on some systems with some fonts, the widget might end up with partial characters inside the bounding rectangle. The right-most visible character in the widget might get clipped by the right edge of the widget, or the bottom line of characters might be partially clipped by the bottom edge of the widget.

Using INNER-CHARS and INNER-LINES ensures that an integral number of characters are visible in a widget on all platforms, so there are no partial characters. These phrases tell the 4GL exactly what you want to appear inside of the widget; however, the outer dimensions of the widget vary somewhat between systems. The widget might overlap another widget or leave too much gap between widgets on another system.

On Windows, when a selection list has input focus, you can press an alphabetic character on the keyboard to select the next entry in the list that begins with that character. The search begins from the current selection and moves forward through the list.

For information on using the editor and selection list widgets, see Chapter 4, “Programming Considerations.”

3.5.3 Frame Widget Portability

Frames and dialog boxes are container widgets. Frames allow you to organize data, action, and graphic widgets. You cannot place widgets in a window unless they are first contained in a frame widget. If you do not specify a frame, Progress uses the default frame.

The dialog box is another window that overlays the main window. The dialog box blocks the user from working with the main interface until the user explicitly dismisses the dialog box. Like a frame, a dialog box contains data, action, and graphic widgets.

At compile time, you can use the preprocessor to adjust the position and size of user interface objects, or use the DEFINE FRAME or FORM statements to describe the layout and processing properties of a certain frame.

The DEFINE FRAME statement defines the layout of a frame for use within a single procedure or within several procedures. The DEFINE FRAME statement does not scope the frame. When using the DEFINE FRAME statement, you can adjust the position and size of user interface objects at run time. When you use the FORM statement, if the frame has not been previously scoped, the FORM statement scopes it to the current block. You can create different versions of the FORM statement and DEFINE FRAME statement for use on different platforms.

See the DEFINE FRAME Statement and FORM Statement reference entries in the *Progress Language Reference*.

At run time, you can adjust the position and size of user interface objects by modifying their geometric attributes. If you do this, use the HIDDEN attribute on the window or frame so that the changes do not cause flashing.

3.5.4 Window Widget Portability

The window widget is the workspace for your application. The window contains all other widgets. Character interfaces use one window, while graphical interfaces can use multiple windows.

These are the window events that are valid only in graphical displays:

- WINDOW-CLOSE
- WINDOW-MAXIMIZED
- WINDOW-MINIMIZED
- WINDOW-RESTORED

3.6 Screen Formatting

The user interface of a Progress application consists of one or more windows, whose size and number affect the space available for frames. Depending on your environment, character or graphical, there are different factors to consider when you lay out frames.

There are two ways a terminal can handle screen formatting: spacetaking and nonspacetaking. A *spacetaking terminal* reserves a character position on both sides of every field for special screen display attributes, such as underlining or highlighting. A *nonspacetaking terminal* does not reserve character positions for special screen display attributes.

NOTE: Although UNIX supports spacetaking and nonspacetaking terminals, Progress supports nonspacetaking terminals only.

3.6.1 Dimensions of Terminals

Progress can accommodate character-based terminal screens of different sizes. On Windows NT, the number of lines is 25. On UNIX, the number of lines varies depending on the terminal model, but is usually 24. Procedures that rely on 25 lines on the screen do not operate correctly on 24-line terminals. Therefore, design your use of the screen based on 24 lines to ensure that your procedures do not depend on that extra screen line.

NOTE: Windows NT 4.0 provides you with a choice of line numbers for a character-based terminal screen. Located in the character window properties sheet, the selection of line numbers includes 25 (default), 43, and 50.

Figure 3–1 shows the dimensions of a character-based window.

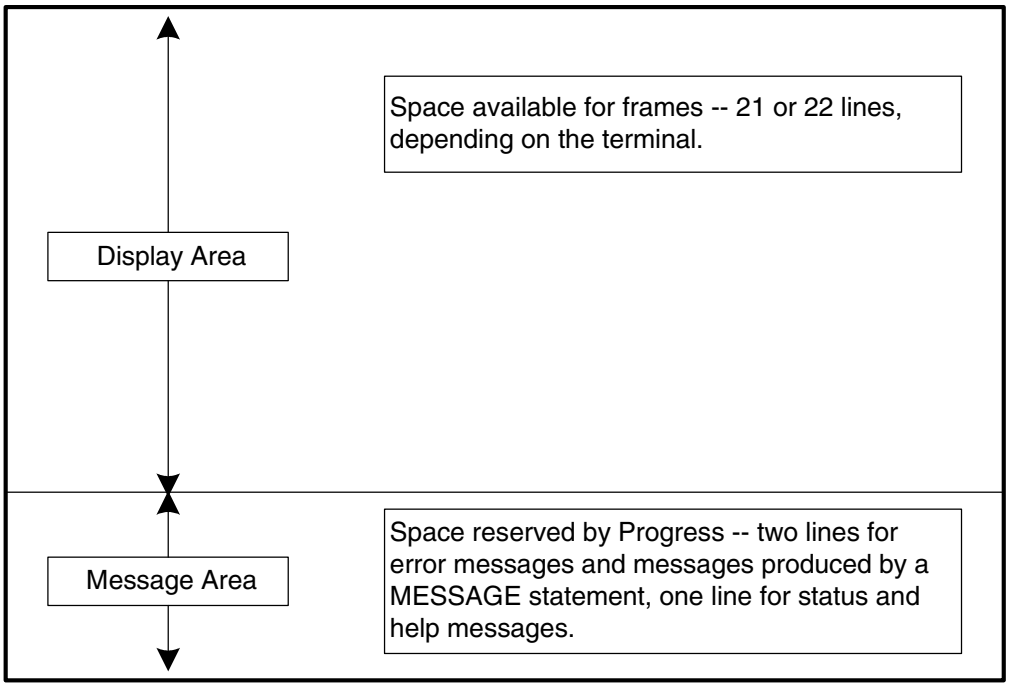

Figure 3–1: Character-based Window

You must size your frames to fit the display area. If some users run your application on terminals that have 24 lines while others run on terminals with more lines, design your frames to fit in the smaller display area to ensure that your frames fit in either display.

You must also size your frames to fit the width of the screen. Most screens are 80 characters wide, although some are wider. You can take advantage of the entire width of your terminal screen (up to 255 characters) when developing an application. However, if you take advantage of a screen width greater than 80 characters, your frames will not fit on a screen with a smaller width.

3.6.2 Dimensions of Graphical Displays

Screen dimensions on different platforms vary. Consider the displays where your applications will run so that your procedures do not depend on screen lines that are not available on some displays. Design your interfaces using the number of lines in the smallest display environment.

On Windows, you can also use the CREATE WINDOW statement to specify window size at run time. The size of a default window on Windows depends on your screen resolution. See the *Progress Language Reference* for more information on the CREATE Widget statement.

In a graphical display, window layouts are generally measured in pixels, not characters. However, Progress lets you plan your window layout in either pixels or in character units. For better portability across different platforms (or across different display resolutions within a single platform), use character units to lay out your screen displays so you do not have to calculate your application layout in pixels and adjust your calculations for different screen resolutions or different platforms.

The window attributes in Table 3–7 let you query the interior size of a maximized window. When a window is maximized, it fills the entire screen. The values these attributes return are the same whether or not the window is maximized.

Table 3–7: Maximized Window Attributes

Attribute	Description
FULL-HEIGHT-CHARS	The interior height of the window in character units
FULL-HEIGHT-PIXELS	The interior height of the window in pixels
FULL-WIDTH-CHARS	The interior width of the window in character units
FULL-WIDTH-PIXELS	The interior width of the window in pixels

The values these attributes return exclude the title bar, menu bar, message area, and status area. The interior of a window cannot exceed the area these attributes return.

See the *Progress Programming Handbook* for more information on windows. For information on window defaults and characteristics on Windows, see the "User-interface Environments" section of this chapter.

3.7 Colors and Fonts

Some attributes of user-interface widgets are specific to particular user interfaces. For the best portability, avoid using these attributes. The following attributes apply to all widgets except field groups, images, and text, and are only valid in character interfaces:

- DCOLOR
- PFCOLOR

To make your application portable between character and graphical interfaces, define all four types of color specifications: FGCOLOR, BGCOLOR, DCOLOR, and PFCOLOR. Each platform uses the two appropriate specifications and ignores the remaining two.

3.7.1 Colors

Progress understands colors according to a zero-based list of color numbers. Progress maps a color number to a particular color whose definition is stored in the environment files.

For applications on Windows (graphical or character), 16 predefined colors are available to use as foreground or background colors. In addition, you can add up to 256 colors in the registry or the `progress.ini` file. For best portability, do not change the first 16 predefined colors in graphical environments. To add a new color not in this default list, map the color to a number greater than 15.

In UNIX character interfaces, you can specify up to 123 color fields in a terminal entry that maps a color to an integer from 5 to 127. Set colors in your environment files so that the color numbers you use in your application are mapped to the same color on each platform.

Colors can be static or dynamic. You can use static colors to prevent users from modifying some colors while allowing them to modify others. You can modify dynamic colors in the 4GL using the SYSTEM-DIALOG COLOR statement or by explicitly setting the color's red, green, and blue values using the COLOR-TABLE system handle. See the SYSTEM-DIALOG COLOR Statement reference entry in the *Progress Language Reference*.

Any entries that you add to the color table on Windows must be defined sequentially with no gaps. For example, since the registry or `progress.ini` file defines color0 through color15, the next color you add must be color16. If you mistakenly add an entry for color17, Progress ignores it if color16 is undefined. Therefore, for static colors to be portable between graphical interfaces, use only contiguous color numbers up to 255. For static colors to be portable across graphical and character interfaces, use contiguous color numbers up to 127.

When you specify a color for a widget in Progress, graphical interfaces consider only the FGCOLOR (foreground color) and BGCOLOR (background color). Character interfaces consider only DCOLOR (display color) and PFCOLOR (prompt color). Therefore, to make your application more portable between character and graphical interfaces, define all four types of color specifications. Each platform uses the two appropriate specifications and ignores the remaining two. See the *Progress Language Reference* for more information on using these options with the FORM statement.

3.7.2 Fonts

Like colors, fonts are stored in the environment files. If you use multiple fonts on different graphical interface platforms, you must make sure that you have fonts that match, particularly in size.

A character unit is based on the default font. The default font specified in the Startup Section of the registry or the `progress.ini` determines the size of the character unit.

Fonts are bound into r-code at compile time. If you build your application with one font and try to deploy it with a different font, the result is unpredictable. The environment file you use at compile time should have the same font definitions as the environment file at run time.

Use the GET-TEXT-WIDTH method of the FONT-TABLE handle to calculate the length of a string in pixels or character units. Use the GET-TEXT-HEIGHT method of the FONT-TABLE handle to calculate the height of a font in pixels or character units using the font number. See the *Progress Language Reference* for more information on these methods.

There is no direct way to set font values in Progress. Font entries can only be changed by the user through the Font system dialog box. You must either use the SYSTEM-DIALOG FONT statement on Windows, or set them manually in the environment files. The SYSTEM-DIALOG FONT statement displays a dialog box that allows the user to select and associate a system font with the specified font number. See the SYSTEM-DIALOG FONT Statement reference entry in the *Progress Language Reference*.

NOTE: On Windows, the AppBuilder runs with both small and large fonts on 800 x 600 and 1,024 x 768 display resolutions, but only runs with small fonts on 640 x 480 displays.

3.7.3 System Dialog Statements

Progress allows you to implement native color and font dialog using the SYSTEM-DIALOG COLOR and SYSTEM-DIALOG FONT statements. Using these dialogs, you can allow the user to select a color or font.

The SYSTEM-DIALOG statements provide a dialog box appropriate to the graphical environment in which they run. The SYSTEM-DIALOG statements are not valid in character interfaces.

You can also use the SYSTEM-DIALOG GET-FILE statement to enter a filename assigned to a character variable. This statement displays a native dialog box that allows you to choose an operating system file. The dialog responds to filenames differently. If the user enters a filename or relative pathname, the Windows dialog box never searches the PROPATH and always returns the full pathname of the entered relative pathname appended to the current common dialog directory.

The SYSTEM-DIALOG PRINTER-SETUP statement displays the standard print dialog box. This statement is ignored in character environments.

Table 3–8 lists the SYSTEM-DIALOG statements.

Table 3–8: SYSTEM-DIALOG Statements

Statement	Description
SYSTEM-DIALOG GET-FILE	Displays a dialog box where you enter a filename that is assigned to a character variable.
SYSTEM-DIALOG COLOR	Displays a dialog box where you choose and associate a system color with the specified dynamic color number.
SYSTEM-DIALOG FONT	Displays a dialog box where you choose and associate a system font with the specified font number.
SYSTEM-DIALOG PRINTER-SETUP	Displays the standard print dialog box. This statement applies only on Windows.

See the *Progress Language Reference* for more information on these statements.

3.8 User-interface Environments

Each user interface that Progress supports provides a place for storing *environment* settings. Environment settings determine how the display appears to the user and how the mouse and keyboard behave.

On Windows platforms (graphical and character), Progress uses either the registry or the `progress.ini` file for user-interface environment settings. For more information on the use of the registry and the `progress.ini` file, see the chapter on user-interface environments in the *Progress Client Deployment Guide*.

On UNIX platforms, Progress searches the PROTERMCAP file for the user-interface environment settings.

3.8.1 Windows Environment Settings

When you install Progress on Windows, the installation program automatically places default `progress.ini` information into the registry. Depending upon which `prowin32` parameters you specify for startup, Progress searches for either the registry or the `progress.ini` file. For more information on the registry, see the chapter on maintaining user environments in the *Progress Client Deployment Guide*.

Progress Software recommends that you modify the Windows environment settings by modifying the `progress.ini` file. You can change the default settings and add your own customized sections. Note that after each modification of this file, you must run the INI2REG utility, which places `progress.ini` information into the registry.

The `progress.ini` file contains sections created by Progress for unique purposes. For example, the [Proedit] section is read by the Procedure Editor. Sections are also platform specific. For example, character client executables read information stored only in the [WinChar Startup] section while the Windows graphical client executables read information stored only in the [Startup] section.

As installed, the `progress.ini` file contains the following sections:

- [Startup]
- [Colors]
- [Default Window]
- [Fonts] for the Progress Application Development Environment (ADE) tools

  NOTE: You can add site-specific fonts to the file. Do not change the first 16 colors; do not change the first 8 fonts. These colors and fonts are reserved for the ADE tools. Changing them could cause the Progress ADE tools to malfunction.

- [Proedit]
- [Debug-Init]

  NOTE: Do not modify information in the Debugger section.

- [WinChar Startup]
- [WinChar Colors]
- [WinChar Default Windows]
- [WinChar Keys]

  NOTE: The [WinChar...] sections are read only by the Windows Character Client.

When Progress searches for environment settings at startup, it first searches the registry, unless the session started with a startup parameter that told Progress to start searching for an initialization file. Progress then searches for an initialization file in the following places:

1. The current working directory (as defined in the Working Directory field in the Program Item properties sheet)
2. The Windows directory
3. The directory that contains the Progress executable
4. The user's path

This search order fosters deployment flexibility because you can:

- Create a customized initialization file.
- Place the file in a consistent place on every machine.
- Let all users on a network share the same file.

Startup Specifications

The [Startup] section specifies the following:

- Terminal-emulation specification
- Display attributes and multi-tasking capabilities
- The default proportional font for alphanumeric data and character fields that do not contain fill characters (such as the parentheses surrounding the area code of a telephone number)
- The default fixed font for integer fields (such as date fields or numeric fields) or for fields with format strings containing fill characters, for example, xxx-xxx, in which the dash (-) is a fill character
- The default font used in updates to fill-in fields in version-specific display mode

For more information on these startup specifications, see the chapter on user-interface environments in the *Progress Client Deployment Guide*.

Font Specifications in Startup

These are the font settings in the [Startup] and [WinChar Startup] sections:

```
DefaultFont=MS Sans Serif, size 8
DefaultFixedFont=CourierNew
```

Available Fonts

There are font table entries for the Progress ADE tools. Progress uses the default system fonts defined in the [Startup] and [WinChar Startup] sections, unless you add additional fonts to the existing font table.

NOTE: Do not change fonts 0 through 7; this could cause the Progress ADE tools to malfunction.

Determine which screen fonts exist in your environment and choose fonts by using the Font common dialog box or by viewing the font settings in the registry.

Once you have selected fonts, add them to the font table using the following syntax:

SYNTAX

```
fontN=typeface,
  { size point size, keyword, script+scriptname }
```

N

An integer that specifies the font table entry.

typeface

A valid typeface name, such as Times or Courier.

point size

The size of the font, in points. If you omit the point size, Progress uses the default point size as defined by the typeface.

keyword

You can optionally specify bold, italic, underline, or strikeout.

For example, add 15 point Times to font table entry 16 as follows, and separate typeface and keyword with a comma:

```
font16=Times, size 15, bold
```

You can also add fonts to the font table using the following formats:

```
fontN=DefaultFont
fontN=DefaultFixedFont
```

Your font table can contain up to 256 entries. Each entry in the table performs two semantic functions: it names a font that exists in your environment and it maps that font to an integer from 0 to 255. The integer is the way an application references the font when making a font assignment.

Available Colors

At run time, Progress stores color specifications in a data structure called the color table. The color table is initialized from the [Colors] and [WinChar Colors] sections. There are 16 colors available to an application for use as either foreground or background colors. You can add up to 256 colors.

Each entry in the [Colors] and [WinChar Colors] sections performs two semantic functions: it defines a color by name or RGB value and maps that color to an integer from 0 to 255. The integer is the way an application references the color when making color assignments to a widget.

A color entry has the following syntax:

SYNTAX

```
colorN={R,G,B | color-name}
```

N

An integer from 0 to 255 that specifies the color table entry.

R

An integer that specifies the amount of red present in the color.

G

An integer that specifies the amount of green present in the color.

B

An integer that specifies the amount of blue present in the color.

color-name

Any of the following color names, mapped to the colors defined on your Windows system:

```
COLOR-SCROLLBAR
COLOR-BACKGROUND
COLOR-ACTIVECAPTION
COLOR-INACTIVECAPTION
COLOR-MENU
COLOR-WINDOW
COLOR-WINDOWFRAME
COLOR-MENUTEXT
COLOR-WINDOWTEXT
COLOR-CAPTIONTEXT
COLOR-ACTIVEBORDER
COLOR-INACTIVEBORDER
COLOR-APPWORKSPACE
COLOR-HIGHLIGHT
COLOR-HIGHLIGHTTEXT
COLOR-BTNFACE
COLOR-BTNSHADOW
COLOR-GRAYTEXT
COLOR-BTNTEXT
COLOR-INACTIVECAPTIONTEXT
COLOR-BTNHIGHLIGHT
```

For example, the following entry specifies red at color table entry 4:

```
color4=128,0,0
```

Any entries you add to the color table must be sequential. For example, since the installed environment defines color0 through color15, the next color you add must be color16. If you mistakenly add an entry for color17, Progress ignores it if color16 is undefined.

For backward compatibility, color0 through color15, as installed, are the same as the colors supported in previous versions. See the *Progress Programming Handbook* for a list of these colors.

Windows Color Palette

Many VGA and Super VGA video cards provide 256-color palettes for the Windows environment. The following restrictions apply to the use of color palettes on Windows:

- Windows reserves 20 palette entries for system use.
- Only one palette can be active at a time. If two applications have 256-color bitmaps, and Application A is active, the image for Application B is most likely rendered incorrectly if it uses a different color palette and runs in the same window session.
- An application's palette is used when the application is active.

Some video cards provide either HI-COLOR or TRUE-COLOR interfaces for Windows. The HI-COLOR interface provides 65,536 colors, and the TRUE-COLOR interface provides 16,777,216 colors. These two interfaces do not use palettes and therefore do not impose the same restrictions on color use. These are the features provided by these environments:

- You can display multiple 256 or TRUE-COLOR bitmaps simultaneously.
- The correct rendering of an image is not dependent on the currently active application.

Windows Color Bitmaps

Progress supports 256-color bitmaps for buttons and images. Each 256-color bitmap has a list of palette entries associated with it. When you display one of these bitmaps in a Progress window, the palette entries are appended to the existing Progress palette.

The following rules and restrictions apply:

- The last 256-color bitmap's palette used in a Progress window overwrites any previous bitmap palettes within that window.
- Progress Software Corporation recommends that you use only one 256-color bitmap palette per Progress window. This ensures that all of the bitmaps within a Progress window display properly when the window is active. Progress applications that change the color palette might significantly impact performance.
- The current 256-color bitmap palette for a Progress window is activated whenever the window is activated.
- The number of colors that are used from the 256-color bitmap are calculated as follows:

  N COLORS = 256 - (RESERVED WIN COLORS) - (Progress CUSTOM COLORS)

- When using **PRINT SCREEN** or **ALT-PRINT SCREEN** to capture screens, the bitmap copied to the Clipboard matches the current display resolution. For example, once **PRINT SCREEN** captures a screen, you can use Paint Brush to cut a 99 x 24 bitmap for use on a Progress button. The resulting disk image is a 256-color bitmap that takes up 3,478 bytes of disk space. The same bitmap, when reduced to the standard 16 colors supplied by Windows, takes up only 1,366 bytes of disk space. There are several shareware and commercial products that can perform these color-reduction operations.

  NOTE: If you are using bitmaps that do not require 256 colors, build them as 16-color bitmaps. Otherwise, you might experience a loss in performance.

Main Window Defaults

The [Default Window] section of the environment file specifies the following for a Windows application's default window:

- **Location** — Specify the location as an x,y pixel value.
- **Size** — Specify the size as a number of rows and columns.

3.8.2 UNIX Environment Settings

The environment file or terminal definition file for UNIX character interfaces is called the PROTERMCAP file. This file is a text file that stores an entry for each supported terminal type and contains special feature definitions especially for use with Progress. Each terminal entry contains the commands and terminal descriptions that Progress uses to communicate with and control the terminal. Each entry includes the following five logical sections:

- Terminal names
- Terminal capabilities
- Progress terminal-specific key functions and capabilities
- Vermont Views key functions
- Version-specific key functions

The PROTERMCAP environment variable ($DLC/protermcap) points to the PROTERMCAP file.

NOTE: There is a 4,096K limit on the size of each PROTERMCAP file.

Setting Up Your Terminal

Use your terminal's setup mode to synchronize your terminal with the color and key code definitions in its PROTERMCAP file.

The PROTERMCAP file supplied with Progress has built-in definitions for many terminals. If your terminal is not included in the PROTERMCAP file supplied with Progress, you must make modifications to that file. Progress uses the shell variable PROTERMCAP to find the PROTERMCAP file. Progress looks for this file whenever you start the software in a UNIX environment.

Specifying Colors

You can specify up to 123 color fields in a terminal entry. At run time, Progress loads these color specifications into the color table.

Each table row performs two semantic functions-defining a foreground/background color pair and mapping that color pair to an integer from 5 to 127. Integers are used by the application to reference colors assigned to a widget. (An application typically assigns two color pairs to a widget-one pair for display mode and one pair for prompt mode.)

Use the following syntax to specify a color field in a PROTERMCAP terminal entry:

SYNTAX

```
COLOR color-number
  [ color-name ]=start-sequence:stop-sequence:
```

color-number

An integer from 5 to 127 that specifies the location of the color in the color table. The color number is the mechanism used to assign a color pair in an application.

color-name

The name of the color. You can use any value for *color-name*, except a Progress keyword. Although this is optional, including a name makes the color section of the terminal entry self-documenting (you cannot embed comments) and also makes the color specifications backward compatible.

start-sequence

The character sequence that starts the color attribute.

stop-sequence

The character sequence that stops the color attribute.

For example, the following fields from the Wyse 370 terminal section define color table locations 5 and 6:

```
COLOR 5 RED/BLACK=\E[31;40m:\E[m:\
COLOR 6 GREEN/BLACK=\E[32;40m:\E[m:\
```

Progress reserves color table locations 0 through 4 as follows:

- Color 0 holds the version-specific color NORMAL, which is set by the terminal initialization (is) field.
- Color 1 holds the version-specific color INPUT and is set and cleared by ue and us fields. As installed, this is underline mode.
- Color 2 holds the version-specific color MESSAGE and is set and cleared by so and se fields. As installed, this is reverse video.
- Color 3 holds the colors used for high intensity and is set and cleared by HS and HR.
- Color 4 holds the colors used for blink and is set and cleared by BB and BR.

NOTE: The PROTERMCAP file does not support spacetaking (embedded attribute) terminals. These are terminals that use spaces on the display to hold control codes for video and color capabilities. If you add support for such terminals, Progress ignores the capability when displaying an interface.

3.9 Events

Event portability encompasses the following issues:

- Keyboard events
- Mouse events

3.9.1 Keyboard Events

Although keyboards vary, Progress defines several hundred standard key codes that map to common key labels. The resource files for your environment determine what key sequences on your keyboard map to each Progress key label.

Progress key codes, key labels, and key functions are interrelated. Progress provides a set of built-in functions to convert from one keyboard event to another. The most portable way to specify keyboard events is to use high-level key function identifiers. For example, instead of

using KEYLABEL, you can use the KEYFUNCTION function, which is portable across platforms. In place of **F1** or **F2**, you can use the **GO** function key. See the *Progress Programming Handbook* for more information on Progress key translation functions.

Key Bindings on Windows (Graphical and Character)

Table 3–9 and Table 3–10 list the common key labels that are mapped to each Progress key function on Windows.

Table 3–9: Static Key Bindings on Windows (Graphical and Character)

Key Function	Key Label
BACKSPACE	BACKSPACE
BACK-TAB	SHIFT-TAB
CURSOR-DOWN	CURSOR-DOWN
CURSOR-LEFT	CURSOR-LEFT
CURSOR-RIGHT	CURSOR-RIGHT
CURSOR-UP	CURSOR-UP
DELETE-CHARACTER	DEL
END	END
HOME	HOME
PAGE-DOWN	PAGE-DOWN
PAGE-UP	PAGE-UP
RETURN	ENTER
TAB	TAB

Table 3–10: User-definable Key Bindings on Windows (Graphical and Character)

Key Function	Key Label
APPEND-LINE	No default
BLOCK	No default
BREAK-LINE	No default
CLEAR	No default
DEFAULT-POP-UP	SHIFT-F10
DELETE-LINE	No default
END-ERROR	ESC[1]
FIND	No default
GET	No default
GO	F2[1]
HELP	F1[2]
INSERT-MODE	INS
NEW-LINE	No default
NEXT-FRAME	F6
PREV-FRAME	SHIFT-F6
PUT	No default
RECALL	No default

[1] The **ESC** key is Windows specific; it is only a modifier key in UNIX

[2] The **F1** and **F2** keys are reversed in the Windows character client (**F1**=GO and **F2** =HELP) but can be remapped.

NOTE: Certain keys such as **ESC**, **HOME**, and **END** (Windows specific, no functionality in UNIX) should not be used for porting.

You can change the user-definable key bindings by editing the [Keys] section (for graphical interfaces) or the [WinChar Keys] section (for character interfaces) of the registry or the `progress.ini` file. Use a single entry to create a key binding; additional entries for the same key function are ignored. Key binding entries have the following syntax:

SYNTAX

```
keyfunction=keylabel[,keylabel...]
```

For example, the following entry maps the Help key function to the **F2** and **CTRL-H** key labels, and the Go key function to the **F1** and **CTRL-X** key label:

```
HELP=F2, CTRL-H
GO=F1, CTRL-X
```

If you specify more than one key label, Progress uses the first instance where it needs to display a key label with a key function, as in "Enter data or press F1 to end."

You can use any combination of **CONTROL**, **SHIFT**, and **ALT** keys. However, **SHIFT** only has an affect when used with ASCII keys.

Key Bindings in UNIX

Table 3–11 lists common key bindings in the PROTERMCAP file for UNIX character environments. The PROTERMCAP file maps key functions to key labels. Key bindings might differ for specific terminals.

Table 3–11: Progress PROTERMCAP Key Bindings *(1 of 7)*

Key Function	Key Label
ABORT	CTRL-\ CTRL-ALT-DEL
APPEND-LINE	CTRL-A
BACKSPACE	BACKSPACE CTRL-H DEL-CHAR

Table 3–11: Progress PROTERMCAP Key Bindings *(2 of 7)*

Key Function	Key Label
BACK-TAB	BACK-TAB CTRL-U SHIFT-TAB
BELL	BELL CTRL-G
BLOCK	CTRL-V SELECT
BOTTOM-COLUMN	ESC-CTRL-B
BREAK-LINE	ESC-B
CANCEL-PICK	ESC-CTRL-X
CHOICES	ESC-CTRL-H
CLEAR	F8 CTRL-Z
CLOSE	ESC-Z
COMPILE	ESC-P
COPY	F11 ESC-C
CURSOR-DOWN	CURSOR-DOWN CTRL-J
CURSOR-LEFT	CURSOR-LEFT CTRL-O
CURSOR-RIGHT	CURSOR-RIGHT CTRL-L

Table 3–11: Progress PROTERMCAP Key Bindings *(3 of 7)*

Key Function	Key Label
CURSOR-UP	CURSOR-UP CTRL-K
CUT	F10 ESC-X
DEFAULT-POP-UP	ESC-U
DELETE-CHARACTER	DEL
DELETE-COLUMN	ESC-CTRL-Z
DELETE-END-LINE	ESC-K
DELETE-FIELD	ESC-CTRL-D
DELETE-LINE	CTRL-D REMOVE
DELETE-WORD	ESC-D
EDITOR-BACKTAB	CTRL-B
EDITOR-TAB	CTRL-G
END	END ESC ->
ESC-.
END-ERROR	F4 PF4 ESC CTRL-E
ENTER-MENUBAR	F3 PF3 ESC-M

Table 3–11: Progress PROTERMCAP Key Bindings *(4 of 7)*

Key Function	Key Label
EXIT	ESC-Q
FIND	CTRL-F FIND
FIND-NEXT	ESC-F
FIND-PREVIOUS	ESC-I
GET	F5 ESC-O
HELP	HELP F2 ESC-? PF2
HOME	HOME ESC-H ESC-h ESC-< ESC-,
INSERT-COLUMN	ESC-CTRL-N
INSERT-FIELD	ESC-CTRL-G
INSERT-FIELD-DATA	ESC-CTRL-F
INSERT-FIELD-LABEL	ESC-CTRL-E
INSERT-MODE	INS F9 CTRL-T CTRL-@
LEFT-END	ESC-LEFT-ARROW

Table 3–11: Progress PROTERMCAP Key Bindings *(5 of 7)*

Key Function	Key Label
MAIN-MENU	ESC-CTRL-M
MOVE	ESC-CTRL-V
NEW	ESC-N
NEW-LINE	CTRL-N INSERT-HERE
NEXT-ERROR	ESC-E
NEXT-FRAME	ESC-CTRL-I
NEXT-WORD	CTRL-W
OPEN-LINE-ABOVE	ESC-L
OPTIONS	ESC-CTRL-O
PAGE-DOWN	PAGE-DOWN ESC-CURSOR-DOWN ESC-DOWN-ARROW NEXT-PAGE NEXT-SCRN
PAGE-LEFT	ESC-W
PAGE-RIGHT	ESC-Y ESC-CTRL-J
PAGE-RIGHT-TEXT	ESC-CTRL-J
PAGE-UP	PAGE-UP ESC-CURSOR-UP ESC-UP-ARROW PREV-PAGE PREV-SCRN

Table 3–11: Progress PROTERMCAP Key Bindings *(6 of 7)*

Key Function	Key Label
PASTE	F12 ESC-V
PICK	ESC-CTRL-P
PICK-AREA	ESC-CTRL-W
PICK-BOTH	ESC-CTRL-Q
PREV-FRAME	ESC-CTRL-U
PREV-WORD	CTRL-P
PUT	F6 ESC-S
RECALL	F7 CTRL-R
REPLACE	ESC-R
REPORTS	ESC-CTRL-A
RESUME-DISPLAY	CTRL-Q
RETURN	RETURN CTRL-M
RIGHT-END	ESC-CURSOR-RIGHT ESC-RIGHT-ARROW
SAVE-AS	ESC-A
SCROLL-LEFT	ESC-CTRL-L
SCROLL-MODE	ESC-T
SCROLL-RIGHT	ESC-CTRL-R
SETTINGS	ESC-CTRL-@

Table 3–11: Progress PROTERMCAP Key Bindings *(7 of 7)*

Key Function	Key Label
STOP	CTRL-C CTRL-BREAK
STOP-DISPLAY	CTRL-S
TAB	TAB CTRL-I
TOP-COLUMN	ESC-CTRL-T

To change any of these mappings or create new ones, use the following syntax in the PROTERMCAP file:

SYNTAX

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

action

The name of a key function.

key-label

The key label as it appears on the keyboard.

code

The characters transmitted when the key is pressed.

As in other sections of the PROTERMCAP file, string values are assigned using an equals sign (=) and the field is terminated with a colon (:). This field in a PROTERMCAP terminal entry defines F7 as transmitting a CTRL-A, followed by an uppercase F, followed by a carriage return, and associates use of F7 with the RECALL function:

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

NOTE: If you create a control code sequence for a key beginning with CTRL that uses an existing Progress code sequence, you must assign a new code for that function. For

example, if you specify CTRL-F in a control code sequence, you can no longer use CTRL-F for FIND. Map another key sequence to enable the FIND function.

3.9.2 Mouse Events

Progress supports two types of mouse events: physical events and portable events. You can use portable events to associate triggers with logical actions of any mouse. You can use the physical events to associate triggers with specific physical actions of a mouse.

Progress uses a logical (portable) model to reference mouse button input. This ensures that your application works in any supported operating environment, such as Windows and UNIX. Progress also provides access to the physical mouse input of your operating environment, if necessary. Both forms of input are available as events in a manner similar to keyboard events.

Portable and Physical Buttons

Progress supports four portable mouse buttons:

- SELECT
- EXTEND
- MENU
- MOVE

Although Progress supports two-, three-, and four-button mice, the standard mouse used with Windows has only two buttons. Therefore, some physical mouse buttons must have double functions or you must use control keys with one or more buttons.

Table 3–12 shows the mappings between the Progress portable mouse buttons and the physical mouse buttons on Windows.

Table 3–12: Mouse Buttons for Windows

Portable Button	Function	Windows
SELECT	Select an object by pointing to it and clicking this mouse button. Any previously selected object becomes deselected.	Left mouse button
EXTEND	Toggle the selection state of an object by pointing to it and clicking this button. This has no affect on any other object; therefore, you can use the EXTEND button to toggle the selection individually on more than one object.	**CTRL** with left mouse button
MENU	If an object has an associated pop-up menu, you can activate that menu by clicking this button.	Right mouse button
MOVE	Drag an object on the screen.	Left mouse button

Physical mouse events take priority over any corresponding portable event. If you define a MOUSE-SELECT-CLICK and a LEFT-MOUSE-CLICK trigger for the same widget, Progress only fires the LEFT-MOUSE-CLICK trigger. Therefore, when choosing the events to associate with a trigger, use portable mouse events whenever possible.

Progress automatically maps a portable mouse event to the mouse key used by the native window system to perform the same event. In other words, Progress supports a window system that uses either the left or the right button as the select button. The portable MOUSE-SELECT-CLICK event automatically maps to the right button for that system.

NOTE: If porting to a system that does not support a mouse, you can replace mouse events with functional events, such as SELECTION, ENTRY, and MENU-DROP.

Low-level and High-level Events

Both portable and physical mouse events fall into two classes: low-level and high-level mouse events. You generate low-level mouse events with the simplest mouse button action in a single

direction, such as up or down; and you generate high-level events with more complex actions, such as click or double-click.

Low-level mouse events take priority over corresponding high-level mouse events. For example, if you define a MOUSE-SELECT-UP and MOUSE-SELECT-CLICK trigger on the same widget, only the MOUSE-SELECT-UP trigger executes. Therefore, use high-level mouse events exclusively whenever possible. If you must use low-level events, do not mix low-level and high-level event triggers on the same widget. Processing both classes of events on the same widget can lead to unintended results.

While Progress always recognizes corresponding low-level and high-level events, it executes only one trigger for both. Progress looks first for a trigger on a low-level event. If there is no trigger on a low-level event, Progress looks for a trigger on a high-level event. Once it finds and fires a trigger, it waits for the next event. However, each graphical interface might recognize and pass high-level events to Progress after recognizing a different combination of low-level events. For example, one system might recognize a double-click on a down-mouse event and another system might recognize the double-click on an up-mouse event. This causes mixed event sequences to differ significantly among interfaces.

Mixing low-level and high-level mouse events creates portability problems among graphical interfaces. Furthermore, the expected event sequences in the same graphical interface might change in future versions of that interface.

Table 3–13 lists the mouse events that apply to all mouse devices, regardless of how the buttons are configured, and indicates the level of each event.

Table 3–13: Portable Mouse Events *(1 of 2)*

Event	User Action	Level
MOUSE-SELECT-DOWN	Press the mouse SELECT button.	Low
MOUSE-SELECT-UP	Release the pressed mouse SELECT button.	Low
MOUSE-SELECT-CLICK	Press and release the mouse SELECT button.	High
MOUSE-SELECT-DBLCLICK	Press and release the mouse SELECT button twice.	High
MOUSE-EXTEND-DOWN	Press the mouse EXTEND button.	Low
MOUSE-EXTEND-UP	Release the pressed mouse EXTEND button.	Low

Table 3–13: Portable Mouse Events *(2 of 2)*

Event	User Action	Level
MOUSE-EXTEND-CLICK	Press and release the mouse EXTEND button.	High
MOUSE-EXTEND-DBLCLICK	Press and release the mouse EXTEND button twice.	High
MOUSE-MENU-DOWN	Press the mouse MENU button.	Low
MOUSE-MENU-UP	Release the pressed mouse MENU button.	Low
MOUSE-MENU-CLICK	Press and release the mouse MENU button.	High
MOUSE-MENU-DBLCLICK	Press and release the mouse MENU button twice.	High
MOUSE-MOVE-DOWN	Press the mouse MOVE button.	Low
MOUSE-MOVE-UP	Release the pressed mouse MOVE button.	Low
MOUSE-MOVE-CLICK	Press and release the mouse MOVE button.	High
MOUSE-MOVE-DBLCLICK	Press and release the mouse MOVE button twice.	High

Table 3–14 lists the mouse events associated with physical mouse buttons and indicates the level of each event.

Table 3–14: Physical Mouse Events *(1 of 2)*

Event	User Action	Level
LEFT-MOUSE-DOWN	Press the left mouse button.	Low
LEFT-MOUSE-UP	Release the pressed left mouse button.	Low
LEFT-MOUSE-CLICK	Press and release the left mouse button.	High
LEFT-MOUSE-DBLCLICK	Press and release the left mouse button twice.	High
RIGHT-MOUSE-DOWN	Press the right mouse button.	Low

Table 3–14: Physical Mouse Events *(2 of 2)*

Event	User Action	Level
RIGHT-MOUSE-UP	Release the pressed right mouse button.	Low
RIGHT-MOUSE-CLICK	Press and release the right mouse button.	High
RIGHT-MOUSE-DBLCLICK	Press and release the right mouse button twice.	High
MIDDLE-MOUSE-DOWN	Press the middle mouse button.	Low
MIDDLE-MOUSE-UP	Release the pressed middle mouse button.	Low
MIDDLE-MOUSE-CLICK	Press and release the middle mouse button.	High
MIDDLE-MOUSE-DBLCLICK	Press and release the middle mouse button twice.	High

4

Programming Considerations

Most of the functionality of your application resides inside triggers. Your application's trigger code performs the tasks for which you created the application. This chapter discusses portability issues involved with these internal tasks, including issues involved with database and operating system access.

This chapter covers the following topics:

- Naming conventions for files, tables, and variables
- Terminal key definitions
- Operating system statements
- R-code
- Coding style
- Storing data
- Retrieving data
- Security
- Accessing C functions
- Interprocess communications

4.1 Naming Conventions for Files, Tables, and Variables

Different operating systems have different restrictions and naming conventions. To ensure portability across platforms, use the following conventions when naming your fields, files, tables, and variables.

4.1.1 Characters to Avoid

Several characters have special meanings or are invalid in one or more supported operating systems. Avoid using these characters:

```
/ \ " ' * ; | ? [ ] ( ) ~ ! $ { } < >
```

Do not use a hyphen as the first character of a filename. Do not use spaces in filenames; spaces delimit filenames.

4.1.2 Name Length

Use a maximum filename length that takes into account your software platform (Windows or UNIX) and your file networking software. Although Windows and UNIX support long filenames, some file networking software does not. For maximum portability, use operating system filenames that have a maximum of eight characters (not including the extension).

Database table names can be from 1 to 32 characters; however, to ensure portability, make the first eight characters unique.

In Windows, if your Progress table names are more than eight characters and the first eight characters of some of the table names are the same, you might not be able to use the standard Data Dictionary dump/reload utilities. These procedures dump data to a text file whose name is the table name with a .d extension. These dump files might overwrite each other. The Data Dictionary generates an error message in this case.

Although you can override the default dump/reload name, avoid duplicate names. Avoid using procedure names that are longer than eight characters or that contain periods. A filename extension of .p or .w is recommended.

Windows lets you create bitmap (`.BMP`), icon (`.ICO`), and cursor (`.CUR`) files for your applications. Bitmap files define monochrome or color bitmaps. Icon and cursor files define a family of images. For maximum flexibility, omit file extensions when referencing an image file to allow Progress to search for a file with the appropriate file extension. Progress first searches the current working directory and then searches the PROPATH for matching filenames and extensions that a platform supports. Windows searches in this order: `.BMP`, `.ICO`, `.CUR`. For more information on images, see Chapter 3, "User Interface Design."

4.1.3 Case Sensitivity

Although operating system filenames are not case sensitive in Windows, they are in UNIX. On systems where operating system filenames are case sensitive, filenames and command names in uppercase are different from those in lowercase. By convention, lowercase is used for most UNIX filenames.

To ensure portability among operating systems, use lowercase when specifying a procedure name in a RUN statement, and make sure your procedure files have lowercase names on UNIX.

4.1.4 File Path Specifications

You can use UNIX pathname syntax in Progress procedures on Windows. Progress automatically converts the UNIX pathname syntax to Windows pathname syntax. This can help maintain portability among operating systems.

Use forward slashes (/) as separators in specifying file paths when using the RUN, INPUT FROM, or OUTPUT TO statements. Although Progress running in Windows supports both forward slashes (/) and backslashes (\), Progress running in UNIX supports only forward slashes. Progress automatically converts forward slashes in pathnames to backslashes. Therefore, use forward slashes for the best portability across operating systems.

Progress can only reference UNIX pathnames of up to 60 characters and Windows pathnames of up to 255 characters.

Use environment variables and relative paths instead of hard coding full paths. This allows you to make global changes more easily and creates more meaningful names. This also helps Progress applications appear customized for each environment.

4.2 Terminal Key Definitions

The operating system you port to might expect certain terminal key mappings that do not exist in the environment where you developed your application.

Alternatively, your code might rely on unique key mappings or operating-system-specific terminal control information stored in the user-interface environment files: the PROTERMCAP file in UNIX and the registry or a `progress.ini` file in Windows. For more information on environments, see Chapter 3, “User Interface Design,” and the *Progress Client Deployment Guide*.

Consider the following portability issues:

- If you are planning to port your applications from one operating system to another, use the tilde (~) as an escape character in your application. While versions of Progress running on

UNIX interpret both the backslash (\) and the tilde as an escape character, versions of Progress running in Windows interpret only the tilde as an escape character.

- Wherever possible, use spaces rather than tabs. Spaces are consistent in size regardless of the operating system, but tabs might vary.
- Use KEYFUNCTION in code logic, which is portable across platforms. Use KEYLABEL in messages to the user.

For more information on keyboard and other events, see Chapter 3, "User Interface Design."

4.3 Operating System Statements

To avoid the limitations of operating-system-specific statements, use the OPSYS function to determine the appropriate operating system. The OPSYS function allows you to build one application that can run on more than one operating system. The OPSYS function identifies the operating system being used so that a single version of a procedure can work differently on different operating systems.

For example, the following procedure produces a listing of the files in your current directory. The OPSYS function determines the operating system you are running Progress on and uses the appropriate operating system command to produce the directory listing:

```
CASE OPSYS:
  WHEN "unix" THEN OS-COMMAND ls.
  WHEN "win32" THEN OS-COMMAND dir.
  OTHERWISE MESSAGE OPSYS "is an unsupported operating system".
END CASE.
```

To use operating system commands from within Progress, use either the operating system statement for the operating system you are running on or the OS-COMMAND statement. For example, use the OS-COMMAND statement on a Windows system and on a UNIX system. If you include the operating system statement in a procedure on a system other than the one named, the procedure compiles but does not run if the flow of control passes through that operating system statement. Progress might report an error when it tries to process an operating system statement on a system other than the one named. See the *Progress Language Reference* for more information on the OPSYS function.

You can also use the built-in preprocessor directive, {&OPSYS}, which expands to a character string that contains the name of the operating system that the file is being compiled on.

The preprocessor is a component of the Progress Compiler. You control the preprocessor by placing preprocessor directives throughout your source code.

The {&OPSYS} preprocessor directive allows you to write code that is conditionally compiled, while the OPSYS function is a run-time function. For more information on preprocessor names, see the "Preprocessor" chapter of the *Progress Programming Handbook*.

4.3.1 Standard System Commands

Where possible, use standard system commands instead of variants unique to certain hardware manufacturers. Table 4–1 lists the OS statements that promote portability by avoiding limitations of operating-system-specific commands. These statements allow you to create a single version of a procedure that works differently on varying operating systems.

Table 4–1: OS Statements *(1 of 2)*

OS Statement	Description
OS-APPEND	Performs a system call to execute the operating system command that appends two files.
OS-COMMAND	Performs a system call to execute an operating system command that you specify.
OS-COPY	Performs a system call to execute the operating system command that copies a file.
OS-CREATE-DIR	Performs a system call to execute the operating system command that creates a new directory.
OS-DELETE	Performs a system call to execute the operating system command that deletes a file or directory. You can delete one or more files, a directory, or an entire directory branch.
OS-DRIVES	Performs a system call to execute the operating system command that returns a comma-separated list of the available drives in Windows.
OS-ERROR	Returns a Progress error code that indicates whether an execution error occurred during the last OS-APPEND, OS-COMMAND, OS-COPY, OS-CREATE-DIR, OS-DELETE, or OS-RENAME statement.

Table 4–1: OS Statements *(2 of 2)*

OS Statement	Description
OS-GETENV	Performs a system call to execute the operating system command that returns a string containing the value of the specified environment variable in the environment in which Progress is running.
OS-RENAME	Performs a system call to execute the operating system command that renames a file or directory.

See the *Progress Language Reference* for more information on these statements.

OS-COMMAND

Use the OS-COMMAND statement to execute an operating system statement that you cannot execute using the OS statements listed in Table 4–1. The OS-COMMAND statement provides a generic, operating-system-independent way to escape to the current operating system, which lets you:

- Execute a Progress or operating system command that has the same syntax on two or more different operating systems
- Start an operating system shell
- Execute an operating system statement that a user enters

The arguments to OS-COMMAND must be appropriate for the current operating system. Therefore, where possible, read these arguments at run time from the user, database table, or environment variables rather than hard coding them. The following procedure prompts the user for an operating system command, then uses the OS-COMMAND statement to execute the command:

```
DEFINE VARIABLE comm-line AS CHARACTER FORMAT "x(70)".
REPEAT:
     UPDATE comm-line.
     OS-COMMAND VALUE(comm-line).
END.
```

The OS-COMMAND statement eliminates the need to use the OPSYS function to determine the operating system. However, if you cannot use the OS-COMMAND statement, use the OPSYS function to determine the operating system you are running on, and use conditional logic to execute the appropriate code using one of the operating-system-specific escape statements.

The NO-WAIT option of OS-COMMAND is valid only in multi-tasking environments. This option causes Progress to pass control to the statement following the OS-COMMAND, without waiting for the operating system command to terminate. If you are using the OS-COMMAND statement to run an independent Windows application, use the NO-WAIT option.

For more information, see the OS-COMMAND Statement reference entry in the *Progress Language Reference*.

INPUT FROM OS-DIR

You generally use the INPUT FROM statement to read the contents of an operating system file; however, you can also read a list of the files in a directory using the OS-DIR option of the INPUT FROM statement. The INPUT FROM statement specifies a new input source. Using INPUT FROM OS-DIR indicates that you want your input to be the filenames found in the directory you specify. If that directory is not a valid directory or you do not have permission to read it, an error condition occurs. Otherwise, Progress generates the directory list and sends it to the calling program through the INPUT stream. An INPUT CLOSE statement discards any unread filenames from the list.

The following example uses the OS-GETENV function to find the path of the `DLC` directory, then uses the OS-DIR option of INPUT FROM to read the contents of the directory:

```
DEFINE VARIABLE search-dir AS CHARACTER.
DEFINE VARIABLE file-name AS CHARACTER FORMAT "x(16)" LABEL "File".
DEFINE VARIABLE attr-list AS CHARACTER FORMAT "x(4)" LABEL "Attributes".
search-dir = OS-GETENV("DLC").
INPUT FROM OS-DIR(search-dir).
REPEAT:
   SET file-name ^ attr-list
          WITH WIDTH 70 USE-TEXT TITLE "Contents of " + search-dir.
END.
INPUT CLOSE.
```

When you use the OS-DIR option, the UNBUFFERED option is ignored. OS-DIR always buffers exactly one filename at a time. When you try to read beyond the last filename in the list, Progress generates the ENDKEY condition.

See the *Progress Language Reference* for more information on the INPUT FROM statement.

4.4 R-code

R-code is the executable code that Progress generates when you compile Progress source code.

This section briefly covers:

- R-code portability and operating system-specific statements
- R-code-imposed limits on program size
- Using cyclic redundancy checks (CRCs) instead of time stamps to enforce consistency between schema and trigger
- Choosing r-code or encrypted source code to ship to customers
- Byte-order of integers in r-code

NOTE: For more information on r-code, see Chapter 1, "About Progress Portability," the "R-code Portability and DataServers" section in Chapter 2, "Database Design," and the r-code appendix of the *Progress Programming Handbook*.

4.4.1 R-code Portability and Operating System-specific Statements

If you want your r-code to be portable, avoid Progress statements that are operating-system specific. For example, the DDE INITIATE statement is specific to Windows.

4.4.2 R-code-imposed Limits on Program Size

R-code does not impose a limit on program size because it is made up of multiple segments, each of which can grow to a maximum of 60K.

4.4.3 Using CRCs Instead of Time Stamps

If you have r-code that contains database triggers and you intend to run the r-code on different systems, you must use cyclic redundancy checks (CRCs), as opposed to time stamps, to ensure consistency between the database schema and the trigger.

NOTE: By default, Progress uses CRCs.

4.4.4 Choosing R-code or Encrypted Source Code to Ship to Customers

Portable r-code is an alternative to encrypted source code. Rather than shipping encrypted source code, which customers must compile, you can ship r-code. You might ship encrypted source code, however, to allow customers to modify include files or modify the schema.

4.4.5 Byte Order of Integers in R-code

R-code, whether on disk or in memory, always stores integers in byte-swapped order, even on machines that store integers in byte-stream format. On such machines, Progress converts integers to byte-stream format at run time.

4.5 Coding Style

For the best portability, isolate your user-interface code using the following techniques:

- Using preprocessor directives
- Using VIEW-AS with the DEFINE VARIABLE statement

4.5.1 Using Preprocessor Directives

Progress provides a language preprocessor that allows you to write applications that are easy to read, modify, and transport to other operating systems. The preprocessor is a component of the Progress Compiler. Before the Compiler analyzes your source code and creates r-code, the preprocessor examines your source code and performs text substitutions.

You control the preprocessor by placing preprocessor directives throughout your source code. A preprocessor directive is a statement that begins with an ampersand (&) and is meaningful only to the preprocessor.

The preprocessor recognizes a built-in preprocessor constant that identifies the window system where a file is being compiled, called &WINDOW-SYSTEM. The possible values include "MS-WINnn" and "TTY." You can use this constant to direct the preprocessor to determine which code to run on each window system, as follows:

```
FORM
  fld1
  fld2
&IF "{&WINDOW-SYSTEM}" = "MS-WIN97" &THEN
  fld3 AT 20
&ELSE
  fld3 AT 15
&ENDIF
   WITH FRAME XYZ.
```

The SESSION system handle also has a WINDOW-SYSTEM attribute. Your application can use this to test the current window system while it is running:

```
IF SESSION:WINDOW-SYSTEM = "MS-WIN97" THEN
                      .
                      .
                      .
```

Both the &WINDOW-SYSTEM preprocessor constant and the SESSION handle WINDOW-SYSTEM attribute perform the same basic function. The &WINDOW-SYSTEM preprocessor constant allows you to write code that is conditionally compiled, while the WINDOW-SYSTEM attribute is a run-time function.

The WINDOW-SYSTEM attribute evaluates as follows:

- If Windows *nn* or Windows NT is running, and the Windows *nn* user interface is running, this attribute evaluates to MS-WIN*nn*. Otherwise, if the Windows *nn* user interface is not running, it evaluates to MS-WINDOWS.
- If the application is not running in a Windows environment, this attribute evaluates to TTY.

Progress supports an override option that enables applications that require a WINDOW-SYSTEM attribute to return the value of MS-WINDOWS for all Microsoft operating systems. To establish this override capability, define the Window System key in the Startup Section of the current environment, which might be in the registry or an initialization file. If the Window System key is located, the WINDOW-SYSTEM attribute returns the value associated with the Window System key on all platforms.

You can also specify offsets and ROW and COLUMN specifications using preprocessor constants, then define the constants separately for different environments. Progress allows you to specify fractional character units so that you can specify precise locations for objects in a graphical environment. In a character environment, the ROW and COLUMN values are truncated to integer values, as follows:

```
&IF "{&WINDOW-SYSTEM}" = "TTY" &THEN
&GLOBAL-DEFINE COL3 20
&ELSE
&GLOBAL-DEFINE COL3 15
&ENDIF
FORM
  fld1
  fld2
  fld3 AT {&COL3}
  WITH FRAME XYZ.
```

4.5.2 Using the VIEW-AS Phrase

If possible, use the VIEW-AS phrase with the DEFINE VARIABLE statement instead of with screen I/O statements. The VIEW-AS phrase describes how a field or variable is represented on the screen. You can use the VIEW-AS phrase as a modifier to the DEFINE VARIABLE statement or as a modifier to a screen I/O statement, such as DISPLAY.

With the DEFINE VARIABLE statement, the VIEW-AS phrase specifies the default type of widget for the variable being defined. Use the VIEW-AS phrase when defining a variable so that code that manipulates the variable, such as an UPDATE statement, does not have to specify the representation. This helps you isolate your user-interface code.

For more information on the VIEW-AS phrase, see the *Progress Language Reference*.

4.6 Storing Data

To start working with data, you first have to copy it from the database. When your procedure copies data from a database, Progress stores that data in memory. When you request any data from a database record, Progress places a copy of the entire database record in a buffer. For each table that you work with, Progress creates and maintains a separate record buffer. Data visible on screen is also stored in memory. As with record buffers, Progress maintains screen buffers for you.

4.6.1 Storing Database Triggers

You can store a database's trigger procedures in an operating system directory or in a Progress r-code library. There are advantages and disadvantages with each storage technique.

When developing the application, it is usually better to store the procedures in a directory. Typically, when deploying or porting an application to another platform, it is advantageous to store your trigger procedures in an r-code library. This is usually faster than running the procedures from an operating system directory, even if the procedures are precompiled. However, you must precompile procedures run from a library. The Trigger (-trig) startup parameter allows you to specify the directory or r-code library containing your trigger procedures.

4.7 Retrieving Data

When you start a Progress application, the relational database management system (RDBMS) uses the commands in the startup script to connect the necessary databases. All the objects (fields, tables, indexes, etc.) contained in the connected databases make up the set of objects that your application can reference. Your procedures interact with the RDBMS to access these database objects.

4.7.1 Using Logical Database Names

When you connect to a database, Progress automatically assigns that database a default logical name for the current Progress session. Progress uses the logical database name to resolve database differences, and stores it in the compiled r-code of any procedures that you compile that refer to the database. Logical database names allow you to change physical databases without recompiling an application.

Use logical database names to maintain consistency in naming conventions when porting your applications to other platforms.

4.7.2 Referencing Database Records

The record phrase is the language structure for referencing the table you want to work with and for helping to specify an individual record or a set of records.

The reference to the table name is both a reference to the database table and the record buffer associated with the table. When two connected databases contain identically named tables, you must specify the database name in the record phrase to make the reference unique, as follows:

SYNTAX

```
database-name.table-name
```

Progress flags all ambiguous references at compile time. If you intend to port your application, always specify the full reference to avoid ambiguity and make code more readable.

4.7.3 Referencing Database Fields

After you retrieve a record, you can work selectively with some of the fields in the record buffer. To avoid ambiguity at this level, use the following syntax:

SYNTAX

```
database-name.table-name.field-name
```

4.7.4 Using the FIND and GET Statements

The FIND statement retrieves an individual record and is the basis for the triggers associated with the navigation buttons. With FIND, you provide enough information for your procedure to access the database and find the record that satisfies your criteria. FIND returns a record from the database. In other words, FIND copies a record from a database to a record buffer.

Similar to the FIND statement, the GET statement returns a record from the database. However, the GET statement returns a record from the database by way of a defined query. With a defined query, the DEFINE QUERY and OPEN QUERY statements provide enough information for Progress to build a result list of all the records that match your criteria. The GET statement works with the result list to determine the correct record and then copies that record from the database to a record buffer.

A defined query lets you build a result list of records that satisfy some criteria. When you use a defined query, Progress knows about all the records that satisfy the criteria. Conversely, when you use FIND, Progress only knows about a single record that satisfies your criteria.

Unlike the FIND statement, the GET statement does not return records in a predictable order by default. The FIND statement uses the primary index of the database table. The defined query might use any index or combination of indexes to order the records in the retrieval process. If you require a definite order, do not rely on the query default. Instead, use the record phrase options to specify an order or use the FIND statement.

4.7.5 Using the Browse Widget

The browse widget represents the result list of a defined query. A browse widget is like a selection list of database records. Instead of listing a single value on each line of the list, browse widgets let you list data from records currently in your environment.

Working with a browse widget usually represents the first step in a user task. The user finds and selects a record with the browse widget and then manipulates data from the record.

An enabled browse widget:

- Accepts input focus by way of the navigation key functions (such as **TAB**) or the mouse

- Responds to the universal key functions (such as **GO** or **HELP**)
- Moves input focus among the records with the cursor keys
- Responds to ENTRY, LEAVE, and VALUE-CHANGED triggers

If you intend to port your graphical application to a character interface, consider the different ways users might interact with the browse widget when you design your interface.

4.7.6 Record Locking

Progress supports two types of record locking-shared and exclusive. When working with DataServers, you must be aware that non-Progress database managers implement different record locking procedures. For example:

- The DataServer for C-ISAM does not support shared record locks; all record locks are exclusive.
- C-ISAM provides two types of record locking (automatic and manual) and two types of file locking (exclusive and manual).
- The DataServer for C-ISAM uses manual locking at the table and record levels. It does not support C-ISAM automatic record locking. The DataServer opens files with manual locks, allowing multiple programs to access the files.

Progress applications rely on the ORACLE RDBMS to handle all locking for the target ORACLE database. Progress locks do not apply to your ORACLE database.

The DB2/400 database supports NO-LOCK, SHARE-LOCK, and EXCLUSIVE-LOCK when you use the Progress Lock Table on the server. If non-Progress applications are running against the database, do not use the Progress Lock Table and limit your access to NO-LOCK and EXCLUSIVE-LOCK.

See the appropriate *Progress DataServer Guide* for a comparison of Progress locks to their non-Progress equivalents.

4.8 Security

Progress table and field security checking can be based on user IDs. UNIX assigns a user ID to each user; Windows does not unless security has been implemented. However, you can use Progress security with Windows by prompting the user for a user ID and password as part of the login process.

To obtain the current user ID, use the Progress USERID function. Use this function when you want to allow only one user ID to a procedure.

On operating systems where security has been implemented, the USERID function returns the user ID of the user executing the function. In Windows where security is not implemented, the function returns a null value. While blank user IDs can connect to the database, they cannot access data protected by compile-time and run-time security.

After Progress starts running, you can use the SETUSERID function to change the current user ID. The SETUSERID function returns a true value and assigns the user ID to the user if the user ID and password are in the _User table. If the user ID is not in the _User table or the password is incorrect, SETUSERID returns a false value and does not assign the user ID to the user.

You can also use the CAN-DO function to perform security checking. The CAN-DO function compares the current user ID with a list of users who have permission to access a specified file.

For more information on the USERID, SETUSERID, and CAN-DO functions, see the *Progress Language Reference*.

User IDs and Progress passwords are case sensitive. For more information on security and user IDs, see the *Progress Database Administration Guide and Reference* and the *Progress Programming Handbook*.

Using DataServers involves following the security guidelines required by both Progress and ORACLE, SYBASE, C-ISAM, and DB2/400. By default, Progress does not impose security on databases (it only allows unrestricted access to databases), so at a minimum, you must follow the guidelines required for your application. See the appropriate *Progress DataServer Guide* for more information on security.

4.9 Accessing C Functions

The Progress Host Language Call (HLC) Interface provides a technique to call C language functions from within Progress applications. You can use Progress HLC to add portable custom features to Progress.

If you intend to port your extensions to other environments, use HLC. If you intend to build extensions for use with Windows only, use dynamic link libraries (DLLs).

A DLL is a module that contains a collection of compiled functions or procedures that Windows applications can access. An application links to these routines at run time rather than at build time, and shares the DLL code with other applications that link to them. Progress lets you link and execute DLL routines from a 4GL procedure. Using these routines, you can write applications that perform a wide range of Windows functions.

DLL entry points work only with Windows (GUI and character). For more information on dynamic link libraries, see the *Progress External Program Interfaces* manual.

To ensure portability, avoid accessing DLL entry points provided by Windows and third-party developers, or use the preprocessor to isolate this type of code. For more information on the OPSYS preprocessor name, see the "Operating System Statements" section. For more information on isolating your code using the preprocessor, see Chapter 3, "User Interface Design," or refer to the *Progress Programming Handbook*.

If you intend to port your HLC application, make your C functions as portable as possible by following these guidelines:

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

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

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

For information on building an HLC executable and a complete listing of all HLC library functions, see the *Progress External Program Interfaces* manual.

4.10 Interprocess Communications

Progress supports several operating-system-dependent techniques of interprocess communication.

4.10.1 I/O with Other Processes

Use the INPUT THROUGH statement to pipe data into Progress from a UNIX or Windows NT process. Use the OUTPUT THROUGH statement to pipe data from Progress to another UNIX or Windows NT process.

You can also use the INPUT-OUTPUT THROUGH statement to pipe the output of a UNIX (or Windows NT) process into a Progress procedure and to pipe data from Progress back to that same UNIX (or Windows NT) process. This allows two-way communication to a program written at the operating system level. You might use this capability to do specialized calculations on data stored in a Progress database or entered during a Progress session.

To ensure portability, avoid using the INPUT THROUGH and OUTPUT THROUGH statements. In place of OUTPUT THROUGH, use the OUTPUT TO statement with an escape to the operating system, as shown in the following example:

```
OUTPUT TO file1.
      .
      .
      .
OUTPUT CLOSE.
CASE OPSYS:
   WHEN "UNIX" OR "Win32" THEN OS-COMMAND program-name < file1.
   OTHERWISE MESSAGE OPSYS "Operating system not supported".
```

In this example, you send the data to `file1`. Then you use 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`.

In place of INPUT THROUGH, use an escape to the operating system with the INPUT FROM statement, as shown in the following example:

```
CASE OPSYS:
   WHEN "UNIX" THEN OS-COMMAND program-name > file1.
INPUT FROM file1.
      .
      .
      .
INPUT CLOSE.
```

First, you use 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`.

4.10.2 Named Pipes

On UNIX and on Windows NT, a non-Progress application (such as a C program or commercial software package) can communicate with a Progress application using named pipes.

Named pipes provide a general exchange mechanism for text data. There is no practical limit to the types of data you can exchange using named pipes. Any data you can access as a character string within Progress, you can read or write to a named pipe.

Progress accesses named pipes already created on your UNIX or Windows NT systems. Progress does not create a named pipe itself. Progress treats a named pipe the same way it treats a text file.

To ensure portability, avoid using named pipes; they are only valid on UNIX or Windows NT systems. See the *Progress External Program Interfaces* manual for more information on named pipes.

4.10.3 System Clipboard

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

CLIPBOARD System Handle

The CLIPBOARD system handle allows you to transfer data between the window system Clipboard and your Progress application. Using the CLIPBOARD attributes, you can paste (read) data from the Clipboard to a Progress field or variable and copy or cut (write) data from a field or variable to the Clipboard.

In Progress, you also have a choice of two data-transfer modes to implement these operations-single-item transfers and multiple-item transfers. In single-item transfers, a single write to the Clipboard immediately replaces all data in the Clipboard, and a single read from the Clipboard returns all data in the Clipboard to the Progress application. In multiple-item transfers, you can format the data transfer into multiple rows of multiple items. Each write to the Clipboard adds an item to a tab and new line-separated list of Clipboard items. Each read from the Clipboard returns one tab or new line-separated item to your Progress application. In one step, you can transfer aggregate units of data between Progress and other applications (such as spreadsheets) that support similar aggregate Clipboard operations.

These data transfers are made with the help of the CLIPBOARD handle attributes. For more information on handle attributes, see the *Progress Language Reference*.

Windows provides default Clipboard operations through control keys, whether or not you implement them with the CLIPBOARD handle. These operations are completely compatible with CLIPBOARD handle operations as single-item operations without any interaction with the MULTIPLE attribute. Also, these operations can be used if an interruption occurs in a 4GL Clipboard operation. Table 4–2 lists Windows default Clipboard operations and the control keys that activate them.

Table 4–2: Clipboard Operations

Operation	Control Key for Editors and Fill-ins	Control Key for Fill-in Fields Only
Cut	**CTRL-X**	**SHIFT-DEL**
Copy	**CTRL-C**	**CTRL-INS**
Paste	**CTRL-V**	**SHIFT-INS**

Supported Environments

In character environments where there is no system Clipboard, Progress supports CLIPBOARD handle operations within a single Progress application. You can cut and paste between fields in one Progress application, but not between one Progress application and another Progress or non-Progress application. In graphical environments, Progress supports CLIPBOARD handle operations using the system Clipboard. This allows data transfers between Progress and non-Progress applications as well as within a single Progress application.

For more information on how Clipboard operations work in Windows, see the *Microsoft Windows User's Guide*. For general information on the system Clipboard, see the *Progress External Program Interfaces* manual.

Index

A

Absolute layout 3–11
  using the UIB 3–11

Arrays 2–4

AT phrase
  COLON-ALIGNED 3–11
  LEFT-ALIGNED 3–11
  RIGHT-ALIGNED 3–11
  specifications 3–11

Attributes, of widgets 3–14

Audience vii

B

BGCOLOR 3–23

Bold typeface, as typographical convention viii

Browse widget 3–18, 4–13
  events 4–13

Buffers
  record 4–11
  screen 4–11

Buttons 3–16
  default 3–16
  images and labels 3–18
  layout 3–16
  using 3–17

C

C functions
  accessing 4–15
  portable 4–16

CAN-DO function 4–15

Case sensitivity, naming conventions 4–3

Character mode
  keybindings 3–37
  PROTERMCAP file 3–32
  specifying colors 3–33

Character set. *See* Code pages

Character units 3–12

Clipboard
  supported environments 4–19
  system handle 4–18

Code pages 2–3

Color table system handle 3–23

Colors

adding 3–23
available 3–29
background 3–23
bitmaps, in Windows 3–31
defaults in character mode 3–33
defaults in Windows 3–30
definitions 3–23
display portability 3–23
dynamic 3–23
foreground 3–23
HI-COLOR 3–31
palette, in Windows 3–31
restrictions, in Windows 3–31
specifying in PROTERMCAP file 3–33
static 3–23
syntax in PROTERMCAP file 3–33
TRUE-COLOR 3–31

Commands, operating system 4–5

COMPILE statement 2–4

Conventions
case sensitivity 4–3
characters to avoid 4–2
database names 4–12
duplicate names 4–2
file naming 4–2
name length 4–2

CREATE Widget statement 3–22

CREATE WINDOW statement 3–21

Custom features, adding to Progress 4–15

D

Data Manipulation Language 2–4

Data types 2–3

Database
default names 4–12
design 2–2
fields 4–13
logical names 4–12

Database objects
naming 2–3
terminology 2–3

Database records, referencing 4–12

DataServers 2–2

DBRESTRICTIONS function 2–4

DCOLOR 3–22

Default window section 3–32

DEFINE FRAME statement 3–19

DEFINE QUERY statement 4–13

DEFINE VARIABLE statement 4–11

Display format, fill characters 3–13

Display order, widgets 3–14

Display portability 3–1
color 3–23
fonts 3–24
widgets 3–14

Displays, dimensions 3–21

Dynamic colors 3–23

Dynamic Link Libraries, in Windows 4–15

E

Editor widget 3–18

Environment settings, in Windows 3–25

Environments, user interface 3–25

Error messages, displaying descriptions xii

Events 3–34
keyboard 3–34
mouse 3–44

F

FGCOLOR 3–23

Field extents. *See* Arrays

Fields, database references 4–13

File
  extensions 3–18
  naming conventions 4–2
  path specifications 4–3
  resource 3–25

Filenames, case sensitivity 4–3

FIND statement 2–4, 4–13

Font table system handle 3–24

Fonts
  available 3–28
  definitions 3–24
  display portability 3–24
  GET-TEXT-WIDTH 3–24
  specifying 3–28
  with UIB 3–24

FORM statement 3–19

Format
  fill characters 3–13
  screen dimensions 3–20, 3–21

Frame widget 3–19

Frames
  absolute positioning 3–11
  attributes 3–19

G

GET statement 4–13

H

Help, Progress messages xii

Host Language Call Interface 4–15

I

I/O with other processes 4–17

IDs and passwords 4–14

Images 3–17
  system arrows 3–17

INNER-LINES/CHARS constructs 3–18

INPUT FROM statement
  OS-DIR option 4–7

INPUT THROUGH statement 4–17

INPUT-OUTPUT THROUGH statement 4–17

Interprocess communications 4–16
  I/O with other processes 4–17
  named pipes 4–18

Isolating code 4–9

Italic typeface, as typographical convention viii

K

Key
  codes 3–35
  definitions 4–3
  functions 3–35
  labels 3–35

Key bindings
  static 3–35
  syntax in progress.ini file 3–37
  syntax in PROTERMCAP file 3–43
  user definable 3–35

Keyboard events 3–34

KEYFUNCTION 4–4

KEYLABEL 4–4

Keystrokes viii

L

Layout, character units and pixels 3–12

Length of filenames and tables 4–2

Locking, of records 4–14

Logical database names 4–12

M

Manual
  organization of vii
  syntax notation ix

Maximized window, attributes 3–22

Messages, displaying descriptions xii

Monospaced typeface, as typographical convention viii

Mouse
  buttons in UNIX 3–45
  buttons in Windows 3–45
  events 3–44
    high-level 3–45
    low-level 3–45
  physical buttons 3–44
  physical events 3–47
  portable buttons 3–44
  portable events 3–46

Moving data 4–11

N

Named pipes, on UNIX 4–18

Names, logical database 4–12

Naming conventions
  case sensitivity 4–3
  characters to avoid 4–2
  database names 4–12
  duplicate names 4–2
  file path specifications 4–3
  files, tables, variables 4–2
  name length 4–2

NO-WAIT option
  OS-COMMAND 4–7

O

Objects, database 2–3

OPEN QUERY statement 4–13

Operating system
  commands 4–5
  statements 4–4

Operating system file
  reading contents
    using INPUT FROM 4–7

OPSYS
  function 4–4
  preprocessor directive 4–5

OS statements 4–5

OS-APPEND 4–6

OS-COMMAND 4–6
  NO-WAIT option 4–7

OS-COPY 4–6

OS-CREATE-DIR 4–6

OS-DELETE 4–6

OS-DIR
  INPUT FROM statement 4–7
  UNBUFFERED option 4–7

OS-DRIVES 4–6

OS-ERROR 4–6

OS-GETENV 4–6

OS-RENAME 4–6

OUTPUT THROUGH statement 4–17

P

Pathnames, syntax 4–3

PFCOLOR 3–22

Pixels 3–12

Preprocessor
  creating portable applications 4–9
  directives 4–9
  &WINDOW-SYSTEM 4–10

progress.ini file 3–25
  available colors 3–29
  available fonts 3–28
  default window section 3–32
  key bindings 3–35
  keys section 3–37
  search order 3–27
  specifying fonts 3–28
  startup specifications 3–27

PROPATH 3–24

PROTERMCAP file 3–32
  color syntax 3–33
  keybindings 3–37
  sections 3–32
  size restrictions 3–32
  specifying colors 3–33

R

R-code 2–4
  portable 4–8

Record buffers 4–11

Record locking 4–14

Records, referencing 4–12

Rectangles 3–17

References
  ambiguous 4–12
  database fields 4–13
  database records 4–12

Registry 3–25
  colors 3–29
  default window section 3–32
  fonts 3–28
  startup specifications 3–27

Resource files 3–25

Retrieving data 4–12

ROWID function 2–4

S

Schema objects. *See* Database objects

Screen
  dimensions 3–20
  formatting
    dimensions of graphical displays 3–21
    dimensions of terminals 3–20
  resolutions 3–22

Screen buffers 4–11

Security 4–14
  CAN-DO function 4–15
  SETUSERID function 4–15
  USERID function 4–15

Selection list widget 3–18

Selection list, using keyboard 3–18

SESSION system handle
  attributes 3–13, 3–22
  WINDOW-SYSTEM attribute 4–10

SETUSERID function 4–15

SIZE phrase 3–18

Stacking order, widgets 3–14

Startup specifications 3–27

Static colors 3–23

Storing
  data 4–11

triggers 4–11

Streams, two-way 4–17

Syntax notation ix

System clipboard 4–18

System dialog statements 3–25
SYSTEM-DIALOG COLOR statement 3–23, 3–24
SYSTEM-DIALOG FONT statement 3–24
SYSTEM-DIALOG GET-FILE statement 3–24

T

Tables
modifying 2–2
naming conventions 4–2

Terminals
control information 4–3
dimensions 3–20
keybindings 3–37
screen formatting 3–20
spacetaking 3–20
specifying colors 3–33

Third-party software 4–15

Triggers, storing 4–11

Two-way streams 4–17

Typographical conventions viii

U

User IDs 4–14

User interface design 3–1

User interface environments 3–25

USERID function 4–15

V

VIEW-AS statement 4–11

W

Widgets 3–14
attributes 3–14
browse 4–13
buttons 3–16
default size 3–13
display order 3–14
editor 3–18
field-level 3–16
frame 3–19
images 3–17
overlapping 3–14
rectangle 3–17
selection list 3–18
window 3–19

Window
attributes 3–22
widgets 3–19
WINDOW-CLOSE event 3–19
WINDOW-MAXIMIZED event 3–19
WINDOW-MINIMIZED event 3–19
WINDOW-RESTORED event 3–19

Window system, SESSION system handle attributes 4–10

Windows
key bindings 3–35
progress.ini file 3–25
registry 3–25

&WINDOW-SYSTEM, preprocessor constant 4–10