

Progress Debugger 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: 4512
Item Number: 81080;9.1C

Contents

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

1. Introduction **1–1**
1.1 Debugger Features and Functions 1–2
1.1.1 Application Control 1–2
1.1.2 Application Data Management 1–3
1.1.3 Execution Monitoring 1–4

1.1.4 Session Management and Information 1–4
1.2 How a Debugging Session Operates 1–6
1.2.1 Interaction Between Debugger and Client 1–6
1.2.2 Debugger and Application Control 1–7
1.3 Procedures and the Debugger Context 1–8
1.3.1 The Debugger Context 1–8
1.3.2 Debugger Context and the Procedure Call Stack 1–9

2. Quick Tutorial 2–1
2.1 Preparation 2–2
2.2 Sample Session 2–2

3. Starting and Stopping the Debugger 3–1
3.1 Debugging Modes 3–3
3.1.1 Application Mode 3–3
3.1.2 Stand-alone Mode 3–4
3.1.3 Stacked Applications 3–5
3.2 Debugging from the Procedure Editor 3–6
3.3 Debugging from the AppBuilder 3–8
3.4 Debugging from the ADE Desktop 3–9
3.5 Debugging from the Operating System 3–10

4. Debugger Window 4–1
4.1 Window Layout 4–2
4.2 Window Panels 4–4
4.2.1 Listing Panel 4–4
4.2.2 Button Panel 4–6
4.2.3 Command Panel 4–6
4.2.4 Data Panel 4–8
4.3 Menu Bar 4–8
4.3.1 Accelerator Keys and Mnemonics 4–8
4.3.2 File Menu 4–9
4.3.3 Search Menu 4–9
4.3.4 Debug Menu 4–11
4.3.5 Options Menu 4–12
4.3.6 Help Menu 4–13

5. Running a Debugging Session 5–1
5.1 Pattern of Operation 5–2
5.2 Entering Debugger Commands 5–3
5.2.1 Intrinsic Functions and the Command Queue 5–3
5.2.2 Break Interrupt Function 5–3
5.2.3 Command Panel Entry 5–4

5.2.4 Button Entry 5–5
5.2.5 Menu Options 5–5
5.2.6 Command File Entry 5–5
5.3 Exiting a Debugging Session 5–6
5.3.1 Exiting in Application Mode 5–6
5.3.2 Debugger Settings on Exit 5–7
5.4 Managing Breakpoints 5–7
5.4.1 Setting Breakpoints 5–8
5.4.2 Removing Breakpoints 5–11
5.4.3 Working with Inactive Procedures 5–12
5.4.4 Handling Internal Procedures and Procedure Files 5–12
5.5 Controlling Procedure Execution 5–13
5.5.1 Interrupting a Running Procedure 5–13
5.5.2 Executing an Interrupted Procedure 5–14
5.5.3 Running Procedures from the Debugger 5–14
5.5.4 Continuing Procedure Execution 5–16
5.5.5 Executing a Specified Number of Lines 5–16
5.5.6 Terminating an Application from the Debugger 5–22
5.6 Examining and Modifying Procedure Data 5–23
5.6.1 Listing Buffers, Parameters, and Variables 5–24
5.6.2 Displaying Data and Attribute Values 5–24
5.6.3 Modifying Data and Attribute Values 5–24
5.7 Displaying Debugger and Session Information 5–25
5.7.1 Getting On-line Help 5–25
5.7.2 Displaying Debugger Session Settings 5–26
5.7.3 Examining and Navigating in the Debugger Context 5–26
5.7.4 Determining the Current Transaction State 5–28
5.7.5 Displaying Debug Listings 5–28
5.8 Defining and Using Macros and Buttons 5–28
5.8.1 Specifying Macro and Button Definitions 5–28
5.8.2 Using Macros 5–30
5.8.3 Using Buttons 5–31
5.8.4 Removing Macros and Buttons 5–33
5.9 Logging a Debugging Session 5–33
5.9.1 Using the LOG and CANCEL LOG Commands 5–33
5.9.2 Automatic Session Logging 5–33
5.10 Debugging Events 5–34
5.10.1 Nonvolatile Events 5–34
5.10.2 Volatile Events 5–35

6. Debugger Files 6–1
6.1 Debugger Initialization Files 6–2
6.1.1 Windows Initialization 6–2
6.1.2 Motif Initialization 6–5

6.2 Debugger Command Files 6–7
6.3 Debug Listing (.dbg) Files 6–8
6.3.1 Listing Creation 6–8
6.3.2 Listing Deletion 6–8
6.3.3 Listing Format 6–9
6.4 Procedure Files and Listing Generation 6–9
6.4.1 Debug Listing Generation from R-code 6–9
6.4.2 Debug Listing Generation from Source Code 6–10
6.4.3 Hidden Procedure Files 6–10
6.5 Debugger Log Files 6–11
6.5.1 Log File Contents 6–11
6.5.2 Log File As a Command File 6–11

7. DEBUGGER and SESSION System Handles 7–1
7.1 DEBUGGER Handle Attributes and Methods 7–2
7.2 Starting the Debugger from the 4GL 7–3
7.2.1 Starting in Application Mode 7–4
7.2.2 Starting in Stand-alone Mode 7–7
7.3 Using the DEBUGGER Handle During a Debugging Session 7–9
7.4 Using the SESSION Handle with a Debugging Session 7–9

A. Command Reference A–1
ASSIGN Command A–2
BREAK Command A–5
BUTTON Command A–8
CANCEL ALL Command A–10
CANCEL BREAK Command A–11
CANCEL BUTTON Command A–13
CANCEL LOG Command A–14
CANCEL MACRO Command A–15
CONTINUE Command A–16
DISPLAY Command A–17
DOWN Command A–19
EXIT Command A–20
FILE Command A–22
HELP Command A–23
INPUT Command A–24
LIST BUFFERS Command A–25
LIST PARAMETERS Command A–26
LIST VARIABLES Command A–27
LOG Command A–28
MACRO Command A–29
NEXT Command A–32
QUIT Command A–33

RUN Command A–34
SHOW BREAKS Command A–35
SHOW BUTTONS Command A–36
SHOW MACROS Command A–37
SHOW STACK Command A–38
SHOW TRANSACTION Command A–39
STEP Command A–40
STOP Command A–41
UP Command A–42

Index Index–1

Figures

Figure 1–1: Progress Debugger/Client Interaction 1–6
Figure 1–2: Procedures Inside and Outside the Debugger Context 1–10
Figure 3–1: Debugging in Application Mode 3–3
Figure 3–2: Debugging in Stand-alone Mode 3–5
Figure 3–3: Debugging from the Character-mode Editor 3–6
Figure 3–4: Debugging from the Windows Editor 3–7
Figure 3–5: Debugging from the AppBuilder in Windows 3–8
Figure 3–6: Debugging from the ADE Desktop 3–9
Figure 4–1: Progress Debugger Window in Windows 4–2
Figure 4–2: Progress Debugger Window in Motif 4–3
Figure 5–1: Set Breakpoint Dialog Box in Windows 5–10
Figure 5–2: Set Breakpoint Dialog Box in Motif 5–10
Figure 5–3: Run Procedure Dialog Box in Windows 5–15
Figure 5–4: Run Procedure Dialog Box in Motif 5–15
Figure 7–1: Alert box with Stack Trace Help Button 7–10
Figure 7–2: Stack Trace Dialog Box 7–10

Tables

Table 4–1: File Menu 4–9
Table 4–2: Search Menu 4–10
Table 4–3: Debug Menu 4–11
Table 4–4: Options Menu 4–12
Table 4–5: Help Menu in Windows Debugger 4–13
Table 4–6: Help Menu in Motif Debugger 4–13
Table 5–1: Breakpoint Management Commands 5–7
Table 5–2: Execution Commands 5–14
Table 5–3: Data Commands 5–23
Table 5–4: Session Information Commands 5–25
Table 5–5: Debugger Variables 5–30
Table 5–6: Variable Values in Buttons 5–32
Table 7–1: DEBUGGER Handle Attributes 7–2
Table 7–2: DEBUGGER Handle Methods 7–2
Table A–1: Variables in Button Definitions A–9
Table A–2: Variables in Macros as Commands A–29

Preface

Purpose

Many application development environments provide debuggers, which are software tools that help you locate run-time bugs (logic errors) by monitoring and controlling program execution. This manual describes the Progress Debugger and how to use it to find logic errors in Progress 4GL applications. This is both a guide and reference that provides complete information on the Debugger, its commands, and techniques for using it to monitor and control the execution of Progress procedures.

Audience

This manual is for any programmer who develops Progress applications and has a working knowledge of the user interface in which their version of the Debugger runs.

Organization of This Manual

Chapter 1, “Introduction”

Describes the features of the Debugger and how it interacts with a Progress application.

Chapter 2, “Quick Tutorial”

Presents a sample debugging session that you can run to get a hands-on sense of how the Debugger operates.

Chapter 3, “Starting and Stopping the Debugger”

Describes how to start and end a debugging session in all supported operating environments.

Chapter 4, “Debugger Window”

Describes the Debugger user interface-a window where you enter commands and monitor application execution.

Chapter 5, “Running a Debugging Session”

Describes how to organize and perform major debugging tasks and how to use the Debugger commands to accomplish each task.

Chapter 6, “Debugger Files”

Describes the files that the Debugger references, including system files, Progress files, and special files created or referenced only by the Debugger.

Chapter 7, “DEBUGGER and SESSION System Handles”

Describes how to use the DEBUGGER system handle, which is a 4GL system handle that allows you to start and control aspects of Debugger operation from a Progress procedure.

Appendix A, “Command Reference”

Describes the syntax and use of each Debugger command.

How to Use This Manual

For an overview of the basic Debugger features, terms, and principles of operation, see Chapter 1, “Introduction.” If you have never used a debugger before, you might want to take a few minutes to go through the sample debugging session described in Chapter 2, “Quick Tutorial.” For a description of the two modes in which you can run the Debugger, see Chapter 3, “Starting and Stopping the Debugger.”

Once you have begun a debugging session, the Debugger online help available from the Help menu of the Debugger window provides most of the information you need to use the Debugger. For examples and more information on each help topic, see the remaining chapters of this manual.

Typographical Conventions

This manual uses the following typographical conventions:

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

The following typographical conventions are used to represent keystrokes:

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

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

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

  CTRL-X

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

  ESCAPE H
  ESCAPE CURSOR-LEFT

Syntax Notation

The syntax for each component follows a set of conventions:

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

  In this example, ACCUM is a keyword:

  SYNTAX

  ```
  ACCUM aggregate expression
  ```

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

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

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

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

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

  SYNTAX

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

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

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

SYNTAX

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

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

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

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

  SYNTAX

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

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

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

  SYNTAX

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

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

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

  SYNTAX

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

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

SYNTAX

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

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

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

  SYNTAX

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

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

  SYNTAX

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

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

  SYNTAX

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

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

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

SYNTAX

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

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

SYNTAX

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

Example Procedures

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

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

  If the name does not start with a listed prefix, the procedure is not available online.

- If they are not available online, they compile as shown, but might not execute for lack of completeness.

Accessing Files in Procedure Libraries on Windows Platforms

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

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

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

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

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

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

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

```
MKDIR prodoc
```

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

```
MKDIR prodoc\langref
```

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

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

PROLIB extracts all examples into `prodoc\langref`.

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

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

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

Extracting Source Files from Procedure Libraries on UNIX Platforms

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

1 ♦ Run the PROENV utility:

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

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

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

```
mkdir prodoc
```

3 ♦ Create the `proghand` directory under `prodoc`:

```
mkdir prodoc/proghand
```

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

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

PROLIB extracts all examples into `prodoc\langref`.

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

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

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

Progress Messages

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

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

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

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

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

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

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

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

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

1 ♦ Start the Progress Procedure Editor:

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

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

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

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

Other Useful Documentation

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

Getting Started

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

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

Progress Installation and Configuration Guide Version 9 for UNIX

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

Progress Installation and Configuration Guide Version 9 for Windows

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

Progress Version 9 Product Update Bulletin

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

Progress Application Development Environment — Getting Started (Windows only)

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

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

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

Progress Master Glossary for 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 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.

Progress Portability Guide

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

WebSpeed

Getting Started with WebSpeed

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

WebSpeed Installation and Configuration Guide

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

WebSpeed Developer's Guide

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

WebSpeed Version 3 Product Update Bulletin

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

Welcome to WebSpeed (Hard copy only)

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

Reference

Pocket Progress (Hard copy only)

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

Pocket WebSpeed (Hard copy only)

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

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

A Guide to the SQL Standard

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

Understanding the New SQL: A Complete Guide

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

1

Introduction

The Progress Application Debugger is an interactive tool that allows you to control and monitor the execution of compilable Progress procedures. When you run a *debugging session* (that is, control or monitor a procedure with the Debugger), you can find and fix data and logic errors without modifying your procedure code. With the Debugger, you can execute one or more lines of a procedure and its subprocedures, stopping and restarting at specified lines. You can also display data from the procedure you are running, including the names and contents of variables it defines, and you can modify any of this data.

In addition, you can use two debugging modes, Distributed Session Debugging and Remote Session Debugging, to examine code associated with a distributed application that uses the Progress AppServer. For more information, see the *Building Distributed Applications Using the Progress AppServer* manual. For information on debugging WebSpeed applications, see the *WebSpeed Developer's Guide*.

The Windows version of the Debugger runs on 32-bit platforms and can debug either GUI or character-mode applications. The UNIX version of the Debugger can debug character-mode applications. However since the Progress Debugger is a GUI-based tool, it must run in OSF/Motif on UNIX platforms.

This chapter describes the following:

- Debugger features and functions
- How a debugging session operates
- How the Debugger interacts with procedures

1.1 Debugger Features and Functions

The Debugger is a utility that you can run with any Progress application that runs in the Debugger environment. When you invoke the Debugger, it runs in a separate process with its own window. The Debugger window allows you to enter commands to control all of its functions. You can enter these commands by typing them in, choosing menu options, clicking on window buttons, or pressing accelerator keys. The Debugger window also provides access to comprehensive online help information on all Debugger features and functions. (For more information on each Debugger command, see Appendix A, "Command Reference.")

You can also invoke the Debugger from the 4GL using the DEBUGGER handle. With this system handle, you can control many features of Debugger operation from a Progress procedure. This provides a range of capabilities - from helping to debug complex event-driven logic to building your own Progress development tools that include the Debugger as a tool option. (For more information on the DEBUGGER handle, see Chapter 7, "DEBUGGER and SESSION System Handles.")

1.1.1 Application Control

During a debugging session, the Debugger allows you to start and stop procedure execution as you want, and even execute procedures that are not currently part of your application. All procedures that the Debugger executes are added to the procedure call stack. The *procedure call stack* is a last-in-first-out (LIFO) memory structure that keeps track of the execution status of each procedure as it calls a subprocedure or invokes a trigger block.

Open and Incremental Execution

If you want to execute a procedure other than the one the Debugger is currently running, you can start it from the Debugger using the RUN command. Whenever the Debugger has control, you can run any application, effectively adding it to the procedure call stack of your current application. This allows you to mix and match procedures dynamically, or to run diagnostic procedures while debugging your application. (Note that you cannot start a procedure that requires parameters or arguments from the Debugger. The Debugger has no way to define them.)

When the Debugger is stopped in an executing procedure, you can continue execution using the CONTINUE command. This allows the procedure to execute until it terminates or reaches the next stopping point.

You can also continue executing the current procedure incrementally using the NEXT and STEP commands. Each incremental execution runs one or more sequential lines of code and returns to the Debugger for your next command.

Procedure Interruption

To selectively interrupt procedure execution, you can set or remove *breakpoints* using the BREAK and CANCEL BREAK commands. A breakpoint is a specific line in a procedure where you want execution to stop temporarily. When the procedure reaches a breakpoint, it halts before executing the specified line and gives control to the Debugger for your next command.

You can also set and remove breakpoints from the 4GL using the DEBUGGER handle SET-BREAK( ) and CANCEL-BREAK( ) methods. This feature is helpful in a large application when you want to trace complex logic or trigger operation without reference to specific line numbers, or for setting breakpoints in include files that might appear in many different procedures.

To interrupt a running procedure and give immediate control to the Debugger, the Debugger provides a *break interrupt* function. This is a special function that bypasses standard Debugger command input to interrupt the procedure wherever it is executing, as if it encountered a breakpoint. It is particularly useful when you want to interrupt a procedure in which you have not set a breakpoint.

Procedure Termination

You can force an interrupted procedure to terminate using the STOP and QUIT commands. These commands effectively insert and execute a corresponding Progress STOP or QUIT statement at the current stopping point.

1.1.2 Application Data Management

Whenever the Debugger has control, you can monitor and make changes to data items and widget attributes in your application.

Data Analysis

You can list the buffers, parameters, and variables that are defined and accessible from an executing procedure using the LIST command, and examine the values of buffer fields, parameters, variables, and widget attributes using the DISPLAY command.

Data Manipulation

You can modify any data items and widget attributes that you can display using the ASSIGN command.

1.1.3 Execution Monitoring

The Debugger provides several ways to examine the state of an application.

Code Path Analysis

During procedure execution, the Debugger window displays the listing of the procedure at each breakpoint or stopping point in its execution. A pointer always indicates the next line to execute. At any stopping point, you can display the listing for any procedure file using the FILE command. This provides a convenient way to set breakpoints for any procedure in your environment, whether or not it is called from the procedure you are running.

You can examine the procedure call stack using the SHOW STACK command, and you can navigate the procedure call stack using the UP and DOWN commands. This allows you to examine the current code path back and forth through the active series of procedure calls and triggers starting from the current stopping point.

From the 4GL, you can use the DISPLAY-MESSAGE( ) method of the DEBUGGER handle to display message strings to the Debugger window, indicating the status of the procedure as it executes.

Transaction Monitoring

You can determine the transaction state of your application using the SHOW TRANSACTION command. This tells you whether or not a transaction is active at the current stopping point.

Breakpoint Analysis

You can display the current breakpoint settings using the SHOW BREAKS command. This command lists the breakpoints currently set for all procedures during the session. The listing for each executing procedure also indicates each line where a breakpoint is set when displayed in the Debugger window.

1.1.4 Session Management and Information

The Debugger provides a number of features that allow you to monitor and customize a debugging session.

Listing Management

The Debugger window provides a search menu that allows you to search forward and backward for text in the displayed procedure listing. You can also search for a line in the listing specified by line number.

Programmable Command Entry

You can store a series of Debugger commands in a text file and direct the Debugger to take its command input from that file using the INPUT command. The Debugger executes all commands from the specified file until it encounters an error or reaches the end of the file. At this point it resumes accepting input from the Debugger window.

You can also customize your Debugger interface by creating your own macro and button definitions using the MACRO and BUTTON commands. A *macro definition* is a series of commands that you can invoke by entering your own command name, and a *button definition* is a series of commands that you can invoke by choosing an associated button in the Debugger window. You can review all current macro and button definitions using the SHOW MACROS and SHOW BUTTONS commands.

Session Log

You can record the Debugger commands executed and their responses for all or selected portions of a debugging session in log files using the LOG and CANCEL LOG commands.

4GL Debugger Control

The DEBUGGER handle provides additional methods and attributes to initialize, start, and control features of a debugging session from within a Progress procedure. For example, it can clear panels in the Debugger window or display and remove the Debugger window from the screen.

Debugger Initialization

You can set up initial Debugger characteristics in initialization files, including window location and size, panel size, macro definitions, and button definitions. You can also change these characteristics during a debugging session and save them for the next session. For more information on Debugger initialization, see Chapter 6, “Debugger Files.”

1.2 How a Debugging Session Operates

During a debugging session, the Debugger and Progress client processes cooperate to execute, monitor, and interrupt application procedures. The Progress client process executes your application procedures, and the Debugger process manages Debugger command input and window management. However, these two processes share the management of application execution. Figure 1–1 shows this interaction.

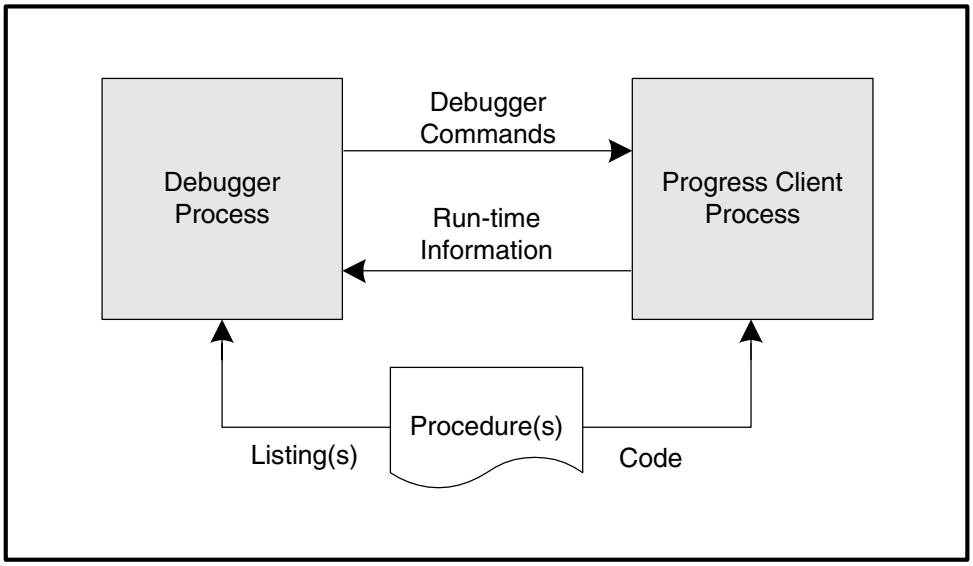


Figure 1–1: Progress Debugger/Client Interaction

1.2.1 Interaction Between Debugger and Client

The Debugger process sends (executes) the commands you enter to the Progress client, for example, setting breakpoints or requesting the contents of the procedure call stack. The client interprets the commands, performs relevant actions (such as setting breakpoints), and sends run-time information to the Debugger process in response. For example, the client sends the procedure and line number of each breakpoint that it encounters or the contents of the procedure call stack for each SHOW STACK command that it receives. The Debugger process uses the run-time information that it receives from the client to manage and display procedure listings.

1.2.2 Debugger and Application Control

In general, either the Debugger or the executing application has control during a debugging session. The Debugger has control when no application is running or the running application is interrupted, for example, at a breakpoint. When the Debugger has control, it can send commands to the client process for execution. The application has control whenever it is being executed by the client, for example, in response to a Debugger control flow command. When the application has control, it interacts with the user and database as if the Debugger were not running.

Application Interaction During Debugger Control

When the Debugger has control, the application is blocked. At this point, the client application is waiting to execute a line of code. The application continues to wait until you give the Debugger a control flow command such as STEP or CONTINUE.

While the Debugger has control, any input you try to enter in the application window is handled differently depending on your window system. In Windows, application input is ignored while the Debugger has control. In OSF/Motif, application input is queued in the system until the Debugger returns control to the application.

Also, unlike Windows, the Motif environment defers application window refresh to the application itself. For example, in Motif, if the application window is covered by another window and you raise it to the top while the Debugger has control, the covered portions remain blank. The application does not refresh its window until you transfer control back to the application from the Debugger with a control flow command.

Debugger Interaction During Application Control

If the application has control (is executing) and you try to execute commands in the Debugger window, the Debugger commands queue up until the Debugger receives control (for example, when the application reaches a breakpoint). You can force the Debugger to receive control during procedure execution by invoking the break interrupt function from the Debugger window, but this also purges all Debugger commands from the command queue without executing the commands.

1.3 Procedures and the Debugger Context

This section introduces several terms that are used throughout this manual to describe the relationship between the Debugger and application procedures that it controls or references.

1.3.1 The Debugger Context

The *Debugger context* consists of the data and execution status of all procedures running under Debugger control.

Procedures and Triggers

Procedures and trigger blocks in the Debugger context include the executing procedure or trigger plus some or all of the procedures and triggers pushed onto the procedure call stack. The number of procedures and triggers on the call stack grows as each subprocedure or trigger calls another and decreases as each subprocedure or trigger returns to the one that called it. The Debugger provides information on a running application by referencing this call stack.

NOTE: In Progress, procedures and trigger blocks are identical with the exception that triggers cannot not accept parameters or arguments.

The executing procedure and all procedures on the call stack (whether inside or outside the Debugger context) are *active procedures*. Active procedures include all those that you run directly or indirectly from the Progress client and any additional procedures that you start from the Debugger with the Debugger RUN command.

When an application has control, all active procedures **except** the currently executing procedure are on the call stack. When the Debugger has control, all active procedures **including** the currently executing procedure are on the call stack. As a result, the executing procedure waits to continue execution.

Persistent Procedures

A persistent procedure is active in the Debugger context in the following instances:

- When the persistent procedure is first executed to instantiate it (RUN ... PERSISTENT).
- When one of its triggers is executed in response to a user or program event.
- When one of its internal procedures is executed from another procedure (RUN ... IN).

Otherwise, the persistent procedure is not on the call stack and is invisible to the Debugger context.

Data

Data in the Debugger context includes all variables, fields, buffers, parameters, and attributes of widgets defined or accessed by procedures inside the Debugger context.

Execution Status

The execution status includes the location of the current *executable line* and the relative position in the call stack of each procedure in the Debugger context plus the current transaction state. An executable line is any 4GL line that contains a run-time statement (for example, FOR, DISPLAY, or CREATE). Note that although the listing monitored by the Debugger contains all the 4GL code for a procedure, the execution status only reflects executable lines; execution stops only on executable lines. *Nonexecutable lines* include comments and static data definition statements (for example, DEFINE or FORM). Static data definition statements take effect only at compile time and generate data items and widgets for a procedure.

1.3.2 Debugger Context and the Procedure Call Stack

Figure 1–2 shows the relationship between active procedures inside and outside the Debugger context in a typical procedure call stack (growing downward in the figure). The Debugger is controlling an application (`your-app.p`) that is running in the Procedure Editor (also a Progress application). In this figure, the executing procedure is `subexec.p`. The procedure call stack includes all procedures down to the executing procedure, including the procedure started in the Progress client. All procedures running from your application are in the Debugger context. Only procedures associated with the Procedure Editor (`edit.p` and `_proedit.p`) are running outside of the Debugger context.

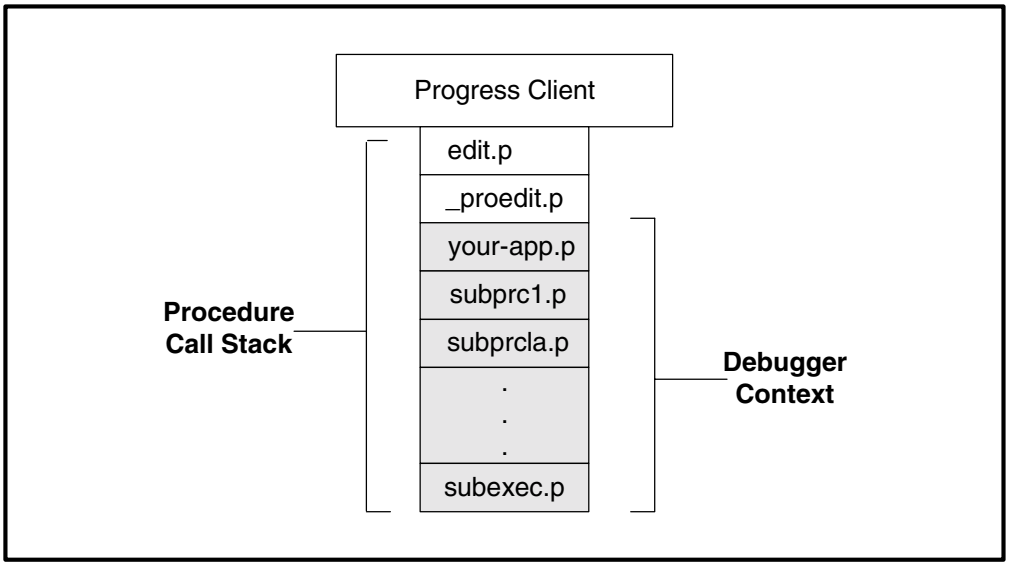


Figure 1–2: Procedures Inside and Outside the Debugger Context

An active procedure (and all its ancestor procedures) can run outside the Debugger context in two ways:

- When the name of the procedure is preceded by an underscore (_), as with `_proedit.p` in Figure 1–2.
- When it invokes the Debugger using the DEBUG method of the DEBUGGER handle.

For more information on using underscores in procedure names, see Chapter 6, "Debugger Files." For more information on the DEBUGGER handle, see Chapter 7, "DEBUGGER and SESSION System Handles."

The Debugger keeps track of run-time information only for active procedures inside the Debugger context. Whether an active procedure is inside or outside the Debugger context depends on how the Debugger and procedure are started. For more information on starting the Debugger and applications under its control, see Chapter 3, "Starting and Stopping the Debugger."

Procedures Inside the Debugger Context

The Debugger allows you to examine the data items, widget attributes, listing, and execution status of any procedure that is active in the Debugger context by bringing it into focus. A procedure can receive Debugger focus by triggering a breakpoint in it or by navigating the call stack (with the UP and DOWN commands). The procedure that has the Debugger focus is the *current procedure*. The default current procedure is the executing procedure on the call stack.

Procedures Outside the Debugger Context

The Debugger allows you to examine the listing and set breakpoints for any procedure that is outside the Debugger context. Procedures outside the Debugger context can include both active procedures and *inactive procedures*.

Active procedures outside the Debugger context include:

- Procedures that are **above** the Debugger context on the procedure call stack
- Instances of persistent procedures that have no code executing

These active procedures are invisible to Debugger commands that examine or control procedure execution in the Debugger context.

Inactive procedures are procedures that are not executing in any context. They include:

- Procedures that have yet to be called
- Procedures that have completed execution and returned to their caller
- Instances of persistent procedures that have been deleted from the session

2

Quick Tutorial

To provide a sense of how to run a debugging session, this chapter presents a short tutorial using a modified sample procedure from the *Progress Programming Handbook*. To use this tutorial, you must have the Progress Application Debugger installed in your Progress environment. The figures in the tutorial include examples from all supported user interfaces, including character mode where applicable. Most Debugger window examples are shown in Windows, except where they explicitly illustrate a comparison with the corresponding Debugger window in OSF/Motif.

NOTE: Since the Progress Debugger is a GUI-based tool, it must run in OSF/ Motif on UNIX platforms.

This chapter describes:

- Preparations for running the tutorial procedure
- A sample debugging session

2.1 Preparation

To run this tutorial, you must do the following:

- Use the PRODB utility to make a copy of the Sports database in your working directory. This new database is used to run the debugging session described in this tutorial. To run PRODB to create the sportsdebug database from the Sports database, use the following command at the DOS prompt:

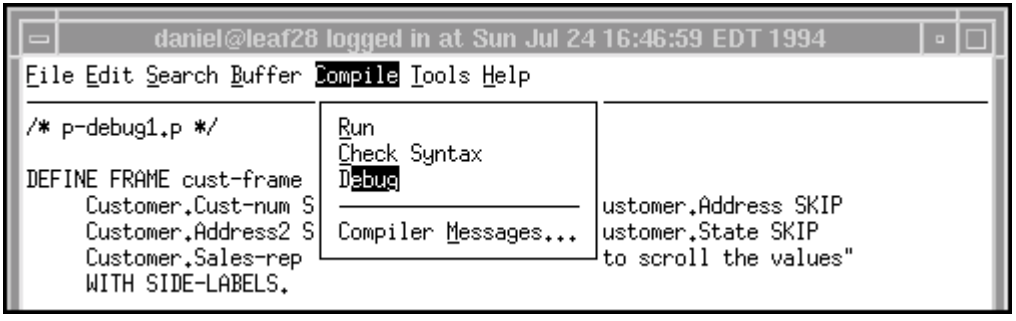
  prodb sportsdebug sports

- Use the PROLIB utility to copy the `p-debug1.p` procedure into your working directory. For instructions on running PROLIB to copy the `p-debug1.p` procedure, see "Example Procedures" in the"Preface".

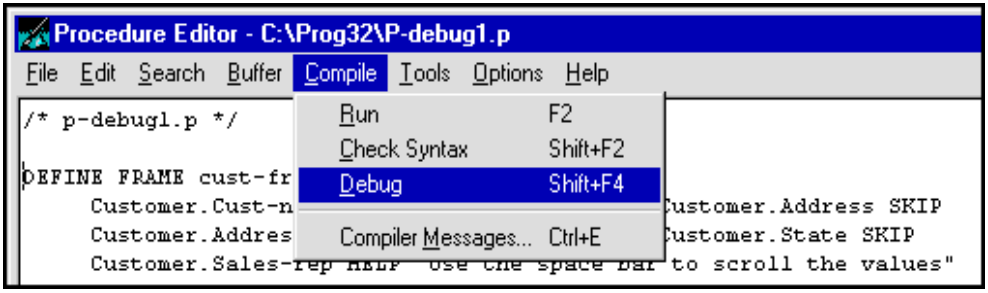
2.2 Sample Session

To begin the session, run the Procedure Editor with the sports database and open `p-debug1.p` in the buffer. Then, follow these steps:

1 ♦ To start the Debugger for `p-debug1.p`, choose the Debug option of the Compile menu. The following figure shows starting the Debugger in character mode:

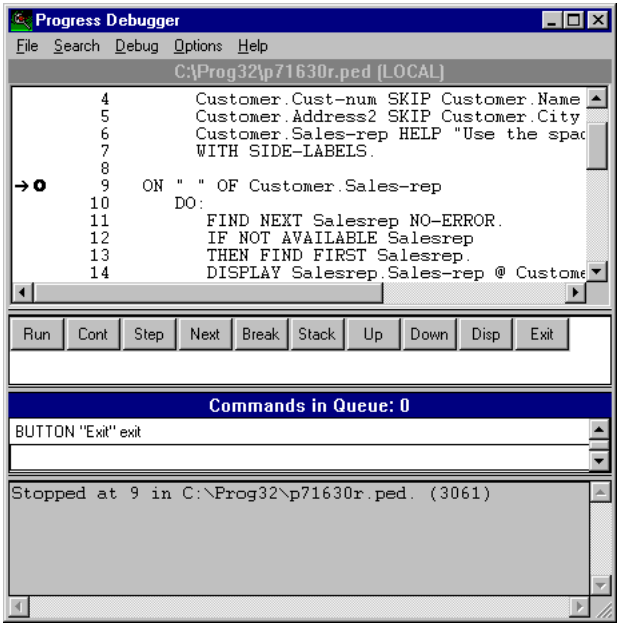


The following figure shows starting the Debugger in Windows:

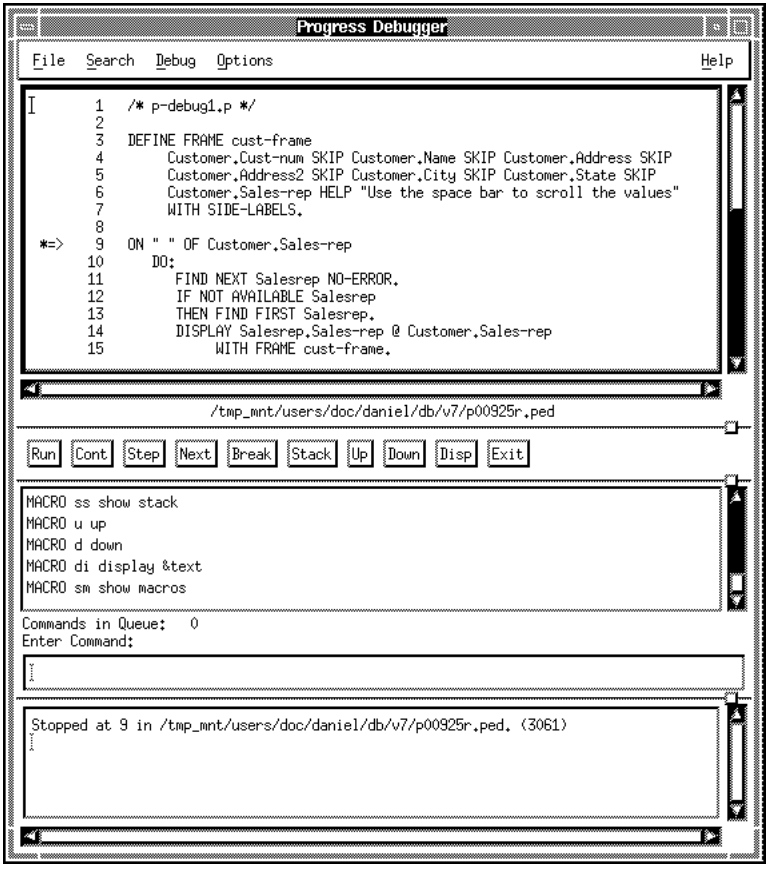


The Debugger window opens as the procedure begins and stops at an automatic breakpoint set on the first executable line.

In Windows, a small octagon indicates a breakpoint and a solid arrow indicates the current executable line on the left side of the procedure listing:



In Motif, an asterisk indicates a breakpoint and an open arrow indicates the current executable line on the left side of the procedure listing:



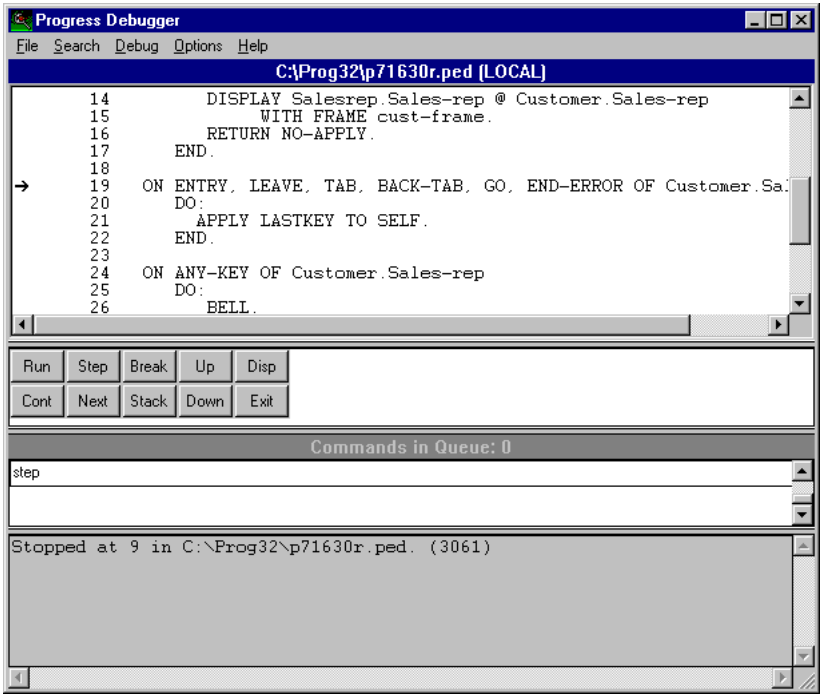
Note how the default buttons appear as the Commands in Queue field decrements; the Debugger executes initial commands to define the buttons.

The panel above the buttons displays the debug listing of the procedure labeled with a temporary buffer filename. The panel below the buttons displays a command entry area and a list of all entered commands (command history). The Windows command entry area is combined with the command history. The bottom panel displays Debugger data and informational messages.

NOTE: You might want to size your Debugger window to match the tutorial. You can adjust the vertical size of each panel by dragging the horizontal dividers between the panels in Windows or the tiny boxes between the vertical scroll bars in Motif. Be sure to leave room for both the command history and the command entry area.

2 ♦ To continue, click the mouse pointer on the Step button.

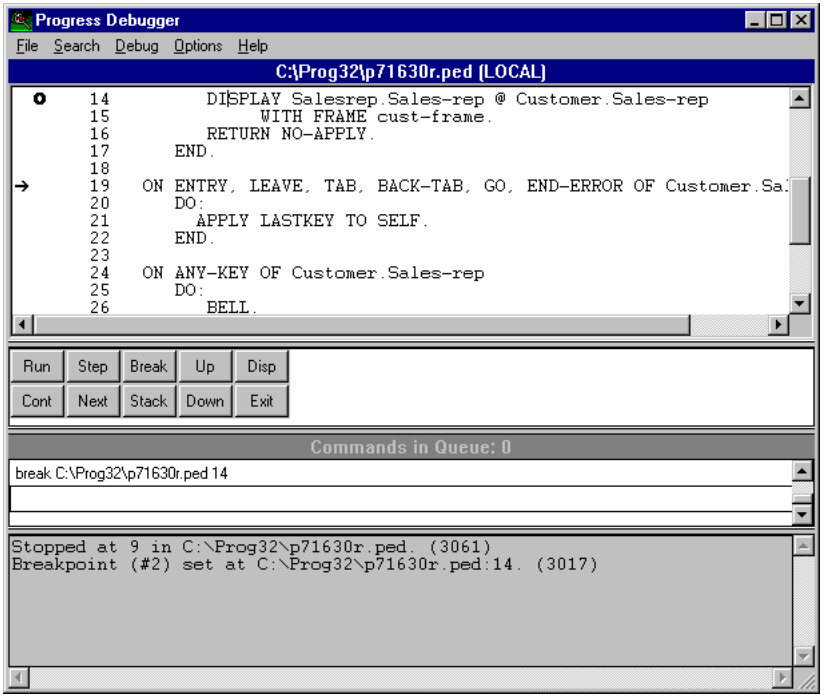
The line pointer advances to the next executable line (line 19) after executing the ON statement that defines a trigger for the space bar (“ ”) specified on lines 9 through 17 as shown in the following figure:



Note the STEP command that scrolled into the command history. You can enter Debugger commands using uppercase or lowercase characters.

3 ♦ Click the mouse anywhere on line 14 and click on the Break button.

This sets a breakpoint on line 14. Note the BREAK command for the current procedure that scrolled into the command history. The Break button definition references the current listing to specify the breakpoint, as shown in the following figure:



4 ♦ Use the mouse to scroll to the bottom of the procedure listing and perform similar actions to set a breakpoint on line 33.

5 ♦ To continue, click on the Cont button.

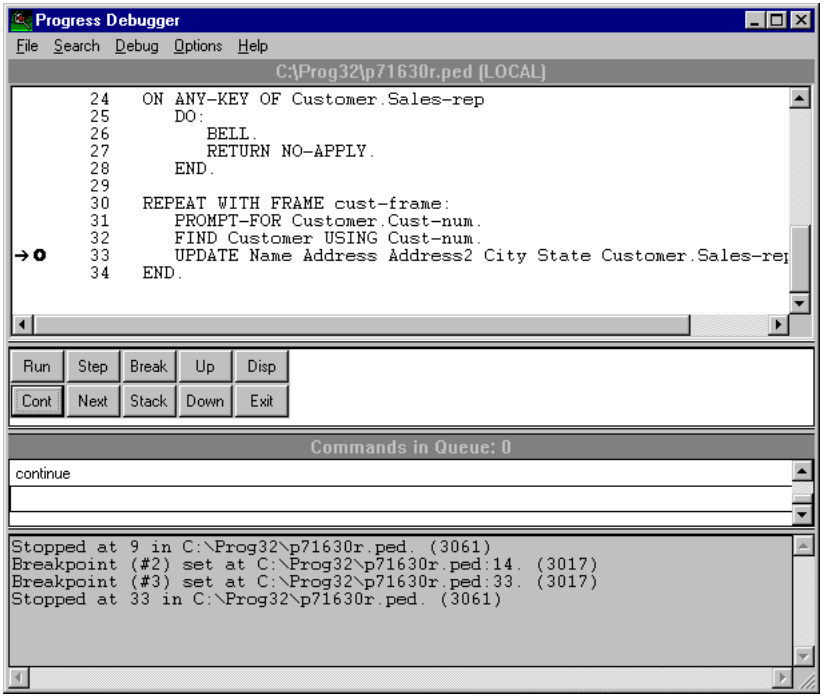
The application window opens displaying a frame that prompts you for a customer number in the Cust-Num field.

6 ♦ Type **10** in this field, as shown below:

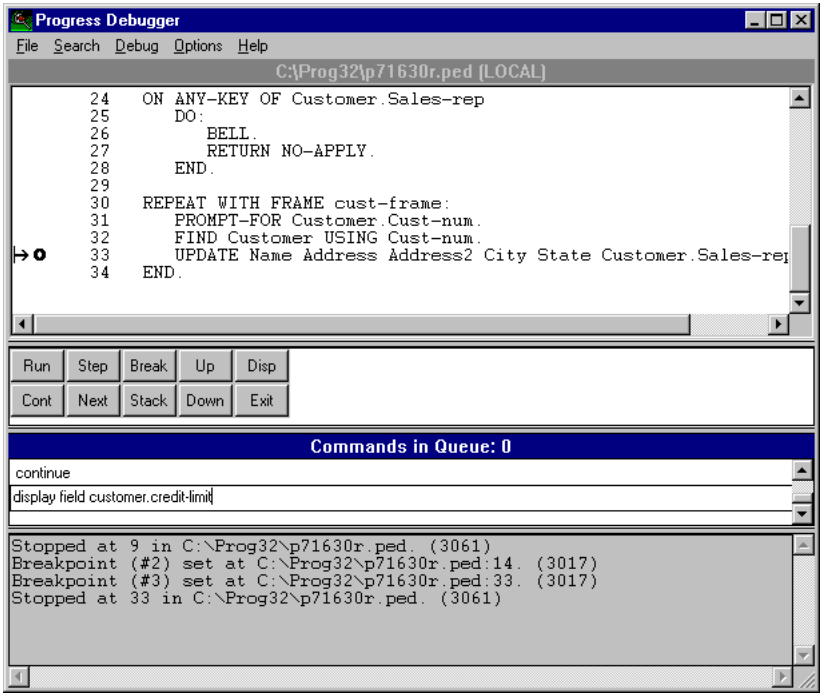


7 ♦ To continue, press Cont.

The Debugger takes control after the procedure retrieves the record for customer number 10 and stops at the breakpoint on line 33, as shown in the figure that follows:



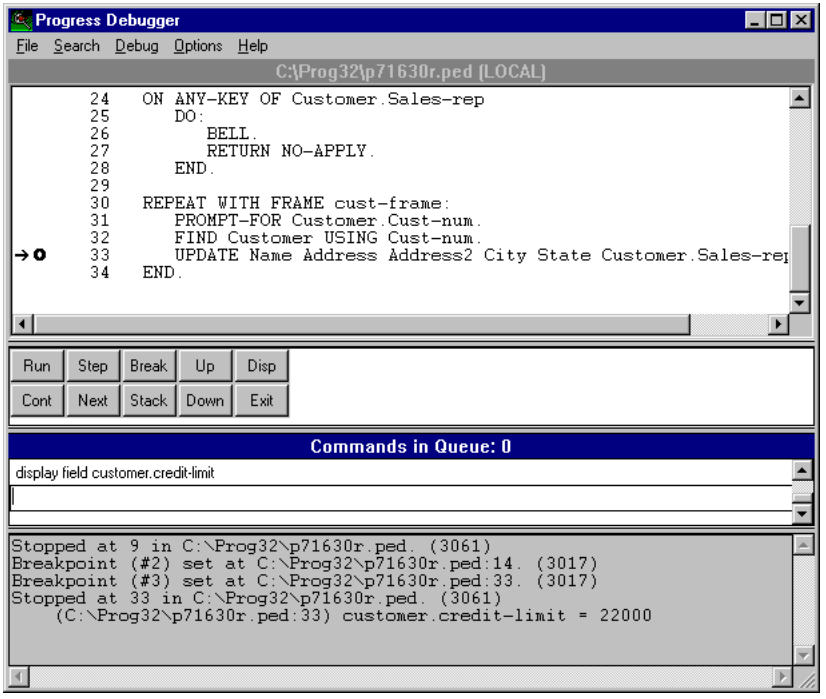
8 ♦ Type **display field customer.credit-limit** in the command entry area:



Note that the Debugger can reference any field in an accessible record buffer, such as credit-limit, even though it is not referenced in the procedure.

9 ♦ To enter the command, press **RETURN**.

The Debugger displays the customer credit limit, 22000, in the bottom panel:

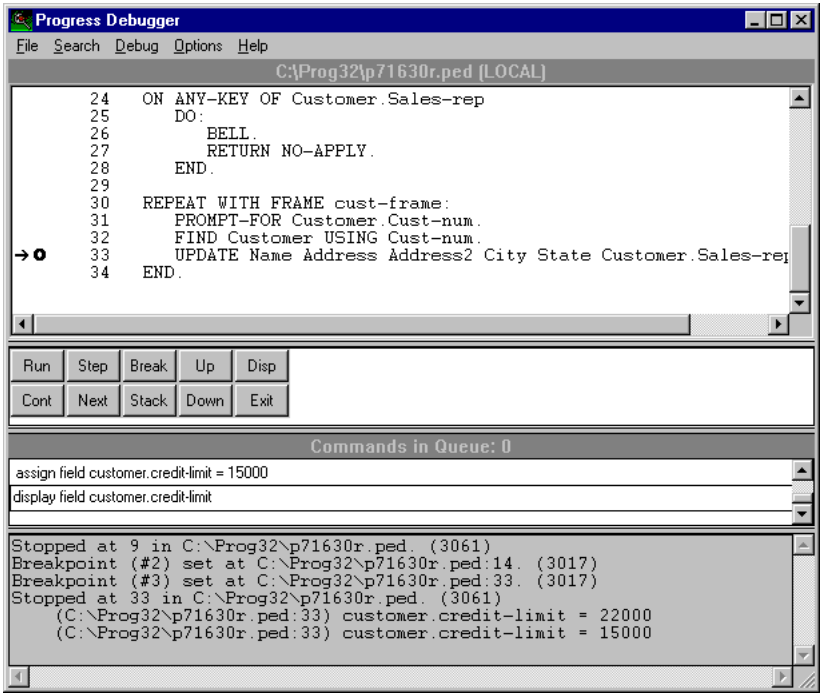


10 ♦ Type **assign field customer.credit-limit = 15000** in the command entry area and press **RETURN**.

The Debugger assigns the new credit limit value.

11 ♦ You can use the command history to display the new value. Click the mouse pointer on **display field customer.credit-limit** in the command history and press **RETURN**.

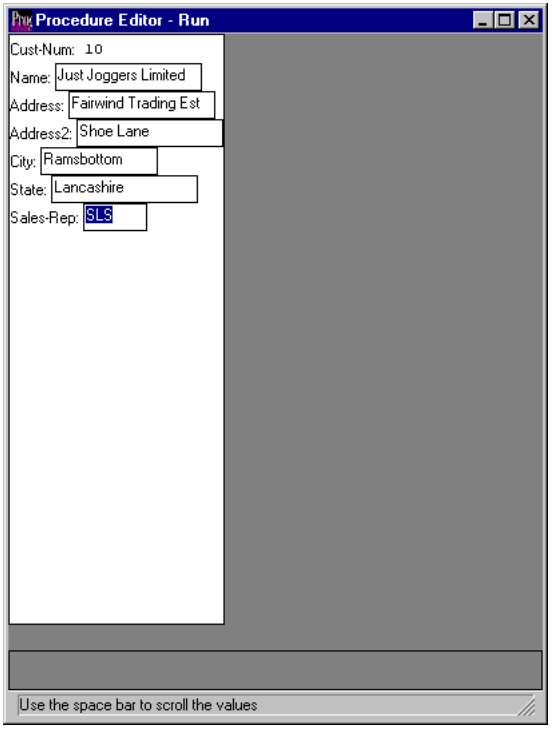
The Debugger displays the new credit limit value, as follows:



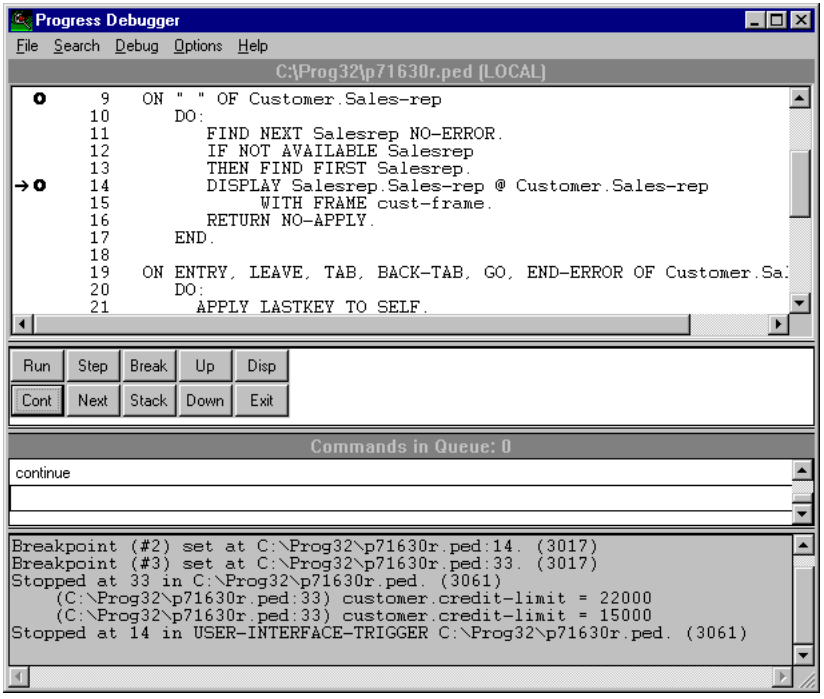
12 ♦ To continue, click on the Cont button.

The procedure blocks on the UPDATE statement, displaying the specified fields for customer number 10.

13 ♦ Tab to the Sales-Rep field and press the **SPACE BAR**, as shown below:

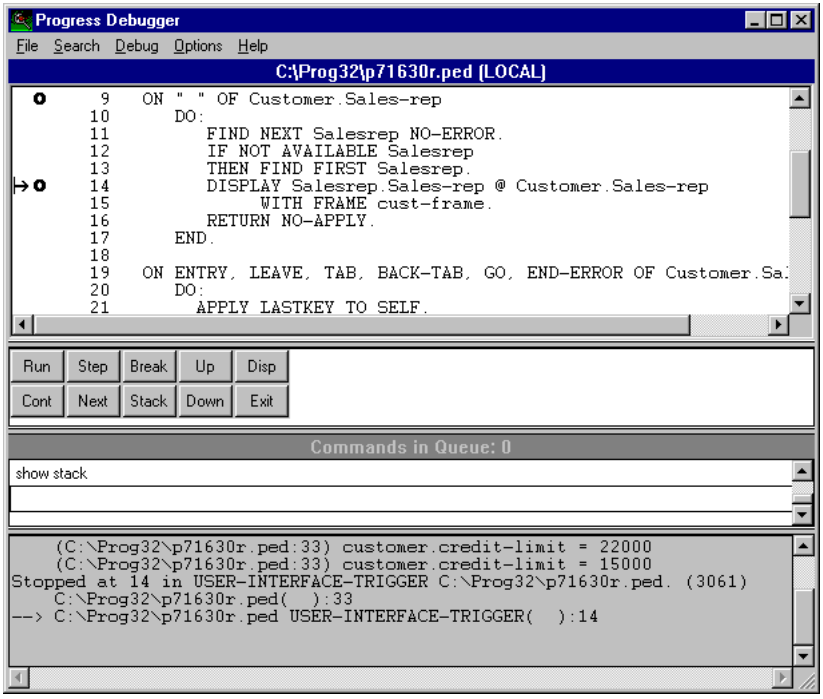


This activates the space bar trigger and the Debugger takes control as the procedure stops at the breakpoint on line 14 in the trigger block. Note that the breakpoint for the trigger does **not** occur on line 9 (the ON statement). The ON statement defines the trigger block (DO statement) on the first executable line where the initial breakpoint is set. The trigger block itself executes in response to the space bar event and stops at the current breakpoint, as shown below:



14 ♦ To see the execution relationship between the procedure and the trigger block, click on the Stack button.

The Stack button enters the SHOW STACK command to display the procedure call stack (see the figure below). The result, in the bottom panel, shows that the trigger is executed as an internal procedure (USER-INTERFACE-TRIGGER( )), stopped at line 14, and that the trigger is invoked by the space bar event while blocked for I/O on line 33 (the UPDATE statement) of the main procedure:



15 ♦ After completing this tutorial and observing the Debugger, you can explore further or type **quit** in the command entry area to exit both the application and the Debugger.

16 ♦ Press the **SPACE BAR** in the application window to return to the Procedure Editor.

3

Starting and Stopping the Debugger

Progress provides several different ways to start a debugging session, depending on where you are working in Progress and how you want the Debugger to interact with your application. You can start a debugging session in the following ways:

- In the Procedure Editor or AppBuilder running an initial application from an Editor or AppBuilder buffer
- From the operating system or the Application Development Environment (ADE) Desktop with no initial application running
- Directly from the 4GL using the DEBUGGER system handle

  NOTE: Using the DEBUGGER handle to start a debugging session is primarily intended for experienced Progress programmers who are writing their own Progress development tools. For more information, see Chapter 7, "DEBUGGER and SESSION System Handles."

When you start a debugging session, the Debugger initializes and runs in either application or stand-alone mode. Debugger initialization creates the Debugger window, reads and processes initialization files, enables procedure and breakpoint monitoring, and allows the DEBUGGER handle to interact with the Debugger from the 4GL. When and how the Debugger gets control depends on the mode used to run it. For more information about these modes see "Debugging Modes" in this chapter.

This chapter describes:

- The debugging modes
- The techniques for beginning and ending a debugging session in each mode

3.1 Debugging Modes

The Debugger can run in one of two modes: application mode or stand-alone mode.

3.1.1 Application Mode

The Debugger runs in *application mode* when you start an initial application using the Debugger options of the Procedure Editor and the AppBuilder. This application runs in the Debugger context, and the Debugger takes control when the application stops at an automatic breakpoint set at the first executable line. This gives you a chance to examine your application listings and set breakpoints before continuing execution. You can also run any other procedures from the Debugger using the RUN command, effectively running them under your initial application. Figure 3–1 shows the relationship between the Debugger in application mode, its initial application, other procedures it might run, and the Progress environment in which it is running.

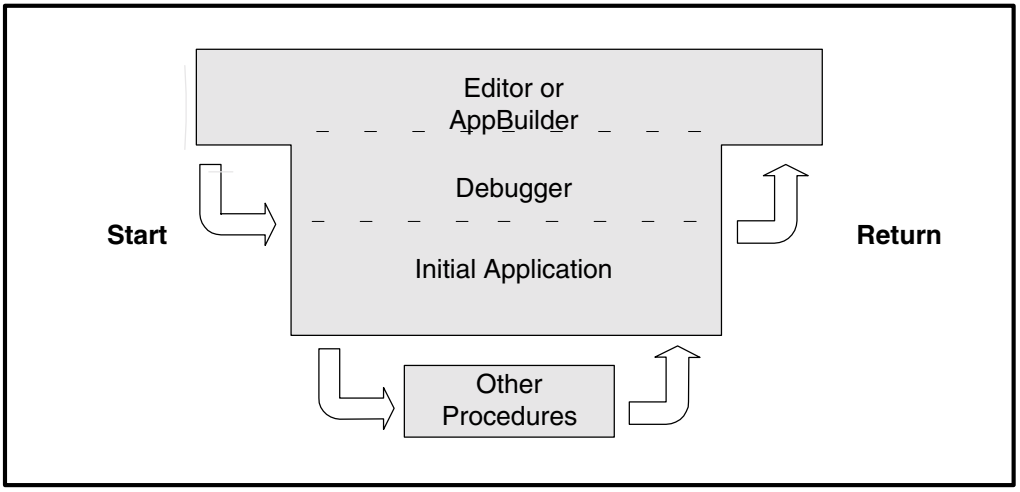


Figure 3–1: Debugging in Application Mode

In application mode, you can start and restart the Debugger and its initial application without leaving your Progress environment, but how the Debugger restarts depends on how you exit.

Exiting Procedures Started with the RUN Command

When you exit a procedure started with the RUN command, it returns control to the Debugger. Any previous application on the stack (including the initial application) waits to resume execution at its current stopping point.

Exiting the Debugger Before the Initial Application

When you exit the Debugger in application mode (using the EXIT command), the Debugger purges itself from the Progress environment and closes any applications started with the RUN command. It then returns control to the initial application, which continues execution from its current interruption point. If you restart (*cold-start*) the Debugger in application mode from the same startup environment, the Debugger completely re-initializes itself.

Exiting the Initial Application Before the Debugger

When you exit the initial application, it returns control to the environment where you started it. If the Debugger is running when you exit, the Debugger clears and closes its window, removes all breakpoints, and closes any open log file, but leaves Debugger session settings such as window size, macro, and button definitions initialized in the Progress environment. If you then run the same or another initial application using the Editor or AppBuilder Debugger option, the Debugger restarts (*warm-starts*) and takes control with a breakpoint at the first executable line. The Debugger window opens with its previous window dimension and button settings, but it does not re-initialize itself (that is, does not reread its initialization files). If you want to run any other Debugger commands or set any breakpoints expected during initialization, you must execute them again in the Debugger after it restarts. (You can also Use command files. For more information, see Chapter 5, "Running a Debugging Session.")

NOTE: You can also run the Debugger in application mode from your own application using the DEBUGGER system handle. This approach offers more flexibility between the Debugger and your application. For more information, see Chapter 7, "DEBUGGER and SESSION System Handles."

3.1.2 Stand-alone Mode

The Debugger runs in *stand-alone mode* when you start it from the operating system or the ADE Desktop. In stand-alone mode, the Debugger takes control immediately as your initial Progress interface and with no initial application running in the Debugger context. In this mode, you must start all application procedures from the Debugger using the RUN command.

Figure 3–2 shows the relationship between the Debugger in stand-alone mode, the application procedures it runs, and the Progress environment in which it is running.

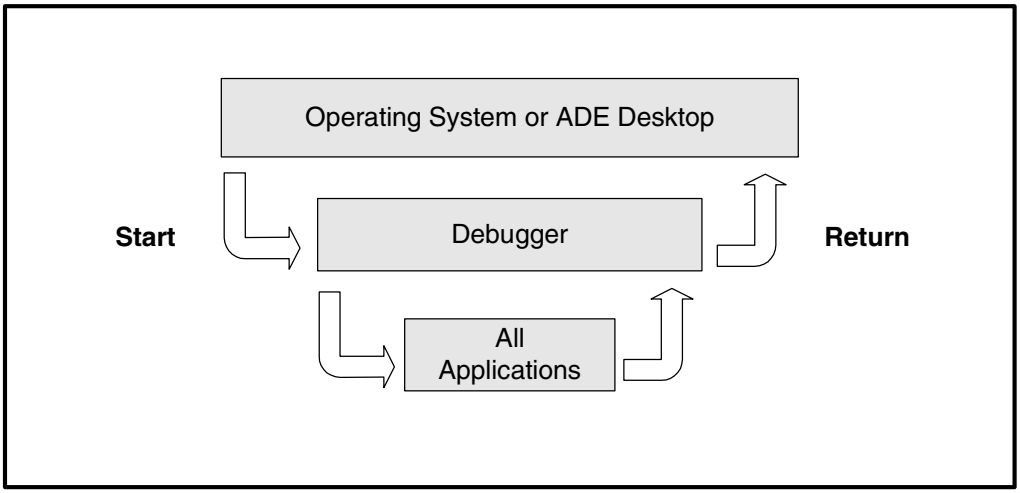


Figure 3–2: Debugging in Stand-alone Mode

When you exit any of these applications, it returns control to the Debugger. When you exit the Debugger, it completely removes itself and any application it is running from the Progress environment, and returns control to the operating system or ADE Desktop. If you restart the Debugger, for example, from the ADE Desktop, it completely re-initializes itself without any application in the Debugger context.

NOTE: You can also run the Debugger in stand-alone mode from your own application using the DEBUGGER system handle. For more information, see Chapter 7, "DEBUGGER and SESSION System Handles."

3.1.3 Stacked Applications

In both application and stand-alone mode, you can stack applications. That is, you can run a new application at any interruption point of an executing application. When you do so, the previous application waits on the procedure call stack. The new application can access any global data defined by previous applications stacked in the Debugger context. When the new application exits, it returns control to the Debugger, and the previous application is ready to continue at its current interruption point. Whenever the Debugger exits, it closes all applications stacked in the Debugger context before it returns to its startup environment.

3.2 Debugging from the Procedure Editor

To start a debugging session from the Procedure Editor, enter a procedure in the Editor buffer and choose the Compile→ Debug option from the menu bar. Figure 3–3 shows the character-mode Editor and Figure 3–4 shows the Windows Editor.

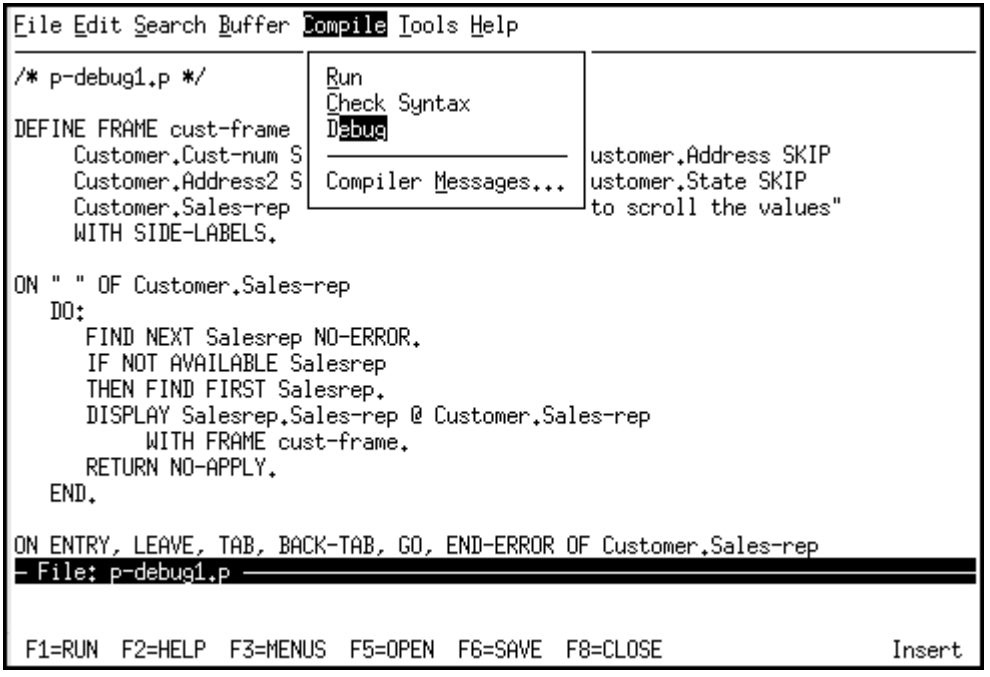


Figure 3–3: Debugging from the Character-mode Editor

NOTE: If you want to debug a 32–bit character client, you would start your debugging session from the Procedure Editor.

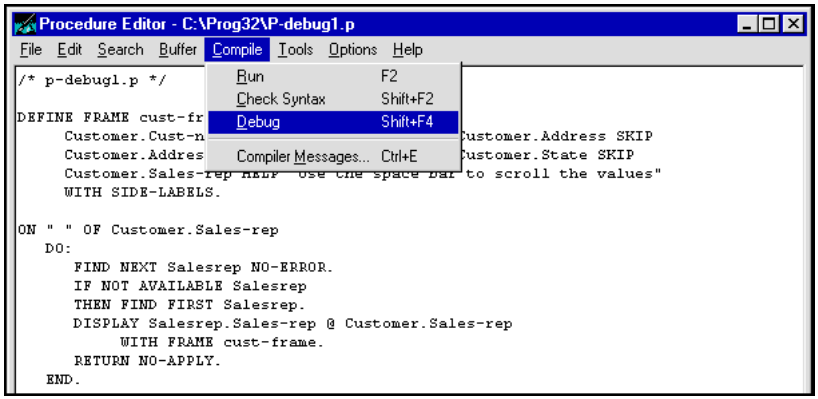


Figure 3–4: Debugging from the Windows Editor

After choosing the Debug option, the Procedure Editor:

1. Saves a file containing your procedure using a random alphanumeric filename with a `.ped` extension.
2. Initializes the Debugger.
3. Sets a breakpoint on the first executable line of your procedure.
4. Runs the procedure.

The procedure immediately gives control to the Debugger at the initial breakpoint. Starting the Debugger from the Editor with the Compile→ Debug option runs the Debugger in application mode, ready to debug the procedure that is running (and any other procedures you choose).

NOTE: You can also start the Debugger from the Editor in stand-alone mode by invoking it with the DEBUGGER system handle. For more information, seeChapter 7, “DEBUGGER and SESSION System Handles.”

3.3 Debugging from the AppBuilder

To start a debugging session from the AppBuilder, enter a window or dialog box application and choose the Compile→ Debug option from the menu bar, as shown in Figure 3–5.

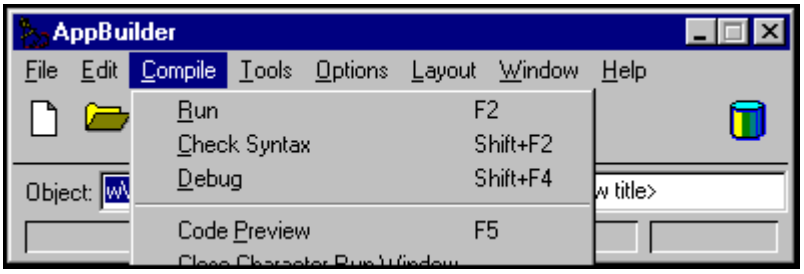


Figure 3–5: Debugging from the AppBuilder in Windows

After choosing the Debug option, the AppBuilder:

1. Saves a file containing your procedure using a random alphanumeric filename with a `.ab` extension.
2. Initializes the Debugger.
3. Sets a breakpoint on the first executable line of your procedure.
4. Runs the procedure.

The procedure immediately gives control to the Debugger at the initial breakpoint. Starting the Debugger from the AppBuilder with the Compile→ Debug option runs the Debugger in application mode, ready to debug the procedure that is running (and any other procedures you choose).

Note that when you debug a procedure from the AppBuilder, the AppBuilder main window iconifies and then restores itself automatically when the Debugger returns.

3.4 Debugging from the ADE Desktop

To start a debugging session from the ADE Desktop, choose the Tools→ Application Debugger option from the menu bar or click on the Application Debugger button. Figure 3–6 shows the Application Debugger button choice in Windows.

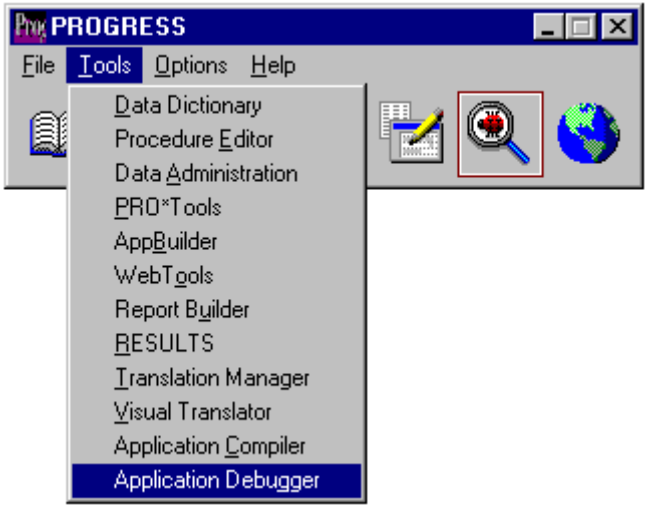


Figure 3–6: Debugging from the ADE Desktop

The Desktop immediately initializes and gives control to the Debugger without an initial application running. Starting from the Desktop runs the Debugger in stand-alone mode, ready to execute and debug any procedures you choose. Note that the Debugger does not automatically set a breakpoint on the first executable line of procedures you execute from the Debugger when you start it from the Desktop. Be sure to set at least one breakpoint before running a procedure.

NOTE: You can use the break interrupt function from the Debugger window to stop execution in any procedure, whether or not it has a breakpoint set. For more information, see Chapter 4, "Debugger Window."

3.5 Debugging from the Operating System

To start a debugging session from the operating system, run a single- or multi-user Progress client session with the Debugger (-debug) startup parameter, as in the following example:

```
pro sports -debug
```

The Debugger startup parameter causes the Progress session to begin by running the Debugger in stand-alone mode. This makes the Debugger the point of entry for all application procedures you want to run. The Debugger begins by allowing you to set breakpoints before using the Debugger RUN command to execute the startup procedure (the default, `prostart.p`, or your own startup procedure. Also, Progress automatically sets a breakpoint on the first executable line of the startup procedure before you run it.

For example, to run a single-user debugging session from the operating system and start debugging the `arrmenu.p` procedure, you run this command:

```
pro sports -p arrmenu.p -debug
```

This command runs the Debugger in stand-alone mode, ready to execute and debug `arrmenu.p` (or any other procedures you choose). An automatic breakpoint is set on the first executable line of `arrmenu.p`.

4

Debugger Window

The Debugger window is your primary means of interaction with the Debugger. It operates independently of any window used by your application. From the Debugger window you can control most features of the Debugger, from setting breakpoints to listing and executing procedures under Debugger control.

The Debugger window provides several components, including panels, menus, and buttons that you can use to invoke Debugger commands and view the results. You can configure these components in various ways, such as the sizes of the panels and the identities and functions of the buttons, and save the configuration for future debugging sessions.

The functional layout of the Debugger window is generally the same in all supported window systems. However, there are some minor variations. Any differences are described in this chapter.

This chapter describes the parts of the Debugger window:

- Layout and components
- Panels and their functions
- Menus and their functions

4.1 Window Layout

After starting a debugging session, the Debugger displays a window similar to those shown in Figure 4–1 for Windows and Figure 4–2 for OSF/Motif.

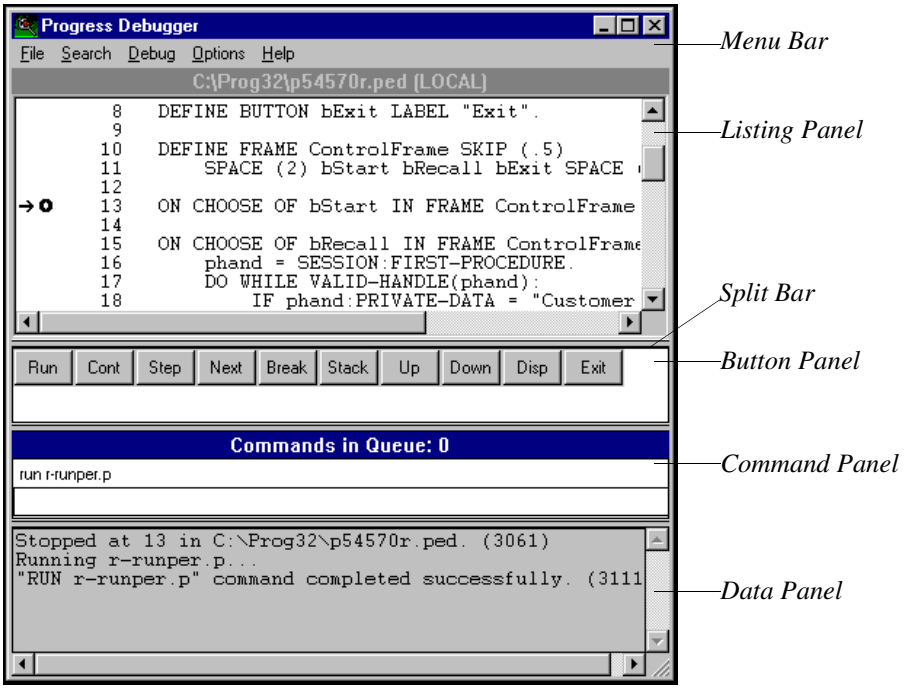


Figure 4–1: Progress Debugger Window in Windows

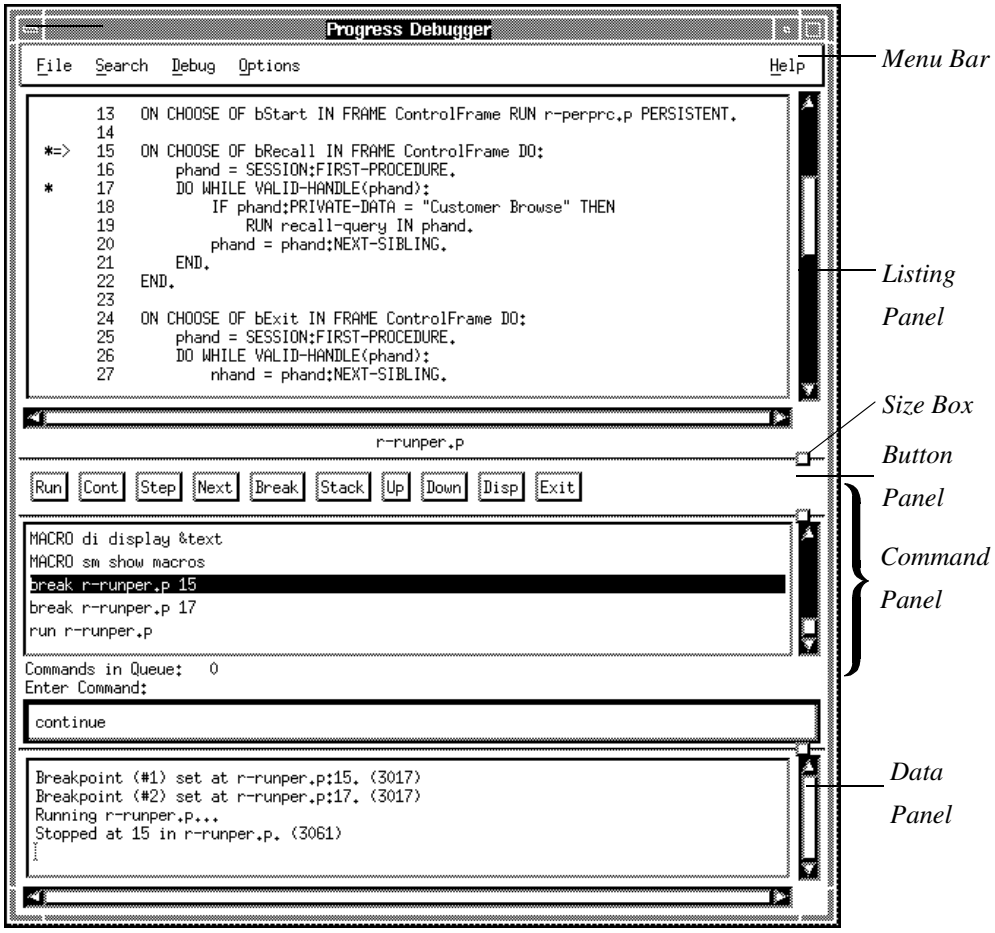


Figure 4–2: Progress Debugger Window in Motif

The exact dimensions and layout of your Debugger window might differ depending on the size and resolution of your monitor and the initial parameters used to start your debugging session. However, the general layout of the window is fixed for each window system, as shown.

The Debugger window contains several panels and a menu bar. You can resize the Debugger window like any other window in your system. You can also resize each panel vertically in relation to the one next to it. The window divides each panel from the next with a horizontal divider. By grabbing a split bar (in Windows) or size box (in Motif) with the mouse and dragging it up or down, you can lengthen or shorten adjacent panels within the window. Each panel also provides scroll bars, as required, to allow you to access portions of the panel that might be vertically or horizontally out of view. In Windows, however, you must use the left and right arrow keys to reach parts of command lines that lie beyond the command panel borders.

4.2 Window Panels

The window panels provide the following functions:

- **Listing panel** - Displays the listing for the current procedure, or any other procedure you choose, and provides default arguments for command buttons and menu options
- **Button panel** - Displays the buttons defined for your session
- **Command panel** - Accepts command input and displays a working history of all commands you enter in a session
- **Data panel** - Displays Debugger messages and all responses to Debugger commands

4.2.1 Listing Panel

The *listing panel* displays the debug listing for both the current procedure and any other procedure whose listing you want to see. The *debug listing* is a specially prepared source listing that allows you to more easily set breakpoints and follow the course of procedure execution in an application. This listing also provides a point of reference for specifying arguments to Debugger commands.

Debug Listing

The debug listing is an expanded listing file that Progress creates during procedure compilation. This file contains a line-numbered listing of the procedure with the text of any embedded preprocessor include files, names, and parameters inserted. For more information on debug listing files and how they are generated, see Chapter 6, "Debugger Files."

Listing Display

The Debugger displays the listing for the executing procedure each time it receives control (for example, at a breakpoint). During Debugger control, you can also display the debug listing for any other active or inactive procedure in the listing panel using the UP, DOWN, and FILE commands.

If the Debugger cannot find the debug listing for the current or specified procedure, it tries to generate it from the procedure source code. If the Debugger can only find the r-code for the procedure, each time it would normally display the listing, the Debugger leaves the listing panel unchanged and displays a message that the listing is not available. For more information on how Progress searches for debug listing files, see Chapter 6, "Debugger Files."

Listing Indicators

When the panel displays the listing for the current procedure, it also displays up to two listing indicators: a *line pointer* that shows the next line to execute and a *breakpoint indicator* that shows each line where a breakpoint is set. The line pointer is an arrow (→ in Windows; => in Motif) that points to the line number of the next executable line. The breakpoint indicator in Windows is a thick-lined octagon and in Motif is an asterisk (*), placed to the left of the line number of each line where a breakpoint is set.

NOTE: In this manual, all example listing panel code fragments use the Motif line pointer (=>) and breakpoint indicator (*).

If you use the FILE command to list a procedure that is not in the Debugger context, breakpoint indicators appear on the lines where breakpoints are set, whether or not the lines are executable. If the procedure is in the Debugger context, all breakpoint indicators are listed on effective executable lines, whether or not they were set on those lines. If you list a procedure in the Debugger context other than the executing procedure, the Windows line pointer becomes a box arrow to indicate that it is waiting on a subprocedure executed from the indicated line; the Motif line pointer becomes a double arrowhead (>>) for the same purpose.

Listing Name

The Debugger window labels the panel with the name of the procedure whose listing it is displaying. For example, the listing panel in Figure 4–1 and Figure 4–2 displays the listing for `r-runper.p`.

NOTE: Any procedure that you run in application mode from a Procedure Editor buffer is displayed using a filename with a `.ped` extension. This is the filename for a temporary procedure file where the Editor writes the 4GL contents of the buffer. The names of temporary procedure files written by the AppBuilder have a `.ab` extension.

Text Selection

The Debugger window provides a text cursor that you can use to select lines and parts of lines to set breakpoints and to specify other arguments to Debugger commands. For more information on using text selections in the listing panel to enter Debugger commands, see Chapter 5, "Running a Debugging Session."

4.2.2 Button Panel

The *button panel* is the area of the Debugger window where buttons are displayed. You can define a button to invoke any combination of Debugger commands and macros, and invoke the definition by choosing the button with the mouse. You can define buttons during a debugging session or at Debugger initialization. The Debugger comes with a default set of buttons defined at initialization.

For more information on defining buttons during a debugging session, see Chapter 5, "Running a Debugging Session." For more information on the default buttons and defining buttons at Debugger initialization, see Chapter 6, "Debugger Files."

4.2.3 Command Panel

The *command panel* provides an area to enter commands from the keyboard, and to record a history of all commands entered and executed during a debugging session. This *command history* includes commands entered from the menu bar, button panel, and command files, as well as the command panel itself. Each entered command echoes and scrolls into the panel scrolling area before it executes. The panel also displays the number of commands entered in the command queue and ready to execute.

Panel Components

The command panel provides a different interface in each window system. In Windows, the panel has two components:

- Commands in Queue field
- Scrolling area to enter commands and display the command history

In Motif, the panel has three components:

- Scrolling area to display the command history
- Commands in Queue field
- Enter Command field to enter commands

The command panels in Figure 4–1 and Figure 4–2 of the "Window Layout" section show the CONTINUE command typed in the command panel and ready to enter in the command queue.

Command Queue

The Debugger places all commands that you enter in a *command queue*. This is a typical first-in-first-out (FIFO) input queue. When the Debugger has control with the queue empty, any single command you enter is executed immediately. However, if you enter several commands using a command file or enter commands directly in the Debugger window while your application has control, all commands remain in the command queue until the Debugger is able to execute them. The Commands in Queue field (queue indicator) displays the number of commands in the Debugger command queue at any moment.

NOTE: You can purge the command queue at any time using the break interrupt function from the Debugger window. For more information, see the "Menu Bar" section of this chapter and Chapter 5, "Running a Debugging Session."

Panel Operation

As each command enters the command queue, the *queue indicator* increments and the command scrolls up to the bottom of the command history. As the Debugger executes each command, the queue indicator decrements. When the Debugger has executed all commands in the queue, the queue indicator displays 0.

Therefore, if you enter a command in the Debugger window and the queue indicator stays set at 1 or greater, you know that the application you are running is in control, and not the Debugger. The Debugger can begin executing these commands only after your application reaches a breakpoint or returns control to the Debugger when it terminates.

Note that the next command the Debugger executes is located on the *n*th line from the bottom of the command history, where *n* is the integer displayed by the queue indicator. For example, suppose the command panel displays the following command history:

```
break 1000
continue
display variable ExecStatus
display field customer.Name
display field customer.Credit
display variable OrdersTotal
```

If the queue indicator is 4, when the Debugger gets control, it will begin by executing the `display variable ExecStatus` command on the fourth line from the bottom.

For more information on entering commands using the command panel and other components of the Debugger window, see Chapter 5, "Running a Debugging Session."

4.2.4 Data Panel

The *data panel* maintains a scrolling display of messages and information during your debugging session. The data displayed includes:

- Session messages from the Debugger
- Responses from Debugger information commands like SHOW and DISPLAY
- Output from the DEBUGGER handle DISPLAY-MESSAGE( ) method

Most Debugger messages are appended with a four-digit message number that identifies the message type for maintenance purposes. For examples of the varied information displayed in this panel, see Chapter 2, "Quick Tutorial," Chapter 5, "Running a Debugging Session." and Appendix A, "Command Reference." For more information on the DISPLAY-MESSAGE( ) method, see Chapter 7, "DEBUGGER and SESSION System Handles," and the *Progress Language Reference*.

4.3 Menu Bar

Unlike other Application Development Environment (ADE) tools, the primary way to access Debugger functions is through commands that you enter at the keyboard. However, you can also use the pull-down menus in the menu bar to access common Debugger commands and functions. As with other tools in the ADE, you can access these menus in more than one way, using the mouse to choose menu options or using accelerator keys and mnemonics to invoke menu options directly from the keyboard. The menu options that invoke Debugger commands place the commands in the command queue as if you typed them into the command panel.

The following sections describe the Debugger menus and how to access them. The description for each menu option that invokes a Debugger command lists the command in parentheses. For more information on entering commands, see Chapter 5, "Running a Debugging Session."

4.3.1 Accelerator Keys and Mnemonics

In the following tables, some menu options have predefined accelerator keys labeled to the right of each menu option. *Accelerator keys* are one or more function keys that you can press to choose a menu option. The listed accelerator keys are predefined in Windows only, and cannot be changed. In Motif, you can define your own accelerator keys for each Debugger menu option in your `.Xdefaults` file. For more information on defining accelerator keys in Motif, see Chapter 6, "Debugger Files."

Each menu option has a *mnemonic*, which is a single letter that you can press to choose a menu option after you select the menu bar. The mnemonic for each menu option is underlined in its label. The mnemonics listed in the tables Table 4–1 through Table 4–5 are for Windows. The mnemonics for Windows and Motif are usually the same. Where there is a difference, use the mnemonic underlined on your Debugger menu.

4.3.2 File Menu

The File menu provides application information and the standard means to end your debugging session. Table 4–1 lists the File menu options and a brief description of each. More detailed descriptions follow the table.

Table 4–1: File Menu

File	
Open...	Displays a debug listing (FILE command).
Exit	Terminates the Debugger (EXIT command).

File→ Open

The Open option displays the debug listing for a procedure. This option opens a dialog box that requests the filename of a procedure. After entering the procedure filename, the debug listing of the procedure is displayed in the listing panel.

File→ Exit

The Exit option terminates the Debugger and removes it from the Progress environment. In both application and stand-alone modes, this returns control to the environment or application in which the Debugger was invoked.

4.3.3 Search Menu

The Search menu allows you to find a specified text or line in your debug listing. Table 4–2 lists the Search menu options and a brief description of each. There are no command equivalents for these options. More detailed descriptions follow the table.

Table 4–2: Search Menu

Search	
Find... Ctrl-F	Finds the specified text.
Find Next F9	Finds the next occurrence of the specified text.
Find Prev Shift+F9	Finds the previous occurrence of the specified text.
Goto Line... Ctrl-G	Goes to the line specified by number.

Search→ Find

The Find option opens a dialog box that requests:

- The text you want to find (Find What)
- Whether to match letter case in the text during the search (Match Case)
- (OSF/Motif only) Whether to wrap around to the listing top or bottom during the search (Wrap at Beginning/End)
- Whether to search backward or forward (Direction: Up or Down)

Choosing the Find Next button (OK in Motif) initiates the search. If found, the Debugger highlights the text, scrolling it into view, if necessary. If the text is not found, the Debugger displays an alert box message.

Search→ Find Next

The Find Next option highlights the next occurrence of the text specified by the Find option, scrolling it into view, if necessary. If Wrap at Beginning/End is turned off for the Find option and there is no next occurrence, the Debugger displays an alert box message (Motif only).

Search→ Find Prev

The Find Prev option highlights the previous occurrence of the text specified by the Find option, scrolling it into view, if necessary. If Wrap at Beginning/End is turned off for the Find option and there is no next occurrence, the Debugger displays an alert box message (Motif only).

Search→ Goto Line

The Goto Line option opens a dialog box that requests the line number of a line in the debug listing you want to see. Choosing OK causes the text cursor to move to the line you specify, scrolling it into view, if necessary. You must specify a line number greater than zero. If the line you specify is beyond the last line, the text cursor moves to the end of the listing.

4.3.4 Debug Menu

The Debug menu controls procedure execution and Debugger-application interaction.Table 4–3 lists the Debug menu options and a brief description of each. More detailed descriptions follow the table.

Table 4–3: Debug Menu

Debug		
Next	F6	Executes one procedure line (NEXT command).
Step	F7	Executes one procedure line (STEP command).
Run...	F2	Executes the specified procedure (RUN command).
Break...	F8	Sets a breakpoint in a procedure (BREAK command).
Interrupt	Ctrl-Break	Invokes the break interrupt function to give the Debugger control.

Debug→ Next

The Next option executes the next line in the executing procedure, stopping on the following line. Any procedure calls on the next line are executed without interruption, unless they contain a breakpoint.

Debug→ Step

The Step option executes the next line in the executing procedure, stopping on the following line. If the next line contains any procedure calls, it steps into and stops on the first line of the first procedure call it encounters.

Debug→ Run

The Run option executes a procedure. This option opens a dialog box that requests a procedure name and executes the procedure you specify. The default is the procedure displayed in the listing panel, if any.

Debug→ Break

The Break option sets a breakpoint. This option opens a dialog box that requests a procedure name and line number, and sets a breakpoint in the procedure and line number you specify. The default is the first executable line of the procedure displayed in the listing panel, if any.

Debug→ Interrupt

The Interrupt option bypasses the command queue to interrupt any executing procedure, giving immediate control to the Debugger. This allows you to enter and execute Debugger commands whenever you want. This option immediately interrupts and halts any executing procedure on the next executable line, and gives control to the Debugger without actually setting a breakpoint. It also clears the Debugger command queue, whether or not a procedure is executing, to accept commands from the Debugger window. There is no command equivalent for this option.

4.3.5 Options Menu

The Options menu allows you to make changes to your Debugger environment. Table 4–4 lists and describes the Options menu option. More detailed information follows the table.

Table 4–4: Options Menu

Options	
Save Settings...	Saves the specified Debugger settings.

Options→ Save Settings

The Save Settings option saves any or all of the following Debugger session settings for future debugging sessions:

- Button definitions
- Macro definitions
- Window and panel sizes

This option opens a dialog box that contains the list of Debugger settings. Check off each setting you want to save when the Debugger exits.

In Windows, the Debugger saves all of these settings in the Registry or in your `PROGRESS.INI` file. In Motif, the Debugger saves button and macro definitions in your initial command file, and saves window and panel sizes in your `.Xdefaults` file. For more information on these files, see Chapter 6, "Debugger Files."

4.3.6 Help Menu

The Help menu accesses on-line help for the Debugger, as well as a cross-reference to the Progress language. Table 4–5 and Table 4–6 summarize the Help menu options as they appear in the Motif Debugger and the Windows Debugger.

Table 4–5: Help Menu in Windows Debugger

Help	
Help Topics	Accesses the Help table of contents, index and search function.
About Debugger...	Gets version information about the current Debugger installation.

Table 4–6: Help Menu in Motif Debugger

Help	
Topics	Accesses the Help table of contents, index and search function.
About Debugger...	Gets version information about the current Debugger installation.

5

Running a Debugging Session

A debugging session alternates between managing the application when the Debugger has control and monitoring or interrupting execution when the application has control. You can invoke a number of actions at any point in this operation. This chapter explains the actions that you can or might want to invoke at each point, and the related consequences.

This chapter describes:

- The typical pattern of operation
- Entering Debugger commands
- Exiting a debugging session
- Managing breakpoints
- Controlling procedure execution
- Examining and modifying procedure data
- Displaying Debugger and session information
- Defining and using macros and buttons
- Logging a debugging session
- Debugging events

5.1 Pattern of Operation

How you run a debugging session varies depending on how you start it and what you want to accomplish. In general, follow these steps to run a debugging session:

1 ♦ Start the Debugger in application or stand-alone mode.

For information on starting a debugging session, see Chapter 3, "Starting and Stopping the Debugger." Before starting, you might want to set up initial starting parameters for the Debugger. For information on initialization, see Chapter 6, "Debugger Files."

All remaining steps require you to enter commands to the Debugger. For more information, see the "Entering Debugger Commands" section in this chapter.

2 ♦ If your application is not already running, run it from the Debugger.

For information on running an application from the Debugger, see the "Controlling Procedure Execution" section. Before running an application, you might want to set up initial breakpoints, macros, and buttons. For information on setting breakpoints, see the "Managing Breakpoints" section. (Also see the "Debugging Events" section for important information on when and how you can interrupt events.) For information on setting up macros and buttons, see the "Defining and Using Macros and Buttons" section.

NOTE: You can interrupt application execution at any time and give control to the Debugger by choosing the Debug→ Interrupt option.

3 ♦ When the Debugger gets control, you can set or remove any breakpoints, define additional macros and buttons, or display information about the debugging session. You can also exit the Debugger at this point.

For information on setting and removing breakpoints, see the "Managing Breakpoints" section. For information on displaying session information, see the "Displaying Debugger and Session Information" section. For information on exiting the Debugger, see the "Exiting a Debugging Session" section.

4 ♦ When you are ready to continue application execution, enter the appropriate command to execute a line at a time, all lines up to the next breakpoint, or a completely new application procedure. You can also force the application to terminate at this point.

For information on continuing application execution, see the "Controlling Procedure Execution" section.

Each time the Debugger has control, you can cycle through the options available in Step 2 through Step 4. Each time your application has control, you can interact with it as if the Debugger were not running.

You can also keep a record of all or selected parts of your debugging session. For information on recording a debugging session, see the "Logging a Debugging Session" section.

5.2 Entering Debugger Commands

You can enter commands to the Debugger:

- With the command panel
- With buttons
- With menu options
- From a command file

From the viewpoint of the Debugger, all of the techniques for entering commands are equivalent. For a complete description of each command, see Appendix A, "Command Reference."

5.2.1 Intrinsic Functions and the Command Queue

All Debugger commands, no matter how you enter them, enter the command queue to be executed. Certain menu options, however, invoke *intrinsic functions*, not Debugger commands. Intrinsic functions are never queued. You can execute them immediately whenever the Debugger window is enabled, and whether or not an application has control. The Options and Help menus and the Interrupt option of the Debug menu all invoke intrinsic functions.

NOTE: Unlike the Help menu and its submenus, the HELP command provides access to the Debugger Help system through the command queue.

5.2.2 Break Interrupt Function

If you need to execute a Debugger command immediately, you can invoke the break interrupt function to force any executing procedure to give control to the Debugger. The break interrupt function also purges the command queue, allowing you to immediately enter and execute a command. If an application procedure has control, this function interrupts and halts procedure execution on the next executable line, and then gives control to the Debugger. The only way to invoke the break interrupt function is by choosing the Debug→ Interrupt option.

5.2.3 Command Panel Entry

You can enter commands in the command panel by:

- Typing them in the command panel
- Selecting previously entered commands from the command history
- Using macros

Typing Commands

To type a Debugger command in Windows, enter it in the command panel immediately below any listed command history. To type a debugger command in OSF/Motif, enter it in the Enter Command field. You can enter multiple commands as a single command by separating them with a semicolon (;), as in the following example:

```
step; show stack; list variables
```

Each single or multiple command entry is placed in the command queue and executed as a single command. For information on the syntax for each command, see Appendix A, "Command Reference."

Re-entering Commands from the Command History

To enter a previously entered command, choose it from the command history as follows:

- In Windows, click on the command in the scrolling area, and press **RETURN**
- In Motif, double-click on the command in the command history

In each environment, the command scrolls into the command history as it is placed in the command queue.

Entering Commands with Macros

Using macros, you can also enter one or more Debugger commands as a single command, and specify up to three parameters to satisfy command arguments. For more information on using macros to enter commands, see the "Defining and Using Macros and Buttons" section.

5.2.4 Button Entry

You can define buttons to enter almost any Debugger command or string of commands. When you choose a button with the mouse, the associated command or command string is placed in the command queue and executed as a single command. Commands in a button definition can also substitute text selections in the listing panel for command parameters. For more information on entering commands with buttons, see the "Defining and Using Macros and Buttons" section.

5.2.5 Menu Options

Most of the Debugger menu options invoke Debugger commands, as if you entered them from the command panel. These commands enter the command queue immediately for execution. However, some menu options invoke commands that take parameters. For these menu options, the Debugger displays dialog boxes in which you enter the parameters. After completing the dialog, the specified command enters the command queue for execution.

The menu options that provide dialog boxes to enter parameters include:

- File→ Open to invoke the FILE command. For more information on this dialog box, see the "Displaying Debugger and Session Information" section.
- Debug→ Run to invoke the RUN command. For more information on this dialog box, see the "Controlling Procedure Execution" section.
- Debug→ Break to invoke the BREAK command. For more information on this dialog box, see the "Displaying Debugger and Session Information" and "Setting Breakpoints" sections.

5.2.6 Command File Entry

You can record a series of commands in a text file and enter them all into the command queue for execution. Use the INPUT command to specify the file. This is useful for entering a long series of commands that you want to execute often. You can record any Debugger command in this file, including additional INPUT commands to execute other command files. The following example enters the debugger commands from the file `commands.txt` into the command queue for execution:

```
input commands.txt
```

Each command line in the file takes up one entry in the command queue. The Debugger inputs and executes all of the commands from this file before executing any other commands.

During command file entry, if the Debugger encounters an unrecognized command, it stops command execution and purges the queue of all remaining commands (including any commands queued after those from the command file). The Debugger then continues to accept command input from the Debugger window. You can also explicitly halt command file execution and purge the command queue at any time by invoking the break interrupt function (Debug→ Interrupt option).

For more information on creating command files for use with the INPUT command, see Chapter 6, "Debugger Files."

5.3 Exiting a Debugging Session

At any time while the Debugger has control, you can end a debugging session by entering the EXIT command or choosing the Exit option from the File menu. This command closes the Debugger window and any application procedure running and started with the Debugger RUN command.

5.3.1 Exiting in Application Mode

When you are debugging the initial application in application mode, you can terminate the debugging session in the following circumstances:

- When the initial application is run from the Procedure Editor, the application returns to the Procedure Editor and the Debugger window clears from the display.
- When the initial application is run from the AppBuilder, the application returns to the AppBuilder and the Debugger window clears from the display.
- When you are using the DEBUGGER handle in the initial application to initialize and invoke the Debugger at a breakpoint, the application returns to the environment where you started it, and the Debugger remains on the display or clears from the display if the application invokes the DEBUGGER:CLEAR( ) method.

For more information on debugging sessions run from the Editor and AppBuilder, see Chapter 3, "Starting and Stopping the Debugger." For more information on debugging sessions run in application mode using the DEBUGGER handle, see Chapter 7, "DEBUGGER and SESSION System Handles."

5.3.2 Debugger Settings on Exit

When the Debugger exits, it closes any open log file. Any buttons, macros, or changes in window size or location that you made during the session are lost, unless you save them using Options→ Save Settings. For more information on saving Debugger settings, see Chapter 4, "Debugger Window," and Chapter 6, "Debugger Files."

5.4 Managing Breakpoints

You can set and remove breakpoints in any procedure, whether or not it is the current procedure, and whether or not the procedure currently exists. A breakpoint is identified by both the procedure name and line number in the **debug listing** (not the source listing) where the breakpoint occurs. Note that a breakpoint occurs on a **physical line** (not a statement) in the procedure. If you want the procedure to break on a particular statement, you must ensure that the statement is the first statement on the line where you set the breakpoint.

NOTE: There are certain user-interface triggers where the Debugger ordinarily ignores all breakpoints. The Debugger cannot take control in these triggers without competing with the window system for the mouse pointer—an effect that, if allowed to happen, causes a deadlock in the window system. Instead, the Debugger displays a message indicating that it cannot take control there. Debugging these triggers requires alternative techniques. For more information, see the "Debugging Events" section in this chapter.

You can manage breakpoints using the commands listed in Table 5–1.

Table 5–1: Breakpoint Management Commands

Function	Command
Set a single breakpoint.	BREAK[1]
Remove one or more specified breakpoints.	CANCEL BREAK
Remove all breakpoints.	CANCEL ALL
Display all current breakpoints in the data panel.	SHOW BREAKS

[1] The Break option of the Debug menu also invokes the BREAK command.

5.4.1 Setting Breakpoints

When you set a breakpoint, you can specify it using either the procedure name, line number, or both, depending on whether the procedure is listed in the Debugger window. As you set it, the Debugger identifies each breakpoint by number, starting at 1. The Debugger indicates the breakpoint location in any listed procedure with a breakpoint indicator. The exact format of the breakpoint indicator differs with each window system. The examples in this section use the Motif format. For more information on breakpoint indicators for each supported window system, see Chapter 4, "Debugger Window."

Breakpoint Settings and References

This example sets a breakpoint on line 16 of the procedure specified by `arrmenu.p` in the BREAK command:

```
BREAK arrmenu.p 16
```

In response, the Debugger displays this message in the data panel:

```
Breakpoint (#2) set at arrmenu.p:16. (3017)
```

The `#2` is the *breakpoint number*. It indicates that the new breakpoint is the second breakpoint set in the session, and `arrmenu.p:16` indicates that the breakpoint is set on line 16 of `arrmenu.p`. You can use this information later to reference each breakpoint accurately and efficiently.

Effective Breakpoints

When you specify a breakpoint line, the Debugger always resolves it to an effective breakpoint during procedure execution. In general, an *effective breakpoint* is the closest executable line on or near the specified line number. More specifically, if the line number specifies a nonexecutable line (for example, a comment or DEFINE statement), the effective breakpoint occurs on the next executable line after the specified line. If the line number is less than 1 or greater than the last executable line, the effective breakpoint occurs on the corresponding first or last executable line in the procedure.

This command sets a breakpoint on the first executable line of the procedure specified by `arrmenu.p`:

```
BREAK arrmenu.p
```

Using only the procedure name always sets a breakpoint on the first executable line of the specified procedure.

The following example attempts to set a breakpoint on line 16 of the listed procedure (if there is a listed procedure). If there is no listed procedure, this example does not set a breakpoint, and displays a message that there is no source in which to set a breakpoint:

```
BREAK 16
```

Again, if you use a line number alone, the BREAK command sets an effective breakpoint on the next or closest executable line.

If the procedure where you set a breakpoint is active in the Debugger context, the Debugger always indicates the breakpoint using the line number of the effective breakpoint if it is different from the specified line number. For example, suppose you list the following procedure in the Debugger context using the FILE command:

```
        1   /*  custrpt.p  - lists all customers  */
        2
        3   for each customer:
        4       display cust-num name address city st zip.
        5   end.
```

If you set a breakpoint on line 1, the Debugger indicates the setting on line 3 in the listing and data panels:

```
        1   /*  custrpt.p  - lists all customers  */
        2
 *      3   for each customer:
        4       display cust-num name address city st zip.
        5   end.
```

```
Breakpoint (#9) set at custrpt.p:3. (3017)
```

If the procedure is inactive, the Debugger always indicates the breakpoint using the specified line number, because it does not, at that point, know how the procedure executes. However, when the procedure is executed, it always breaks on the effective breakpoint line, regardless of the line number used to specify the breakpoint.

The Debug→ Break Option

Choosing Debug→ Break also invokes the BREAK command. However, the Debugger displays a dialog box that prompts for the procedure name and line number for the breakpoint. This dialog box provides a default procedure name from the procedure displayed in the listing panel and a default line number from the displayed line selected with the text cursor.

For example, if you have `r-runper.p` listed in the Debugger window and placed the text cursor on line 25 in the listing panel, the dialog box displays `r-runper.p` and 25 as the default procedure and line number. Figure 5–1 shows the Set Breakpoint dialog box in Windows; Figure 5–2 shows it in Motif.

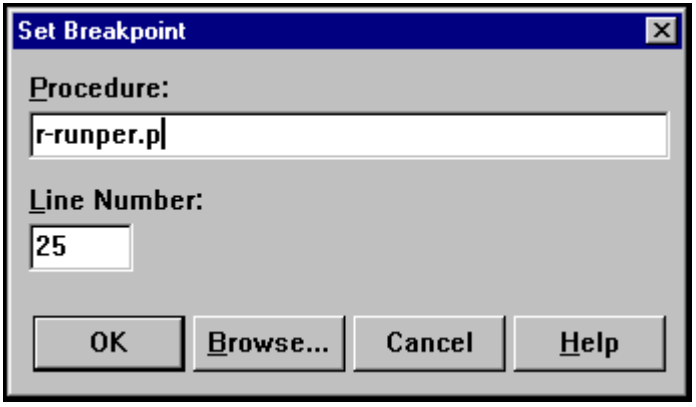


Figure 5–1: Set Breakpoint Dialog Box in Windows

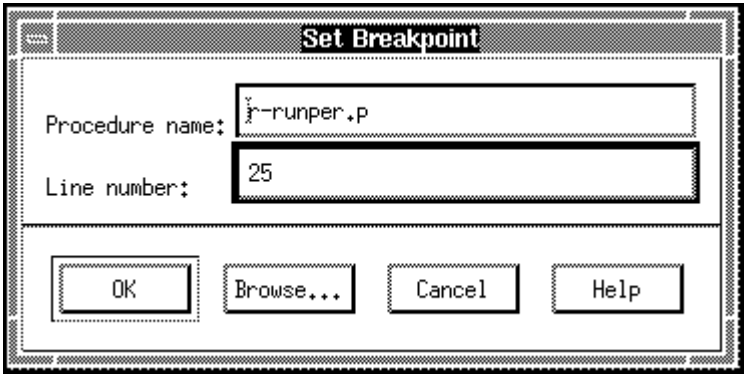


Figure 5–2: Set Breakpoint Dialog Box in Motif

You can change these defaults as needed. Choosing the Browse button opens a dialog box in which you can enter or select a procedure filename. The Help button displays a Help window that describes how to use this dialog box.

When you choose the OK button, the Break option attempts to set the breakpoint using the BREAK command.

5.4.2 Removing Breakpoints

To remove breakpoints during a debugging session, you must enter either the CANCEL BREAK command to remove one or more specified breakpoints or the CANCEL ALL command with the BREAKS option to remove all of them.

Using the Procedure Name and Line Number

You can remove a single breakpoint by specifying the procedure name and line number with the CANCEL BREAK command. You must specify the procedure name exactly as you specified it in the BREAK command or Debug→ Break option. If the procedure is inactive, you must specify the line number you used to set it. If the procedure is active, you can specify either the line number used to set it or, if different, the effective executable line.

If you set more than one breakpoint on nonexecutable lines that are effective for the same executable line, the SHOW BREAKS command reports multiple breakpoints on that same executable line. To cancel all breakpoints on that line, you must cancel each one using its effective executable line number, using each of the original specified line numbers, or using the ordinal breakpoint numbers (see the "Using Breakpoint Numbers" section).

For example, if line 7 is the first executable line of `arrmenu.p` and you set a breakpoint on lines 1, 3, and 5 to prevent `arrmenu.p` from breaking at line 7, you must cancel each breakpoint on lines 1, 3, and 5 or cancel the breakpoint on line 7 three times. Using the effective breakpoint line number cancels the breakpoints on the nonexecutable line numbers in the order they were set.

Using Breakpoint Numbers

Specifying breakpoint numbers is the most efficient way to cancel one or more individual breakpoints with the CANCEL BREAK command. No two breakpoints have the same number. The following command cancels four breakpoints by number:

```
CANCEL BREAK #9 #6 #2 #10
```

The BREAK command displays the number of each breakpoint when you set it. You can also identify the numbers of all current breakpoints using the SHOW BREAKS command. For more information, see the "Displaying Debugger Session Settings" section.

5.4.3 Working with Inactive Procedures

If you are setting or removing breakpoints in an inactive procedure, you might find it helpful to display the debug listing for the procedure in the listing panel using the FILE command. Note that breakpoint indicators appear in the procedure listing even if the procedure is inactive. However, the breakpoints indicated are on the specified lines, not necessarily on the effective executable lines. The effective breakpoints are indicated when you actually run the procedure. For more information on the FILE command, see the "Displaying Debugger and Session Information" section.

5.4.4 Handling Internal Procedures and Procedure Files

The procedure name in a breakpoint specification can identify either a procedure filename or an internal procedure name. If both a procedure file and internal procedure exist by the same name (including extension) and you set a breakpoint using this name, Progress applies the breakpoint to both the internal and external procedures. The only way to distinguish the two in the same debugging session (and calling procedure) is to reference the external procedure file by a relative or absolute pathname. You must use the same pathname in breakpoint specifications, Debugger RUN commands, and procedure RUN statements that refer to the external procedure file. Any reference to the procedure name alone always defaults to an internal procedure, if it exists, in the executing procedure file.

For example, if procedure file `arrmenu.p` calls both internal procedure `custrpt.p` and an external procedure with the pathname `/db/v9/custrpt.p` you can set the following breakpoints to stop on the first executable line of each procedure:

```
BREAK custrpt.p
BREAK /db/v9/custrpt.p
```

All calls to the external procedure must also refer to `/db/v9/custrpt.p`.

You can also set a breakpoint in an internal procedure by instead referencing the calling procedure name and line number. This eliminates any confusion in setting the breakpoint between an internal procedure and external procedure file. For example, if procedure file `arrmenu.p` calls internal procedure `custrpt.p` and the first line of `custrpt.p` is line 33 in `arrmenu.p`, you can unambiguously set a breakpoint on the first line of `custrpt.p` using the following setting:

```
BREAK arrmenu.p 33
```

5.5 Controlling Procedure Execution

The purpose of the Debugger is to control procedure execution so that you can closely follow the actual decisions and code paths that a procedure takes as it runs. An initial application procedure starts up in one of two ways:

- Together with the Debugger, stopped at a breakpoint on the first executable line (application mode). From here you can continue procedure execution with a control flow command like CONTINUE or start another procedure with the RUN command.
- After the Debugger begins (stand-alone mode). From here you must start initial procedure execution with the RUN command.

For more information on starting a debugging session in application or stand-alone mode, see Chapter 3, "Starting and Stopping the Debugger."

5.5.1 Interrupting a Running Procedure

Regardless of how you start it, the current procedure runs until one of the following operations occurs:

- The procedure reaches a breakpoint, giving control to the Debugger
- You interrupt it using the break interrupt function (choosing Debug→ Interrupt), giving control to the Debugger
- The procedure completes execution, returning control to either the Debugger or another startup environment (for example, the Procedure Editor)

NOTE: Any procedure you start from the Debugger always returns to the Debugger when it terminates. All other procedures return to their startup environment, whether it is the Editor, AppBuilder, operating system, or another procedure.

5.5.2 Executing an Interrupted Procedure

Any time the Debugger has control at a procedure interruption, you can continue or terminate procedure execution using one of the control flow commands listed in Table 5–2.

Table 5–2: Execution Commands

Function	Command
Start any inactive procedure, or restart any active procedure recursively.	RUN[1]
Continue running the executing procedure from its current stopping point.	CONTINUE
Execute one or more lines of the executing procedure.	NEXT[1]
Execute one or more lines of the executing procedure, stepping into any trigger or subprocedure.	STEP[1]
Terminate the executing application as if it executed a 4GL QUIT statement.	QUIT
Terminate the executing application as if it executed a 4GL STOP statement.	STOP

[1] The Run, Next, and Step options of the Debug menu invoke the corresponding commands.

Note that when you enter a RUN or CONTINUE command, the line pointer and data panel are updated at the next breakpoint or break interrupt point. When you enter a STEP or NEXT command, the line pointer is updated when the specified number of lines are executed or at the next breakpoint or break interrupt point, whichever happens first.

5.5.3 Running Procedures from the Debugger

The RUN command executes the specified procedure from the beginning. If you are already executing a procedure when you invoke the RUN command, the Debugger pushes the executing procedure onto the call stack, and begins executing the specified procedure as if it were called from the pushed procedure. If the specified procedure is the same as the pushed procedure or any other active procedure, it executes recursively. In any case, the specified procedure becomes the current procedure.

NOTE: Unlike the 4GL RUN statement, you cannot pass procedure parameters, use the PERSISTENT option, or use the IN option with the Debugger RUN command. However, you can run any internal procedure within a persistent or nonpersistent external procedure by first making the specified external procedure the current procedure. For information on setting the current procedure, see the "Examining and Navigating in the Debugger Context" section.

When the specified procedure completes execution, it returns control to the Debugger at the point where it was executed with the RUN command. If there is a procedure on the procedure call stack, the Debugger reinstates it as the executing procedure with the line pointer at its current stopping point.

The Debug→ Run Option

You can invoke the RUN command by choosing the Debug→ Run option. The Run option opens a dialog box that prompts you for the name of your application procedure. Figure 5–3shows the Run Procedure dialog box in Windows; Figure 5–4 shows it in Motif.



Figure 5–3: Run Procedure Dialog Box in Windows

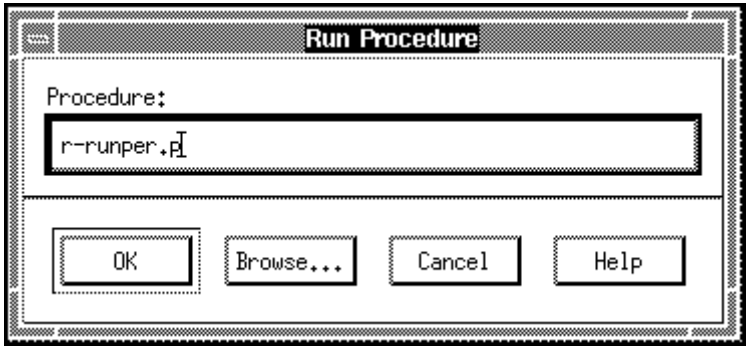


Figure 5–4: Run Procedure Dialog Box in Motif

This dialog box uses any procedure listed in the Debugger window as the default. You can change this default as needed. Choosing the Browse button opens a dialog box in which you can enter or select a different procedure filename. The Help button displays a Help window that describes how to use this dialog box.

When you choose the OK button, the Run option attempts to execute the specified procedure using the RUN command.

5.5.4 Continuing Procedure Execution

The CONTINUE command continues execution from the current stopping point of the executing procedure up to the next interruption or termination point. If a breakpoint or break interrupt occurs in a procedure called directly or indirectly from the continued procedure, execution stops in the procedure where the interruption occurs. The interrupted procedure then becomes the current procedure and its listing is displayed in the Debugger window.

5.5.5 Executing a Specified Number of Lines

Both the STEP and NEXT commands execute a procedure one or more lines at a time. However, the STEP command "steps into" any called subprocedure, continuing for the specified number of lines in the called subprocedure. The NEXT command "steps over" any called subprocedure, treating it as a statement and continuing execution for the specified number of lines in the current procedure. The NEXT command also honors breakpoints in the current or any called subprocedure, and completes execution at the first breakpoint encountered, stopping in a called subprocedure, if necessary. The STEP command ignores breakpoints; it always executes the number of lines you specify up to the application termination point.

STEP Command Subprocedure Processing

When the STEP command executes any line that calls a subprocedure, it performs these actions:

1. Executes any statements on the line before the subprocedure call.
2. Pushes the current (calling) procedure onto the procedure call stack, and executes the called subprocedure.
3. Continues execution from the first executable line of the called subprocedure up to the last line specified for the STEP command.
4. When the STEP command stops in a subprocedure, the subprocedure becomes the current procedure and the Debugger displays its listing with the line pointer on the next line to execute.

For example, suppose you are running the procedure fragments `dbmain.p` and `dispmsg.p`. In the Debugger, you are stopped on line 23 of `dbmain.p` and you have set a breakpoint on line 13 of `dispmsg.p`:

```
        1   /***** PROCEDURE: dbmain.p *****/
        2
        3   DEFINE NEW SHARED VARIABLE MsgArray
        4       AS CHARACTER FORMAT "x(40)" EXTENT 10 INITIAL [?].
                  .
                  .
                  .
 =>    23   IF dberror THEN DO:
       24      HIDE.
       25      DisplayedMsg = TRUE. RUN dispmsg.p. InitialState = TRUE.
       26      RETURN.
       27   END. /* dberror */
```

You then enter this STEP command:

```
STEP 7
```

The procedure dbmain.p executes through line 25 (two lines) where it calls dispmsg.p after executing the assignment to DisplayMsg. The procedure dispmsg.p then continues execution for five more executable lines (including the procedure call plus lines 9 through 13), and stops on line 14 displaying the following listing. Note that line 12 is considered nonexecutable and part of line 13 because it contains only part of a statement:

```
        1   /***** PROCEDURE: dispmsg.p *****/
        2
        3   DEFINE SHARED VARIABLE MsgArray /* Array of message lines */
        4       AS CHARACTER FORMAT "x(40)" EXTENT 10.
        5
        6   DEFINE VARIABLE MsgText AS CHARACTER FORMAT "x(4000)".
        7   DEFINE VARIABLE LineNo AS INTEGER.
        8
        9   LineNo = 1.
       10   MsgText = "".
       11   REPEAT WHILE (LineNo <= EXTENT(MsgArray) AND
       12                 MsgArray[LineNo] <> ?):
*      13       MsgText = MsgText + MsgArray[LineNo] + "~012". /* + NL */
 =>    14       LineNo = LineNo + 1.
       15   END.
       16
       17   MESSAGE MsgText VIEW-AS ALERT-BOX MESSAGE BUTTONS OK.
                  .
                  .
                  .
       24   MsgArray = ?.
```

NEXT Command Subprocedure Processing

When the NEXT command executes a line that calls a subprocedure, it performs these actions:

1. Executes any statements on the line before the subprocedure call.
2. Executes the called subprocedure.
3. If there are no breakpoints in the called subprocedure, it completes the called subprocedure and any remaining statements on the line.

   If it reaches a breakpoint, it pushes the current (calling) procedure onto the procedure call stack, and stops on the breakpoint line, making the procedure where the breakpoint occurred the current procedure and displaying its listing in the Debugger window.

For example, suppose you are running the procedure fragments `dbmain.p` and `dispmsg.p`. In the Debugger, you are stopped on line 23 of `dbmain.p` and you have set a breakpoint on line 13 of `dispmsg.p`:

```
        1   /***** PROCEDURE: dbmain.p *****/
        2
        3   DEFINE NEW SHARED VARIABLE MsgArray
        4       AS CHARACTER FORMAT "x(40)" EXTENT 10 INITIAL [?].
                  .
                  .
                  .
 =>    23   IF dberror THEN DO:
       24      HIDE.
       25      DisplayedMsg = TRUE. RUN dispmsg.p. InitialState = TRUE.
       26      RETURN.
       27   END. /* dberror */
```

You then enter this NEXT command:

```
NEXT 3
```

The procedure dbmain.p executes through line 25 (two lines) where it calls dispmsg.p after executing the assignment to DisplayMsg. The procedure dispmsg.p then continues execution through line 12, and stops at the breakpoint on line 13 displaying the following listing:

```
        1   /***** PROCEDURE: dispmsg.p *****/
        2
        3   DEFINE SHARED VARIABLE MsgArray /* Array of message lines */
        4       AS CHARACTER FORMAT "x(40)" EXTENT 10.
        5
        6   DEFINE VARIABLE MsgText AS CHARACTER FORMAT "x(4000)".
        7   DEFINE VARIABLE LineNo AS INTEGER.
        8
        9   LineNo = 1.
       10   MsgText = "".
       11   REPEAT WHILE (LineNo <= EXTENT(MsgArray) AND
       12                 MsgArray[LineNo] <> ?):
 *=>   13       MsgText = MsgText + MsgArray[LineNo] + "~012". /* + NL */
       14       LineNo = LineNo + 1.
       15   END.
       16
       17   MESSAGE MsgText VIEW-AS ALERT-BOX MESSAGE BUTTONS OK.
                  .
                  .
                  .
       24   MsgArray = ?.
```

Without the breakpoint in dispmsg.p, dbmain.p executes three executable lines stopping on line 26 at the RETURN statement.

STEP and NEXT Commands Inside a Subprocedure

From the current stopping point in the subprocedure, you can now execute any control flow command. If you invoke a STEP or NEXT command on the last executable line of the subprocedure, it performs these actions:

1. Executes the last line of the subprocedure.

2. Returns to the calling procedure, executing any remaining statements on the line of the subprocedure call.

3. Continues on to the next executable line after the subprocedure call and executes the number of executable lines specified for the STEP or NEXT command minus one.

Thus if you invoke a one-line STEP or NEXT command on the last line of the subprocedure, it stops execution on the next line after the subprocedure call.

For example, suppose you are stopped on the last executable line of `dispmsg.p`:

```
        1   /***** PROCEDURE: dispmsg.p *****/
        2
        3   DEFINE SHARED VARIABLE MsgArray /* Array of message lines */
        4       AS CHARACTER FORMAT "x(40)" EXTENT 10.
        5
        6   DEFINE VARIABLE MsgText AS CHARACTER FORMAT "x(4000)".
        7   DEFINE VARIABLE LineNo AS INTEGER.
        8
        9   LineNo = 1.
       10   MsgText = "".
       11   REPEAT WHILE (LineNo <= EXTENT(MsgArray) AND
       12                 MsgArray[LineNo] <> ?):
       13       MsgText = MsgText + MsgArray[LineNo] + "~012". /* + NL */
       14       LineNo = LineNo + 1.
       15   END.
       16
       17   MESSAGE MsgText VIEW-AS ALERT-BOX MESSAGE BUTTONS OK.
                  .
                  .
                  .
 =>    24   MsgArray = ?.
```

You then enter the NEXT command:

```
NEXT
```

This both executes the last line in `dispmsg.p` and completes execution of the line in `dbmain.p` (line 25) that called dispmsg.p, stopping on line 26 of `dbmain.p` and displaying the following listing:

```
        1   /***** PROCEDURE: dbmain.p *****/
        2
        3   DEFINE NEW SHARED VARIABLE MsgArray
        4       AS CHARACTER FORMAT "x(40)" EXTENT 10 INITIAL [?].
                  .
                  .
                  .
       23   IF dberror THEN DO:
       24      HIDE.
       25      DisplayedMsg = TRUE. RUN dispmsg.p. InitialState = TRUE.
 =>    26      RETURN.
       27   END. /* dberror */
```

Menu Options for the STEP and NEXT Commands

You can execute the STEP and NEXT commands by choosing the corresponding Debug→ Step and Debug→ Next options. Each option immediately invokes the specified command to execute a single line.

5.5.6 Terminating an Application from the Debugger

The QUIT and STOP commands, when executed, immediately terminate the running application. In general, the QUIT command terminates in exactly the same way as the 4GL QUIT statement, and the STOP command terminates in exactly the same way as the 4GL STOP statement. In other words, the QUIT command completes the active transaction, and the STOP command backs it out.

Unwinding the Procedure Call Stack

If you started the running application from the Debugger using the RUN command, both the QUIT and STOP commands terminate the application, returning control to the Debugger. You must enter additional QUIT or STOP commands for each additional application on the procedure call stack you want to terminate. If the Debugger is running in application mode and the initial application is next on the stack, the initial application terminates and returns to its startup environment. If the initial application is a startup (-p) procedure and the Debugger is running in application mode using the DEBUGGER system handle, entering the STOP command restarts the startup procedure (unless the STOP is trapped by an ON STOP statement). If the Debugger is running in stand-alone mode from the operating system (-debug), entering the STOP command returns control to the Debugger.

NOTE: Any application you start from the Debugger returns to the Debugger, whether you terminate it from within the application or with the Debugger QUIT or STOP command.

5.6 Examining and Modifying Procedure Data

You can examine and modify a range of data objects that are accessible to the current procedure using the commands listed in Table 5–3.

Table 5–3: Data Commands

Function	Command
List the names of buffers, parameters, and variables defined in the current procedure.	LIST BUFFERS LIST PARAMETERS LIST VARIABLES
Display the value of specified fields, parameters, variables, and attributes for widgets and handles in the current procedure.	DISPLAY FIELD DISPLAY PARAMETER DISPLAY VARIABLE DISPLAY *widget*:*attribute* DISPLAY *handle*:*attribute*
Modify the value of specified fields, parameters, variables, and attributes for widgets and handles in the current procedure.	ASSIGN FIELD ASSIGN PARAMETER ASSIGN VARIABLE ASSIGN *widget*:*attribute* ASSIGN *handle*:*attribute*

Note that these commands operate on data accessible to any active procedure in the Debugger context, not just the executing procedure. To access data in an active procedure other than the executing procedure, you must navigate the call stack using the UP and DOWN commands to choose a new current procedure. For more information on navigating the call stack, see the "Displaying Debugger and Session Information" section.

5.6.1 Listing Buffers, Parameters, and Variables

You can list the names of all buffers, parameters, or variables that are accessible from the current procedure using the corresponding LIST BUFFERS, LIST PARAMETERS, and LIST VARIABLES command. The Debugger lists the name of each data object with important qualifying information. For example, a LIST BUFFERS command might produce the following output:

```
Buffers for r-ordbug.p:6
    CustBuf (SHARED BUFFER) for Customer
    order (IMPLICIT BUFFER)
    order-line (IMPLICIT BUFFER)
    item (IMPLICIT BUFFER)
```

This shows that the procedure `r-prdbug.p` is stopped at line 6 and accesses one shared buffer and three implicit (database default) buffers.

5.6.2 Displaying Data and Attribute Values

You can display the value of any specified field, parameter, variable, or widget attribute that is accessible from the current procedure using the corresponding DISPLAY FIELD, DISPLAY PARAMETER, DISPLAY VARIABLE, DISPLAY *widget*:*attribute*, or DISPLAY *handle*:*attribute* command. The DISPLAY FIELD command accepts (and requires) any buffer qualifier that you need to uniquely specify the field.

A *widget* can include any legal 4GL reference to a user-interface widget, system handle, or procedure handle. You can qualify user-interface widget references as you do in the 4GL to specify the type of widget and its container, as necessary. An *attribute* can include any attribute associated with the specified *widget*. Note that you cannot query (and thereby invoke) the methods of a *widget*.

5.6.3 Modifying Data and Attribute Values

You can modify the value of any specified field, parameter, variable, or widget attribute that is accessible from the current procedure using the corresponding ASSIGN FIELD, ASSIGN PARAMETER, ASSIGN VARIABLE, ASSIGN *widget*:*attribute*, or ASSIGN *handle*:*attribute* command. The ASSIGN FIELD command accepts (and requires) any buffer qualifier that you need to uniquely specify the field.

A *widget* can include any legal 4GL reference to a user-interface widget, system handle, or procedure handle. You can qualify user-interface widget references as you do in the 4GL to specify the type of widget and its container, as necessary. An *attribute* can include any attribute associated with the specified *widget* and that is also writable at the stopping point of the current procedure.

Note that the value you assign must be a literal value that conforms to the data type of the data item or attribute that you are modifying. Expressions are not allowed.

5.7 Displaying Debugger and Session Information

You can display Debugger and session information using the commands listed in Table 5–4.

Table 5–4: Session Information Commands

Function	Command
Get on-line help on Debugger commands and functions.	HELP
List session settings.	SHOW BREAKS SHOW BUTTONS SHOW MACROS
Examine and navigate the Debugger context.	SHOW STACK UP DOWN
Determine the current transaction state.	SHOW TRANSACTION
Display procedure debug listings.	FILE

5.7.1 Getting On-line Help

The HELP command accesses on-line help for all Debugger commands. From the command help, you can also access most of the information available from the Help menu, which accesses the complete Debugger on-line help system. For more information, see Chapter 4, "Debugger Window."

You can also use the HELP command to get help directly on a specific command by entering HELP followed by the first one or two words of the desired command in the command panel. This example provides information on the LIST BUFFERS command:

```
HELP LIST BUFFERS
```

5.7.2 Displaying Debugger Session Settings

The SHOW BREAKS, SHOW BUTTONS, and SHOW MACROS commands list your current session breakpoints, buttons, and macros in order of creation (most recent last).

The SHOW BREAKS command also indicates the unique number of each breakpoint in the session to help you easily select breakpoints to cancel. This command lists all breakpoints, including those in inactive as well as active procedures. For an inactive procedure, the listed breakpoints may not be the effective breakpoints. Once the procedure is active in the Debugger context, the SHOW BREAKS command lists all its breakpoints as effective breakpoints.

Both the SHOW BUTTONS and SHOW MACROS commands list each button or macro with its definition fully expanded. That is, all macros included in the definition are expanded to their constituent Debugger commands and parameters. For more information on button and macro definitions, see the "Defining and Using Macros and Buttons" section.

5.7.3 Examining and Navigating in the Debugger Context

You can use the SHOW STACK, UP, and DOWN commands to examine and move around in the Debugger context, which includes the procedure stack accessible to the Debugger plus the executing procedure. These capabilities allow you to review your whole execution environment from the startup to the current executing procedure. By moving from one active procedure to another, you also focus on a new current procedure in the Debugger context. This allows you to examine and modify data that is accessible to that procedure. For more information on procedures and the Debugger context, see Chapter 1, "Introduction."

NOTE: An instance of a persistent procedure can be active in the Debugger context (and therefore made current) only when your application executes the main-line code, triggers, or internal procedures of the persistent procedure.

Examining the Procedure Call Stack

The SHOW STACK command displays a trace of the procedure call stack and executing procedure in the Debugger context. The procedure call stack contains a record of each procedure (*stack entry*) above the executing procedure that is waiting for a called procedure to return. The SHOW STACK command displays one line per stack entry, starting with the initial application procedure running in the session. Each line lists a procedure name and line number where the named procedure is waiting for the next procedure down to return. A *context arrow* points to the current procedure in the Debugger context. The default current procedure is the executing procedure (not on the stack) at the bottom of the list. If the executing procedure is the only procedure in the Debugger context, it is listed by itself.

NOTE: If the Debugger is running a procedure, and, at an interruption point, you execute another procedure with the RUN command, the current procedure is pushed on the call stack, and the newly executed procedure becomes the current procedure. This is true whether you run an inactive procedure or an active procedure recursively. The SHOW STACK command lists any such procedure "call" invoked from the Debugger as if it were invoked using the 4GL RUN statement from the procedure immediately above it. However, an additional message indicates that the procedure was started from the Debugger. For more information on running procedures from the Debugger, see the "Controlling Procedure Execution" section.

Navigating the Procedure Call Stack

You can enter the UP and DOWN commands to move up and down (*change focus*) in the Debugger context. Each focus change moves the context arrow up or down to the next procedure, which becomes the current procedure. Each time you change the current procedure, the Debugger displays the listing for the new current procedure. The listing indicates the breakpoints set for the current procedure, with the line pointer on the line where it is stopped. For a procedure on the call stack, this is the line where it is waiting for the next procedure down in the Debugger context to return. For the executing procedure, it is the current interruption point (the next line to execute).

Changing the current procedure with the UP and DOWN commands allows you to examine and modify data accessible to the new current procedure. Thus, the LIST and DISPLAY commands now reference data objects in the current procedure, whether or not it is the executing procedure.

However, the UP and DOWN commands do **not** change the course of procedure execution. (In other words, the execution status does not change.) Any Debugger control flow command you enter (for example, CONTINUE or NEXT) continues running the executing procedure from its current stopping point.

NOTE: If the procedure with prior focus was on the call stack (with its listing displayed), the debug listing for the currently executing procedure is not displayed unless you interrupt it with a breakpoint or break interrupt.

5.7.4 Determining the Current Transaction State

The SHOW TRANSACTION command indicates whether a transaction is active in the executing application. However, it only indicates the transaction state at the current stopping point in the executing procedure. If you move up and down the call stack, changing the current procedure, the reported transaction state does not change.

5.7.5 Displaying Debug Listings

The FILE command displays the debug listing for any procedure in the listing panel, whether or not it is active or part of the application you are running. If the listed procedure is not active in the Debugger context, the listing is displayed with all breakpoint indicators at specified lines, whether or not the lines are executable. However, when you run the procedure in the Debugger context, the breakpoint indicators adjust to show the effective breakpoints.

The listed procedure provides defaults for the Break and Run options of the Debug menu, whether or not it is the current procedure. It also specifies the default procedure for the BREAK and CANCEL BREAK commands. For more information, see the "Setting Breakpoints," "Removing Breakpoints," and "Continuing Procedure Execution" sections.

NOTE: The FILE command does **not** change focus in the Debugger context. The current procedure remains unaffected by the FILE command.

You can also enter the FILE command using the File→ Open option. Choosing this option opens a dialog box in which you can enter or select a procedure to list. When you choose the OK button, the Open option enters the FILE command to display the listing.

5.8 Defining and Using Macros and Buttons

You can define macros and buttons for the Debugger that allow you to enter one or more Debugger commands as a single command. A macro is identified by a unique name and is invoked like a command. A button is identified by a unique label in the button panel and, like other window or dialog buttons, is invoked by choosing it with the mouse.

5.8.1 Specifying Macro and Button Definitions

You can define both macros and buttons in much the same way, using the MACRO and BUTTON commands, as shown in the following syntax description:

SYNTAX

```
{ MACRO | BUTTON } identifier command [ ; command ] ...
```

identifier

The name of your macro or the label for your button. Both macro names and button labels are case sensitive. You cannot specify a Debugger command name as a macro name, but you **can** specify a Debugger command name as a button label. (That is, you cannot redefine a Debugger command as a macro, but you can label a button with a command name.) You can also enclose button labels in double quotes (". . .") to embed spaces anywhere in the label. If *identifier* specifies a previously defined macro name or button label, the current macro or button definition replaces the previous one.

command

Specifies any previously defined macro or Debugger command, except the MACRO or BUTTON commands themselves. (That is, you cannot define a new macro or button inside another macro or button definition.) The entire macro or button definition must be entered on one line. When you specify a previously defined macro in a new macro or button definition, the Debugger immediately replaces the previously defined macro with its component commands so that its definition becomes part of the new macro or button definition.

Immediately after you define a macro, it is available as a command or to include in other macro or button definitions. Immediately after you define a button, the Debugger visibly places the button in the last available place in the button panel, making it available for use.

Most macro and button definitions are relatively simple. For example, this command defines a macro and a button for the HELP command:

```
macro h help; button HELP help
```

However, several Debugger commands take parameters such as procedure names and line numbers. You can specify these parameters in both macro and button definitions using the Debugger variables listed in Table 5–5. When used in buttons, these variables take their values from the listing panel in the Debugger window; when used in macros, their values depend on whether the macro is used as a command or as part of a button definition.

Table 5–5: Debugger Variables

Variable	Usage	Value
&FILE	BUTTON	Set to the name of the procedure that is listed in the Debugger window.
	MACRO	When invoked as part of a button definition, set to the name of the procedure that is listed in the Debugger window. When invoked as a command, set to the corresponding value supplied on the command line.
&LINE	BUTTON	Set to the line number of the procedure line selected in the listing panel.
	MACRO	When invoked as part of a button definition, set to the line number of the procedure line selected in the listing panel. When invoked as a command, set to the corresponding numeric value supplied on the command line.
&TEXT	BUTTON	Set to the text selected in the listing panel.
	MACRO	When invoked as part of a button definition, set to the text selected in the listing panel. When invoked as a command, set to the corresponding value supplied on the command line.

5.8.2 Using Macros

To execute a macro definition, you can enter the macro name like any Debugger command, except that you must enter the name using the same letter case used in the definition. You must also enter the value for each Debugger variable that you use in the definition, specifying each value according to the variable's first order of appearance in the definition. Note that each variable can reference only one value per macro or button, but you can use each variable as many times as you want in the macro or button definition.

For example, suppose that you define the following macro:

```
macro SetAndGo break &file &line; run &file
```

To set a breakpoint on line 16 of the procedure `arrmenu.p` and run the procedure, you must enter the SetAndGo macro command:

```
SetAndGo arrmenu.p 16
```

Note that the &FILE and &TEXT variables provide the same capabilities in a macro definition when the macro is entered as a command. That is, they each take a text value as a parameter. For example, suppose you define the following SetAndDebug macro using the SetAndGo macro. This macro sets a breakpoint in a procedure, runs the procedure, and at the next breakpoint, executes the Debugger commands in the specified command file:

```
macro SetAndDebug SetAndGo; input &text
```

To set a breakpoint on line 16 of the procedure `arrmenu.p`, run the procedure, and execute the Debugger commands in `commands.txt`, you must enter the SetAndDebug macro command:

```
SetAndDebug arrmenu.p 16 commands.txt
```

When you use a macro in a button definition, there is no need to enter parameters for the macro. As described in Table 5–5, all macro variables take their values from the listing panel when included in a button definition.

5.8.3 Using Buttons

To execute a button definition, click on the corresponding button in the button panel. However, first ensure that values are selected in the listing panel for any Debugger variables in the button definition. (This includes any variables in macros that are part of the button definition.) For example, suppose you define a button using the SetAndGo macro defined in the "Using Macros" section.

```
button " Set and Test " SetAndGo; show stack
```

This button sets a breakpoint on the selected line of a procedure, runs the procedure, and displays the procedure call stack at the first breakpoint. To use this button with line 16 of `arrmenu.p`, you must perform the following steps:

1 ♦ Ensure that `arrmenu.p` is displayed in the listing panel using the FILE command. This provides the value for the &FILE variable, `arrmenu.p`.

2 ♦ Place the text cursor in the listing panel on line 16 of the procedure. This selects and provides the value for the &LINE variable, 16.

3 ♦ Click on the Set and Test button in the button panel. This sets the breakpoint on line 16 of `arrmenu.p`, runs the procedure, and displays the procedure call stack at the first breakpoint.

Specifying Button Parameters

In general, each Debugger variable in a button definition takes its value from the contents of the listing panel when you choose the button. You can use the techniques described in Table 5–6 to assign values to these variables in the listing panel.

Table 5–6: Variable Values in Buttons

Variable	To Assign the Value from the Listing Panel
&FILE	• Enter the FILE, UP, or DOWN command. The value is the name of the listed procedure. • Interrupt an application with a breakpoint or the break interrupt function. The value is the name of the interrupted procedure.
&LINE	• Click the mouse anywhere on a listing line, leaving the mouse cursor on the line. The value is the number of the selected listing line. • Use the mouse to select any text on the line (see &TEXT). The value is the number of the line where text is selected.
&TEXT	• Place the mouse pointer on the text, and double-click to select a space-delimited string. The value is the space-delimited string selected with the mouse. This technique does not work in Windows and does not work in Motif for strings surrounded by quotes or other punctuation; instead, use the following technique. • Drag the mouse pointer from the first to the last character of the selected string. The value is the string included in the mouse selection. Use this technique to include spaces or to omit surrounding quotes or other unwanted characters from the selection.

5.8.4 Removing Macros and Buttons

To remove a single macro or button during a debugging session, enter the CANCEL MACRO or CANCEL BUTTON command. You must specify the macro name or button label exactly as you defined it, including any quotes. You can also enter the CANCEL ALL command with the MACROS or BUTTONS option to remove all macros or buttons (respectively) from your session.

5.9 Logging a Debugging Session

The Debugger provides a logging function to record the history of a debugging session. This session history includes all Debugger commands entered in the command queue along with command responses displayed in the data panel. The Debugger stores this information in a file named `prodebug.log` or another file you specify. For more information on this file, see Chapter 6, “Debugger Files.”

5.9.1 Using the LOG and CANCEL LOG Commands

You can turn on Debugger logging any time the Debugger has control by entering the LOG command. Logging continues throughout the Debugging session until you turn it off by entering the CANCEL LOG command or exiting the Debugger. At any time, you can turn logging on again with the LOG command. If you already have logging turned on and enter the LOG command for a different file, the Debugger automatically turns off logging to the previous log file and continues logging to the new file.

Every time you restart logging, the Debugger replaces the previous session history with the current history starting at the point you enter the LOG command, unless you specify a different log filename. If you specify a different file, the new session history goes into the new log file with no effect on previous log files.

5.9.2 Automatic Session Logging

You can also turn on logging automatically when you run the Debugger by including the LOG command in your Debugger initial command file. For more information on the initial command file, see Chapter 6, “Debugger Files.”

5.10 Debugging Events

From the viewpoint of the Debugger, there are two kinds of events: nonvolatile events and volatile events. The triggers for *nonvolatile events* can always be interrupted by the Debugger. However, the triggers for *volatile events* can only be interrupted by the Debugger on certain deployment platforms. All database (DB) events are nonvolatile events. Nonvolatile user-interface (UI) events include most high-level widget events, all keyboard events, and all mouse events. Volatile UI events include:

- All direct manipulation events.
- The MENU-DROP high-level widget event.

The following sections describe techniques for debugging both nonvolatile and volatile events.

5.10.1 Nonvolatile Events

If your application uses a large number of DB and UI triggers, it might help to include DEBUGGER handle statements in the code to debug the application. Especially with UI events, the combinations of UI actions that cause them to fire are not always clear because the user has most of the control over when UI actions occur. If you also have cascading events (events firing within events, for example using the APPLY statement), the order of execution can be even harder to follow.

DEBUGGER Handle Methods

You can provide some built-in event debugging capability by invoking the SET–BREAK( ) method at the start of each trigger block. Once the Debugger is initialized, this method (with no arguments) sets a breakpoint on the first executable line after the line that invokes the SET–BREAK( ) method. This saves having to find the procedure names and line numbers for all the trigger blocks you might want to break on during a debugging session. You might also want to use the DISPLAY–MESSAGE( ) method to display data and attribute values as each trigger executes.

R-code Impacts

When the Debugger is not initialized, DEBUGGER handle statements have no functional affect and incur no more r-code performance or memory penalty than variable assignment statements. Thus, when you put your application into production, you can often leave these debugging statements in the source code without any noticeable affect on application or system performance.

5.10.2 Volatile Events

Volatile events occur whenever the Debugger and window system are in competition for the mouse pointer. If the Debugger is allowed to interrupt one of these events, it tries to grab the mouse pointer at the same time the window system is trying to process the event for the same mouse pointer. Thus, the window system cannot finish processing the event until the Debugger returns control to the application, and the Debugger cannot return control to the application until it gets control of the mouse pointer. The result is a system deadlock that you can break only by shutting down the window manager and restarting or, if necessary, rebooting.

Debugger Detection

To prevent this system deadlock, the Debugger detects when an event is volatile and does not allow any trigger code to be interrupted for the event. Instead, when it encounters a breakpoint during one of these events, it displays a message like this indicating the location of the breakpoint and that it is processing a volatile event:

```
Ignore breakpoint at USER-INTERFACE-TRIGGER ... (4346)
```

If the Debugger is about to complete a NEXT or STEP command in a volatile event trigger, it completes trigger execution and stops at the first available line after the trigger completes.

Volatile Events in Windows

To debug a volatile event in Windows, you can use the DISPLAY–MESSAGE( ) method of the DEBUGGER handle to display a message string to the data panel when the event trigger executes. Invoke the method from inside the trigger block or procedure that services the event. Using this technique allows you to display information about the event as it is triggered, including data and widget attribute values.

Volatile Events in Motif

Since your application runs in character mode, all UI events are effectively nonvolatile. This is because the application does not use the Motif window manager at all. Therefore, the Debugger never competes with the application for window manager events.

6

Debugger Files

During a debugging session, the Debugger directly or indirectly references several files, including:

- Initialization files to set initial parameters for a debugging session
- Command files to execute any combination of commands and macros
- Debug listing files derived from procedure files
- Procedure files, in source code or r-code form
- Log files to record a history of your debugging session

This chapter describes these files, how the Debugger uses them, and how you can interact with them to manage debugging sessions.

6.1 Debugger Initialization Files

Each time you start the Debugger, it reads one or more files to initialize itself. This initialization is performed differently in each graphical environment, but generally includes settings that perform the following actions at startup:

- Sizing and positioning the Debugger window on the display.
- Defining initial macros and buttons.
- Executing initial Debugger commands.
- Specifying the font size for the listing panel (Windows only).
- Specifying an alternative Debugger executable (Windows only).
- Specifying accelerator keys for Debugger menu options (OSF/Motif only).

Progress provides default initialization files in your Progress installation directory as specified for each window system. The settings provided in these files depend on your system environment. You can modify some of the default settings in these files using a text editor. The function of these settings and how you can modify them is described in the following sections for each graphical environment.

6.1.1 Windows Initialization

You can specify Debugger initialization parameters in Windows using the Registry or the `PROGRESS.INI` file provided with your Progress installation. Note that if you plan to make changes to the defaults, make them to a copy stored in your working directory.

Search Order for PROGRESS.INI

When the Debugger starts up, it searches either in the Registry or for a `PROGRESS.INI` file. For information about the search order for the `PROGRESS.INI` file, see the chapter on maintaining user environments in the *Progress Client Deployment Guide*.

PROGRESS.INI Sections for Debugger Initialization

This file provides three sections for Debugger initialization, labeled with the following headings:

- [Debug-Init]
- [Debug-Macros]
- [Debug-Buttons]

CAUTION: There are certain parameters in the PROGRESS.INI file that you must never change in a text editor. These are labeled **Do Not Change** in the following descriptions. You can set them reliably only from the Debugger using the Options→ Save Settings menu option. Any attempt to change these parameters outside of the Debugger can result in loss of data.

[Debug-Init] Section

This section includes several parameters that specify basic initialization options for the Debugger. They determine how the Debugger and its window start up in your environment, and include:

- **Location (Do Not Change)** — Specifies Debugger window location and state. This parameter is described more fully in your installation PROGRESS.INI file. The Debugger resets this parameter when you save the Debugger window settings. For more information, see Chapter 4, "Debugger Window."

- **Pane0, Pane1, Pane2, and Pane3 (Do Not Change)** — Specifies panel characteristics. This parameter is described more fully in your installation PROGRESS.INI file. The Debugger resets this parameter when you save the Debugger window settings. For more information, see Chapter 4, "Debugger Window."

- **SourceFontHeight =** ***font-size*** — Specifies the window's relative point size for the font used in the listing panel.

- **DebuggerName =** ***filename*** — Specifies the name of an alternative executable for the Debugger process. Ordinarily, you have no reason to change this. If the parameter is not specified, Progress uses the default Debugger executable, PRODEBUG.EXE, in the executable installation directory.

- **DebuggerInput =** ***filename*** — Specifies the name of an initial command file that the Debugger executes at startup. This is a portable way to set up initial macros and buttons because it uses the same syntax as Debugger commands. It is also the only method you can use to run Debugger commands at initialization. The specified file executes before either of the [Debug-Macros] or [Debug-Buttons] sections described below. A common use of this file is to turn on logging at the start of a debugging session with the LOG command. The file format is exactly the same as any command file that you can enter using the INPUT command. For more information, see the "Debugger Command Files" section in this chapter.

[Debug-Macros] Section

This section includes a list of parameters that define initial macros. The macros defined in this section take effect after any defined in the initial command file specified by the DebuggerInput parameter in the [Debug-Init] section. Specify these parameters in the following form:

SYNTAX

```
Macron = macro-name , command [ ; command ] ...
```

Each macro parameter is identified by an integer (*n*) that must start at 0 and increment by 1 for each additional macro definition. The name of the macro (*macro-name*) can include any alphanumeric characters, and is case sensitive. Each command (*command*) can specify any Debugger command, except the BUTTON or MACRO commands, or a previously defined macro command. The maximum length of each macro definition to the right of the equal sign (=) is 255 characters.

In general, you can define an initial macro by using Debugger variables and following the same rules as any macro definition entered from the Debugger. For more information, see the MACRO command reference entry in Appendix A, "Command Reference," and the "Defining and Using Macros and Buttons" section in Chapter 5, "Running a Debugging Session."

NOTE: If you define and save macros during a debugging session, the Debugger stores the macro definitions in this section of your `PROGRESS.INI` file. For more information on saving macro definitions, see Chapter 4, "Debugger Window."

[Debug-Buttons] Section

This section includes a list of parameters that define initial buttons. The buttons defined in the [Debug-Buttons] section take effect after any buttons defined in the initial command file specified by the DebuggerInput parameter in the [Debug-Init] section. Specify these parameters in the following form:

SYNTAX

```
Buttonn = button-label , command [ ; command ] ...
```

Each button parameter is identified by an integer (*n*) that must start at 0 and increment by 1 for each additional button definition. The label for the button (*button-label*) can include any alphanumeric characters, including embedded spaces, and is case sensitive. Unlike button labels defined using the BUTTON command, no quotes are allowed in these parameter definitions. Each command (*command*) can specify any Debugger command, except the BUTTON or MACRO commands, or a previously defined macro command. The maximum length of each button definition to the right of the equal sign (=) is 255 characters.

Except for quoting the button label (for example, to embed spaces), you can define an initial button by using Debugger variables and following the same rules as any button definition entered from the Debugger. For more information, see the BUTTON Command in Appendix A, "Command Reference," and the "Defining and Using Macros and Buttons" section in Chapter 5, "Running a Debugging Session."

NOTE: If you define and save buttons during a debugging session, the Debugger stores the button definitions in this section of your `PROGRESS.INI` file. For more information on saving button definitions, see Chapter 4, "Debugger Window."

6.1.2 Motif Initialization

In Motif, Progress initializes the Debugger at startup using a combination of your resource definition file (usually `.Xdefaults`) and an initial command file.

Initial Command File

You can use an initial command file to define initial macros, buttons, and breakpoints, and to run any other Debugger commands at startup. A common use of this file is to turn on logging at the start of a debugging session using the LOG command. The initial command file runs exactly like an INPUT command file executed at the start of a debugging session.

The Debugger first looks for a file named `.prodebug` in your home directory and executes it as your initial command file. If it cannot find that file, it looks for a file named `system-prodebug` in your `$DLC` directory. Progress provides a default `system-prodebug` file with your installation. For more information on command files, see the"Debugger Command Files" section.

NOTE: If you define buttons and macros in a debugging session and save them using the Save Settings option of the Options menu, the Debugger stores the definitions in your initial command file. For more information on saving button and macro definitions, see Chapter 4, "Debugger Window."

Resource Definition File Settings for Window Dimensions

Your resource definition file allows you to define panel dimensions for your Debugger window. To specify panel dimensions for each panel, set the following parameters to the desired number of pixels:

- **prodebug*listing.width** — Listing panel width.
- **prodebug*listing.height** — Listing panel height.
- **prodebug*button.height** — Button panel height.
- **prodebug*command.height** — Command panel height.
- **prodebug*data.height** — Data panel height.

NOTE: All panels conform to the width specified for the listing panel.

For example, the following parameters define an initial Debugger window that fits conveniently on a Sun workstation:

```
prodebug*listing.width:    540
prodebug*listing.height:   350
prodebug*button.height:   30
prodebug*command.height:  170
prodebug*data.height:     85
```

NOTE: Once the window is displayed, you can change its dimensions with the mouse. You can then save its current dimensions using the Save Settings option of the Options menu. The Debugger saves them as the new default dimensions in your resource definition file. For more information, see Chapter 4, “Debugger Window.”

Resource Definition File Settings to Define Accelerator Keys

Your resource definition file allows you to define accelerator keys for Debugger menus. To define accelerator keys for each menu option, use parameters with this format:

SYNTAX

```
prodebug*Menu-option.accelerator:     X-translation-string
prodebug*Menu-option.acceleratorText: accelerator-label
```

For these parameters, *Menu-option* is the menu option label exactly as it appears in the Debugger menu. The *X-translation-string* is the standard X translation string that specifies the key or key combination you want to assign. The *accelerator-label* is the string you want displayed to the right of the menu option that identifies your accelerator.

For example, the following parameters assign the **CTRL-C** key combination to the Debugger break interrupt function invoked by the Interrupt option of the Debug menu:

```
prodebug*Interrupt.accelerator:           Ctrl <Key>C
prodebug*Interrupt.acceleratorText:       Ctrl+C
```

For more information on accelerator parameters and X translation strings, see your *X-Window System User's Guide — OSF/Motif Edition.*

6.2 Debugger Command Files

A Debugger command file is any text file that contains only valid Debugger commands. You can execute a command file using the Debugger INPUT command. Each line of the file can contain any command or semicolon-separated (;) list of commands that you enter in the command panel of the Debugger window. If you include INPUT commands in a command file, they cannot specify the command file in which they are included. That is, a command file cannot include cyclic references.

In Windows, you can use any command file as an initial command file by assigning its name to the DebuggerInput parameter of the `PROGRESS.INI` file. For more information on using initial command files in Windows, see the "Debugger Initialization Files" section.

In Motif, you can use any command file as an initial command file by copying it to `.prodebug` me directory. For more information on using initial command files in Motif, see the "Debugger Initialization Files" section.

6.3 Debug Listing (.dbg) Files

A debug listing file contains a line-numbered listing of a procedure with all preprocessor directives expanded and all include files inserted. The Debugger displays this listing in the listing panel to indicate breakpoints and the next line to execute in the current procedure. You can also display the debug listing for any procedure using the FILE command.

6.3.1 Listing Creation

You can generate a debug listing in one of two ways:

- Automatically — When you run a procedure or display a procedure using the FILE command during a debugging session.
- Manually — When you compile a procedure explicitly using the COMPILE statement from the 4GL or the Application Compiler from the ADE Desktop.

Automatic Listing Generation

Automatic listing generation is appropriate for most debugging situations where you are working with new source procedures that you have not yet compiled. Automatic generation creates temporary listing files that Progress removes when you end your session and exit to the operating system. These temporary listing files have eight-character alphanumeric names with a `dbg` extension. Progress stores temporary debug listing files in either your current working directory or the directory specified by the Temporary Directory (-T) startup parameter. For more information on how and when Progress automatically generates debug listings, see the "Procedure Files and Listing Generation" section.

Manual Listing Generation

Manual listing generation can save significant time in a debugging session, especially when debugging very large procedures. For more information on how to generate debug listings with the COMPILE statement, see the *Progress Language Reference*, and for information on the Application Compiler, see the *Progress Basic Development Tools* manual.

6.3.2 Listing Deletion

There might be occasions when a session terminates leaving temporary debug listing files in your directory. This can occur during a system crash or any other abnormal termination of your application. You can safely delete these temporary debug listings from your directory since the Debugger creates them automatically.

6.3.3 Listing Format

The format of a debug listing file is similar to standard Progress compiler listings. Like compiler listings, debug listings expand all preprocessor, include file, and parameter references. The main difference is that the text in debug listings contains only numbered source code lines, without headings, page formatting, or any of the other internal data information (such as buffer and frame usage) usually provided by compiler listings.

6.4 Procedure Files and Listing Generation

You can debug procedure files in both source (`.p`, `.w`) and r-code (`.r`) form. You can set breakpoints for procedure files, whether or not they exist, and whether or not they are currently reachable from any application that you run.

NOTE: Procedure filenames for source code can have any or no extension. The listed extensions (`.p`, `.w`) represent common Progress conventions on many systems. However, r-code files must have the identifying extension established for your operating system, usually `.r`. Progress automatically generates r-code files with the correct extension during compilation.

Progress makes every attempt to provide the debug listing for the procedure that you are debugging. If necessary, Progress generates the debug listing automatically ("on the fly"). Any automatic listing generation creates a temporary file that is deleted at the end of your session. For more information on temporary debug listings, see the "Debug Listing (.dbg) Files" section.

The method Progress uses to provide a debug listing depends on the form of the procedure that Progress has available to execute: the r-code or the source code.

6.4.1 Debug Listing Generation from R-code

If Progress runs the r-code version, Progress tries to locate an existing debug listing for it. The r-code contains references to all associated source, include, and debug listing files, including their cyclic redundancy check (CRC) values. When your application runs an r-code file, Progress attempts to provide the debug listing using this method:

1. If the r-code file references a debug listing whose path prefix is compatible with the current operating system, Progress uses the path prefix together with the current PROPATH to locate the file and ensure that the r-code and listing CRC values match.

2. If it cannot find the listing file or the CRC values do not match, Progress tries to regenerate the debug listing using the source and include file references stored in the r-code, and saves the new listing as a temporary file.

3. If Step 2 fails because a source or include file cannot be found, Progress uses any debug listing found in Step 1 with a bad CRC.

4. If the debug listing file cannot be found or created in any form, Progress displays a warning message and continues executing the r-code.

For all file search and creation attempts in the previous steps, Progress verifies each source, include, and debug listing file with the corresponding CRC value stored in the r-code. If the CRC value does not match a located file, Progress displays a warning message and continues using the file with a bad CRC value.

6.4.2 Debug Listing Generation from Source Code

If all you have is the source code for a procedure, Progress automatically generates the debug listing during compilation and saves it as a temporary file. Progress generates temporary debug listings from source both when you execute a procedure for a debugging session and when you just display the procedure in the listing panel using the FILE command.

6.4.3 Hidden Procedure Files

The Debugger ignores any procedure file (*hidden procedure*) whose filename begins with an underscore (_). Although you can execute a hidden procedure, it is hidden from the Debugger in several ways:

- A hidden procedure never enters the Debugger context and does not honor any breakpoints that you set in it.
- The Debugger never displays the debug listing for a hidden procedure.
- While a hidden procedure executes, all procedures above it in the procedure call stack are also hidden from the Debugger as well, whether or not they are hidden procedures themselves.

The Progress ADE tools that invoke the Debugger use hidden procedures to keep the tools themselves invisible to the Debugger. You can use the same technique when writing your own development tools that invoke the Debugger.

6.5 Debugger Log Files

After you turn on logging with the LOG command, the Debugger records a history of your debugging session in either a log file named `prodebug.log` or another file that you specify. By default, it creates the log file in your temporary directory, that is, your current working directory or the directory specified with the Temporary Directory (-T) startup parameter. If you specify an absolute pathname, the log file is created with the specified path. If you specify a relative pathname, the file is created relative to the temporary directory.

6.5.1 Log File Contents

The data recorded in a log file includes all of the following information from the point in a debugging session when logging is turned on for the file:

- Commands entered by any means, whether directly in the command panel or indirectly using command files, macros, buttons, and menu options.
- Information displayed in the data panel in response to each command.

6.5.2 Log File As a Command File

A Debugger log file is not suitable for input as a command file because it contains both Debugger commands and responses to them. To turn it into a Debugger command file, you must use a text editor to strip the file of all command responses, leaving the commands only. You can then rerun the part of a debugging session recorded in the original log file by running the new command file using the Debugger INPUT command.

7

DEBUGGER and SESSION System Handles

The DEBUGGER system handle has attributes and methods that allow you to start and control a debugging session from the 4GL. It is primarily intended for experienced Progress programmers who write their own Progress development tools and want to initialize and control Debugger interaction from these tools. For example, the Progress Application Development Environment (ADE) uses the DEBUGGER handle to invoke and manage the Debugger from ADE tools such as the Procedure Editor and ADE Desktop. The DEBUGGER handle can also aid in tracing program flow through the complex code paths typical of event-driven applications.

The SESSION system handle has an attribute that allows you to specify whether you want an alert box to include a Help button that provides access to a Stack Trace dialog box.

The following sections describe:

- The DEBUGGER handle attributes and methods
- Invoking the Debugger using the DEBUGGER handle
- Using miscellaneous attributes and methods during a debugging session
- Using the SESSION handle during a debugging session

7.1 DEBUGGER Handle Attributes and Methods

The DEBUGGER handle attributes and methods provide a limited set of capabilities primarily designed to start up and clean up after a debugging session from the 4GL. A few additional capabilities duplicate some basic Debugger commands and provide a means of performing certain debugging tasks that are more convenient from the 4GL.

Table 7–1 lists the function of each DEBUGGER handle attribute.

Table 7–1: DEBUGGER Handle Attributes

Attribute	Function
TYPE	Returns the widget type of the Debugger, the standard value for all system handles, “PSEUDO-WIDGET”.
VISIBLE	Displays or removes the Debugger window on the screen.

Table 7–2 lists the function of each DEBUGGER handle method.

Table 7–2: DEBUGGER Handle Methods *(1 of 2)*

Method	Function
CANCEL-BREAK( )	Removes a specified breakpoint. Provides the function of the Debugger CANCEL BREAK command.
CLEAR( )	Cancels all breakpoints and logging; clears the listing, command, and data panels; and otherwise returns the Debugger to the state it was in immediately after initialization.
DEBUG( )	Initializes and immediately gives control to the Debugger in stand-alone mode while the invoking procedure blocks.
DISPLAY-MESSAGE( )	Displays a character string in the data panel of the Debugger window.

Table 7–2: DEBUGGER Handle Methods *(2 of 2)*

Method	Function
INITIATE( )	Initializes but does not immediately give control to the Debugger. When the invoking procedure encounters a breakpoint, it gives control to the Debugger in application mode and becomes the current procedure in the Debugger context.
SET-BREAK( )	Sets a specified breakpoint. Provides the function of the Debugger BREAK command.

NOTE: For information on Progress attributes and methods, see the *Progress Programming Handbook*.

To use any of these attributes or methods within a procedure, except DEBUG( ) or INITIATE( ), you must first initialize the Debugger. That is, you must perform one of the following actions:

- Start the Debugger from the operating system or from the 4GL using the DEBUG( ) method, and then run the procedure from the Debugger using the RUN command.
- Invoke the INITIATE( ) method. Typically, you then set an initial breakpoint to start up the Debugger using the SET-BREAK( ) method.

7.2 Starting the Debugger from the 4GL

Any technique you use to run the Debugger from the 4GL runs the Debugger in either application or stand-alone mode. (For an introduction to application and stand-alone mode, see Chapter 1, "Introduction.") Each technique allows your application to interact with the Debugger context in a different way, allowing more or less control of the Debugger from your invoking application. These techniques include:

- **Application mode** — Executes the INITIATE( ) and SET-BREAK( ) methods in the invoking application. At the first breakpoint specified by a SET-BREAK( ) method, the Debugger runs and takes control with the **invoking** procedure as the initial procedure in the Debugger context. In effect, the invoking procedure debugs itself. You can also use this technique to implement a Debugger tool that debugs a specified subprocedure under 4GL control, and optionally make the invoking procedure invisible to the Debugger context. This is the technique used by the Procedure Editor and AppBuilder to implement the Debug option of the Compile menu.

- **Stand-alone mode (as a tool of your application)**— Executes the DEBUG( ) method in the invoking procedure. The invoking procedure blocks until the Debugger returns (exits). In effect, the Debugger runs as a subroutine or tool of the invoking procedure and can only debug procedures that you run from the Debugger. In other words, the invoking procedure is **not** part of the Debugger context. This is the technique used by the ADE Desktop to invoke the Debugger.

- **Stand-alone mode (from the Procedure Editor)**— Executes the DEBUG( ) method as part of a two-line procedure in the Procedure Editor. This technique starts the Debugger directly from an Editor buffer without reference to any procedure. In effect, the Debugger runs as a stand-alone tool in the Editor and can only debug procedures that you start from the Debugger.

7.2.1 Starting in Application Mode

This technique is useful both for debugging complex code paths in the invoking application and for running the Debugger as a tool from one procedure to debug other procedures. It is especially helpful in large applications under development where the line numbers change constantly and you want to follow the code paths in a consistent manner—especially in an event-driven application with a large number of triggers. When implementing a Debugger tool in application mode, you can also specify a subprocedure for the Debugger to use as its initial executing procedure.

Debugging the Invoking Procedure

To start the Debugger from the 4GL so that the invoking procedure becomes the initial executing procedure, follow these steps:

1 ♦ Add a DEFINE VARIABLE statement that defines a logical variable you can use to assign the return value for the DEBUGGER handle.

2 ♦ Add a DEBUGGER handle statement that invokes the INITIATE( ) method before the point where you want to begin debugging the invoking procedure. This initializes the Debugger, but does not immediately start it.

3 ♦ Add a DEBUGGER handle statement that invokes the SET-BREAK( ) method to set at least one breakpoint that occurs after the statement where you set it. The SET-BREAK( ) method that sets the breakpoint must also execute after the statement that invokes the INITIATE( ) method in Step 2.

4 ♦ Run the invoking procedure.

For example, in the following listing fragment, the Debugger is initialized on line 5, and the procedure stops at a breakpoint on line 13. The Debugger takes control at this point. From here, you can continue executing the invoking procedure under Debugger control, stopping at and continuing from all breakpoints:

```
        1   DEFINE VARIABLE procname
        2       AS CHARACTER EXTENT 6.
        3   DEFINE VARIABLE Selection
        4       AS INTEGER FORMAT "9".
        5   DEFINE VARIABLE debug AS LOGICAL. debug=DEBUGGER:INITIATE().
        6   /* Initialize the list of procedure names to be run */
        7   procname[1] = "custedit.p".
        8   procname[2] = "custrpt.p".
        9   procname[3] = "ordedit.p".
       10   procname[4] = "ordrpt.p".
       11   procname[5] = "itemedit.p".
       12   procname[6] = "itemrpt.p". debug=DEBUGGER:SET-BREAK().
 * =>  13   DISPLAY "S A L E S   O R D E R   S Y S T E M" SKIP(1)
       14       WITH COLUMN 18 NO-BOX.
       15
                  .
                  .
                  .
```

NOTE: In this example, the DEBUGGER handle statements share a line with existing code (line 5 is originally blank). This preserves known application line numbers for other breakpoints you plan to set after the debugging session begins.

When you exit the Debugger, it returns to the invoking procedure, which continues execution from its current stopping (or breaking) point as if the Debugger were never initialized. If you exit the procedure before exiting the Debugger, the Debugger continues running until you either exit the procedure startup environment or explicitly exit the Debugger. When the invoking procedure exits, it returns to its startup environment.

Debugging a Specified Subprocedure

To use the same technique to debug a subprocedure called from the invoking procedure, follow these steps:

1 ♦ Add a DEFINE VARIABLE statement that defines a logical variable you can use to assign the return value for the DEBUGGER.

2 ♦ Add a DEBUGGER handle statement that invokes the INITIATE( ) method before the point where you want to begin debugging the called subprocedure. This initializes the Debugger, but does not immediately start it.

3 ♦ Add a DEBUGGER handle statement that invokes the SET-BREAK( ) method to set a breakpoint at the first executable line of the subprocedure you want to debug. This statement must execute after the statement that invokes the INITIATE( ) method in Step 2, but before the statement that calls the subprocedure.

4 ♦ Run the invoking procedure.

The following listing fragment prompts the user to enter the name (`procname`) of a procedure to debug (line 30). It then initializes the Debugger (line 33) and sets a breakpoint on the first executable line of `procname` (line 34). When the invoking procedure calls `procname` (line 35), `procname` breaks immediately on its first executable line and gives control to the Debugger. The Debugger then displays the listing for `procname` in the Debugger window as the current procedure:

```
 1   DEFINE VARIABLE debug    AS LOGICAL.
 2   DEFINE VARIABLE procname AS CHARACTER INITIAL "".
           .
           .
           .
30   SET procname LABEL "Enter name of procedure to debug" DEBLANK
31       WITH SIDE-LABELS.
32   IF procname <> "" THEN DO:
33       debug = DEBUGGER:INITIATE().
34       debug = DEBUGGER:SET-BREAK(procname).
35       RUN VALUE(procname).
36   END.
           .
           .
           .
```

NOTE: Using this technique, the invoking procedure executes in the Debugger context. Thus, if you enter a SHOW STACK command, the invoking procedure appears on the call stack. You can set breakpoints in the invoking procedure and break into it with the Debugger. In this type of application, you might not want the invoking procedure running in the Debugger context like this. You can prevent the invoking procedure from entering the Debugger context by prepending its procedure name with an underscore (_). In this case, the Debugger ignores the invoking procedure completely, and it never appears on the call stack.

Although this is a simple example, it illustrates the basic elements required to build a Progress development tool that can start debugging sessions for specified application procedures.

7.2.2 Starting in Stand-alone Mode

This technique is useful for running the Debugger as a tool started from one application to debug other applications. The invoking procedure runs the Debugger much like a subroutine, but does not specify any procedure for the Debugger to run as its initial current procedure. You must run all procedures you want to debug from the Debugger after it starts using the RUN command.

Invoking the Debugger As a Tool of Your Application

To start the Debugger from the 4GL while blocking the invoking procedure, follow these steps:

1 ♦ Add a DEFINE VARIABLE statement that defines a logical variable you can use to assign the return value for the DEBUGGER handle.

2 ♦ Add a DEBUGGER handle statement that invokes the DEBUG( ) method at the point where you want to start the Debugger.

3 ♦ Run the invoking procedure.

In the following example, the procedure presents six execution options, each of which invokes a different procedure. However, if you choose an invalid option, the procedure invokes the DEBUG( ) method and blocks at the following statement (line 29) while the Debugger runs:

```
        1   DEFINE VARIABLE debug AS LOGICAL.
        2   DEFINE VARIABLE procname AS CHARACTER EXTENT 6.
        3   DEFINE VARIABLE Selection AS INTEGER FORMAT "9".
        4
        5   /* Initialize the list of procedure names to be run */
        6   procname[1] = "custedit.p".
        7   procname[2] = "custrpt.p".
        8   procname[3] = "ordedit.p".
        9   procname[4] = "ordrpt.p".
       10   procname[5] = "itemedit.p".
       11   procname[6] = "itemrpt.p".
       12
       13   DISPLAY "S A L E S   O R D E R   S Y S T E M" SKIP(1)
       14       WITH COLUMN 18 NO-BOX.
       15
       16   REPEAT WITH SIDE-LABELS COLUMN 13 TITLE "M A I N   M E N U":
       17       FORM
       18         SKIP(2) "          Enter/Display/Update     List All"
       19         SKIP(1) "Customers          1                   2   "
       20         SKIP(1) "Orders             3                   4   "
       21         SKIP(1) "Items              5                   6   "
       22         SKIP(2) "           Press END KEY to exit           ".
       23       UPDATE SKIP(1) Selection AUTO-RETURN.
       24       HIDE.
       25       IF Selection GE 1 and Selection LE 6
       26          THEN RUN VALUE(procname[Selection]).
       27       ELSE DO:
       28          debug = DEBUGGER:DEBUG().
  =>   29          MESSAGE  "Incorrect selection. Please try again.".
       30       END.
       31   END.
```

When the Debugger first starts, it lists the invoking procedure, with the line pointer on the line that contains the blocked statement. Entering a control flow command (such as NEXT) to execute the blocked statement (the MESSAGE statement in the example) has no effect because the invoking procedure is **not** running in the Debugger context.

As a result, while the invoking procedure waits on the blocked statement, the Debugger runs on its own allowing you to control any **other** procedure that you start from it.

NOTE: Typically, you do not want to provide any indication of the invoking application in the Debugger window for this type of application. You can suppress the listing of the invoking procedure in the Debugger window by prepending the procedure name with an underscore (_). In this case, the Debugger starts up with a clear listing panel, ignoring the invoking procedure completely.

When you exit the Debugger, it returns control to the invoking procedure, which continues execution from the blocked statement (line 29 in the example). In stand-alone mode, you cannot otherwise access the invoking procedure until the Debugger returns control.

Starting a Stand-alone Debugger from the Procedure Editor

This technique is an alternative to running the Debugger from the Procedure Editor using the Compile menu. When you start the Debugger using the Compile menu, the Editor starts the Debugger in application mode, passing it the current buffer as the initial executing procedure. (For more information, see Chapter 3, "Starting and Stopping the Debugger.") This technique starts the Debugger directly from the Editor in stand-alone mode by invoking the DEBUG( ) method as part of a two-line procedure:

```
DEFINE VARIABLE debug AS LOGICAL.
debug = DEBUGGER:DEBUG().
```

When you run this procedure, the Debugger starts and takes control immediately, without any current procedure in the Debugger context. You must run all application procedures from the Debugger using the RUN command. When an application completes execution, it returns to the Debugger, and when you exit the Debugger, it returns to the Editor. This is useful when you want to work from the Editor and debug several different applications without leaving the Debugger.

7.3 Using the DEBUGGER Handle During a Debugging Session

For information on using all the DEBUGGER handle attributes and methods, see the DEBUGGER Handle reference entry in the *Progress Programming Handbook*.

7.4 Using the SESSION Handle with a Debugging Session

If you want alert boxes to include a Help button that provides access to a Stack Trace dialog box, use the SESSION handle and set the DEBUG-ALERT attribute to TRUE. The following shows sample code to use if you want alert boxes to include a Help button:

```
SESSION:DEBUG-ALERT = TRUE.
```

NOTE: You can also enable this feature when you start a Progress client using the `-debugalert` client startup parameter

Figure 7–1 shows an alert box with a Help button:

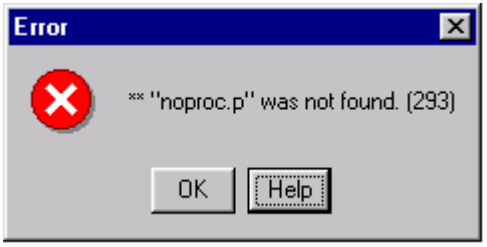


Figure 7–1: Alert box with Stack Trace Help Button

When you choose the Help button, a Stack Trace dialog box opens. This dialog box displays the 4GL stack trace for the currently executing Progress session. The stack trace displays procedures beginning with the root procedure as the first row and the procedure called (that is, the procedure that generated the alert box) as the last row.

Figure 7–2 shows an example of a Stack Trace dialog box:

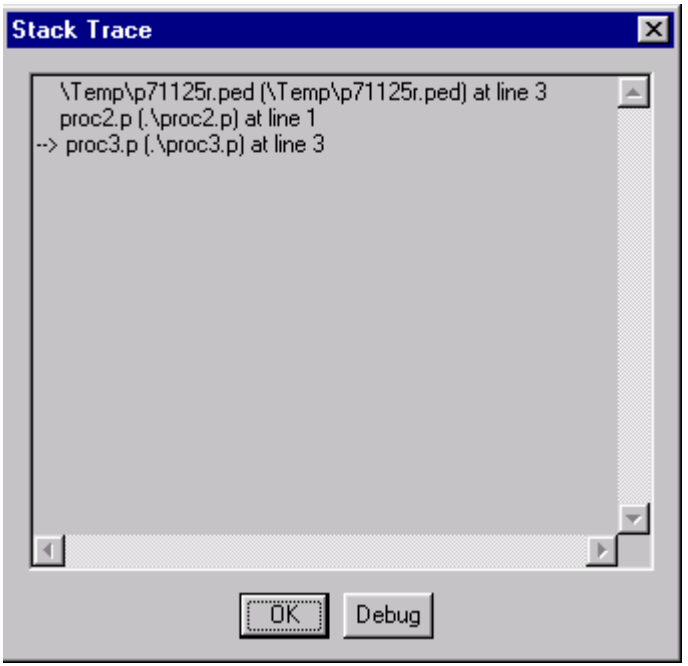


Figure 7–2: Stack Trace Dialog Box

To start the Debugger from the Stack Trace dialog box, do the following:

1 ♦ Choose the Help button in the Alert dialog box to open the Stack Trace dialog box.

2 ♦ Choose the Debug button in the Stack Trace dialog box. The Stack Trace dialog box closes, and you return to the initial Alert dialog box.

3 ♦ Choose OK in the Alert dialog box to start the Debugger.

NOTE: If you choose the OK button in the Stack Trace dialog box instead of the Debug button, you return to the Alert dialog box without starting the Debugger. If you choose the OK in the Alert dialog box at this point, you return to the editor from which you ran your code without starting the Debugger.

A

Command Reference

This appendix contains reference entries for the Debugger commands that you can enter using the Debugger window and command files. They are listed in alphabetical order.

There are a number of Debugger features that are referenced, but not fully explained, in these entries. Among the most important of these are:

- Techniques for starting the Debugger
- Break interrupt function
- Debugger window and its features
- How to run a debugging session and interact with an application
- Files accessed by the Debugger
- DEBUGGER system handle and its attributes

For more information on all of these topics, see the earlier chapters in this guide. For additional information on the DEBUGGER system handle, see the DEBUGGER Handle reference entry in the *Progress Language Reference*.

ASSIGN Command

Modifies the value of a specified variable, field, parameter, or attribute in the current procedure.

SYNTAX

```
ASSIGN [ VARIABLE | FIELD | PARAMETER ] name = constant
```

```
ASSIGN handle :attribute = constant
```

```
ASSIGN widget-name :attribute
  [ IN FRAME frame-name ] = constant
```

```
ASSIGN { FRAME | MENU } widget-name :attribute = constant
```

```
ASSIGN
  { SUB-MENU | MENU-ITEM } widget-name :attribute
  [ IN { MENU | SUB-MENU } menu-name ] = constant
```

PARAMETERS

[VARIABLE | FIELD | PARAMETER]

Specifies the type of data object you want to modify. If you do not specify this option, the Debugger attempts to resolve *name* for all object types and reports any ambiguity in the data panel.

name

The name of a variable, field, or parameter defined in the current procedure. If *name* specifies a field, it must also specify any necessary buffer qualifier that you need to uniquely identify the field.

constant

A literal value that conforms to the data type of the specified data object.

handle

The name of a variable, field, parameter, or system handle in the current procedure that contains a valid HANDLE, WIDGET-HANDLE, or COM-HANDLE value.

attribute

The name of an attribute of the specified widget or the name of a property of a specified com object.

widget-name

The name of a static widget defined in the current procedure.

[IN FRAME *frame-name* **]**

Specifies the name of the frame (*frame-name*) that contains the specified field-level widget. If you do not specify this option, the widget in the most recently defined frame is used.

{ FRAME **|** MENU **|** SUB-MENU **|** MENU-ITEM **}** *widget-name*

Specifies the type of widget for a frame, menu, submenu, or menu item widget (respectively) whose attribute you want to assign. If you do not specify a widget type for these widgets, the Debugger reports an error in the data panel. Note that you do not use a widget type to specify a field-level widget.

[IN **{** MENU **|** SUB-MENU **}** *menu-name* **]**

Specifies the name of the menu or submenu (*menu-name*) that contains the specified submenu or menu item widget. If you do not specify this option, the widget from the most recently defined menu or from the first submenu of the menu tree is used.

NOTES

- If no prior transaction is active, an ASSIGN command starts and ends a custom transaction for itself exactly like the following transaction block:

```
blk:
DO TRANSACTION ON ERROR UNDO blk, LEAVE blk:
    ASSIGN name = constant.
END.
```

- If you are assigning to a database field, *name* can specify any field in an accessible record buffer, whether or not the current procedure references the field.
- If *name* is a database field that has ASSIGN triggers defined for it, those ASSIGN triggers execute; if the table to which this field belongs has WRITE triggers, the WRITE triggers execute according to the current transaction.
- If any database trigger fires with breakpoints set in it, the Debugger receives control first at each breakpoint and once more after all triggers have completed.
- You cannot assign values to schema fields.
- To modify data objects accessible to a particular active procedure, use the UP or DOWN command to give it focus as the current procedure in the Debugger context.
- You can assign a value to an attribute at any stopping point in the current procedure as long as the attribute is writable from the 4GL at that point.
- You cannot reference more than one com object property name or COM-HANDLE value in a single ASSIGN command.

SEE ALSO

DISPLAY Command, LIST BUFFERS Command, LIST PARAMETERS Command,LIST VARIABLES Command, SHOW TRANSACTION Command

BREAK Command

Sets a breakpoint on an executable line in a procedure.

SYNTAX

```
BREAK { proc-name [ line-no ] | line-no }
```

PARAMETERS

proc-name

The name of a Progress procedure.

line-no

An integer that specifies a line number in the listed or specified procedure.

NOTES

- When you set a breakpoint, the Debugger echoes the setting in the data panel, including the ordinal breakpoint number, procedure name, and line number. If the procedure is displayed in the listing panel, the breakpoint indicator appears next to the corresponding line.
- If you do not specify a procedure name, the breakpoint is set on the specified line in the procedure displayed in the listing panel.
- If you specify only a procedure name, the breakpoint is set on the first executable line if the procedure is active in the Debugger context, and on just the procedure name if the procedure is inactive. When the inactive procedure becomes active, the breakpoint resolves to the first executable line.
- If you specify a line number for an active procedure in the Debugger context, the breakpoint is set on the first executable line on or after the specified line. If there is no executable line on or after the specified line, the breakpoint is set on the last executable line. For example, if you enter BREAK 10 for an active procedure displayed with the following lines, the breakpoint is actually set on line 12:

```
10   /* Set Startup Temperature */
11   DEFINE VARIABLE InitialTemp AS INTEGER.
12   ASSIGN InitialTemp = 32.
```

- If you specify a line number (*line-no*) for an inactive procedure, the breakpoint is set on the specified line. When the procedure becomes active in the Debugger context, the specified breakpoint adjusts to the corresponding effective breakpoint line.

- When setting a breakpoint, there is a special distinction between specifying a line number value less than 1 and specifying a value of 1 or greater. Any value for *line-no* less than 1 (for example, 0 or -1) specifies the first executable line of the **main procedure** in the procedure file. However, a positive value for *line-no* specifies the first executable line on or after *line-no* in the procedure file. For example, suppose the **procedure file** starts out like this:

```
1       DEFINE VARIABLE lStart AS LOGICAL INITIAL TRUE.
2       PROCEDURE ShowStart:
3           IF lStart THEN MESSAGE "Procedure is starting ...".
4       END.
5
6       MESSAGE "Hello World!".
7       RUN ShowStart.
8       lStart = FALSE.
                    .
                    .
                    .
```

If you specify a breakpoint at line 0, -1, or any negative value, the breakpoint actually occurs at line 6, the first line that executes in the main procedure. If you specify a breakpoint at line 1 or 2, the breakpoint occurs at line 3, the first executable line in the file, which happens to be the first executable line of an internal procedure.

This distinction also affects procedures containing the Trigger Phrase used to define triggers in widget definitions. For example, suppose the procedure file starts out like this:

```
1       DEFINE BUTTON bOK LABEL "OK"
2           TRIGGERS:
3               ON CHOOSE
4                   MESSAGE "OK Pressed!".
5           END TRIGGERS.
6       MESSAGE "Hello World!".
                    .
                    .
                    .
```

Again, if you specify a breakpoint at line -1, the breakpoint occurs on line 6, but if you specify the breakpoint at line 1, it actually occurs at line 4, which is the first executable line of a trigger block.

NOTE: There is no distinction between line -1 and line 1 for an ON statement because the ON statement is itself an executable statement. Thus, the executable lines of **its** trigger block are not breakpoint candidates in this scenario.

SEE ALSO

CANCEL ALL Command, CANCEL BREAK Command, SHOW BREAKS Command

BUTTON Command

Creates a button in the Debugger button panel.

SYNTAX

```
BUTTON label button-definition
```

PARAMETERS

label

A quoted or unquoted string to display as a button label. Both quoted and unquoted labels are case sensitive. If you want blanks embedded in the label, you must quote it.

button-definition

One or more Debugger commands or defined macros separated by semicolons (;).

NOTES

- If *label* specifies an existing button, *button-definition* redefines the button.
- If the definition specified by *button-definition* contains macros, the Debugger immediately replaces each macro (and each macro within a macro) with its definition in the button definition. Thus, because a button definition is stored only in terms of Debugger commands, you cannot specify an undefined macro in the button definition.
- The Debugger adds new button definitions to the right end of the current set of buttons in the button panel. If the button panel is too small to accommodate a new button definition, you must resize or scroll the button panel to see the new button.
- When you choose a button from the Debugger button panel, the Debugger commands in the button definition are entered together as one command in the Debugger command queue. They echo on a single line in the command panel when executed.
- You can include single-valued Debugger variables to substitute parameters within a button definition. The values of these variables are set from the Debugger listing panel when you choose the button, as listed in Table A–1.

Table A–1: Variables in Button Definitions

Variable	Value
&FILE	The name of the procedure displayed in the listing panel.
&LINE	The line number of the procedure line selected in the listing panel.
&TEXT	The text selected in the listing panel.

EXAMPLES

This example defines a button in the Debugger button panel:

```
BUTTON "Push Me to Continue" CONTINUE
```

This example defines a button using one Debugger variable that specifies the filename for two commands:

```
BUTTON debug BREAK &file; RUN &file
```

SEE ALSO

CANCEL ALL Command, CANCEL BUTTON Command, MACRO Command, SHOW BUTTONS Command

CANCEL ALL Command

Removes all breakpoints, macros, or buttons.

SYNTAX

```
CANCEL ALL { BREAKS | MACROS | BUTTONS }
```

PARAMETERS

{ BREAKS | MACROS | BUTTONS }

Specifies whether you want to remove breakpoints, macros, or buttons.

NOTE

When you remove all buttons, they are all immediately cleared from the button panel.

SEE ALSO

CANCEL BREAK Command, CANCEL BUTTON Command, CANCEL MACRO Command

CANCEL BREAK Command

Removes a single breakpoint specified by combinations of procedure names and line numbers or any selection of breakpoints specified by breakpoint number.

SYNTAX

```
CANCEL BREAK
  {   proc-name [ line-no ]
    | line-no
    | #n [ #n ] ...
  }
```

PARAMETERS

proc-name

The name of a Progress procedure.

line-no

An integer that specifies a line number in the listed or specified procedure.

n

An integer that specifies the ordinal breakpoint number of any current breakpoint displayed by the SHOW BREAKS command. The preceding # symbol is required.

NOTES

- When using the procedure (*proc-name*) and line number (*line-no*), you must specify the procedure exactly as you did to set the breakpoint. If the procedure is inactive you must specify the line number exactly as you did when you set it. However, if the procedure is active in the Debugger context, you can specify either the original line number or, if different, the number of the executable line where the breakpoint applies. For example, the following CANCEL BREAK commands do **not** necessarily remove the same breakpoint, even though both breakpoints have the same effect during execution:

```
CANCEL BREAK chkreg.p
CANCEL BREAK chkreg.p 1
```

For more information on specifying breakpoints, see the BREAK Command.

- When you cancel a breakpoint, the Debugger echoes the result in the data panel, including the ordinal breakpoint number, procedure name, and line number of the canceled breakpoint. If the procedure is displayed in the listing panel, the breakpoint indicator disappears from the corresponding line.
- If you specify only a line number, the breakpoint is canceled on the specified line for the procedure displayed in the listing panel.
- If you specify only a procedure name, the breakpoint is canceled on the first executable line if the procedure is active in the Debugger context, or on just the procedure name if the procedure is inactive.

SEE ALSO

BREAK Command, CANCEL ALL Command, SHOW BREAKS Command

CANCEL BUTTON Command

Removes an existing button definition.

SYNTAX

```
CANCEL BUTTON label
```

PARAMETERS

label

A quoted or unquoted string that specifies the label of an existing button on the Debugger button panel.

NOTES

- When you remove a button definition, the button is immediately cleared from the button panel.
- You must specify *label* in exactly the same manner as in the original button definition, with the same embedded spaces and letter case.
- When you specify a nonexistent button, the Debugger displays an error message.

SEE ALSO

BUTTON Command, CANCEL ALL Command, SHOW BUTTONS Command

CANCEL LOG Command

Stops logging Debugger commands and messages to the current log file and closes the file.

SYNTAX

```
CANCEL LOG
```

NOTE

The LOG Command starts the logging of Debugger commands and messages to a log file.

SEE ALSO

LOG Command

CANCEL MACRO Command

Removes an existing macro definition.

SYNTAX

```
CANCEL MACRO macroname
```

PARAMETERS

macroname

The name of an existing macro.

NOTES

- You must specify *macroname* with the same letter case used to specify it in the original macro definition.
- When you specify a nonexistent macro, the Debugger displays an error message.

SEE ALSO

CANCEL ALL Command, MACRO Command, SHOW MACROS Command

CONTINUE Command

Resumes execution of 4GL statements in the executing procedure.

SYNTAX

```
CONTINUE
```

NOTES

- When you execute this command, execution continues up to the next breakpoint, break interrupt, or procedure termination.
- You can enter CONTINUE as CONT.

SEE ALSO

NEXT Command, STEP Command

DISPLAY Command

Displays the value of a specified variable, field, parameter, or attribute in the current procedure. The Debugger displays the information in the data panel.

SYNTAX

```
DISPLAY [ VARIABLE | FIELD | PARAMETER ] name
```

```
DISPLAY handle :attribute
```

```
DISPLAY widget-name :attribute [ IN FRAME frame-name ]
```

```
DISPLAY { FRAME | MENU } widget-name :attribute
```

```
DISPLAY { SUB-MENU | MENU-ITEM } widget-name :attribute
  [ IN { MENU | SUB-MENU } menu-name ]
```

PARAMETERS

[VARIABLE | FIELD | PARAMETER]

Specifies the type of data object you want to inspect. If you do not specify this option, the Debugger attempts to resolve *name* for all object types and reports any ambiguity in the data panel.

name

The name of a variable, field, or parameter defined in the current procedure. If *name* specifies a field it must also specify any necessary buffer qualifier that you need to uniquely identify the field.

handle

The name of a variable, field, parameter, or system handle in the current procedure that contains a valid HANDLE, WIDGET-HANDLE, or COM-HANDLE value.

attribute

The name of an attribute of the specified widget, or the name of a property of a specified com object.

widget-name

The name of a static widget defined in the current procedure.

[IN FRAME *frame-name*]

Specifies the name of the frame (*frame-name*) that contains the specified field-level widget. If you do not specify this option, the widget in the most recently defined frame is used.

{ FRAME | MENU | SUB-MENU | MENU-ITEM } *widget-name*

Specifies the type of widget for a frame, menu, submenu, or menu item widget (respectively) whose attribute you want to inspect. If you do not specify a widget type for these widgets, the Debugger reports an error in the data panel. Note that you do not use a widget type to specify a field-level widget.

[IN { MENU | SUB-MENU } *menu-name*]

Specifies the name of the menu or submenu (*menu-name*) that contains the specified menu or submenu item widget. If you do not specify this option, the widget from the most recently defined menu or from the first submenu of the menu tree is used.

NOTES

- To display data objects accessible to a particular active procedure, use the UP Command or DOWN Command to give it focus as the current procedure in the Debugger context.
- You cannot query (and as a result invoke) methods from the Debugger.
- You cannot reference more than one com object property name or COM-HANDLE value in a single DISPLAY command.

SEE ALSO

ASSIGN Command, LIST BUFFERS Command, LIST PARAMETERS Command, LIST VARIABLES Command

DOWN Command

Moves down a specified number of entries (Progress triggers, internal procedures, and external procedures) in the procedure call stack.

SYNTAX

```
DOWN [ integer ]
```

PARAMETERS

integer

An integer that specifies the number of triggers or procedures to move down the call stack. The default value is 1.

NOTES

- The listing panel displays the current entry and updates the line pointer to the currently executing statement. It also shows any breakpoints in the current entry with the breakpoint indicator.
- The current stack entry becomes the current procedure or trigger in the Debugger context. Thus, all LIST commands and DISPLAY Command now refer to data objects accessible to the current entry, and line numbers specified in the BREAK Command also default to the current entry.
- The SHOW STACK Command lists the stack with the context arrow pointing to the current entry.
- Although the current procedure changes with each DOWN Command, the executing procedure does not change. Any control flow command, such as NEXT Command or CONTINUE Command, immediately makes the executing procedure the current procedure and continues execution from the current stopping point.

SEE ALSO

SHOW STACK Command, UP Command

EXIT Command

Clears all breakpoints, destroys the Debugger window, and returns all Debugger resources to the system.

SYNTAX

```
EXIT
```

NOTES

- The effect of the EXIT command depends upon the technique used to start the Debugger.

  a. If the Debugger was started in stand-alone mode using the Debugger (-debug) startup parameter, the EXIT command terminates all work in process, terminates the Debugger, and exits to the operating system.

  b. If the Debugger was started by invoking the DEBUGGER handle INITIATE and SET-BREAK methods from a Progress procedure (in application mode), the EXIT command terminates all work in process, terminates the Debugger, and returns control to the Progress client session. If the invoking procedure is still running, control returns to the procedure at its current stopping point.

  c. If the Debugger was started by invoking the DEBUGGER handle DEBUG method from a Progress procedure (in stand-alone mode), the EXIT command terminates all work in process, terminates the Debugger, and returns control to the invoking procedure after the statement that invoked the DEBUG method.

  NOTE: The difference between techniques b and c is that in b, the invoking procedure runs in the Debugger context, continuing to execute while the Debugger is running; in c, the invoking procedure blocks while the Debugger is running and continues to execute only after the Debugger exits.

d. If the Debugger was started from the Procedure Editor or AppBuilder using the Debug option of the Compile menu, the Debugger is running in application mode, and the EXIT command works as in technique b—that is, if the procedure run from the Editor or AppBuilder is still running, the Debugger returns to that procedure, continuing execution from its current stopping point; otherwise, the Debugger returns to the Editor or AppBuilder.

e. If the Debugger was started from the ADE Desktop, the Debugger is running in stand-alone mode, and the EXIT command works as in technique c—that is, the Debugger returns to the ADE Desktop.

- Under all circumstances, the EXIT Command terminates any executing procedures started with the Debugger RUN Command and undoes all work associated with them.
- For more information on the Debugger modes of operation (stand-alone and application mode), see Chapter 3, "Starting and Stopping the Debugger."

FILE Command

Displays the debug listing file for a specified external Progress procedure in the Debugger listing panel.

SYNTAX

```
FILE filename
```

PARAMETERS

filename

The name of an external Progress procedure file.

NOTE

If the listed procedure is not active in the Debugger context, breakpoint indicators reflect the specified breakpoint lines, not necessarily the effective breakpoint lines. The breakpoint indicators adjust to show the effective (executable) lines where the breakpoints occur when the procedure becomes active in the Debugger context. You can always review breakpoint settings for any procedure using the SHOW BREAKS Command.

SEE ALSO

DOWN Command, RUN Command, UP Command

HELP Command

Accesses help information on the Debugger or a specified Debugger command.

SYNTAX

```
HELP [ debug-command ]
```

PARAMETERS

debug-command

The first word or first two words of a Debugger command.

NOTES

- If you enter the HELP command without options, the Debugger displays an index of all Debugger commands that you can choose from.
- This command accesses the same Debugger and Progress information as the Debugger Help menu, but from a different entry point. From this point, you can reach most of the remaining Debugger Help information by following the Help links and options in the Help system.

EXAMPLES

This example displays an index of all Debugger commands that begin with the word SHOW:

```
HELP SHOW
```

This example displays help information about the SHOW BUTTONS Command for the Debugger:

```
HELP SHOW BUTTONS
```

INPUT Command

Reads a Debugger command file and places the commands from the file in the Debugger command queue for execution.

SYNTAX

```
INPUT filename
```

PARAMETERS

filename

The name of a text file that contains Debugger commands.

NOTES

- The Debugger executes commands from the command queue in the order they are read from the file. This is true even if a command in the queue passes control to a Progress procedure. When control returns to the Debugger, the remaining commands in the command queue execute before the Debugger gets input from the Debugger window.
- If there is a syntax error in a command file, the Debugger executes all commands prior to the bad command. All commands following and including the bad command are purged from the Debugger command queue, and the Debugger continues accepting input from the Debugger window.
- You can force the Debugger to immediately accept input from the Debugger window by invoking the break interrupt function (the Debug→ Interrupt menu option). This function purges the command queue at any point in its execution, allowing the Debugger to accept input from the Debugger window.

SEE ALSO

LOG Command

LIST BUFFERS Command

Lists the names and status of all implicit and explicit buffers defined by the current procedure. The Debugger displays this information in the data panel.

SYNTAX

```
LIST BUFFERS
```

NOTES

- The Debugger displays the type and table name associated with each buffer. These are the valid buffer types:
  - PARAMETER
  - NEW SHARED
  - SHARED
  - WORK-TABLE
  - TEMP-TABLE
- To list buffers accessible to a particular active procedure, use the UP Command or DOWN Command to give it focus as the current procedure in the Debugger context.

SEE ALSO

DISPLAY Command, LIST PARAMETERS Command, LIST VARIABLES Command

LIST PARAMETERS Command

Lists the names, data types, extents, and data direction of all parameters passed to the current procedure. The Debugger displays this information in the data panel.

SYNTAX

```
LIST PARAMETERS
```

NOTE

To list parameters accessible to a particular active procedure, use the UP Command or DOWN Command to give it focus as the current procedure in the Debugger context.

SEE ALSO

DISPLAY Command, LIST BUFFERS Command, LIST VARIABLES Command

LIST VARIABLES Command

Lists the names, data types, and extents of all variables defined in the current procedure. The Debugger displays this information in the data panel.

SYNTAX

```
LIST VARIABLES
```

NOTE

To list variables accessible to a particular active procedure, use the UP Command or DOWN Command to give it focus as the current procedure in the Debugger context.

SEE ALSO

DISPLAY Command, LIST BUFFERS Command, LIST PARAMETERS Command

LOG Command

Starts session logging from this point to a specified log file. The Debugger then writes all Debugger commands and associated messages to the file.

SYNTAX

```
LOG [ filename ]
```

`filename`

The name of a log file. If *filename* specifies a simple filename, the Debugger creates the log file in your temporary directory. If *filename* specifies a relative pathname, the Debugger creates the file relative to the temporary directory in the directory you specify. If *filename* specifies an absolute pathname, the Debugger creates the log file in the specified directory.

If you do not specify *filename*, the Debugger creates the log file with the name `prodebug.log` in your temporary directory.

NOTES

- Your temporary directory is the directory specified by the Temporary Directory (-T) startup parameter, or the current working directory by default.
- If you execute the LOG Command for a log file and logging is already started for the specified file, the Debugger displays an error message and ignores the request.
- If you execute the LOG Command for a log file while logging is already started for a previous log file, the Debugger stops logging and closes the previous file, then starts logging to the new file.
- Output to the log file is flushed with each write to keep the file current.
- The LOG Command does not append. If a file with the specified log file name already exists, its contents are overwritten.

SEE ALSO

CANCEL LOG Command, INPUT Command

MACRO Command

Creates a Debugger macro.

SYNTAX

```
MACRO macroname macro-definition
```

PARAMETERS

macroname

The name of the new macro. The macro name is case sensitive and cannot specify a Debugger command name.

macro-definition

One or more Debugger commands and user-defined macro names separated by semicolons (;).

NOTES

- If *macroname* specifies the name of an existing macro, *macro-definition* redefines the macro.
- When you enter a macro as a command, the macro command (not its definition) is echoed in the command panel.
- You can include single-valued Debugger variables to accept values in a macro definition. The values of these variables are set differently depending on whether you invoke the macro as a command or as part of a button definition. If the macro is invoked as part of a button definition, the variables take their values from the listing panel as defined for button definitions. Otherwise, they take the corresponding values entered on the command line, as listed in Table A–2.

Table A–2: Variables in Macros as Commands *(1 of 2)*

Variable	Value
&FILE	A corresponding quoted or unquoted string.

Table A–2: Variables in Macros as Commands *(2 of 2)*

Variable	Value
&LINE	A corresponding numeric value.
&TEXT	A corresponding quoted or unquoted string.

- When you enter a macro that is defined with Debugger variables as a command, you must specify each variable value in the same order, from left to right, as it first appears in the macro definition.

EXAMPLES

This macro definition executes the BREAK Command using the &FILE and &LINE Debugger variables:

```
MACRO stop-at BREAK &FILE &LINE
```

To invoke the stop-at macro as a command, you must specify the macro name followed by a filename and line number:

```
stop-at myapp.p 36
```

The stop-at macro expands to this Debugger command:

```
BREAK myapp.p 36
```

When you invoke a macro as a command, you can specify only one value for each Debugger variable used, no matter how many times you use it. Thus, you can never specify more than three variable values for a macro command. For example, this macro definition executes the BREAK Command and RUN Command using the &FILE Debugger variable:

```
MACRO debug BREAK &FILE; RUN &FILE
```

To invoke the debug macro, you only have to enter the macro name with one filename parameter:

```
debug myapp.p
```

The debug macro then expands to these Debugger commands:

```
BREAK myapp.p; RUN myapp.p
```

This macro definition executes the CANCEL BREAK Command using the &TEXT variable. This allows all formats of the CANCEL BREAK Command to be invoked by the macro command:

```
MACRO cb CANCEL BREAK &TEXT
```

These commands all show valid uses of the cb macro:

```
cb "myapp.p 33"
cb "#2 #4 #5 #8"
cb 33
```

SEE ALSO

BUTTON Command, CANCEL ALL Command, CANCEL MACRO Command, SHOW MACROS Command

NEXT Command

Runs a specified number of executable lines from the current stopping point in the executing procedure.

SYNTAX

```
NEXT [ integer ]
```

PARAMETERS

integer

An integer that specifies the number of executable lines to run. The default value is 1.

NOTES

- If the NEXT command executes a statement in a procedure that calls a trigger or subprocedure, the Debugger does not step **into** the trigger or subprocedure. The NEXT command executes the trigger or subprocedure and any other executable statements on the same line, and then proceeds to the next executable line of the calling procedure until a total of *integer* lines are executed.
- If the NEXT command encounters a breakpoint, it stops execution on the affected line, regardless of the value of *integer*.
- If the Debugger encounters a breakpoint in a trigger or subprocedure as part of a NEXT command, execution stops at the breakpoint and the executing trigger or procedure becomes the current procedure. You can continue execution from the breakpoint. Thus, although the NEXT command does not **step** into a subprocedure or trigger, it does **break** into one if a breakpoint occurs.
- If the NEXT command returns from a subprocedure, the return to the next executable line of the calling procedure counts as one line of the specified *integer* lines.
- When you return from a trigger or subprocedure that the NEXT command has broken into, the calling procedure executes any remaining statements on the line that invoked the trigger or subprocedure and proceeds to the next executable line.

SEE ALSO

CONTINUE Command, STEP Command

QUIT Command

Immediately terminates the executing application as if it invoked the 4GL QUIT statement.

SYNTAX

```
QUIT
```

NOTE

- As with the 4GL QUIT statement, any active transaction is completed before the application terminates.
- If the application was started from the Debugger with the RUN Command, it terminates and returns control to the Debugger.
- If the application was started together with the Debugger in application mode, it terminates and returns control to the application startup environment (the operating system, Procedure Editor, or AppBuilder).

SEE ALSO

EXIT Command, STOP Command

RUN Command

Runs a specified procedure, bringing it into the Debugger context.

SYNTAX

```
RUN procedure
```

PARAMETERS

procedure

The name of a Progress procedure.

NOTES

- You cannot specify input or output parameters for *procedure*.
- If the executing procedure file contains an internal procedure named *procedure*, the RUN command executes the internal procedure. Otherwise, it attempts to execute an external procedure named *procedure*.
- Before *procedure* runs, the executing procedure is pushed onto the call stack as if *procedure* were called from the executing procedure at its current stopping point.
- When *procedure* reaches a breakpoint or is stopped using the Debugger break interrupt function, the Debugger displays the listing for *procedure* in the Debugger window with all current breakpoints marked and the line pointer indicating the next line to execute.
- Any database activity is subject to the current transaction state. Any breakpoints set inside *procedure* are honored, and when the procedure completes, Progress returns to the Debugger for input.
- If a STOP Command or QUIT Command is encountered in *procedure*, the procedure terminates and returns control to the Debugger. The STOP or QUIT statements do not affect any other procedures above *procedure* on the call stack.

SEE ALSO

FILE Command

SHOW BREAKS Command

Displays all presently set breakpoints in the data panel in the order of creation.

SYNTAX

```
SHOW BREAKS
```

NOTE

Each breakpoint is identified by the procedure name and line number, as well as a unique breakpoint number that you can use to reference it.

SEE ALSO

BREAK Command, CANCEL ALL Command, CANCEL BREAK Command, SHOW STACK Command, SSHOW TRANSACTION Command

SHOW BUTTONS Command

Displays all current button definitions in the data panel and in the order of creation.

SYNTAX

```
SHOW BUTTONS
```

SEE ALSO

BUTTON Command, CANCEL ALL Command, CANCEL BUTTON Command, SHOW BREAKS Command, SHOW MACROS Command

SHOW MACROS Command

Displays all current macro definitions in the data panel and in the order of creation.

SYNTAX

```
SHOW MACROS
```

SEE ALSO

CANCEL ALL Command, CANCEL MACRO Command, MACRO Command, SHOW BREAKS Command, SHOW BUTTONS Command

SHOW STACK Command

Displays the procedure call stack and all associated input parameter values in the data panel.

SYNTAX

```
SHOW STACK
```

NOTES

- The Debugger can only report input parameter values if it is initialized prior to execution of the procedure.
- If you navigate the call stack with the UP Command and DOWN Command, the SHOW STACK command lists the call stack with the context arrow pointing to the procedure that is currently displayed in the listing panel. This is your current procedure for all LIST commands and DISPLAY Command and the default procedure for all breakpoint references.

SEE ALSO

DOWN Command, UP Command

SHOW TRANSACTION Command

Displays a message in the data panel indicating whether there is a transaction active in the executing procedure.

SYNTAX

```
SHOW TRANSACTION
```

NOTES

- This command returns the transaction state of the executing procedure, whether or not the listed procedure is on the stack or outside the Debugger context.
- You can enter SHOW TRANSACTION as SHOW TRANS.

SEE ALSO

ASSIGN Command

STEP Command

Runs a specified number of executable lines from the current stopping point in the executing procedure.

SYNTAX

```
STEP [ integer ]
```

PARAMETERS

integer

An integer that specifies the number of executable lines to execute. The default value is 1.

NOTES

- If the STEP command executes a statement in a procedure that calls a trigger or subprocedure, the Debugger steps **into** the trigger or subprocedure and continues until a total of *integer* lines are executed. If the STEP command stops in the trigger or subprocedure, the trigger or subprocedure is now the current procedure, and you can continue execution from the current stopping point.
- The STEP command ignores breakpoints and executes every executable line that it encounters.
- Stepping into a trigger or subprocedure to the first executable line and stepping out of a trigger or subprocedure from the last executable line each count as one line of the specified *integer* lines.
- When you return from a trigger or subprocedure that the STEP command has stepped into, the calling procedure executes any remaining statements on the line that called the trigger or subprocedure and proceeds to the next executable line.

SEE ALSO

CONTINUE Command, NEXT Command

STOP Command

Immediately terminates the executing application as if it invoked the 4GL STOP statement.

SYNTAX

```
STOP
```

NOTES

- As with the 4GL STOP statement, any active transaction is backed out before the application terminates.
- If the application was started from the Debugger with the RUN command, it terminates by returning control to the Debugger.
- If the application was started together with the Debugger in application mode, it terminates by closing the Debugger window and returning control to the application startup environment (for example, Procedure Editor or AppBuilder). If the application was invoked from the operating system using the Startup Procedure (-p) startup parameter, and the Debugger was started in application mode from the 4GL using the DEBUGGER handle, the application terminates by closing the Debugger window and then restarting the startup procedure (unless the STOP is trapped by an ON STOP statement).

SEE ALSO

EXIT Command, QUIT Command

UP Command

Moves up a specified number of entries (Progress triggers, internal procedures, and external procedures) in the procedure call stack.

SYNTAX

```
UP [ integer ]
```

PARAMETERS

integer

An integer that specifies the number of triggers or procedures to move up the call stack. The default value is 1.

NOTES

- The listing panel displays the current entry and updates the line pointer to the currently executing statement. It also shows any breakpoints in the current entry with the breakpoint indicator.
- The current stack entry becomes the current procedure or trigger in the Debugger context. Thus, all LIST commands and DISPLAY Command now refer to data objects accessible to the current entry, and line numbers specified in the BREAK command also default to the current entry.
- The SHOW STACK Command lists the stack with the context arrow pointing to the current entry.
- Although the current procedure changes with each UP Command, the executing procedure does not change. Any control flow command, such as NEXT Command or CONTINUE Command, immediately makes the executing procedure the current procedure and continues execution from the current stopping point.

SEE ALSO

DOWN Command, SHOW STACK Command

Index

Symbols

_ (underscore) files 6–10, 7–6, 7–8

A

Accelerator keys
  defining in Motif 6–7
  described 4–8

Active procedures
  debugger context 1–11
  defined 1–8

Alert-boxes 7–9

Application mode 3–3
  debugger exit 5–6
  DEBUGGER handle 7–3, 7–4
  hiding the invoking procedure 7–6

Applications
  stacking 3–5

ASSIGN command A–2
  example 2–9
  using 5–24

Attributes
  displaying 5–24
  modifying 5–24

Audience xi

B

Bold typeface
  as typographical convention xiii

BREAK command A–5
  example 2–6
  using 5–8

Break interrupt function
  command entry 5–3
  command queue 4–7
  defined 1–3

Break option
  debug menu 4–12

Breakpoint analysis
  features 1–4

Breakpoint indicator 4–5

Breakpoint number 5–8

Breakpoints
  defined 1–3
  effective 5–8
  example 2–7
  managing 5–7
  references for removing 5–11
  references for setting 5–8

removing 5–11
using procedure name and line number 5–11
using the breakpoint number 5–11
setting 5–8
active procedures 5–9
debug menu 5–10
inactive procedures 5–9, 5–12
internal procedures and procedure files 5–12
trigger example 2–12

BUTTON command A–8
using 5–28

Button definitions
saving 4–12

Button panel 4–6

Button variables 5–29

Buttons 5–28
defining 1–5, 5–28
initializing
Motif 6–5
MS-Windows 6–2, 6–3, 6–4
removing 5–33
using 5–31
variables A–8

C

Call stack. *See* Procedure call stack

CANCEL ALL command A–10
using 5–11, 5–33

CANCEL BREAK command A–11
using 5–11

CANCEL BUTTON command A–13
using 5–33

CANCEL LOG command A–14
using 5–33

CANCEL MACRO command A–15
using 5–33

Changing focus
Debugger context 5–27

Code path analysis
features 1–4

Com object property
with ASSIGN command A–3, A–4
with DISPLAY command A–18

COM-HANDLE value
with ASSIGN command A–3, A–4
with DISPLAY command A–17, A–18

Command entry
break interrupt 5–3
buttons 5–5
command history example 2–10
command panel 5–4
files 5–5
menu options 4–8, 5–5
techniques 5–3

Command files 6–7
using 5–5

Command history
command entry 5–4
defined 4–6
next command to execute 4–7

Command panel 4–6
command queue 4–7
components 4–6
operation 4–7

Command programming
features 1–5
macros and buttons 5–28

Command queue
break interrupt 4–7
command entry 4–7
indicator. *See* Queue indicator
intrinsic functions 5–3

Commands
ASSIGN 5–24
BREAK 5–8
BUTTON 5–28

CANCEL ALL 5–11, 5–33
CANCEL BREAK 5–11
CANCEL BUTTON 5–33
CANCEL LOG 5–33
CANCEL MACRO 5–33
CONTINUE 5–16
DISPLAY 5–24
DOWN 5–27
entering 5–3
buttons 5–5
command files 5–5
command panel 5–4
menu options 5–5
mouse 5–4
multiple commands 5–4
EXIT 5–6
FILE 5–28
HELP 5–25
INPUT 5–5
LIST BUFFERS 5–24
LIST PARAMETERS 5–24
LIST VARIABLES 5–24
LOG 5–33
MACRO 5–28
NEXT 5–16
QUIT 5–22
RUN 5–14
SHOW BREAKS 5–26
SHOW BUTTONS 5–26
SHOW MACROS 5–26
SHOW STACK 5–27
STEP 5–16
STOP 5–22
UP 5–27

Commands in Queue field
function 4–7
location 4–6

CONTINUE command A–16
using 5–16

CRC. *See* Cyclic redundancy check

Current procedure
changing focus 5–27
defined 1–11
setting 5–26

Cyclic redundancy check 6–9

D

Data
debugger context 1–9
examining and modifying 5–23

Data analysis
features 1–3

Data manipulation
features 1–3

Data panel 4–8

.dbg files 6–8
creating 6–8
deleting 6–8
generating
r-code 6–8
source code 6–9

Debug listing
defined 4–4
displaying 4–4
name 4–5

Debug listing files 6–8
creating 6–8
deleting 6–8
format 6–9
generating 6–9
r-code 6–9
source code 6–10

Debug menu 4–11
break option 4–12, 5–10
interrupt option 4–12
next option 4–11, 5–22
run option 4–12, 5–15
step option 4–11, 5–22

-debug startup parameter 3–10

Debugger
exiting 5–6
features 1–2, 1–3, 1–4
focus 1–11
intrinsic functions 5–3
logging function 5–33
automatic 5–33

modes of operation 3–3
overview 1–6
starting from the 4GL. *See* DEBUGGER system handle
starting from the ADE Desktop 3–9
starting from the AppBuilder 3–8
starting from the operating system 3–10
starting from the Procedure Editor 3–6
variables 5–29, A–8, A–29
button values 5–32
macro values 5–32

Debugger context
active procedures 1–8
current procedure 1–11
defined 1–8
focus 1–11
inactive procedures 1–11
navigating in and examining 5–26
procedure call stack 1–9

DEBUGGER handle
application mode 7–4
attributes and methods 7–2
blocking the invoking procedure 7–7
code path analysis 7–4
DEBUG method 7–7, 7–9
debugging tools 7–7
described 7–1
DISPLAY-MESSAGE method
nonvolatile events 5–34
volatile events 5–35
INITIATE method 7–4
invoking procedure 7–4
SET-BREAK method 7–4
nonvolatile events 5–34
stand-alone in the Editor 7–9
stand-alone mode 7–7
starting the Debugger 7–3
subprocedure debugging 7–5

DEBUGGER system handle
attributes 7–2
methods 7–2

Debugger window
button panel 4–6
command panel 4–6
data panel 4–8
layout 4–2
listing panel 4–4
menu bar. *See* Menus
overview 4–1
panels 4–4
saving sizes 4–12
size
saving 4–12

Debugger, Progress
defined 1–1
versions 1–1

Debugger-application interaction 1–7
application control 1–7
debugger control 1–7

Debugger-client interaction 1–6

Debugging events. *See* Event debugging

Debugging modes
DEBUGGER handle 7–3

Debugging sessions
defined 1–1
exiting 5–6
application mode 5–6
debugger settings 5–7
logging 5–33
automatic 5–33
operating pattern 5–2
overview 1–6
sample 2–2
terminating 5–6

Dialog boxes
debug menu
break option 5–10
run option 5–15

DISPLAY command A–17
example 2–8
using 5–24

Displaying status and information 5–25

Distributed Applications 1–1

DOWN command A–19
using 5–27

E

Effective breakpoints 5–8

Error messages
  displaying descriptions xx

Event debugging 5–34
  MS-Windows 5–35
  nonvolatile events 5–34
  volatile 5–35

Events
  nonvolatile 5–34
  volatile 5–34

Examining the Debugger context 5–27

Example procedures xvii

Executable line
  defined 1–9

Executing procedures 5–13

EXIT command A–20
  using 5–6

Exit option
  file menu 4–9

Exiting the Debugger 5–6

F

Fields
  displaying 5–24
  modifying 5–24

FILE command A–22
  using 5–28

&FILE Debugger variable 5–29, 5–32, A–8, A–29

File menu 4–9
  exit option 4–9
  open option 4–9, 5–28

Files
  command 6–7
  .dbg files 6–8
  debug listing 6–8
  initialization 6–2
  log 6–11
  procedure. *See* Procedure files
  .prodebug 6–7
  prodebug.log 5–33, 6–11
  progress.ini. *See* PROGRESS.INI file
  resource definition 6–5
  system-prodebug 6–5
  .Xdefaults 6–5

Find next option
  search menu 4–10

Find option
  Search menu 4–10

Find prev option
  Search menu 4–10

4GL debugger control
  features 1–5

4GL lines
  executable and non-executable 1–9

Functions
  intrinsic
    command queue 5–3
    defined 5–3

G

Goto line option
  search menu 4–11

H

Help
  Progress messages xx

HELP command A–23
  using 5–25

Help menu 4–13

I

Inactive procedures
defined 1–11

Initial command file
Motif 6–5
MS-Windows 6–3

Initialization
features 1–5
Motif 6–5
MS-Windows 6–2

INPUT command A–24
using 5–5

Internal procedures 5–12

Interrupt option
debug menu 4–12

Italic typeface
as typographical convention xiii

K

Keystrokes xiii

L

&LINE Debugger variable 5–29, 5–32, A–8, A–29

Line pointer 4–5
stacked procedures 4–5

Lines
executable and nonexecutable 1–9

LIST BUFFERS command A–25
using 5–24

LIST PARAMETERS command A–26
using 5–24

LIST VARIABLES command A–27
using 5–24

Listing management
features 1–4

Listing name 4–5

Listing panel 4–4
displaying 4–4
indicators 4–5
label 4–5
text selection 4–5

LOG command A–28
using 5–33

Log files 6–11
contents 6–11
converting to command files 6–11
using 5–33

Logging a debugging session 5–33

M

MACRO command A–29
using 5–28

Macro definitions
saving 4–12

Macros 5–28
defining 5–28
initializing
Motif 6–5
MS-Windows 6–2, 6–3, 6–4
removing 5–33
using 5–30
variables A–29
variables in 5–29

Macros, defining 1–5

Manual
syntax notation xiv

Menu bar. *See* Menus

Menu mnemonics 4–8

Menus 4–8
accelerator keys. *See* Accelerator keys

Debug
setting breakpoints 5–10
debug 4–11
executing procedure lines 5–22
running procedures 5–15
file 4–9
help 4–13
mnemonics 4–8
options 4–12
search 4–9

Messages
displaying descriptions xx

Mnemonics 4–9

Monospaced typeface
as typographical convention xiii

N

Navigating in the Debugger context 5–27

NEXT command A–32
subprocedures 5–18, 5–20
using 5–16

Next option
debug menu 4–11

Nonexecutable line
defined 1–9

O

Open option
file menu 4–9

Options menu 4–12
save settings option 4–12

P

Panels 4–4

Parameters
displaying 5–24
modifying 5–24

Persistent procedures
debugger context 1–8
internal procedures 5–15

Procedure
continuing execution 5–14, 5–16
executing 5–13
executing lines 5–16
interrupting execution 5–13
starting 5–14
terminating
Debugger 5–22

Procedure call stack 1–9
defined 1–2
examining 5–27
navigating 5–27

Procedure execution
features 1–2

Procedure files 6–9
CRC values 6–9
generating source code listings 6–10
hidden 6–10
r-code listing generation 6–9

Procedure interruption
features. *See* Break interrupt function

Procedure names
leading underscore (_) 1–10

Procedure termination
features 1–3

Procedures
debugger context 1–8
examples of xvii
executing
Debug menu 5–15
executing lines
debug menu 5–22
persistent
internal procedures 5–15
making active 1–8
running from the Debugger 5–14

.prodebug file 6–5

prodebug.log file 6–11
using 5–33

progress.ini file 6–2
[Debug-Buttons] section 6–4
[Debug-Init]section 6–3
[Debug-Macros]section 6–4
search order 6–2

Q

Queue indicator 4–7
location 4–6
next command to execute 4–7
operation 4–7

QUIT command A–33
using 5–22

R

Resource definition file 6–5
accelerator key definitions 6–7
Debugger window dimensions 6–6

RUN command A–34
using 5–14

Run option
debug menu 4–12

S

Sample session 2–2

Save settings option
options menu 4–12

Saving debugger settings 4–12

Search menu 4–9
find next option 4–10
Find option 4–10
find prev option 4–10
goto line option 4–11

SESSION handle
alert boxes 7–9

Stack trace 7–10

Session handle
described
Alert dialog box 7–1

Session information 5–25

Session logging
features 1–5

Setting breakpoints 5–8

SHOW BREAKS command A–35
using 5–26

SHOW BUTTONS command A–36
using 5–26

SHOW MACROS command A–37
using 5–26

SHOW STACK command A–38
example 2–13
using 5–27

SHOW TRANSACTION command A–39
using 5–28

Stacking applications 3–5

Stand-alone mode 3–4
DEBUGGER handle 7–4, 7–8
hiding the invoking procedure 7–7
Procedure Editor 7–4

Starting the Debugger
ADE Desktop 3–9
AppBuilder 3–8
4GL 7–3
operating system 3–10
Procedure Editor 3–6
stand-alone mode 7–9

STEP command A–40
example 2–5
subprocedures 5–16, 5–20
using 5–16

Step option

debug menu 4–11

STOP command A–41
using 5–22

Syntax notation xiv

system-prodebug file 6–5

T

Terminating an application 5–22

&TEXT Debugger variable 5–29, 5–32, A–8, A–29

Text selection
listing panel 4–5

Transaction monitoring
features 1–4

Trigger blocks
debugger context 1–8

Typographical conventions xiii

U

UP command A–42
using 5–27

V

Variables
Debugger 5–29, A–8, A–29
button values 5–32
macro values 5–32
displaying 5–24
modifying 5–24

W

WebSpeed Applications 1–1

Window layout 4–2

X

.Xdefaults file 6–6
accelerator key definitions 6–7
Debugger window dimensions 6–5