Programming with VisualAge® RPG

Version 6.0 for Windows®
Programming with VisualAge® RPG

Version 6.0 for Windows®
Eighth Edition (June 2005)

This edition applies to Version 6.0 of IBM WebSphere Development Studio Client for iSeries and to all subsequent releases and modifications until otherwise indicated in new editions.

This edition replaces SC09-2449-06.

Changes or additions to the text and illustrations are indicated by a vertical line to the left of the change or addition.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

IBM welcomes your comments. You can send your comments to:

IBM Canada Ltd. Laboratory
Information Development
D1/817/8200/MKM
8200 Warden Avenue
Markham, Ontario, Canada L6G 1C7

You can also send your comments electronically to IBM. See “How to Send Your Comments” on page xi for a description of the methods.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1994, 2005. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
# Contents

**About this Book** ................................................. ix  
Who Should Use This Book ....................................... ix  
Prerequisite and Related Information ........................... ix  
How to Use This Book ............................................ ix  
The VisualAge RPG Library ........................................ x  
How to Send Your Comments ....................................... xi  
Accessing Online Information ...................................... xi  
  Using Online Books .............................................. xi  
  Publications in PDF Format ...................................... xi  
  Using Online Help .............................................. xi  

**What’s New in This Release** ................................. xiii  
Changes in WebSphere Development Studio Client  
for iSeries Version 5.1.2 ........................................... xiii  
Changes in WebSphere Development Studio Client  
for iSeries Version 5.1 ............................................ xiv  
Changes in WebSphere Development Studio Client  
for iSeries Version 5.0 ............................................ xiv

## Part 1. A First Look at Client/Server Applications ........................................ 1

**Chapter 1. Creating a Client/Server Application** ........................................... 3  
About the Sample Application ....................................... 3  
Building the Sample Application .................................... 3  
Deciding What to Show the User ..................................... 5  
  Welcome to the Video Store Catalog ............................... 5  
  Browsing by Category ............................................ 5  
  Searching for Specific Titles .................................... 6  
  Previewing Titles ................................................ 6  
  Modifying and Submitting Orders ................................ 6  
  Submitting Orders ................................................. 6  
High-Level Window Design .......................................... 6  
Creating the Comedy Window ....................................... 7  
  Creating the GUI ................................................ 7  
  Setting Attributes .............................................. 7  
  Adding Program Logic ........................................... 8  
Creating the Preview Window ...................................... 11  
  Creating the GUI ................................................ 11  
  Setting Attributes at Design Time ............................... 12  
  Setting Attributes at Run Time ................................ 12  
  Adding Program Logic ........................................... 12  
Creating Messages ................................................. 15  
Creating the Online Help .......................................... 15  
  Context-sensitive help .......................................... 15  
  Creating Help push buttons .................................... 16  
A Review of Visual Programming ................................... 17

**Chapter 2. Planning Your Application** ........................................... 19  
Enabling Secure Java Applications ............................... 19  
Decide What Functions to Provide ................................ 19  
Help Your Users .................................................. 19  
Keep Window Design Simple ....................................... 20

Number of Windows ............................................... 20  
Content of Each Window ......................................... 20  
Plan Your Code Effectively ....................................... 21  
Keep the User Informed .......................................... 21  
Use a Consistent Style ............................................. 21  
Anticipate Translation Issues ...................................... 22

## Part 2. Working with Parts ........................................ 23

**Chapter 3. Programming with Parts** ........................................... 25  
Getting and Setting Part Attributes ................................ 25  
  Referencing Parts in Your Program ............................ 25  
Responding to Events .............................................. 26  
System Attributes .................................................. 27  
Working with Event and System Attributes ....................... 27  
Coding Static Text and Entry Field Parts ......................... 28  
  Creating and Retrieving Entry Field Parts .................. 28  
  Operation Codes for Window Parts ............................ 29  
  Using Window Operation Codes on Parts with  
     Identical Names ............................................ 30

**Chapter 4. Sample Programs for VisualAge RPG** ........................................... 33  
Before You Begin .................................................. 34  
  Building the Examples ......................................... 34  
  Running the Examples ......................................... 34  
  Accessing an iSeries 400 Server ............................ 34

**Chapter 5. Common Attributes** ........................................... 35  
ParnName Attribute ............................................... 35  
  ParentName Attribute ......................................... 35  
  PartType Attribute ............................................ 35  
  Color Attributes .............................................. 37  
  Enabled Attribute ............................................ 37  
  Size and Position Attributes ................................ 37  
  Visible Attribute .............................................. 38  
  Focus Attribute ............................................... 38  
  UserDatAttribute ............................................. 39  
  Label Attribute .............................................. 39  
    Label Substitution .......................................... 39  
    Translation Tips .......................................... 39

**Chapter 6. Using Data Transfer** ........................................... 41  
A Typical Data Transfer Scenario ............................... 41  
Parts That Support Data Transfer ................................ 41  
Enabling Parts for Data Transfer ................................ 41  
Data Transfer Example .......................................... 42

**Chapter 7. Using Parts** ........................................... 45  
ActiveX ......................................................... 46  
  Adding ActiveX Controls ...................................... 47  
  Setting Properties ............................................ 47  
  Calling Methods ............................................... 48  
  Responding to Events ......................................... 50

© Copyright IBM Corp. 1994, 2005
| ODBC/JDBC Interface Part Example | 124 |
| Outline Box | 137 |
| Special Height and Width Settings | 137 |
| Pop-up Menu | 138 |
| Progress Bar | 139 |
| Progress Bar Example | 139 |
| Push Button | 140 |
| Setting a Default Push Button | 140 |
| Setting a Mnemonic | 140 |
| Assigning Command Keys | 141 |
| Signaling Events | 141 |
| Radio Button | 142 |
| Setting a Mnemonic | 142 |
| Grouping Radio Buttons | 142 |
| Setting the State of a Radio Button | 144 |
| Signaling Events | 144 |
| Slider | 145 |
| Getting and Setting the Slider Value | 145 |
| Signaling Events | 145 |
| Slider Example | 146 |
| Spin Button | 151 |
| Setting Spin Button Values | 151 |
| Getting the Spin Button Value | 151 |
| Preventing User Input | 152 |
| Spin Button Example | 152 |
| Static Text | 155 |
| Changing the Text of a Static Text Part | 155 |
| Getting Static Text Values | 155 |
| Getting and Setting Information for a Window | 155 |
| Editing Output | 156 |
| Status Bar | 157 |
| Status Bar Example | 157 |
| Subfile | 158 |
| Creating a Subfile Part | 159 |
| Maximum Number of Fields per Subfile | 159 |
| Operation Codes for Manipulating Subfile Parts | 159 |
| Loading a Subfile | 159 |
| Determining the Subfile Size | 159 |
| Getting the Record Count | 159 |
| Reading and Updating Records | 160 |
| Changing Subfile Fields | 160 |
| Hidden Fields | 161 |
| Formatting Subfile Fields | 161 |
| Enabling Tabbing | 161 |
| Subfile Example | 161 |
| Signaling Events | 170 |
| Submenu | 171 |
| Timer | 172 |
| Displaying the Timer Icon | 172 |
| Setting the Interval | 172 |
| Generating Tick Events | 173 |
| Getting the Timer Value | 173 |
| Controlling the Timer Using Timer Modes | 173 |
| Timer Example | 173 |
| Vertical Scroll Bar | 179 |
| Window | 180 |
| Window with Canvas | 181 |
| Displaying a Window | 181 |
| Resizing a Window | 183 |
| Setting the Focus | 185 |
| Window List | 185 |
| Terminating a Program | 186 |
| Clearing Fields on a Window | 187 |
| Example of a Window Part | 187 |
| *Component | 188 |
| Using the *component part | 188 |
| Displaying a File Open/Save As dialog | 188 |
| Selecting a printer | 189 |
| Using Plugins | 190 |

**Part 3. Working with iSeries Data**

**Chapter 8. iSeries Connectivity**

| Defining iSeries Information | 193 |
| Notebook Considerations | 193 |
| Setting Up a Server | 194 |
| Setting a Server at Design Time | 194 |
| Setting a Server at Run Time | 194 |
| Using Data Areas | 195 |
| Using iSeries 400 Database Files | 196 |
| Level Checking | 199 |
| Locking Database Files | 199 |
| Overriding Database Files | 199 |
| iSeries 400 Database I/O Considerations | 200 |
| Using Record Blocking to Improve Performance | 200 |
| iSeries 400 Servers Used | 200 |
| Controlling Server Connections at Run Time | 201 |
| Setting the Remote Location name | 201 |
| Connecting to a remote location | 203 |
| Sample Program Using the Signon API | 205 |
| Handling Server Sign-On Errors | 208 |
| Explicitly Handling File Open Errors | 208 |
| Explicitly connecting to the server | 209 |
| Setting up a general program error handler | 209 |
| Using the Security File for Applets | 210 |

**Chapter 9. Reusing iSeries Applications**

| Reuse Scenario | 213 |
| Importing Display Files | 217 |
| Converting Display Files | 218 |
| Reusing UIM Help | 222 |
| UIM and IPF functions that use the same tags | 222 |
| Equivalent UIM and IPF functions that use different tags | 223 |
| UIM Functions with no IPF equivalents | 224 |
| Reusing RPG Source | 224 |

**Part 4. Advanced Topics**

**Chapter 10. Debugging Your Application**

| Starting the Debugger | 227 |
| Displaying the Assembly Code | 228 |
| Loading the DLL Occurrence | 228 |
| Entering Debug Startup Information | 229 |
| Setting a Breakpoint | 229 |
| Running with Breakpoints | 231 |
| Using the Mouse or Keyboard to Start Debug Functions | 231 |
About this Book

This book is a guide for using VisualAge® RPG to develop client/server applications. It describes the steps at every stage of the application development cycle, from design to packaging and distribution. Programming examples are included, to clarify the concepts and the process.

Who Should Use This Book

This book is written for programmers who will be using VisualAge RPG to develop client/server applications. It assumes that you are familiar with developing RPG applications on iSeries™ 400™ systems.

Prerequisite and Related Information

Use the iSeries Information Center as your starting point for looking up iSeries and AS/400e technical information. You can access the Information Center in two ways:

- From the following Web site:
  http://www.ibm.com/eserver/iseries/infocenter
- From CD-ROMs that ship with your OS/400 order:
  iSeries Information Center, SK3T-4091-00. This package also includes the PDF versions of iSeries manuals, iSeries Information Center: Supplemental Manuals, SK3T-4092-00, which replaces the Softcopy Library CD-ROM.

The iSeries Information Center contains advisors and important topics such as CL commands, system application programming interfaces (APIs), logical partitions, clustering, Java™, TCP/IP, Web serving, and secured networks. It also includes links to related IBM® Redbooks and Internet links to other IBM Web sites such as the Technical Studio and the IBM home page.

How to Use This Book

Note: For information on the product, see Getting Started with WebSphere Development Studio Client for iSeries, SC09-2625-06.

The Programming with VisualAge RPG book consists of the following parts:

A First Look at Client/Server Applications
This part describes the steps involved in creating a client/server application with VisualAge RPG. It walks you through the design and development of a sample application, and discusses design issues.

Working with Parts
This part contains tips about creating a graphical user interface with VisualAge RPG parts and writing program logic to drive those parts. It does not describe how to use every operation code, nor does it describe the details of every attribute or event. For such information, see the VisualAge RPG Language Reference and VisualAge RPG Parts Reference.

Working with iSeries 400 Data
This part discusses how to set up your application to access data on an
iSeries 400 server, and how to reuse existing server applications by converting them to VisualAge RPG applications that run on a programmable workstation (PWS).

Advanced Topics
This part highlights the many features you can add to your VisualAge RPG application. It covers topics such as printing from your application, editing output, using the debugger, using picture and sound files, creating online help, adding messages, and running your application on a DBCS system. It also describes the many different ways VisualAge RPG applications can share data and communicate.

Distributing Your Application
This part discusses how to package the VisualAge RPG runtime code and your application. It also describes how to install the runtime code and the application on a user's PWS.

The VisualAge RPG Library
The VisualAge RPG library contains the following publications:

*Programming with VisualAge RPG*

This book contains specific information about creating applications with VisualAge RPG. It describes the steps you have to follow at every stage of the application development cycle, from design to packaging and distribution. Programming examples are included to clarify the concepts and the process of developing VisualAge RPG applications.

*VisualAge RPG Parts Reference*

This book provides information on the VisualAge RPG parts, part attributes, part events, and event attributes. It is a reference for anyone who is developing applications using VisualAge RPG.

*VisualAge RPG Language Reference*

This book provides information about the RPG IV language as implemented using the VisualAge RPG compiler. It contains:

- Language fundamentals such as the character set, symbolic names and reserved words, compiler directives, and indicators
- Data types and data formats
- Error and exception handling
- Specifications
- Built-in functions, expressions, and operation codes.

For an overview of the entire product, see *Getting Started with WebSphere Development Studio Client for iSeries*.

For a list of related publications, see the Bibliography at the end of this book.

You can also find the most current information about IBM WebSphere Development Studio Client for iSeries on the following online source:

**The Development Studio Client Home Page**

[ibm.com/software/ad/wdsc/]
How to Send Your Comments

Your feedback is important in helping us to provide the highest quality information possible. IBM welcomes any comments about this book or any other iSeries documentation.

- If you prefer to send comments by mail, use the following address:
  IBM Canada Ltd. Laboratory
  Information Development
  D1/817/8200/MKM
  8200 Warden Avenue
  Markham, Ontario, Canada L6G 1C7
- If you prefer to send comments electronically, use this e-mail address: toreador@ca.ibm.com
- If you prefer to send comments by fax, use this number:
  1-845-491-7727

Be sure to include the following:
- The name of the book
- The publication number of the book
- The page number or topic to which your comment applies.

Accessing Online Information

VisualAge RPG contains a variety of online books and online help. You can access the help while you are using the product, and can view the books either while you are using the product, or independently.

Using Online Books

To view an online book, either:
- Select the name of the book from the Help pull-down menu of the VisualAge RPG GUI Designer or the editor window.
- Access the books from the Start menu. Select Programs → IBM WebSphere Development Studio Client for iSeries. Then select Documentation.

Publications in PDF Format

VisualAge RPG publications are available in Portable Document Format (PDF) from the iSeries Information Center at URL http://www.ibm.com/eserver/iseries/infocenter.

Note: You need the Adobe Acrobat Reader, Version 3.01 or later for Windows, to view the PDF format of our publications on the workstation. If your location does not have the reader, you can download a copy from the Adobe Systems Web site (http://www.adobe.com).

The following VisualAge RPG publications are available in PDF format:
- Programming with VisualAge RPG
- VisualAge RPG Parts Reference
- VisualAge RPG Language Reference

For information on the product, see Getting Started with WebSphere Development Studio Client for iSeries, SC09-2625-06.
Using Online Help

Online help is available for all areas of VisualAge RPG. To get help for a particular window, dialog box, or properties notebook, select the Help push button (when available).

Note: To view help that is in HTML format, your workstation must have a frames-capable Web browser, such as Netscape Navigator 4.04 or higher, or Microsoft® Internet Explorer 4.01 or higher. (Recommended browser is Netscape Navigator 4.6 or Internet Explorer 5.0)

Using context-sensitive help
To receive context-sensitive help at any time, press F1. The help that appears is specific to the area of the interface that has input focus. Input focus can be on menu items, windows, dialog boxes, and properties notebooks, or on specific parts of these.

For context-sensitive help on dialog boxes, click on the question mark (when available) in the top right-hand corner of the window. A question mark will appear beside the mouse arrow. Click on a word or field and help information on that specific field will be displayed.

Using language-sensitive help
To receive language sensitive help, press F1 in an edit window. If the cursor is on an operation code, you receive help for that operation code; otherwise, you receive help for the current specification.
What’s New in This Release

Changes in WebSphere Development Studio Client for iSeries Version 6.0 include:

- **VARPG Run-time Library Version Check:** When installing or launching a VARPG application, the application will check the version of the VARPG run-time library installed is sufficient.
- **Application Packaging Utility:**
  - Added a choice to disable repeated messages. (eg. "file exist...")
  - If a specified directory already exists, allow override.
- **Project Description Text:** Enable project description text to be set when the project is saved for the first time. This description is then displayed in the Open Component dialog.
- **Backup & Restore Utilities:** Support adding & displaying a short project description.
- **Open File dialog:** Allow selection of multiple files, and selection of a folder name.
- **(For Citrix users) New "Use Separate Folder" option:** Added to the User Preference dialog.
  
The "Use Separate Folder" check box is added for Citrix server users only. If checked, the security file that stores the user ID and password (file FVDSEC.TXT) is stored in a different directory for each user. For example, for user username, the file is stored in c:\Documents and Settings\username\Application Data\IBM\VARPG. At run time, if you want to use a different security file for each user, create a file called fvdvarpg.ini in x:\vrpgtr\system (x:\vrpgtr is where the VARPG runtime is installed) that contains only this line:
  
  USE_SEPARATE_FOLDER=TRUE
  
  If Use Separate Folder is unchecked(default), all users will share the same security file.

Part changes:

- **Container:** Allow setting of READONLY for the entire container. If the record number is set to 0, READONLY applies to the whole container.
- **Line Graph:** Allow clicking a point on a line graph to set group & point values. Previously, this operation was allowed on Bar or Pie graphs only. When clicked on, the group & point value is set and the point is highlighted with a circle.

This publication includes updates from the Readme files of previous releases and other technical corrections.

Changes are noted by a vertical bar (|).

Changes in WebSphere Development Studio Client for iSeries Version 5.1.2

- **Java Version:** JDK 1.4.2 is supported.
- **Component Package Utility:** The utility now remembers the library settings in the build options dialog box.
- **ODBC Support:** Corrected a problem which caused an "invalid cursor state" error.

Part changes:
• *component:
  – Attributes SelFolder, DlgPrompt, and FolderName can be used to select a folder.
  – Attribute FocusPart returns the name of the part that has the focus.
• Subfile: Attribute ByteComp can be set when sorting a subfile column. When this value is set to 1, the sorting algorithm does a byte-by-byte comparison of the two strings (the default is still to do a language sensitive comparison).
• Media: This part now works on Windows XP.

Changes in WebSphere Development Studio Client for iSeries Version 5.1

• Print Settings: The printer file Open function now uses the printer settings from a previously shown Printer Selection dialog in the application.
  – A new file spec keyword called DEVMODE is added for printer files. You can use it to specify a Windows operating system GDI DEVMODE data structure to directly set printer settings for the printer file open.
  – A new attribute called PRTDEVMODE is added to the *component part. You can use it to pass the address of a DEVMODE data structure to the *COMPONENT SELPRINTER attribute’s Printer Selection dialog, to prime the print settings shown initially on the dialog.
• DSPLY opcode: For the free-form syntax version of the DSPLY operation, *DFT can be coded as a placeholder for a blank second operand. For example, DSPLY message *DFT reply.
  – The response operand is no longer required on the DSPLY operation, but is optional.
  – The Factor 2 operand, which is the message-window-definition-name, is now optional as well.
• Subfile part: Can now make the left columns remain static when scrolling horizontally. Use the FreezeCol attribute to control the number of columns kept static.
  Can now also set the font for individual records in a subfile.
• Container part: A new Presentation Manager (PM) compatible style allows you to quickly add records to a container part. You also no longer experience any problems when you add a large number of items to the container part.
• ActiveX events: You can now access and change the parameters of an ActiveX event when it occurs. You can also use a pointer type parameter when you call an ActiveX method.
• Canvas part: JPEG format support is added. Also, you can now use a VKeyPress event. The event occurs when you press a virtual key on your screen, and when one of the parts in the canvas has focus.
• GUI Designer: When you open a project, you can now press a character key to scroll to the project whose name begins with that key.

Changes in WebSphere Development Studio Client for iSeries Version 5.0

• Launching from the IDE: You can launch the VARPG GUI Designer from the workbench.
• Language enhancements made in ILE RPG V5R2 and V5R1 are now available.
• Qualified GUI attribute access: A new syntax is supported for accessing GUI part attributes in expressions or free-form calculations, as an alternative to the %GETATR and %SETATR built-in functions.
• Free-form action subroutines: When you define a new event, the corresponding action subroutine is added to the program source code in a free-form style. This happens when the location where the subroutine is added falls within a
free-form section. In essence, this applies when an /END-FREE directive follows
the end of the calculation specifications, but before any output, procedure, or
compile-time data specifications.

- **Client conversion support for iSeries database fields with CCSID 65535:** Until
  now, VARPG programs have not been able to work with iSeries database fields
  specified with CCSID 65535, which designates that no conversion be done when
  accessing the field data. The client workstation operating in ANSI could not
  understand the received EBCDIC data.

  A new File specification keyword CVTHEX has been added for
  externally-described remote disk files. When specified, any character fields with
  a CCSID of 65535 in the file will be converted to the workstation CCSID for use
  in the application. (The client-side conversion process uses the server sign-on job
  CCSID in place of the field’s 65535 CCSID to perform the conversion.) Note that
  CVTHEX is not supported when compiling to Java.

- **PRTFMT(*SYS) for Windows device context printing:** A new File specification
  keyword PRTFMT(*SYS) has been added for printer files when you compile to
  Windows. This new keyword enables the files to perform output using a device
  context and perform graphics device interface calls to the operating system,
  instead of the usual raw text output. After opening the file, the device context
  handle is copied to positions 81 to 84 of the printer file INFDS, so that the
  application can reference it when making its own Windows GDI calls. Note that
  PRTFMT does not apply when compiling to Java.

- **ActiveX:** The main application and its components can now access the same
  type of ActiveX part.

- JPEG support is added to the Graphic push button.

- **Subfile part:** You can set a subfile row as bold, italic, strikeout, or underscore.
  However, you must set the Index attribute first, and set the FontArea to 3. For
  example, if you wanted to set a row to be italic, you would need to set the
  Index to 4, the FontArea to 3, an the FontItalic to 1. You can also use GETATR
  for these features, in a specific row (indicated by the Index attribute). The run
  time environment is changed so that defined messages are now shown as is.
  Previous behavior deleted empty rows, and ignored the CR/NewLine control
  character at the end of each row.

- **Subfile part:** The runtime can now perform validation for the Character type
  subfile field.

- **List Box part:** The TOPITEM attribute is added to the List box part.

- Corrected a problem which produced an unrecoverable error when using the
  Open File dialog.
Part 1. A First Look at Client/Server Applications

Chapter 1, “Creating a Client/Server Application,” on page 3
Walks you through the design and implementation of a client/server application.

Chapter 2, “Planning Your Application,” on page 19
Helps you plan and design a graphical user interface for your new client/server application.
Chapter 1. Creating a Client/Server Application

This section describes how to create a client/server application using VisualAge® RPG. A sample application is used to describe the following stages of the development process:
1. Designing what the user will see and do with the application.
2. Creating the graphical user interface (GUI).
3. Setting attributes for the GUI parts.
4. Writing the program logic to drive the GUI.
5. Writing messages and online help for the application.

About the Sample Application

The sample application, called the Video Store Catalog, was created with VisualAge RPG, and is in the VisualAge RPG Samples folder. It provides an online catalog that customers can use to buy videos, and has a preview component that customers can use to view video clips before they place their order.

The information about the videos is stored in a database on the iSeries 400 server. Customers using the Video Store Catalog are actually viewing data stored on the iSeries server. The information that customers provide, such as their name and phone number, is also stored on the host.

Note: Another sample application, Video Store Cashier, can access the same database on the host. It uses the information from the customers’ orders to update the inventory of the video store and to bill customers. The Video Store Cashier is not discussed in this section. For information about building and running it, see the comments in the CASHIER.VPG file in the Video Store Cashier project folder.

Building the Sample Application

The Video Store Catalog application requires the files defined in the F specification to be on an iSeries server. The WDSC\samples\vidcust subdirectory has the save file VIDEOSTORE.sav of the library with the required files.

To upload the necessary files and build the application and its associated preview component, follow these steps:
1. Ensure that both VisualAge RPG and the VisualAge RPG Samples are installed.
2. Upload and restore the VIDEOSTORE.sav file on your iSeries server as follows:
   a. On your iSeries server, create a save file named VIDEOSTORE in any library, for example, USER.
   b. On your workstation, change your current directory to vidcust and issue the ftp command as follows:
      x:\...\WDSC\samples\vidcust>ftp HOSTNAME

      c: is the drive where you installed the product and HOSTNAME is the name of the iSeries server where you created the save file. (You can use your server’s TCP/IP address, instead.)
   c. Enter your user ID when prompted for it.
   d. Enter your password when prompted for it.
   e. After you are signed on, make the USER library your current library. Enter:
ftp>cd user
f. Specify the file transfer as binary. Enter:
ftp>binary

The 200 Representation type is binary IMAGE message appears.
g. Transfer the save file. Enter:
ftp>put VIDEOSTORE.sav

The File transfer completed successfully. message indicates that the VIDEOSTORE.sav was uploaded.
h. Enter: ftp>quit
The VIDEOSTORE save file should now be in library USER on your iSeries server.
i. Use RSTLIB to restore the library and retain the same name - VIDEOSTORE - on your iSeries server. Otherwise, change the REMOTE_FILE_NAME parameter in the Catalog.rst file to the name you restored the save file to.
j. In the Catalog.rst file, change the REMOTE_LOCATION_NAME parameter to point to the Remote location name of your iSeries server.

3. Build the Video Store Catalog sample application. Select Build>Windows or Build>Java from the pop-up menu of the Video Store Catalog project folder in the VisualAge RPG Sample Applications folder. When the project builds successfully, an executable program, CATALOG.EXE for Windows, or CATALOG.CLS for Java is created in the Video Store Catalog project folder.

4. Build the associated Preview component. Select Build>Windows or Build>Java from the pop-up menu of the Preview project folder. When the project builds successfully, a dynamic link library (COMMON.DLL) for Windows, or a COMMON.CLS file for Java is created.

Use one of the following methods to run the Video Store Catalog application:
• Select Run>Windows or Run>Java from the pop-up menu of the Video Store Catalog project folder.
• Open the Video Store Catalog project folder and double-click on the CATALOG.EXE icon.
• Type catalog on a command line.

Notes:
1. The multimedia aspects of this application require additional hardware and software. To run the audio in the preview, you must have a sound card on your system. To run the video clip in the preview, you must have Media Player installed. Java applications require the Java Media Framework (JMF) API.
2. The video clips are .AVI files (for Windows) or .MOV files (for Java) that are stored in the Preview folder.

3. To build and run Java applications, you must have Sun’s Java 2 Software Development Kit (J2SDK) Version 1.2, or higher, installed on your workstation. If you do not have the J2SDK, you can download it from Sun Microsystems at the following URL:
http://java.sun.com/products/
After installing the J2SDK, set the PATH environment variable to point to the location of both the Java compiler and the Java Runtime Environment (JRE). For example, if your home directory for the J2SDK is c:\jdk1.2, add the following path statement: c:\jdk1.2\bin
Deciding What to Show the User

A key step in creating your application is to decide what you want users to do with your application, and then to determine what you need to provide so that they can do it.

During the planning stages of the Video Store application, we decided that customers should be able to list the videos in a particular category (such as Action/Adventure or Comedy). They should also be able to list the videos that are made by their favorite director, feature their favorite actor, or are among the top-10 sellers in the store. To help them decide whether they want to buy a particular video, they should be able to preview it. After they find the video they want to buy, they can place their order and then pay for their purchase at the cashier counter.

Now that we have itemized what the customers should be able to do with the application, we can design what they will see when they display the video catalog. This is the time to start designing the content, number, and order of windows in the application.

Welcome to the Video Store Catalog

The main window, or entry point to the application, is the Video Catalog — Welcome window. It sets the stage for what customers can do with the catalog. To use the catalog, the customers must press a graphic push button to select from the following choices:

Browse by Category...

New releases...

Top 10 Best Sellers...

Search for Specific Titles...

Help Catalog

Selecting Help Catalog displays the Get Help on using the Catalog window. If they press one of the other push buttons, another window is displayed from which they can perform other actions, such as view lists, preview clips, or submit a purchase order.

Browsing by Category

Selecting Browse by Category displays the Video Catalog — Categories window. This window presents a list of video categories to choose from:

- Action/Adventure
- Horror
- Children
- Western
- Science Fiction
- Romance
- Comedy
- Classics

To select a category, the customer presses its associated push button. This displays the Video Titles window, which lists the items for that category. Customers can preview some of the titles, add a title to their order, delete it from their order if they change their mind, and submit their order to the cashier.
Searching for Specific Titles
The Video Catalog — Search window lets customers search for a video by category, title, director, or actor. After they specify the search criteria and press the Search push button to initiate a search of the database, the results are displayed in the Video Titles window.

Previewing Titles
Customers can preview a video that is on a list by reading a review of it or, if they have the appropriate hardware and software, by viewing a clip of it with associated audio.

Modifying and Submitting Orders
The Video Catalog — Review/Order My Selections window lets customers modify their order. They can delete videos from the list, change the number of copies they want to buy, and change the type of medium they want the video to be on (tape or laser disk). This window is displayed when customers select a video from a list in a Video Titles window and then press the Review/submit order push button.

When all the information for their order is entered, customers submit their order from this window.

Submitting Orders
When customers submit their orders, they must provide their name, address, and phone number on the Video Catalog — Order Reference window. This information is stored in a database on the iSeries 400 server.

High-Level Window Design
Discussing how to create every one of the windows in the Video Store Catalog is beyond the scope of this section. The following sections describe how you could use the GUI Designer to create two windows resembling the Comedy and the Preview windows, modify some of their part attributes, and write some of the associated program logic. You find these windows by taking the following path through the Video Store Catalog application:

<table>
<thead>
<tr>
<th>Video Catalog — Welcome</th>
<th>Pressing the Browse by Category push button on the Video Catalog — Welcome window displays the Video Catalog — Categories window.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Video Catalog — Categories</td>
<td>Pressing the Comedy push button on the Video Catalog — Categories window displays the Video Catalog — Comedy window.</td>
</tr>
<tr>
<td>Video Catalog — Comedy</td>
<td>Pressing the Preview button after selecting a title on the Video Catalog — Comedy window displays a Preview window.</td>
</tr>
</tbody>
</table>
Creating the Comedy Window

The Comedy window displays the list of comedy videos that customers can purchase. This section describes how you can create a window similar to this one.

Creating the GUI

Select a window with canvas part from the parts palette with the right mouse button, move the pointer icon onto the project view of the GUI Designer, and right-click again. This becomes the design window, on which you put the following parts from the palette: group box, push button, static text, and subfile.

Aligning the parts

You can use the alignment tools in the GUI Designer to size, align, and space the parts so that they resemble those shown in Figure 1. For information on using these tools, see the online help or HTML tutorial.

Setting Attributes

After a part is placed and positioned on the window, you can modify the default settings for the part attributes using its properties notebook. To do this, right-click on the part and select Properties from the part’s pop-up menu.

Some of the part attributes you can modify are described below.
Window attributes
You can select the items you want to appear on the window (such as system menu, title bar, and minimize and maximize buttons), and configure the border of the window. By default, the window uses the system font and has a white background. You can change the font and color.

Canvas attributes
By default, the canvas part uses the system font, and is the same color as the folder background. You can change the font and background color of the canvas part. You can also place a graphic on the canvas part.

Subfile attributes
By default, the subfile part is created with no columns. If you know the database field names, you can create subfile entry fields using the GUI Designer. Otherwise, you can reference the existing fields in the database by following these steps:
2. Specify the iSeries 400 server and library information to view the database field information.
3. Select the appropriate fields from the Fields list box with the right mouse button, move the pointer icon onto the subfile part in the design window, and right-click again.
   The new subfile entry field inherits the attributes from the original field:
   Length is set to the column width, and Type is set to the data type.

Set the style and data type for a subfile entry field using the appropriate properties notebook. For example, you can set the length, or the type of data.

Push button attributes
Label each of the push buttons to indicate its purpose to the user. To create a mnemonic for each push button, put the mnemonic identifier before a character in the label. For Windows, use an ampersand (&). Note that we put an ellipsis ( ... ) in the label of the Review/submit order push button so that users know that they will have to provide more information after they press the button to place their order.

For each push button, specify what action will happen when the user presses it. For example, for the Preview push button, an action subroutine will be performed; for the Help push button, help for the window will be displayed. You can set this information on the Action tab of the push button’s properties notebook. (See "Creating the Online Help" on page 15 for related information.)

Adding Program Logic
Program logic is required to drive certain GUI functions. This section describes some of the Video Store Catalog’s program logic. (The source file, CATALOG.VPG, is in the Video Store Catalog folder.)

Note: You can type the program logic for a particular event by invoking an edit session from the GUI Designer. For example, to add program logic to the Press event for a particular push button, select Events>Press from the push button’s pop-up menu.

Displaying the Comedy window
To have the Comedy window appear when the user presses the Comedy push button on the Video Titles — Categories window, do the following:
1. Write an action subroutine to handle the **Press** event for the **Comedy** push button.

We wrote the COMEDYGPB action subroutine (shown in Figure 2) to handle this event. When the user presses the push button, the COMEDYGPB subroutine calls the brComedy user subroutine. This subroutine reads the database and calls another user subroutine, dspbrowse, to check if the database is empty. If it is, a message is displayed. If it is not empty, control returns to the brComedy user subroutine, the title of the window is changed, and the results of the database search are displayed.

```
** Categories window action-link subroutines
**
**
**
******************************************************************************
* This routine is executed when the Comedy graphic push button in the
* Categories window is pressed.
*
C COMEDYGPB BEGACT PRESS CATW
C z-add 0 srchdir
C z-add 0 srchact
C exsr brComedy
C ENDACT
******************************************************************************
```

*Figure 2. Handling the PRESS event*

2. Write program logic to read the Comedy video titles from the database and populate the subfile part with the list of titles. Call the dspbrowse subroutine to check whether the database is empty. If the database is not empty, set the title for the browse window to display the found comedy titles. Otherwise, display message number MSG0001 to inform the user that no match was found in the database. See Figure 3 on page 10.
We wrote an action subroutine (see Figure 4 on page 11) to handle the PRESS event for the Preview push button on the Comedy window. When the user presses the button, the PREVIEWPB action subroutine is called to start the common component that displays the Preview window.

**Displaying the Preview Window**

Figure 3. Reading the iSeries Database and showing the results window
Creating the Preview Window

The Preview window uses the multimedia capabilities of the operating system to give customers a glimpse of a video clip. This section describes how you can create a window that resembles the above.

**Note:** To run the audio in the preview, you must have a sound card on your system. To run the video clip, you must have a Media Player installed. Java applications require the Java Media Framework (JMF) API.

**Creating the GUI**

Point-and-click the following parts onto a window with canvas part to create a window that resembles the Preview window:

- Media part
- Multiline edit part
- Push button parts
- Static text parts
Setting Attributes at Design Time

After the parts are placed and positioned in the window, you can set the part attributes using their respective properties notebook. Some of the attributes you can set are described below.

**Media part attributes**
Push button parts are used to control the playback of the video clip: **Play**, **Pause**, **Record**, and **Stop**. The **AudioMode** attribute sets the operating mode of the media part.

**Static text attributes**
You can change the font attribute of a static text part to make it stand out from the other text on the display. Resize the static text part so that it is large enough to hold the longest text. (It is a good idea to leave a bit of extra room, if your application is to be translated in the future.)

**Multiline edit part**
In the code, we set the multiline edit (MLE) part called ABSTMLE to accept text. In the part’s properties notebook, we indicated that the part is read-only.

Setting Attributes at Run Time

To change the title of the displayed window at run time, we used the SETATR operation code to set the **Label** attribute (see Figure 6 on page 13).

See also “Getting and Setting Part Attributes” on page 25.

Adding Program Logic

You have to provide some program logic to drive certain GUI functions on the Preview window. This section describes some of the program logic (see Figure 6 on page 13) for the Preview component.

**Specifying the video to preview**
The video selected in the Comedy window determines which video preview is played. The **preview** action subroutine reads which video is to be used, and then determines the file name of the actual video file.

**Controlling the video**
You can write code to control video by using the media part. In our sample, the media part is used to play a digital video file associated with the video selected.

Push buttons with the associated AudioMode attribute control the playback of the video file:

1. Pause
2. Play
3. Record
4. Stop

The code for the Preview component follows:
Figure 6. The Preview Component (Part 1 of 3)
**Figure 6. The Preview Component (Part 2 of 3)**

```rpg
* PBPLAY BEGACT PRESS PREVIEWW
  C if loaded='N'
  C eval %setatr('previeww':'audio':'FileName')=videofile
  C move 'Y' loaded 1
  C endif
  C eval %setatr('previeww':'audio':'audioMode')=2
  C ENDACT
*

**Figure 6. The Preview Component (Part 3 of 3)**

```rpg
* PBPAUSE BEGACT PRESS PREVIEWW
  C eval %setatr('previeww':'audio':'audioMode')=1
  C ENDACT
*

* PBRECORD BEGACT PRESS PREVIEWW
  C eval %setatr('previeww':'audio':'audioMode')=3
  C ENDACT
*

* PBSTOP BEGACT PRESS PREVIEWW
  C eval %setatr('previeww':'audio':'audioMode')=4
  C ENDACT
*

* CANCELBP BEGACT PRESS PREVIEWW
  C move *on Flg
  C STOP
  C ENDACT
*

* PREVIEWW BEGACT CLOSE PREVIEWW
  C if Flg='ON'
  C eval Fldx='*DEFAULT'
  C else
  C eval Fldx='*NODEFAULT'
  C endif
  C ENDACT Fldx
*
**Creating Messages**

To add messages, select *Project>Define messages* from the GUI Designer. The Define Messages window appears. Select *Create*, and then select the type of message you want to create (for example, information or warning). Type the actual text for the message, and any additional information or second-level help, in the spaces provided.

VisualAge RPG automatically generates a message ID for the message you create. Reference that message ID in your code. For example, in Figure 3 on page 10, MSG0001 is used by the DSPLY operation code.

---

**Creating the Online Help**

We added different types of help to the Video Store Catalog application. The following sections describe how you can replicate some of this help.

**Context-sensitive help**

Add context-sensitive help to the *Browse by Category* graphic push button part by selecting *Help text* from the part’s pop-up menu. This starts an edit session that already contains information similar to that shown in Figure 7.

```
:h1 res=01.PSB0000C
:p.Help
```

*Figure 7. Edit session for adding online help*

The :h1 res=01. is a heading tag containing a resource identifier. The resource identifier is automatically generated — do not edit this text. The heading appears directly after this tag; it is used on the help panel and listed in the help index at run time. By default, the name of the part for which you are adding text is used as the heading. You should replace that with a heading that identifies the purpose of the help panel and is more meaningful to users. Type the actual help text after the :p. tag. By default, the word *Help* appears in the edit session.

An example of help text from the Video Store Catalog application is shown in Figure 8. The help panel that is generated from that source and displayed at run time is shown in Figure 9 on page 16.

```
:h1 res=12.Browse by Category
:p.Select this to browse videos by categories.
```

*Figure 8. Help for the Browse by Category graphic push button*
Creating Help push buttons

To create the help graphic push button at the bottom of the Preview window or the Comedy window, select a graphic push button from the parts palette with the right mouse button, move the pointer icon onto the design window, and right-click again. In the properties notebook, specify the image to be displayed on the graphic push button, and specify that you want to display help when the Press event occurs.

Figure 10 shows the source for the push button that provides help for the Welcome window. The :link. tag is used to link related pieces of help information so that users can find the appropriate information quickly and easily. You place this tag around text that is related to the help text in another panel. The text between the :link. and :elink. tags is highlighted in the runtime application (see Figure 11 on page 17). By selecting the highlighted text, the user jumps to the related target help panel. The resource id (resid) of the target panel is a parameter of the link tag.

```
:h1 res=22.Get Help on using the Catalog
:p.Select one of the graphic push buttons.
:p.
Press this button to browse by categories.
:p.
Press this button to view the new video releases.
:p.
Press this button to view the 10 best ranked.
:p.
Press this button to search for specific titles.
```

Figure 10. Help for the Welcome window
For more information about creating online help for your application, see the following topics:

- Chapter 13, “Tips for Creating Online Help with IPF,” on page 245
- Chapter 14, “Tips for Creating and Using Windows Help,” on page 249
- Chapter 15, “Tips for Creating JavaHelp,” on page 253

Figure 11. Example of help for a window that contains a hypertext link

A Review of Visual Programming

The steps described in the previous sections are similar to the ones you will take when you create your own application using VisualAge RPG. These steps are:

1. Deciding What to Show the User
   Before you begin creating a new application, you should select the purpose of your application, how it will be presented to the user, and how it will communicate with other applications.

2. Creating the GUI using the GUI Designer
   After you have designed the application, you can use the GUI Designer to create the graphical user interface. VisualAge RPG provides a catalog of GUI parts for you to choose from, and gives you the ability to create user-defined parts to suit your needs. You can select the parts that you want to appear in the interface, and select their positions on a design window. Customize the parts as required.
   See the online help for information about creating windows, adding parts to the window, and aligning and customizing them.

3. Getting and Setting Attributes
   You can set some part attributes during design time using the part’s properties notebook. You can also use GETATR and SETATR operations codes, or %getatr and %setatr built-in functions, to get or set the attributes for a part during run time. When getting or setting part attributes, you reference a part by using the name defined for it in the GUI Designer.
   For more information about the parts, and how you can get and set part attributes, refer to VisualAge RPG Parts Reference.

4. Writing Program Logic
   Each part responds to a set of predefined events. Events are typically generated as a result of some user interaction with the GUI. For example, selecting a push button signals a Press event. Events can also be generated by your program. For example, the DDE client part generates a Timeout event if it is unable to start a conversation with a server program within a predetermined period of time.
You respond to events in your program by coding the BEGACT (begin action) and ENDACT (end action) operation codes. The code between these operation codes, called an action subroutine, is executed for a particular event. If you do not code an action subroutine, no action is taken when the event occurs.

5. Adding Messages and Online Help

In addition to creating the GUI and writing some program logic to make it run, you can add messages and online help to your application.
Chapter 2. Planning Your Application

This section discusses what you should do before you begin coding a new application or converting an existing OS/400 application into a VisualAge RPG application.

If you are creating a new application, this is the time to decide on its purpose, how it will be presented to the user, and how it will communicate with other applications.

If you are planning to reuse an existing OS/400 application, this is the time to evaluate the old character screen displays and decide how to improve them using the power of graphical user interface parts. (For more information about reusing existing applications, see Part 3, “Working with iSeries Data,” on page 191.)

The information in this section will help you design an application that meets the user’s needs and is practical to implement.

Enabling Secure Java Applications

If you plan to deploy Java applications for use on the World Wide Web, note that only systems running on the OS/400, Version 4 Release 4, or later, support the Secure Sockets Layer (SSL) specification. Data flow between workstation applications and servers running on earlier OS/400 versions will not be secure.

For information on setting up SSL support for VisualAge RPG, see Appendix D, “Secure Sockets Layer (SSL) Setup,” on page 449. If your VARPG applications will be running applets that use the client security file, see “Using the Security File for Applets” on page 210.

Decide What Functions to Provide

First, determine what the main purpose of your application is, and what functions you must provide to address it. After you determine the core functions, tackle the advanced functions such as dynamic data exchange (DDE) and printing.

Help Your Users

Users will have varying degrees of experience with GUls. Consider providing online help that is tailored for the level of knowledge of a typical user. VisualAge RPG makes it easy for you to add online help to the GUI. You can add four kinds of help:

Context-sensitive help
Help information that is adapted to the current context of a choice, object, or group of choices or objects.

Task help
Information about tasks the user can perform with your application.

ToolTip help
Hover-type help about the tools available to the user.

Window-level help
Information about the contents of a window.
Other ways to help your users are to give them all the information they need to complete a task, and to provide meaningful prompts and labels on GUI parts. You can use an ellipsis (...) to indicate that more information is needed before a particular action can be performed. (For example, use Display... to tell users that they will be prompted for more information before the display action is carried out.) Do not use an ellipsis on a label if the action will be performed immediately after the button is pressed. For example, a Help button does not need an ellipsis because the help information is displayed as soon as the button is pressed. You can also provide help in the form of a static text field that is updated when the mouse pointer moves over different parts of your interface.

You can minimize the amount of information that your users have to provide by setting default values. For example, you can use a combination box part to give users the option of selecting from a list of commonly used choices. This prevents key-in errors at run time.

Keep Window Design Simple

There are two basic aspects to good window design:
• The number and structure of windows in your application
• The content of each window

Number of Windows

It is a good idea to have one main window from which the user can initiate all of the main tasks. Provide secondary windows for additional information that users must specify to complete a task.

Avoid a lot of nested windows because too many layers make a simple task look complex. Also remember that too many windows will clutter the screen, especially if the user has more than one application running. Users can also get lost if they have many windows on their screen.

Try to minimize the number of parts in each window. This will increase performance when windows are displayed. An application with many windows and few parts per window will perform better than the same application with fewer windows and more parts per window.

Content of Each Window

Group all related information in one place. Use the group box part and the outline box part to visually indicate which radio buttons are related.

Use graphic images and icons to identify tasks or to complement the words in the window. Make sure that all text is spelled correctly.

Position the parts in a window in a neat, logical manner. You do not have to plan the position of some parts because their position is predetermined. For example, a menu bar part is always located just below the window’s title bar.

If windows have common parts, you should display those parts in a consistent location. This makes it easier for users to find common information.
Plan Your Code Effectively

After you design the GUI, you must decide what code is needed to support actions that will be performed by users. VisualAge RPG helps you create the GUI without writing a lot of code — it does the routine tasks for you. All parts have default attributes, which you can modify using the GUI Designer or in your program. You have to explicitly set other attributes. For example, if you want users to be able to view graphics by pressing a push button labeled Display..., you must do more than point and click the push button part from the palette onto your design window: at the very least you must set the label attribute to read Display... and write logic to find and display the data.

Keep the User Informed

Use messages to provide particularly important or urgent information. Give detailed but not verbose messages that describe the problem and, if at all possible, explain how to correct it. VisualAge RPG provides three ways of displaying messages, and you should choose the method most appropriate for the type of information you want displayed:

Window
To provide urgent information that the user should know about; for example, a process that was not completed successfully.

Message subfile
To provide information about a choice, or to contain a message about the completion of an action or process.

Second-level message help
To provide an extra level of detail that may not be required at all times by all users; for example, to describe a course of action that novice users may not know about.

For long processes, you may want to add a progress indicator to keep the user informed.

Plan to provide text, visual, or audio cues to users to present exceptions. For example, you may want to show that a push button is not available by dimming the text on it. No one can plan for every possible user action, so you should also plan how your application will inform users about actions it cannot interpret. For example, you may want to display a message if a user tries to exit a file without saving the changes made during an edit session.

Use a Consistent Style

Use consistent terms to minimize confusion. For example, if you use Login on one window, do not use User ID to refer to the same concept somewhere else.

Use consistent mnemonics across the application windows. A mnemonic is a letter key that can be pressed to select a choice or perform an action. The letter key corresponds to the underlined letter of a choice on a push button or in a menu. For example, if you use Save to represent the save function on a push button in one window, use it on every window that has that push button.
Anticipate Translation Issues

Even if your current plans do not include translating the application into another language, you should design and create your application so that it could be easily translated in the future. By doing so you will have less to rework if the need to translate arises.

Consider keeping the executable code separate from the text. This way you can use the text in the appropriate language with the standard executable code.

**Note:** There are other reasons why you should consider separating the text from the code. You can correct errors in the text and make changes to terminology in future releases more easily.

You can create separate message files for each language and assign different file extensions to each. Each of the message files should have identical message numbers but text written in a different language. You can build one .EXE for all languages simply by using the appropriate message file. For example, an English version of the compiled message file could be named SAMPLE.ENG and a German version could be named SAMPLE.GER. You can instruct users to rename the appropriate message file to SAMPLE.MSG before running the application. For more information, see Chapter 16, “Working with Messages,” on page 259.

Also keep in mind that translation can change the sizing requirements for text (such as labels), entry fields, buffer, and windows. When sizing a part in the GUI Designer that will have a substitution label, keep in mind that translated text may be longer than the original.

If you use mnemonics, remember that the mnemonic character may be different for different languages.
Part 2. Working with Parts

Chapter 3, “Programming with Parts,” on page 25
Provides an overview of the general programming tasks you must do to drive the GUI parts.

Chapter 4, “Sample Programs for VisualAge RPG,” on page 33
Describes how to use sample programs for some VisualAge RPG parts.

Chapter 5, “Common Attributes,” on page 35
Describes attributes that are common to most parts and how you can use them.

Chapter 6, “Using Data Transfer,” on page 41
Describes how you can use data transfer to manipulate the value of some parts.

Chapter 7, “Using Parts,” on page 45
Contains helpful hints about using VisualAge RPG parts.
Chapter 3. Programming with Parts

This section presents some tips for programming with parts. Topics include how to get and set part attributes, reference parts in your program, respond to events and system attributes, work with event and system attributes, and code static text and entry field parts.

Getting and Setting Part Attributes

You can set some part attributes during design time using the part’s properties notebook.

Some attributes can be set or retrieved at run time in the application by using:

- **Qualified Attribute Names**: Part attributes can be accessed directly in expressions or free-form calculations using the qualified naming syntax of `window-name.part-name.attribute-name`.
- **Built-in functions %getatr and %setatr**: These are useful when the window or part may vary at run time, and for which the name is identified in program variables in the application code.
- **The fixed-format operation codes GETATR and SETATR** were the earliest form of accessing part attributes, and can only be used to reference parts on the same window as the part that generated the event.

For example, if the part that generated the event is on WINDOW1, the fixed operation codes can reference only parts on WINDOW1. If a GETATR or SETATR operation code references a part on another window, compile-time errors occur for single-link action subroutines because the compiler verifies that the referenced part exists on that window. Run time errors occur for multiple-link action subroutines.

To reference parts on different windows, use the other alternatives.

For more information about the attributes and where you can set them, refer to the VisualAge RPG Parts Reference.

See the VisualAge RPG Language Reference for information about using these operation codes and built-in functions.

Referencing Parts in Your Program

When getting or setting part attributes, you reference a part by using the name defined for it in the GUI Designer. The name must follow AS/400 system naming conventions. Specifically, the name:

- Must not exceed 10 characters in length. Only SBCS characters are allowed. Characters must be letters A-Z, numbers 0-9, @, #, $, or _ (underscore).
- Must begin with the letters A-Z, @, #, or $.
- Can be entered in upper case or lower case.
- Must **not** have embedded blanks.
- Must **not** be an extended name (that is, must not be in double-quotation marks).

**Note:** When your program is running, you can reference only those parts that have been **created**. Parts are created when the window they are on is also created. Creating a window or part loads it into memory. Any attempt to reference a part that is not yet created results in a **Part not found** message.
### Responding to Events

Each part responds to a set of predefined events. You can use one of the following methods to obtain a list of predefined events:

1. Refer to the *VisualAge RPG Parts Reference* for a complete list.
2. Press F1 when focus is on the part in the palette or catalog to get a general description of the part and a list of the attributes and events associated with it.
3. In the GUI Designer, invoke the pop-up menu for the part, and select the **Events** item.

Events are typically generated as the result of some interaction with the user interface. For example, pressing a push button signals a **Press** event. Events can also be generated by your program. For example, the DDE Client part generates a **Timeout** event if it is unable to start a conversation with a server program within a predetermined time period. If your program changes the text value of an entry field part, a **Change** event is signaled by the entry field.

You respond to events in your program by coding the BEGACT (begin action) and ENDACT (end action) operation codes. The code between these operation codes, called an **action subroutine**, is executed for a particular event. When you create an action subroutine for a specific event, an **action link** is defined. If you did not code an action subroutine for a particular event, no action is taken when the event occurs. The code in an action subroutine is executed until the ENDACT operation code is reached. Therefore, if you coded EXSR operation codes within an action subroutine, these subroutines (called **user subroutines**) are also executed.

You cannot invoke an action subroutine using the EXSR operation code. You can, however, invoke a particular action subroutine by more than one action. For example, you can have code that is executed when a push button is pressed or when a menu item is selected. You can review which events have action subroutines and modify link events to action subroutines in the Action Subroutines window. To display the Action Subroutines window:

1. Select **Edit source code** from the **Project** menu in the GUI Designer. This starts an edit session.
2. From the edit session, select **Edit>Action subroutines**. The Action Subroutines window appears.

**Event attributes** contain data that is relevant to an event. For example, the **MouseMove** event stores the X and Y coordinates to indicate where the mouse was located when the event occurred. Before you can use event attributes in your program, they must be defined on definition specifications. The name of the event attribute is the name of the entity on the definition specification. Because the compiler does not verify the length of the variable and some attributes have varying lengths, be sure to specify a length large enough to contain the expected value.

**Note:** Event attributes cannot be changed by your program. Therefore, they cannot appear in a result field or as the target field for an EVAL operation.

The *VisualAge RPG Parts Reference* describes all the event attributes.

The following example illustrates how the %**MouseX** and %**MouseY** event attributes can be defined and used in a program.
System Attributes

System attributes pertain to your application rather than to a specific part.

As with event attributes, system attributes must be defined on a definition specification. They cannot be modified by your program.

VisualAge RPG supports the following system attributes:

Table 1. System Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>%DspHeight</td>
<td>Returns the height of the screen at run time, in pixels.</td>
<td>Numeric</td>
<td>4</td>
</tr>
<tr>
<td>%DspWidth</td>
<td>Returns the width of the screen at run time, in pixels.</td>
<td>Numeric</td>
<td>4</td>
</tr>
</tbody>
</table>

Working with Event and System Attributes

Each event attribute is valid for a particular event and can be used only within action subroutines that are linked to that event. For example, if you use an event attribute for the MouseMove event within an action subroutine that is linked to the ReSize event, a runtime error is issued. Type checking is performed only on event attributes at run time. If you define a character field for a numeric event attribute, this error is detected only at run time.

System attributes can be used anywhere within your program because they are not linked to any particular event. Type checking is performed on system attributes at compile time.

Event and system attributes must be defined on a definition specification before they can be used throughout the VisualAge RPG component. They are treated by the compiler as read-only fields in automatic storage. Any nested and active action subroutine has its own copy of an event attribute.
For example, assume that the ENT0000A+CHANGE+WIN1 action subroutine is linked to window WIN1, entry field part ENT0000A, and event CHANGE.

Also assume that the PSB0000A+PRESS+WIN1 subroutine is linked to window WIN1, push button part PSB0000A, and event PRESS.

Each action subroutine has its own storage for %PART because event attribute fields are in automatic storage:
- In action subroutine PSB0000A+PRESS+WIN1, %PART contains ‘PSB0000A’.
- In action subroutine ENT0000A+CHANGE+WIN1, %PART contains ’ENT0000A’.
- When action subroutine ENT0000A+CHANGE+WIN1 completes and action subroutine PSB0000A+PRESS+WIN1 continues executing, %PART contains ‘PSB0000A’, not ‘ENT0000A’.

### Coding Static Text and Entry Field Parts

The following section contains some tips for coding static text and entry field parts.

#### Creating and Retrieving Entry Field Parts

**Note:** This section also applies to static text parts. For simplicity, only entry field parts are mentioned in the text.
When a READ is performed, where does VisualAge RPG store the retrieved value? When a WRITE is performed, what value does VisualAge RPG use to set the value?

For each entry field part, VisualAge RPG creates a field with the same name as the part. This field is defined to match the definition of the Text attribute (or the Label attribute for static text parts). For example, if there is an entry field part called ENT00012, and the Text attribute is defined as character 20, then VRPG automatically defines a 20-character field called ENT00012. You can use this field in your program.

You can override the definition of the field on a definition specification by defining a field of the same name. However, the definition of the field must comply with the rules of VisualAge RPG concerning type and length compatibility. For example, the field must be the same length as the attribute definition. For numeric fields, the field does not have to be the same type as the attribute definition.

When you run your application, the value of an entry field is initialized by the value that you supplied in the GUI Designer. However, you can overwrite this value by setting the INZ keyword in your definition specification, or by moving a value into the program field. In these cases, the value stored in each of these fields does not necessarily match the value that you see on the screen for the corresponding part.

If you store a different value in a field in a user or action subroutine, VisualAge RPG does not reflect that new value on the screen. Therefore, the value stored in the field is different from what is displayed on the screen. To reflect the stored value on the screen, you must use a WRITE operation or a SETATR operation.

The same holds true for SHOWWIN. When a window is first opened, the values that appear on the screen correspond to the values supplied for the parts in the GUI Designer. If you change the stored value for a corresponding VisualAge RPG field before you show the window, then the value in the field does not match what is seen on the screen. To make the two values identical, you have to perform a WRITE operation or a set attribute operation in an action subroutine that is linked to the Create event for the window. This synchronizes the value stored in the field with the value on the screen, and the user sees only the new value when the window is shown.

In general, it is a good idea to use an action subroutine that is linked to the Create event to set values that should appear on the screen when a window is opened.

**Operation Codes for Window Parts**

Several operation codes have been enhanced in VisualAge RPG to operate on windows and their parts: READ, WRITE, CLEAR, and RESET. These operation codes can be used with windows and affect static text and entry field parts.

**READ**  Performs get attribute operations on all the affected static text and entry field parts.

**WRITE**  Performs set attribute operations on all of the affected static text and entry field parts.

**CLEAR**  Sets all numeric entry field parts to zero, and all character entry field parts to blanks. (It does not operate on static text parts.)
RESET

Sets static text and entry field parts back to their initial values.

The window operation codes use these attributes:

Text    Attribute of entry field parts that is used to perform READ, WRITE, CLEAR, and RESET operations.

Label   Attribute of static text parts that is used to perform READ, WRITE, and RESET operations.

Using Window Operation Codes on Parts with Identical Names

You can have two entry fields with the same name, two static text parts with the same name, or even an entry field part that has the same name as a static text part, as long as the parts belong to different windows. This section describes how to avoid inadvertently setting the value of one of these parts to the value of another.

Only one program field is created for a given part name. If there is an entry field part in window W1 named MYPART, and an entry field part in window W2 named MYPART, then one VisualAge RPG field is created, called MYPART. The compiler creates the definition to match one of the part definitions.

If you have more than one part with the same name, the compiler will issue an error message if the parts do not have compatible definitions. Parts are compatible if they accept the same type of data (numeric or character), are the same length, and (if numeric) have the same number of decimal positions.

If the parts that share a field have different initial values, then the initial value of the field is set depending on the part the compiler encounters first when creating the internal fields for entry field parts. This can vary from one build to another, so when multiple parts share the same field you should not depend on the field having a specific initial value, unless you set all the initial values to be the same.

Performing an operation on one of the windows containing one of these parts itself, or on one of the parts, results in the entry field containing a value that matches the screen value of the part involved in the operation. However, the field contains a value that probably does not match the screen values of the other parts on other windows that share this field. Even though multiple parts share the same field, an operation on any of these parts affects only the part specified on the operation or contained in the window specified on the operation. The other parts that share the field are not affected.

Example

The following example shows what can happen when you set the value of one of the parts to a value of another part when the parts share a field.

1. Define the fields: The Entry field A01 in window W1 is defined as character 10, and the Entry field A01 in window W2 is defined as character 10. The value on the screen for W1 is 78893, and the value on the screen for W2 is 885364. Field A01 contains the value 0000000000. These are the initial values.

<table>
<thead>
<tr>
<th></th>
<th>W1</th>
<th></th>
<th>W2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A01</td>
<td>78893</td>
<td>A01</td>
<td>885364</td>
</tr>
</tbody>
</table>

Program Field

A01 0000000000
2. **Perform a READ on W1:** Field A01 now contains 78893. This matches entry field A01 in W1.

3. **Perform a WRITE on W2:** The screen value of entry field A01 in W2 is now 78893.

4. **Perform a CLEAR on W2:** Field A01 now contains blanks. This matches entry field A01 in W2. The Entry field A01 in W1 - value on screen is 78893. The Entry field A01 in W2 - value on screen is blank.

5. **Perform a GETATR on entry field A01 in W1 with target field A01:** Field A01 now contains 78893. This matches entry field A01 in W1.

If you want all the parts that share a field to display the same value, then you have to perform SETATR operations on all the parts using the field as the source value, or perform WRITE operations on all the windows that contain one of these parts.

It is recommended that you give unique names to all entry field parts in your component to avoid accidentally setting the value of one of the parts to the value of another.
Chapter 4. Sample Programs for VisualAge RPG

The Samples folder (in the VisualAge RPG projects folder) contains the source code and the runtime version of the sample applications discussed in this part of the book. Table 2 lists the sample programs.

Table 2. Sample programs for VisualAge RPG

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Animation</td>
<td>Animation control part example</td>
</tr>
<tr>
<td>ActiveX</td>
<td>ActiveX control example</td>
</tr>
<tr>
<td>Bean</td>
<td>Java bean part example</td>
</tr>
<tr>
<td>Calendar</td>
<td>Calendar part example</td>
</tr>
<tr>
<td>Component Reference Part</td>
<td>Component reference example</td>
</tr>
<tr>
<td>Container</td>
<td>Container part example</td>
</tr>
<tr>
<td>Customer Maintenance*</td>
<td>Customer maintenance example</td>
</tr>
<tr>
<td>DDE Client</td>
<td>DDE client part example</td>
</tr>
<tr>
<td>DDE Hotlink</td>
<td>DDE hot link example</td>
</tr>
<tr>
<td>Drag and Drop</td>
<td>Data transfer example</td>
</tr>
<tr>
<td>Graph</td>
<td>Graph part example</td>
</tr>
<tr>
<td>Image*</td>
<td>Image part example</td>
</tr>
<tr>
<td>Listbox</td>
<td>List box part example</td>
</tr>
<tr>
<td>Message Subfile</td>
<td>Message subfile part example</td>
</tr>
<tr>
<td>Multiline Edit</td>
<td>Multiline edit part example</td>
</tr>
<tr>
<td>Notebook</td>
<td>Notebook part example</td>
</tr>
<tr>
<td>Odbcceld</td>
<td>ODBC/JDBC interface part example</td>
</tr>
<tr>
<td>Popup Menu</td>
<td>Pop-up menu part example</td>
</tr>
<tr>
<td>Progress</td>
<td>Progress bar part example</td>
</tr>
<tr>
<td>Resize</td>
<td>Resize example</td>
</tr>
<tr>
<td>Runtime_control_of_server_connections</td>
<td>Control of server connection using the Signon API example</td>
</tr>
<tr>
<td>Scroll</td>
<td>Scroll bar part example</td>
</tr>
<tr>
<td>Slider**</td>
<td>Slider part example</td>
</tr>
<tr>
<td>Spin Button</td>
<td>Spin button part example</td>
</tr>
<tr>
<td>Subfile*</td>
<td>Subfile part example</td>
</tr>
<tr>
<td>Timer</td>
<td>Timer part example</td>
</tr>
<tr>
<td>VARPG Plug-in</td>
<td>Vendor plug-in example</td>
</tr>
<tr>
<td>Video Store Cashier*</td>
<td>Video store cashier example</td>
</tr>
<tr>
<td>Video Store Catalog*</td>
<td>Video store catalog example</td>
</tr>
<tr>
<td>Welcome</td>
<td>Welcome example</td>
</tr>
</tbody>
</table>
Notes:
1. * This example requires data on an iSeries 400 server.
2. ** Also shows how to use the BackMix and ForeMix attributes

Before You Begin

Before you can run the sample applications, you must install the VisualAge RPG component. The associated samples are in the Samples folder, which is in the VRPG Projects folder.

Read the comments in the sample programs. The comments contain tips and requirements, as well as any restrictions.

Before you can build and run Java applications, you must have Sun Microsystem’s Java 2 Software Development Kit (J2SDK) Version 1.2, or higher, installed on your workstation. If you do not have the J2SDK, you can download it from Sun Microsystems at the following URL:
http://java.sun.com/products/

After installing the J2SDK, set the PATH environment variable to point to the location of both the Java compiler and the Java Runtime Environment (JRE). For example, if your home directory for the J2SDK is c:\jdk1.2, add the following path statement: c:\jdk1.2\bin

If you plan to run VisualAge RPG applets inside a browser, the international version of the JRE must be installed on the client workstation.

Building the Examples

If you want to run most of the samples, you must first build the application.

To build one of the sample programs, display the pop-up menu for the sample’s folder, and select Build>Windows or Build>Java.

Running the Examples

To run a sample program, display the pop-up menu for the program, and select Run>Windows or Run>Java.

Accessing an iSeries 400 Server

Some sample programs, such as the subfile example, access data on an iSeries 400 server. The data files used by these programs are not shipped with VisualAge RPG. However, the source file comment section describes the file layout for that example. You must create the data file on the server and supply data.

To run these examples, start the GUI Designer on the sample program, and use the Define iSeries Information notebook in the Servers pull-down menu to do the following:
1. Change the remote location parameter to point to the server that you want to access.
2. Change the remote file name parameter so that you can access the appropriate data file for the example.

See Chapter 8, “iSeries Connectivity,” on page 193 for more information about defining iSeries 400 information.
Chapter 5. Common Attributes

This section lists the attributes that are common to most parts and describes how you can use them.

**PartName Attribute**

All parts have a name. VisualAge RPG automatically generates this name when the part is created. You can change the name of the part in its properties notebook or by editing it directly in the tree view of the GUI Designer’s project window. The “component part name cannot be edited.

*Note:* You cannot change the part name at run time.

Each window must have a unique name, and all parts on a given window must have unique names. Parts on different windows can have the same name, except for subfile part names, which must be unique across your component.

The compiler implicitly defines a field name for entry field and static text parts using the **PartName** attribute. You can use that name in your program if you want to refer to the value of these parts. For more information, see Chapter 3, “Programming with Parts,” on page 25.

If you change the name of a part, you must change all references to that part in your program source. If you attempt to reference the old name of a part that has been renamed, you will get either compile errors, or a runtime error indicating that the part could not be found.

**ParentName Attribute**

The **ParentName** attribute returns the name of the parent part. The parent is the window on which a part is placed. For a window part, the parent is the window itself.

**PartType Attribute**

You can use the **PartType** attribute to determine the type of a part in your program. **PartType** returns the type of the part as defined by VisualAge RPG. The value returned for VisualAge RPG parts consists of the string **FVDES** followed by the part type. For example, for an entry field part, the part type would be **FVDESEntryField**. One exception to this rule is the component reference part; it has a prefix of **FVDESV**.

Table 3 summarizes the **PartType** attribute value for each VisualAge RPG part.

<table>
<thead>
<tr>
<th><strong>PartType</strong> attribute</th>
<th>VisualAge RPG part</th>
</tr>
</thead>
<tbody>
<tr>
<td>FVDESOCX</td>
<td>ActiveX</td>
</tr>
<tr>
<td>FVDESAnimationControl</td>
<td>Animation control</td>
</tr>
<tr>
<td>FVDESCalendar</td>
<td>Calendar</td>
</tr>
<tr>
<td>FVDESCanvas</td>
<td>Canvas</td>
</tr>
</tbody>
</table>
Table 3. The PartType attribute for VisualAge RPG parts  (continued)

<table>
<thead>
<tr>
<th>PartType attribute</th>
<th>VisualAge RPG part</th>
</tr>
</thead>
<tbody>
<tr>
<td>FVDESCheckBox</td>
<td>Check box</td>
</tr>
<tr>
<td>FVDESComboBox</td>
<td>Combination box</td>
</tr>
<tr>
<td>FVDESContainerControl</td>
<td>Container</td>
</tr>
<tr>
<td>FVDESCVComponentReference</td>
<td>Component reference</td>
</tr>
<tr>
<td>FVDESDDEClient</td>
<td>DDE client</td>
</tr>
<tr>
<td>FVDESEntryField</td>
<td>Entry field</td>
</tr>
<tr>
<td>FVDESGraphic</td>
<td>Graph</td>
</tr>
<tr>
<td>FVDESCGraphicPushButton</td>
<td>Graphic push button</td>
</tr>
<tr>
<td>FVDESGroupBox</td>
<td>Group box</td>
</tr>
<tr>
<td>FVDESHScrollBar</td>
<td>Horizontal scroll bar</td>
</tr>
<tr>
<td>FVDESImage</td>
<td>Image</td>
</tr>
<tr>
<td>FVDESJavaBean</td>
<td>Java Bean</td>
</tr>
<tr>
<td>FVDESListBox</td>
<td>List box</td>
</tr>
<tr>
<td>FVDESAudio</td>
<td>Media</td>
</tr>
<tr>
<td>FVDESLayoutPanel</td>
<td>Media panel</td>
</tr>
<tr>
<td>FVDESMediaPanel</td>
<td>Media panel</td>
</tr>
<tr>
<td>FVDESMedialObject</td>
<td>Menu item</td>
</tr>
<tr>
<td>FVDESMultilineEdit</td>
<td>Multiline edit</td>
</tr>
<tr>
<td>FVDESNotebook</td>
<td>Notebook</td>
</tr>
<tr>
<td>FVDESNotebookPage</td>
<td>Notebook page</td>
</tr>
<tr>
<td>FVDESODBCInterface</td>
<td>ODBC/JDBC Interface</td>
</tr>
<tr>
<td>FVDESOutlineBox</td>
<td>Outline box</td>
</tr>
<tr>
<td>FVDESPopupMenu</td>
<td>Pop-up menu</td>
</tr>
<tr>
<td>FVDESProgressBar</td>
<td>Progress bar</td>
</tr>
<tr>
<td>FVDESPushButton</td>
<td>Push button</td>
</tr>
<tr>
<td>FVDESRadioButton</td>
<td>Radio button</td>
</tr>
<tr>
<td>FVDESSlider</td>
<td>Slider</td>
</tr>
<tr>
<td>FVDESSpinButton</td>
<td>Spin button</td>
</tr>
<tr>
<td>FVDESStaticText</td>
<td>Static text</td>
</tr>
<tr>
<td>FVDESStatusBar</td>
<td>Status bar</td>
</tr>
<tr>
<td>FVDESSubfile</td>
<td>Subfile</td>
</tr>
<tr>
<td>FVDESSubmenu</td>
<td>Submenu</td>
</tr>
<tr>
<td>FVDESTimer</td>
<td>Timer</td>
</tr>
<tr>
<td>FVDESVerticalScrollBar</td>
<td>Vertical scroll bar</td>
</tr>
<tr>
<td>FVDESFrameWindow</td>
<td>Window</td>
</tr>
</tbody>
</table>
Color Attributes

You can change the color of most parts by using the **BackColor** and **ForeColor** attributes. The attribute values are numbers that represent specific colors. The compiler provides a set of figurative constants, such as *RED* and *GREEN*, that you can use to set the colors. Refer to the *VisualAge RPG Language Reference* for these names.

You can specify a color mix for the part by using the **ForeMix** and **BackMix** attributes. The value of these attributes represents a mixture of the primary colors of red, green, and blue. This is often referred to as the **RGB** color value. This RGB value is a string consisting of three values separated by colons (:). Each value represents the intensity of red, green, and blue, in that order. The value of each color is between 0 and 255.

In the following code example, the background color mix of a static text part is set to a medium shade of blue:

```
CSRN01Factor1+++++++OpCode(E)+Factor2++++++Result++++++Len++D+HiLoEq..
*     'ST1'    setatr   '000:000:128' 'BackMix'
```

For a more detailed example on how to specify the **ForeMix** and **BackMix** attributes, see "Slider" on page 145.

---

**Enabled Attribute**

When a part is enabled, it can respond to user interaction and generate events. For example, when an enabled push button is pressed, it generates a **Press** event that may then be handled by your program.

You may not want a part to be enabled until a certain condition exists. In the case of the push button, you may want the user to be able to press it only when an item has been selected in a list box.

When a part is not enabled, it does not respond to user interaction, and its label isdimmed.

To enable a part, set its **Enabled** attribute value to 1. If you do not want it to be enabled, set this value to 0.

---

**Size and Position Attributes**

You can use the **Height** and **Width** attributes to indicate the size, in pixels, of most parts.

You can also use the **Left**, **Bottom** and **Top** attributes to specify the position of the part in its containing part (usually a window). The position value is also expressed in pixels. When you position any part, the values are relative to the top left corner.

These attributes are useful because they dynamically change the size and position of the parts at run time. For example:

* If a user can resize a window, you may want to code an action subroutine to handle the **ReSize** event and alter the position of the parts on that window so that they remain centered within the window. If you do not do this, the parts will remain relative to the top left corner of the window.
• If your application runs on systems that use monitors with different resolutions, you can use the `%DspWidth` and `%DspHeight` system attributes to ensure that the windows are visible regardless of the screen resolution. You may want to position the window in the center of the screen, or at some other coordinate.

Here's a calculation that can be done in the `Create` event for a window. This example calculates the appropriate coordinates to center a window called `Window1`; then moves the window to a new coordinate before displaying it on the screen.

```c
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++

C CRNR01 Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq

* * Declare the display size system attributes
D%DspHeight $ 4P0
D%DspWidth $ 4P0

* * Handle create event for Window1.
* * Gets screen size and calculates pixel coordinates to
* * position the window in the center of the screen.

C C WINDOW1 BEGACT CREATE WINDOW1

* * Get the windows dimensions:
C 'Window1' GETATR 'Width' winWidth 4 0
C 'Window1' GETATR 'Height' winHeight 4 0

* * Calculate new coordinates to center window
C %DspWidth SUB winWidth newLeft 4 0
C %DspHeight SUB winHeight newBottom 4 0
C newLeft DIV 2 newLeft
C newBottom DIV 2 newBottom

* * Center the window and make it visible
C 'Window1' SETATR newLeft 'Left'
C 'Window1' SETATR newBottom 'Bottom'
C 'Window1' SETATR 1 'Visible'
C ENDACT
```

## Visible Attribute

You can use the `Visible` attribute to specify when you want a part or a window to be displayed. For example, you may want a push button to appear on the screen only at run time. To do this, when you create the push button in the GUI Designer, go to the properties notebook for the part and set the visible flag off. Then, at run time, set the `Visible` attribute value to 1 when you want the push button to appear.

## Focus Attribute

The area of a window where a user can interact with the interface has input focus. The part that has input focus must be enabled to respond to user actions, such as the pressing of a key or a button.

There are times when you want to focus on a part in your program so that the user can use it immediately. For example, if you are checking several entry fields and require the user to re-enter information in a particular entry field, you would set the `Focus` attribute value to 1 for that entry field. When you set the attribute, the cursor appears where the user can begin typing data into the entry field.
In addition to giving focus to a particular part, you can determine if a part has gained focus. A part gains focus when the user selects it by either tabbing to it or by selecting it with the mouse. When this happens, a **GotFocus** event is generated for the part. Conversely, a **LostFocus** event is generated when a part loses focus.

### UserData Attribute

All parts support the **UserData** attribute. Use this attribute to assign any text string to a part. This string has no effect on the value of any other attribute of the part, and it is not displayed. The **UserData** attribute can contain a maximum of 65,535 characters.

### Label Attribute

Several parts have a **Label** attribute. This is descriptive text that explains the purpose of the part. The text that appears on a push button is an example of this attribute.

The following parts have a **Label** attribute:
- Check box
- Container
- Group box
- Menu item
- Push button
- Radio button
- Static text
- Window
- Window with canvas

### Label Substitution

You can substitute the text of a label by using a symbol when you set the **Label**, **TabLabel**, or **InfoLabel** attributes. This is particularly useful if you are developing applications for use with other languages, because it allows you to translate labels and messages that you defined in the GUI Designer.

When you specify label substitution for a part, an entry is made in the component’s message file. Multiple parts can use the same label substitution (for example, ^OK). The same message file entry is used for all references.

You define a substitution label for a part in its properties notebook by specifying a string, with no imbedded blanks, preceded by the caret (^) symbol. This adds a message to the message file. Select Project>Define Messages... to invoke the message editor and add the message text you want to replace the **Label** attribute when the application is run.

### Translation Tips

When sizing a part in the GUI Designer that will have a substitution label, keep in mind that translated text may be longer than the original.

If you use mnemonics, remember that the mnemonic character may be different for other languages.

You can have more than one translated message file in the runtime subdirectories by assigning different file extensions to each. For example, an English version of the compiled message file could be named SAMPLE.ENG and a German version.
named SAMPLE.GER. You can instruct the user to copy the appropriate message file to SAMPLE.MSG before running the application.
Chapter 6. Using Data Transfer

This section discusses how you can use data transfer to manipulate the value of some parts.

Note: Data transfer is not supported for Java applications.

A Typical Data Transfer Scenario

In a typical data transfer scenario the user selects a part (called the source part) with the mouse, drags it to another part (called the target part) and releases the button to drop the value on the target part. This transfers information from the source part to the target part, and the target part can then act upon that information.

Note: The part itself is not being moved with data transfer. It is the value of the part that is being transferred.

Parts That Support Data Transfer

The following table lists the parts that support data transfer, and the data that each of them acts on.

<table>
<thead>
<tr>
<th>Part</th>
<th>Data Transferred</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combination box</td>
<td>The value in the entry field portion of the combination box, or the selected item for a drop-down list type of combination box</td>
</tr>
<tr>
<td>Entry field</td>
<td>The Text attribute value</td>
</tr>
<tr>
<td>List box</td>
<td>The selected item</td>
</tr>
<tr>
<td>Message subfile</td>
<td>The selected message</td>
</tr>
<tr>
<td>Multiline edit</td>
<td>The Text attribute value</td>
</tr>
<tr>
<td>Static text</td>
<td>The Label attribute value</td>
</tr>
</tbody>
</table>

Enabling Parts for Data Transfer

If you want a part to be a source part, set the DragEnable attribute for it to a value of 1 and if you want it to be a target part, set the DropEnable attribute for it to a value of 1. You can set these attributes in the part’s properties notebook or in your program. You can not reset these attributes during run time: once you enable a part for data transfer, it remains set.

After you have set the DragEnable and DropEnable attributes, you can drag the source part and drop it on the target part. This causes the Drop event to occur for the target part.

Note: The DragEnable and DropEnable attributes are not supported in Java applications.
Data Transfer Example

In the following example, a window has two entry fields named EF1 and EF2. The **DragEnable** attribute is set for EF1, and the **DropEnable** attribute is set for EF2. The text value of EF1 can then be dragged and dropped onto EF2.

Entry field EF2 only allows certain values. In the **Drop** event for this part, the action subroutine checks that the dropped value is valid. If the value is not valid, a message is added to the message subfile part.

```
**************************************************************************
* * Program ID . . : DragDrop *
* * Description . . : Sample program to illustrate how to respond *
* * to the DROP event, and access the dropped data. *
* * Note: Drag and drop is not supported for Java *
* *
**************************************************************************
* * Define the DROP data event attribute *
* D%Data S 5A
**************************************************************************
* * Window . . : MAIN *
* *
* * Part . . . : PB_EXIT *
* *
* * Event . . : PRESS *
* *
* * Description: End the program *
* *
**************************************************************************
 C PB_EXIT BEGACT PRESS MAIN
 C move *on *inlr
 C ENDACT
```

*Figure 12. Sample Drag and Drop Code (Part 1 of 2)*
***
 Window . . : MAIN
 ***
 Part . . : EF2
 ***
 Event . . : DROP
 ***
 Description: This action subroutine will get control when a value has been 'dropped' onto the entry field EF2
 By checking the %Data event attribute, it will determine if the dropped value is allowed for the entry field and add a message to the message subfile part accordingly.

*********************************************************************
C EF2 BEGACT DROP MAIN

* Clear the message subfile
C  'Msg1' setatr 0 'RemoveMsg'
*
* Check that dropped value is allowed
C if %Data <> 'Yes ' and
C  %Data <> 'No ' and
C  %Data <> 'Maybe'
C  'Msg1' setatr 1 'AddMsgID'
C endif
*
C ENDACT
*********************************************************************

***
 Window . . : MAIN
 ***
 Part . . : EF2
 ***
 Event . . : CHANGE
 ***
 Description:

*********************************************************************
C EF2 BEGACT CHANGE MAIN
C ENDACT
*********************************************************************

Figure 12. Sample Drag and Drop Code (Part 2 of 2)
Chapter 7. Using Parts

This section contains helpful hints about using VisualAge RPG parts. Each part description contains a list of the attributes and events which apply to the part. The following parts are described in detail.

- “ActiveX” on page 46
- “Animation Control” on page 53
- “Calendar” on page 54
- “Canvas” on page 56
- “Check Box” on page 58
- “Combination Box” on page 60
- “Component Reference” on page 65
- “Container” on page 67
- “DDE Client” on page 74
- “Entry Field” on page 75
- “Graph” on page 78
- “Graphic Push Button” on page 80
- “Group Box” on page 82
- “Horizontal Scroll Bar” on page 83
- “Image” on page 84
- “Java Bean” on page 89
- “List Box” on page 92
- “Media” on page 101
- “Media Panel” on page 103
- “Menu Bar” on page 105
- “Menu Item” on page 106
- “Message Subfile” on page 108
- “Multiline Edit” on page 112
- “Notebook” on page 117
- “Notebook Page” on page 118
- “Notebook Page with Canvas” on page 119
- “ODBC/JDBC Interface” on page 120
- “Outline Box” on page 137
- “Pop-up Menu” on page 138
- “Progress Bar” on page 139
- “Push Button” on page 140
- “Radio Button” on page 142
- “Slider” on page 145
- “Spin Button” on page 151
- “Static Text” on page 155
- “Status Bar” on page 157
- “Subfile” on page 158
- “Submenu” on page 171
- “Timer” on page 172
- “Vertical Scroll Bar” on page 179
- “Window” on page 180
- “Window with Canvas” on page 181
- “Component” on page 188

Note: The parts are presented in alphabetical order, not in the order in which you must use them. You must use the window part or the window with canvas part first to start building your application, and then add other parts as required.
For more information about part attributes, events, and event attributes, see the VisualAge RPG Parts Reference. For additional programming tips, see Chapter 3, “Programming with Parts,” on page 25.

ActiveX

* Restriction: This part is unsupported in Java applications.

Use the ActiveX part to add ActiveX control objects to your project. Your applications can then access their attributes and monitor for events. (ActiveX controls are developed and provided by third party vendors.)

You must be familiar with the ActiveX controls you are adding. The VARPG GUI Designer cannot control the functions provided by these parts.

Note: VARPG only works with ActiveX controls that have interfaces written in C++. Check with your ActiveX control provider to make sure that VARPG will work with the ActiveX control you want to use.
Part Attributes

- **Activate**
- **AddEvent**
- **Bottom**
- **DeActivate**
- **HasPrpPage**
- **Height**
- **Left**
- **Method**
- **OCXProp**
- **OCXPropIdx**
- **OCXValue**
- **Parameter**
- **partindex**
- **parentname**
- **partname**
- **parttype**
- **refresh**
- **RetVal**
- **RmvEvent**
- **ShowProp**
- **Top**
- **UserData**
- **Visible**
- **Width**

Applicable Events

- **Create**
- **Destroy**
- **OCXEvent**

Adding ActiveX Controls

To add an ActiveX control to your project, click on the ActiveX part in the parts palette. Click the mouse pointer onto the design window where you want the ActiveX part placed. An Insert Object dialog appears. Select the ActiveX control you want to work with.

An ActiveX control can contain properties, methods, and events that a programmer can manipulate, call, and respond to respectively. The term **ActiveX control** refers to the actual ActiveX component you are using. Examples include a graph, calendar, or spreadsheet control. The term **ActiveX part** refers to the ActiveX part found on the VARPG parts palette.

Setting Properties

You can set the properties of an ActiveX control at build or run time. To set properties during build time, open the ActiveX Part Properties notebook from the design window. Right-click on the ActiveX icon in the design window and select **Properties**. If a properties editor is available for the ActiveX control, it will be displayed, as well. You can then directly edit the property values.

The properties, methods, and events of the ActiveX control appear on the Information page of the ActiveX Part Properties notebook. Select the appropriate radio button to view what is available.

Setting or getting a property during run time requires two steps. First, set the ActiveX part’s **OCXProp** attribute to the property name of the ActiveX control you are using. Use the **OCXValue** attribute to set or get the property.

The following example sets the Depth property for an ActiveX pie chart control:

```
C 'PieChart' Setatr 'Depth' 'OCXProp'
C 'PieChart' Setatr DepthValue 'OCXValue'
```

**Note:** The **OCXValue** attribute takes a string value. The VisualAge RPG run time will handle the appropriate conversions before forwarding the value to your ActiveX control.

The **OCXPropIdx** attribute sets or retrieves the index for an ActiveX string array property type. You can use this attribute together with the **OCXProp** and **OCXValue** attributes to manipulate string array property types.

For example, the following code sets the first element of the property **DataFiles** to `c:\temp\Sample.mdb`, that is, `DataFiles[0] = 'c:\temp\Sample.mdb'`.

Chapter 7. Using Parts  47
To set an element of a multi-dimensional array property, pass the index value for each element in a string as 'n1 n2 n3', where n1 is the index for element 1 and so on. For example, the following code sets 3DImageData[2][4][7] to 200:

```
C EVAL %SETATR('WINNEWJOB':ocxname:'OCXPROP')=
C '3DImageData'
C EVAL %SETATR('WINNEWJOB':ocxname:'OCXPROPIDX')='2 4 7'
C EVAL %SETATR('WINNEWJOB':ocxname:'OCXVALUE')
C = 200
```

Each time the OCXProp attribute is set, the internal index value for an array at runtime is reset to NULL. So, to set or get the array properties, first set OCXProp to the property name. Then, set OCXPropIdx to the index value in the array. Lastly, set or get the OCXValue attribute.

The ActiveX part properties notebook shows the property types in the Information tab. If the type of a property is a string array, the Information tab displays String[][]...[], where the number of [] pairs indicates the dimension of the array. For a numeric array, Numeric[][]...[] is displayed. For a non-array property, either "Numeric" or "String" is displayed. In the following example, the properties are all string arrays:

```
Calling Methods
```

The VisualAge RPG IDE attempts to compile a listing of available methods for your ActiveX control. This list is available on the Information page of the ActiveX Part Properties notebook. For each method, the parameters are shown with brace brackets containing IN for an input parameter, OUT for an output parameter, or nothing if no information is available.

**Note:** It may not be possible for the builder to discover all available methods for your ActiveX control. Consult the control's documentation for a complete listing.
Calling an ActiveX control’s methods is performed at run time by setting the ActiveX part’s **Method** attribute. The **Method** attribute takes a string value containing the method name, followed by zero or more parameter values separated by a comma. The syntax is:

```
method_name, value1, value2, ...
```

The following example calls the *AboutBox* method of an ActiveX pie chart control. The method takes no parameters.

```c
C 'PieChart' Setatr 'AboutBox' 'Method'
```

The next example calls the *AddData* method of an ActiveX pie chart control. This method takes two parameters: a string label and a float value.

```c
D datax C const('AddData, ItemX, 3.0')
C 'PieChart' Setatr datax 'Method'
```
Responding to Events

VisualAge RPG can respond to any events generated by the ActiveX control you are using. ActiveX control events are accepted by VisualAge RPG as an **OCXEvent**. You can use the %RealName event to retrieve the actual name of the event.

To receive an event from an ActiveX control, you must first register the event with the VisualAge RPG run time. Set the ActiveX part’s AddEvent attribute to the string name of the event to be received. You can also set the AddEvent attribute to *ALL to receive all events generated by the ActiveX control. To unregister an event, set the RmvEvent attribute to the string name of the event.

* Declare the event attribute
D %RealName s 20a

* Register the event with the VisualAge RPG run time
C  'DataQ' Setatr 'Click' 'AddEvent'

* Respond to the OCXEvent
C DATAQ  BEGACT  OCXEVENT  FRA000000B
C  if  %RealName = 'Click'
* Do something here
C  endif

When an ActiveX part fires an event, the event may contain some parameters. Prior to WDSC V5.1, these parameters were ignored. They were not available to the application. Now, with the introduction of these two attributes in V5.1, the event parameters are available to the application.

To access a parameter, these are the steps:

1. When an OCXEVENT fires, in its event action-subroutine, the parameter index should be set first:
   C  'ocx'  setatr  2  'PARAMINDEX'

2. Get the parameter and put it into a Character variable "param":
   C  'ocx'  getatr  'Parameter'  param

3. If the parameter is a numeric value, convert "param" into numeric format.

The above steps are for accessing a non-pointer parameter. For pointer parameters, the following must be done:

Take for example an ActiveX control, DieRoll, which simulates a dice roll, which has the event,
   BeforeRollTo(PTR to Variant number, PTR to Boolean useNumber )

The above info shows that the event has 2 parameters. The first one, number, is a pointer to a Variant structure which contains a bstrVal member, and this bstrVal may contain a value from "one" to "six", the number of dots on a dice. The value of vstrVal can be retrieved and be changed to a different number in the action subroutine. The second one, useNumber, is a pointer to a Boolean value. If this value is set to be "true", the Dice control will roll to the number set here (e.g., suppose number="Four"; if it is set to "five", the dice will display 5 dots. If there is no change, the dice will display 4 dots).

Please note that in order to pass a parameter back to the control, the control must define the parameter as a pointer (to desired types), and allow the pointer to be
passed back (some pointer parameters are for "read" only, applications can not change their value. This is controled by the ActiveX control developer.)

How to access and change the BOOL parameter of the event:

```
#include <windows.h>

int main()
{
    int result = 0;
    // Code...
    return result;
}
```

To access and change the first parameter of the event, it is helpful to look at the VARIANT structure first:

```
struct VARIANT {
    VARTYPE vt; // Indicate data type of this structure. VT_BSTR(8) is a string.
    unsigned short wReserved1;
    unsigned short wReserved2;
    unsigned short wReserved3;
    union {
        unsigned char bVal; // VT_UI1
        short iVal; // VT_I2
        long lVal; // VT_I4
        float fltVal; // VT_R4
        double dblVal; // VT_R8
        VARIANT_BOOL xbool; // VT_BOOL
        DATE date; // VT_DATE
        BSTR bstrVal; // VT_BSTR
        short FAR* piVal; // VT_BYREF|VT_I2
        long FAR* pIVal; // VT_BYREF|VT_I4
    }
}
```

NOTE: Refer to MicroSoft's ActiveX control document for more info.

In this case, "number" is defined as a string, so the type, vt, is VT_BSTR (which is 8), and "bstrVal" member in the union will point to a string (like "five").

When the event fires, the first parameter will point to a structure like this. Use this pointer, the values in the structure can be changed. This is how to do it:

```
#include <windows.h>

int main()
{
    VARIANT var;
    // Code...
    return 0;
}
```

Chapter 7. Using Parts 51
d typeStrParam 1 2b 0
d reserved1Param 3 4b 0
d reserved2Param 6b 0
d reserved3Param 7 8b 0
d bstrValParam * String

d B0 9 12b 0
d padding2Param 13 16b 0

d=Change the 'bstrValParam' member of the structure.

d NewBSTRVal s 20A based(bstrValParam)

c OcX2 BegAct OcxEvent MAIN

c if %realname='BeforeRollToVariant'

c*Get the pointer (char string), and transform it into a pointer.

c*The first parameter is a pointer to a VARIANT structure.

c 'ocx2' setatr 1 'paramindex'

c*Please note that the length of the returned "PARAMETER" string

c*varies. E.g., the length of the first parameter here is 8, while

c*the second one is 7. So, a long-enough variable need to be

c*defined to receive the string, and it's need to be processed so

c*that there's no SPACE characters in it before it is converted

c*to a numeric value.

c 'ocx2' getatr 'Parameter' param

c eval PTR_Value = 0

c move param PTR_Value

c*Change the value of 'bstrValParam' member of the structure.

c eval strFour='four'+"00"

c eval NewBSTRVal=strFour

This way, the contents of the structure pointed to by the first parameter can be changed. The above code will make the Dice display 4 dots.
Animation Control

In Windows applications, the animation control part plays video files with the AVI extension. This part differs from the media part in that the video is actually played independently of the program logic. One typical use of this part is to display an AVI file that shows some progress, such as a file being moved from one folder to another.

The animation control part plays video files with no sound. The AVI file cannot be in compressed format, unless it was compressed with the Running-Length Encoded (RLE) method.

In Java applications, the animation control part is used to play back an animated GIF file sequence using the NbrOfImage attribute.

Part Attributes

<table>
<thead>
<tr>
<th>FileName</th>
<th>FrameRate</th>
<th>Handle*</th>
<th>Left</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode</td>
<td>NbrOfImage</td>
<td>ParentName</td>
<td>PartName</td>
</tr>
<tr>
<td>PartType</td>
<td>Top</td>
<td>UserData</td>
<td>Visible</td>
</tr>
</tbody>
</table>

* Note: See the attribute description for restrictions.

Applicable Events

Create          Destroy
Calendar

The calendar part represents a monthly calendar. By clicking on one of the month arrows, the user can navigate the calendar by going to the next or previous month.

You also have complete program control over the calendar such as going to a specific date, determining which date the user has selected, and adding short text comments to individual days in the calendar.

**Part Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Border</td>
<td>Bottom</td>
</tr>
<tr>
<td>ClearMonth</td>
<td>ClearYear</td>
</tr>
<tr>
<td>ColorMix</td>
<td>Date</td>
</tr>
<tr>
<td>DateUnder</td>
<td>Day</td>
</tr>
<tr>
<td>DayNumPos</td>
<td>DayNumRect</td>
</tr>
<tr>
<td>FontArea</td>
<td>FontBold</td>
</tr>
<tr>
<td>FontSize</td>
<td>FontStrike*</td>
</tr>
<tr>
<td>Handle*</td>
<td>Height</td>
</tr>
<tr>
<td>Month</td>
<td>MonthArrow</td>
</tr>
<tr>
<td>OutlineRcl</td>
<td>ParentName</td>
</tr>
<tr>
<td>Refresh</td>
<td>ShowRects</td>
</tr>
<tr>
<td>Top</td>
<td>UserData</td>
</tr>
<tr>
<td>WeekDay</td>
<td>WeekDayIdx</td>
</tr>
<tr>
<td>YearIdx</td>
<td>YearLen</td>
</tr>
<tr>
<td>ClearAll</td>
<td>Color</td>
</tr>
<tr>
<td>ColorArea</td>
<td>DateText</td>
</tr>
<tr>
<td>DayLen</td>
<td>Enabled</td>
</tr>
<tr>
<td>FontItalic</td>
<td>FontName</td>
</tr>
<tr>
<td>FontUnder*</td>
<td>FrmtString</td>
</tr>
<tr>
<td>HRule</td>
<td>Left</td>
</tr>
<tr>
<td>MonthLen</td>
<td>PartType</td>
</tr>
<tr>
<td>PartName</td>
<td>TipText</td>
</tr>
<tr>
<td>PartType</td>
<td>VRule</td>
</tr>
<tr>
<td>Visible</td>
<td>Year</td>
</tr>
</tbody>
</table>

* Note: See the attribute description for restrictions.

**Applicable Events**

<table>
<thead>
<tr>
<th>Event</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Click</td>
<td>Create</td>
</tr>
<tr>
<td>MouseDown</td>
<td>Destroy</td>
</tr>
<tr>
<td>MouseUp</td>
<td>MthChange</td>
</tr>
<tr>
<td>MouseEnter</td>
<td>MouseExit</td>
</tr>
<tr>
<td>MthChange</td>
<td>YearChange</td>
</tr>
</tbody>
</table>

**Determining Which Date the User Selected**

The **DateUnder** attribute can be used to determine which date the mouse pointer is over. In the following example, the date is being retrieved when the user clicks on the calendar. Note that we are checking that the **DateUnder** value is not blank. This is because the click event occurs no matter where the mouse is. If the mouse is not over a specific date, the **DateUnder** attribute will be blank.

Also note that the date is returned as character string in the form of YYYYMMDD:
Using Date Index Attributes

Several attributes are provided that allow you to access the calendar without affecting what is currently being displayed. In this example, a comment is being added to a specific day in the calendar.

* Set index to date to reference
  C 'Call'  Setatr '19971210' 'DateIdx'

* Set comment for the day
  C 'Call'  Setatr 'Vacation' 'DateText'

Figure 13. Example of the calendar part

Figure 14. Adding a comment
Canvas

Use the canvas part with a window or a notebook page if you want to place more than one part on your window or notebook page. You can point and click various parts onto the canvas, position them, and organize them to produce a graphical user interface.

The canvas part occupies the client area of either a window or a notebook page. If there is no canvas in your window or notebook page, then you can put only one part on the client area, unless the parts are extensions to the window, such as menu bars and message subfiles.

More often than not, you will be creating windows and notebook pages with more than one part on them. In that case, you should use the notebook page with canvas part and the window with canvas part. They save you a step because they already contain the canvas part.

At build time, you can also include a bitmap image as the canvas background by specifying the FileName attribute. This image can be tiled by setting the Tile attribute. For Java applications, you can include GIF or JPG images as the background.

Notes:
1. The canvas part, the window (without canvas) part, and the notebook page (without canvas) part are located on the parts catalog, not the parts palette. If you want to use them frequently, you can add them to the parts palette.
2. If parts located on a canvas part have the default font setting specified (Default System Font), they will inherit the font definition specified for the canvas part. To apply a certain font to the majority of parts on a canvas, specify that font for the canvas part rather than for each individual part.

For related information, see:
- “Window” on page 180
- “Window with Canvas” on page 181
- “Notebook Page” on page 118
- “Notebook Page with Canvas” on page 119.
## Part Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BackColor</td>
<td>BackMix</td>
</tr>
<tr>
<td>FileName</td>
<td>FontBold</td>
</tr>
<tr>
<td>FontSize</td>
<td>FontStrike*</td>
</tr>
<tr>
<td>Height</td>
<td>Left</td>
</tr>
<tr>
<td>PartType</td>
<td>ReadOnly</td>
</tr>
<tr>
<td>Top*</td>
<td>UserData</td>
</tr>
</tbody>
</table>

* Note: See the attribute description for restrictions.

## Applicable Events

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Click</td>
<td>Create</td>
</tr>
<tr>
<td>MouseDown</td>
<td>MouseMove</td>
</tr>
<tr>
<td>VKeyPress</td>
<td></td>
</tr>
</tbody>
</table>
Check Box

Use the check box part if you want the user to choose between two clearly distinguishable states; for example, on and off.

A label associated with the check box describes what its setting is when it is selected.

Typically, you use a group of check boxes to provide a list of options. The user can select one or more check boxes, or not select any. The options are not mutually exclusive; therefore, selecting one check box has no effect on others on the window. If you want the user to be able to select only one option from two or more, use radio buttons instead. See “Radio Button” on page 142 for more information.

To set the state of a check box, the user can either click on it with the mouse, press the space bar on the keyboard when the check box is in focus, or (if you have assigned one) press the mnemonic key.

**Part Attributes**

<table>
<thead>
<tr>
<th>AddLink*</th>
<th>AllowLink*</th>
<th>BackColor</th>
<th>BackMix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bottom</td>
<td>Checked</td>
<td>Enabled</td>
<td>Focus</td>
</tr>
<tr>
<td>FontBold</td>
<td>FontItalic</td>
<td>FontName</td>
<td>FontSize</td>
</tr>
<tr>
<td>FontStrike*</td>
<td>FontUnder*</td>
<td>ForeColor</td>
<td>ForeMix</td>
</tr>
<tr>
<td>Handle*</td>
<td>Height</td>
<td>Highlight*</td>
<td>Label</td>
</tr>
<tr>
<td>Left</td>
<td>ParentName</td>
<td>PartName</td>
<td>PartType</td>
</tr>
<tr>
<td>Refresh</td>
<td>RemoveLink*</td>
<td>ShowTips</td>
<td>TipText</td>
</tr>
<tr>
<td>Top</td>
<td>UserData</td>
<td>Visible</td>
<td>Width</td>
</tr>
</tbody>
</table>

* Note: See the attribute description for restrictions.

**Applicable Events**

<table>
<thead>
<tr>
<th>Create</th>
<th>Destroy</th>
<th>MouseEnter</th>
<th>MouseExit</th>
</tr>
</thead>
<tbody>
<tr>
<td>MouseMove</td>
<td>Popup</td>
<td>Select</td>
<td></td>
</tr>
</tbody>
</table>
Setting the State of a Check Box Part

Set the Checked attribute to one of the following values to describe the state of the check box:

1  The check box is set and the state is turned on.
0  The check box is not set and the state is turned off.

The check box contains a check mark when its state is on.

Setting a Mnemonic

Note: Mnemonics are not supported in Java applications.

To specify a mnemonic key for the check box, place the mnemonic identifier in front of a character in the check box label. For Windows, use an ampersand (&). This character is the mnemonic key and will be displayed on your interface with an underscore (for example, Visible). The underscore informs users that they can select the check box by pressing the underlined character on the keyboard.

Signaling Events

When the user selects a check box to turn the state either on or off, a Select event is signaled.
Combination Box

A combination box provides the user with the option of entering specific information, or selecting from a list of commonly used choices.

It consists of an entry field with an attached list box that presents a list of values which the user can scroll through. If the user selects one of these values, it will appear in the entry field and replace any existing text. Alternatively, the user can type a value, that does not have to match any of the listed ones, directly into the entry field.

Combination boxes come in different styles. You can select the style you want from the part’s properties notebook.

For related information, see:
- “List Box” on page 92
- “Entry Field” on page 75

Part Attributes

<table>
<thead>
<tr>
<th>AddItemEnd</th>
<th>AutoScroll*</th>
<th>BackColor</th>
<th>BackMix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bottom</td>
<td>Case*</td>
<td>Count</td>
<td>DelimChar</td>
</tr>
<tr>
<td>DeSelect*</td>
<td>DragEnable*</td>
<td>DropEnable*</td>
<td>Enabled</td>
</tr>
<tr>
<td>FieldExit</td>
<td>FirstSel</td>
<td>Focus</td>
<td>FontBold</td>
</tr>
<tr>
<td>FontItalic</td>
<td>FontName</td>
<td>FontSize</td>
<td>FontStrike*</td>
</tr>
<tr>
<td>FontUnder*</td>
<td>ForeColor</td>
<td>ForeMix</td>
<td>GetItem</td>
</tr>
<tr>
<td>Handle*</td>
<td>Height</td>
<td>Index</td>
<td>InsertItem*</td>
</tr>
<tr>
<td>ItemKey</td>
<td>Left</td>
<td>ParentName</td>
<td>PartName</td>
</tr>
<tr>
<td>PartType</td>
<td>ReadOnly</td>
<td>Refresh</td>
<td>RemoveItem</td>
</tr>
<tr>
<td>Search*</td>
<td>SearchType*</td>
<td>Selected</td>
<td>SelectItem</td>
</tr>
<tr>
<td>Sequence*</td>
<td>SetItem</td>
<td>SetTop*</td>
<td>Showtips</td>
</tr>
<tr>
<td>SizeToFit*</td>
<td>Text</td>
<td>TipText</td>
<td>Top</td>
</tr>
<tr>
<td>UseDelim</td>
<td>UserData</td>
<td>Visible</td>
<td>Width</td>
</tr>
</tbody>
</table>

* Note: See the attribute description for restrictions.
Applicable Events

<table>
<thead>
<tr>
<th>Change</th>
<th>Create</th>
<th>Destroy</th>
<th>Drop*</th>
</tr>
</thead>
<tbody>
<tr>
<td>DropDown*</td>
<td>Enter</td>
<td>GotFocus</td>
<td>KeyPress</td>
</tr>
<tr>
<td>LostFocus</td>
<td>MouseEnter</td>
<td>MouseExit</td>
<td>MouseMove</td>
</tr>
<tr>
<td>Popup</td>
<td>Select</td>
<td>VKeyPress</td>
<td></td>
</tr>
</tbody>
</table>

* Note: See the event description for restrictions.

Selecting the Type of Combination Box

In the combination box properties notebook, choose the type of combination box you want to create:

Combination box
Displays both the entry field and the list box portions of the combination box.

Drop-down combination box
Displays the entry field portion. The list box portion is hidden until the down arrow (↓) is pressed.

Drop-down list combination box
Displays the entry field portion. The list box portion is hidden until the down arrow (↓) is pressed. The entry field does not accept text; it is used only to display the selection from the list box portion.

Note: All types of combination boxes support dragging from the entry field portion. Only the simple combination box supports dragging from the list box portion. All types support dropping onto the entry field portion, not onto the list box portion.

Adding and Setting the Initial Sequence of Items

You can use the combination box properties notebook to place an initial list of items in a combination box at design time.

By default, items are displayed in the combination box in the order in which you added them. If you want them displayed in a more precise order, then before you start adding them, set the Sequence attribute to either ascending, descending, or index. This sorts the items in ASCII collating sequence as they are added.

You cannot use the Sequence attribute to change the order of items that are already in the combination box.

Adding Items at Run Time

You can insert items into a combination box at run time by using the InsertItem attribute. The order in which items are displayed is determined by the Sequence attribute.

Updating Items in a List

You can update items that are already in the list. Use the Index attribute to indicate which item you want to change, and then use the SetItem attribute to set the changed data.

Note: The SetItem attribute replaces an item to its original location in a list, regardless of the value you set for the Sequence attribute. For example, if
you had set the Sequence attribute to ascending order before the combination box was filled, the items appear in the combination box in ascending sequence; however, if you then retrieve an item, change its value, and use the SetItem attribute to replace it in the combination box, the item is inserted in the same position it was in before. Therefore, the list may or may not be in ascending sequence after the change.

**Setting the Top of the List**
Use the SetTop attribute to specify which list item should appear at the top of the combination box. Setting this item does not reorder the items in the list; it scrolls the list so that the item you select is displayed at the top, followed by the items that came after it.

**Removing Items**
Use the RemoveItem attribute and the Index value of the item you want to remove. Index values start at 1. When an item is removed from the list, all items following the removed item are moved up one position in the list.

To remove all items in the list, specify an Index value of 0.

**Selecting and Deselecting Items**
The user can select and deselect items by using the mouse or the keyboard. You can select and deselect items using the Selected and DeSelect attributes in your program. Before you use these attributes, set the Index attribute.

**Retrieving a User-Selected Item**
When the user selects an item from the list in a combination box, that item is placed into the entry field. You can use the Text attribute to get the item. Also, you can use the FirstSel attribute to determine the index of the item that was selected.
The user may also type a value into the entry field portion of the combination box. This value does not have to be one of those in the list. If you want the user to be able to select only items that are in the list, set the **ReadOnly** attribute value to 0.

You can use the **Count** attribute to determine if there are items to retrieve.

### Using Keys

Both the list box and the combination box allow you to add items to the list that consist of a 'key' portion and a 'data' portion. When items are added to the list, only the data portion of the item is displayed. When the user selects an item you can programatically retrieve the key portion of the item.

To enable keys in a list, you must check the 'Use separator' check box on the 'Separator' page of the parts settings notebook and specify a separator character. The default separator character is the semicolon (;). The items in the list consist of the key portion, followed by the separator, followed by the data portion. For example:

01;Shipping

As an example, assume you wish to display a list of departments in a list allowing the user to make a selection. In your database you store the department as a 2 character field but you want the user to see the descriptive name. You would add the following data to the list:

- 01;Shipping
- 02;Manufacturing
- 03;Payroll
- 04;Distribution

**Note:** With the combination box you can add the default list on the Data page of its settings notebook.

When the user makes a selection from the list the following code could be used to get the key portion of the item

```c
C 'Combo1' Getatr 'FirstSel' X 2 0
C 'Combo1' Setatr X 'Index'
C 'Combo1' Getatr 'ItemKey' Key
```
Setting the Entry Field Text

When a combination box is first displayed, its entry field is blank. If you want to place one of the list items in the entry field, set the SelectItem attribute value with the index of the item to be used.

Signaling Events

The Select event is signaled when:
• The user selects an item that is in a combination box.
• You select an item in the list in your program.
• The user selects an item that is already selected.

The Enter event is signaled when:
• The user double-clicks on an item that is in the combination box
• The Enter key is pressed when the list box has focus and an item has been selected.

In your action subroutine for these events, you can use the Selected attribute to determine which item was selected.
The component reference part enables one VARPG component to communicate with another. You use the component reference part to affect a part on the other component. The component being referenced must be running in the same application as the component reference part.

The component reference part also monitors a specified event in the other component. When the monitored event occurs, a Notify event is signaled by the component reference part.

**Part Attributes**

<table>
<thead>
<tr>
<th>AddSrcEvt</th>
<th>AttrValue</th>
<th>Bottom</th>
<th>CompName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>NotSrcEvt</td>
<td>NotSrcPart</td>
<td>NotSrcWin</td>
</tr>
<tr>
<td>ParentName</td>
<td>PartName</td>
<td>PartType</td>
<td>RefAttr</td>
</tr>
<tr>
<td>RefParent</td>
<td>RefPart</td>
<td>RmvSrcEvt</td>
<td>UserData</td>
</tr>
<tr>
<td>Visible</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Applicable Events**

- Create
- Destroy
- Notify

**Referencing Part Attributes in Other Components**

There are two methods you can use to reference an attribute of a part in another component:

1. Define the attribute in the properties notebook of the component reference part.
2. Set the appropriate component reference part attributes at run time.

Before you can reference part attributes in another component, you must ensure that the other component is running. Use the START operation code to start another component.

The following code fragment illustrates how a component reference part in one component can change the value of a part attribute in another. In this example, the FileName attribute of an image part (IMG1) on window WIN01 in component COMPB is being updated with a new value.

```c
* Change the bitmap for image part IMG1 on
* window WIN01 in component COMPB
C 'CR1' Setatr 'COMPB' 'CompName'
C 'CR1' Setatr 'WIN01' 'RefParent'
C 'CR1' Setatr 'IMG1' 'RefPart'
C 'CR1' Setatr 'FILENAME' 'RefAttr'
C 'CR1' Setatr 'D:\PIC.BMP' 'AttrValue'
```
Monitoring for Events in Another Component

You can use the component reference part in one component to monitor for an event that occurs in another component running in the same application. Define the event to be monitored in the component reference part’s properties notebook, or at run time by setting the appropriate attributes. When the event being monitored in the other component occurs, a Notify event is signaled by the component reference part.

The following code fragment shows how a component reference part can be set at run time to monitor for an event in another component. In this example, the event being monitored is the Press event for a push button called PB1 on window WIN01 in component COMPB.

```
* Monitor for the PRESS event of push button
* PB1 on window WIN01 in component COMPB
C 'CR1' Setatr 'COMPB' 'CompName'
C 'CR1' Setatr 'PRESS' 'NotSrcEvt'
C 'CR1' Setatr 'PB1' 'NotSrcPart'
C 'CR1' Setatr 'WIN01' 'NotSrcWin'
*
Container

Use the container part to store related records. The records can be shown in an icon view, tree view, text tree view, or details view.

**Part Attributes**

<table>
<thead>
<tr>
<th>AddRcd</th>
<th>Arrange</th>
<th>BackColor</th>
<th>BackMix</th>
</tr>
</thead>
<tbody>
<tr>
<td>BlankChar</td>
<td>Bottom</td>
<td>ChildCount</td>
<td>ChildList</td>
</tr>
<tr>
<td>Collapsed</td>
<td>ColNumber</td>
<td>Count</td>
<td>DeleteRcd</td>
</tr>
<tr>
<td>DeSelect</td>
<td>EditItem</td>
<td>Enabled</td>
<td>ExtSelect*</td>
</tr>
<tr>
<td>FirstSel</td>
<td>Focus</td>
<td>FontBold</td>
<td>FontItalic</td>
</tr>
<tr>
<td>FontName</td>
<td>FontSize</td>
<td>FontStrike*</td>
<td>FontUnder*</td>
</tr>
<tr>
<td>ForeColor</td>
<td>ForeMix</td>
<td>GetNewID</td>
<td>GetRcdFld</td>
</tr>
<tr>
<td>GetRcdIcon</td>
<td>GetRcdText</td>
<td>Handle*</td>
<td>Height</td>
</tr>
<tr>
<td>InUse*</td>
<td>Label</td>
<td>Left</td>
<td>MinIcon</td>
</tr>
<tr>
<td>ParentId</td>
<td>ParentList</td>
<td>ParentName</td>
<td>PartName</td>
</tr>
<tr>
<td>PartType</td>
<td>ReadOnly</td>
<td>RecordID</td>
<td>Refresh</td>
</tr>
<tr>
<td>RemoveRcd</td>
<td>Selected</td>
<td>SelectRcd</td>
<td>SetRcdFld</td>
</tr>
<tr>
<td>SetRcdIcon</td>
<td>SetRcdText</td>
<td>SetTop*</td>
<td>ShowTips</td>
</tr>
<tr>
<td>SortAsc</td>
<td>SortDesc</td>
<td>TipText</td>
<td>Top</td>
</tr>
<tr>
<td>UserData</td>
<td>View*</td>
<td>Visible</td>
<td>VisTitle</td>
</tr>
<tr>
<td>VisTitlSep</td>
<td>Width</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* **Note:** See the attribute description for restrictions.

**Applicable Events**

- Click
- Click
- DblClick
- GotFocus
- MouseEnter
- Popup
- Collapsed
- Destroy
- KeyPress
- MouseExit
- Select
- ColSelect
- Enter
- LostFocus
- MouseMove
- VKeyPress
- Create
- Expanded
- MouseDown
- MouseUp

**Adding Columns to a Container**

In a details view, a record corresponds to a row in the container part, and each column in that row corresponds to a field in that record. Before you can add a record to a container, you must add the columns required to display the fields by using the container part’s properties notebook.

You have to specify which of the following four types of columns you want to create:

**Object text**

An object text column displays the descriptive text that is specified when the record is created. You can change the text at run time with the `SetRcdText` attribute. Users can change this text at run time by pressing the Alt key and selecting the field with the mouse. To get the value of this column, use the `GetRcdText` attribute.
Object icon

An object icon column displays the icon file that is specified when the record is created. You can change the icon file name at run time using the SetRcdIcon attribute.

Text

Text is a string containing additional information. A text column can contain any string value. The text cannot have any blanks in it. If you want a blank to appear at run time, use an underscore (_) character in the string. Use the SetRcdFld attribute to change this text at run time. Users can change the text at run time by pressing the Alt key and selecting the field with the mouse. To get the value of this column, use the GetRcdFld attribute.

Icon

An icon shows additional graphical record information. An icon column displays an icon file. Use the SetRcdIcon to change the icon file name at run time.

You can add up to 20 columns to a container part. Of these, up to 15 can be some combination of object text and text columns, and up to five can be object icon columns and icon columns. Note that under Windows, only the first column can contain an icon.

The number of columns and column types cannot be changed at run time. If you add records that have more fields than the number of columns in the container, the extra fields are ignored.

Adding Records to a Container

In your code, use the AddRcd attribute to add records to a container. This attribute consists of a string of the following:

ID Text FileName ParentID {field_data field_data ...}

Blanks are used as delimiters. The parameters are:

ID

A unique numeric value you give to a record. This number must be greater than zero. To ensure that you assign a unique ID to each record you create, use the GetNewID attribute.

Text

The text that is displayed with the object icon. The text can be changed at run time by using the SetRcdText attribute.

FileName

The name of a file containing the icon image for an object icon column. This icon is displayed in the container’s Icon and Tree views. You can change the icon file name at run time using the SetRcdIcon attribute.

ParentID

The unique ID of the record under which this record will appear. If this record does not have a parent record, put a 0 in this field.

Field_data

Additional information for a record that is displayed in a text or icon column. Each field_data value updates a corresponding column in the container part. If you want to have an empty text column or icon container column, you must specify an underscore (_) in the corresponding field_data parameter.

The following code fragment shows the parameters that are specified to add a sample record to a container. No column data is added in this example.
Use the **Count** attribute to determine how many records are in a container.

### Updating Container Columns

Once a record has been added to a container, the data in the record fields is displayed in the corresponding columns of the container. You can update data in individual container text columns by updating the record fields.

**Note:** You can update only the data in the columns; you cannot change the number of columns in the container. The number of columns is set when you create the container in the GUI Designer.

To update a column, set the **RecordID** attribute to the record that corresponds to that column, and set the **ColNumber** attribute to the field on that record that contains the updated data. The following code fragment illustrates how to update the third column in a container:

```c
* * Set the record id to be referenced
C 'CT1' Setatr NextIDN 'RecordID'
* * Reference the third column in the record
C 'CT1' Setatr 3 'ColNumber'
* * Update the column with the new data
C 'CT1' Setatr 'Larry' 'SetRcdFld'
```

If you want the new column value to contain imbedded blanks, use the underscore character to represent each blank. The underscore characters are replaced by blanks when the column is updated. For example:

```c
* 'New data' is set in the column
C 'CT1' Setatr 'New_data' 'SetRcdFld'
```
Use the GetRcdFld attribute to retrieve the contents of a record field. Set the RecordID attribute to the unique ID of the record, and the ColNumber attribute to the desired column number.

Removing Records from a Container

Use the RemoveRcd attribute to remove records from a container part and to remove the record ID that uniquely identifies that record. To remove all records in the container, set the record ID value to zero.

The following code fragment illustrates how a record is removed from a container:

```
* * Get ID of first selected record
C 'CT1' Getatr 'FirstSel' TmpID 6 0
* * If a record was selected, remove it from the container
C If TmpID <> 0
C 'CT1' Setatr TmpID 'RemoveRcd'
C EndIf
* 
```

Changing the Container View

To change the view, use the View attribute. The container part can display the following views of the data: icon, tree icon, tree text, and details.

Icon view

An icon view has each record represented by an icon, with text beneath it. Child records are not displayed in icon view. You specify the icon file name and the descriptive text in the record structure when you add the record to the container.

You can change the icon and icon text at run time by using the SetRcdIcon and SetRcdText attributes.

To have the icons displayed in rows in the container, set the Arrange attribute to 1.

To use mini icons, set the MiniIcon attribute to 1 or check the Mini Icon box on the properties notebook’s ‘Style’ page.
Tree view

In a tree view, records are presented in a hierarchy. The tree icon view displays each record with its icon and icon text beside it. If a record has child records, a plus sign is displayed next to its icon. Selecting the plus sign shows all the records related to this record. The tree text view displays records in the same way as the tree icon view, except in text-only mode, without icons.

Connecting lines are drawn between related records to show their relationship.

Figure 15. Sample icon view
Details View

In a details view, records are shown one after the other with each column displayed (similar to a subfile). Child records are not displayed in this type of view.

Figure 16. Sample tree view
Enter data in the entry fields below, and press Add Record or F6 to add a new record to the container. To remove a record, or change the container view, press the Right mouse button.

<table>
<thead>
<tr>
<th>Employee Number</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VARPG Container Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>Anil</td>
</tr>
<tr>
<td>Kimberly</td>
</tr>
</tbody>
</table>

![Image of VARPG Container Example]

**Figure 17. Sample details view**

If the container is not large enough to display all records at once, scroll bars are automatically added.

To change the view, use the **View** attribute.

**Mini Icons**

This option allows the programmer to specify whether the icons contained in the container part will be shown as regular icons, or as mini icons. This will only affect the icon view, and will leave the tree and details views unchanged.
DDE Client

* Restriction: This part is unsupported in Java applications.

Use the DDE client part to exchange data with other applications, such as spreadsheet applications, that support the dynamic data exchange (DDE) protocol.

The exchange is called a **DDE conversation**. The application that initiates the conversation is the **client**, and the application that responds is the **server**. To determine if an application supports DDE, refer to the documentation that came with it.

The DDE client part supports both **cold-link** and **hot-link conversations**. A cold-link conversation consists of a client program making explicit requests to the server program. A hot-link conversation consists of a server program automatically updating the client program when its data changes.

You can configure cold-link or hot-link conversations from the DDE client part’s properties notebook and in your program logic.

**Part Attributes**

<table>
<thead>
<tr>
<th>Attributes</th>
<th>AppName</th>
<th>Bottom</th>
<th>DDEAddLink</th>
<th>DDEMode</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDERmvLink</td>
<td>Execute</td>
<td>Format</td>
<td>Item</td>
<td></td>
</tr>
<tr>
<td>Left</td>
<td>ParentName</td>
<td>PartName</td>
<td>PartType</td>
<td></td>
</tr>
<tr>
<td>Poke</td>
<td>Request</td>
<td>TimeOut</td>
<td>Top</td>
<td></td>
</tr>
<tr>
<td>Topic</td>
<td>UserData</td>
<td>Visible</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Applicable Events**

<table>
<thead>
<tr>
<th>Events</th>
<th>Create</th>
<th>Data</th>
<th>Destroy</th>
<th>ExecuteAck</th>
</tr>
</thead>
<tbody>
<tr>
<td>PokeAck</td>
<td>Terminate</td>
<td>TimeOut</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Entry Field

Use the entry field part if you want the user to input something that you cannot predict the value of. An entry field is an area into which the user can type or place text. Its boundaries are usually indicated. The user can scroll through the text in the entry field if more information is available than is currently visible.

You can configure the entry field part to accept character, numeric, or double-byte character set (DBCS) data.

You can also make the entry field read-only, so that it contains information that cannot be directly altered by the user.

You can point and click on an entry field part in the parts palette and then click it onto the subfile part to create a subfile entry field.

Part Attributes

<table>
<thead>
<tr>
<th>AddLink*</th>
<th>Alignment</th>
<th>AllowLink*</th>
<th>AutoScroll*</th>
</tr>
</thead>
<tbody>
<tr>
<td>AutoSelect</td>
<td>BackColor</td>
<td>BackMix</td>
<td>Bottom</td>
</tr>
<tr>
<td>CapsLock</td>
<td>Copy</td>
<td>CsrAtEnd</td>
<td>Cut</td>
</tr>
<tr>
<td>DataType</td>
<td>Delete</td>
<td>DragEnable*</td>
<td>DropEnable*</td>
</tr>
<tr>
<td>Enabled</td>
<td>FieldExit</td>
<td>Focus</td>
<td>FontBold</td>
</tr>
<tr>
<td>FontItalic</td>
<td>FontName</td>
<td>FontSize</td>
<td>FontStrike*</td>
</tr>
<tr>
<td>FontUnder*</td>
<td>ForeColor</td>
<td>ForeMix</td>
<td>Handle*</td>
</tr>
<tr>
<td>Height</td>
<td>InsertMode*</td>
<td>Left</td>
<td>Masked</td>
</tr>
<tr>
<td>ParentName</td>
<td>PartName</td>
<td>PartType</td>
<td>Paste</td>
</tr>
<tr>
<td>ReadOnly</td>
<td>Refresh</td>
<td>RemoveLink*</td>
<td>ShowTips</td>
</tr>
<tr>
<td>Text</td>
<td>TextEnd</td>
<td>TextLength</td>
<td>TextSelect</td>
</tr>
<tr>
<td>TextStart</td>
<td>TipText</td>
<td>Top</td>
<td>UserData</td>
</tr>
<tr>
<td>Visible</td>
<td>Width</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Note: See the attribute description for restrictions.
Applicable Events

<table>
<thead>
<tr>
<th>Event</th>
<th>Event</th>
<th>Event</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change</td>
<td>Click</td>
<td>Create</td>
<td>DblClick</td>
</tr>
<tr>
<td>Destroy</td>
<td>Drop</td>
<td>GotFocus</td>
<td>KeyPress</td>
</tr>
<tr>
<td>Link*</td>
<td>LostFocus</td>
<td>MouseDown</td>
<td>MouseEnter</td>
</tr>
<tr>
<td>MouseExit</td>
<td>MouseMove</td>
<td>MouseUp</td>
<td>Popup</td>
</tr>
</tbody>
</table>

* Note: See the event description for restrictions.

Using the InsertMode Attribute

In Windows, insert mode is always on. It cannot be turned off.

Using the Text Attribute

Use the Text attribute to get or set the value of an entry field. The field you use to do this must be defined as the same type as the entry field. For example, if you are getting the value of a numeric entry field, the field that receives the value must also be defined as numeric.

Getting and Setting Information for a Window

During compilation, the compiler implicitly defines fields in your program with the same name as the entry field part, and with the same data type and length. By using the READ and WRITE operation codes with a window name specified in factor 2, the Text attribute value is automatically copied to or from these fields. The READ and WRITE operation codes are most useful if you have many entry fields in your user interface because you do not have to execute a series of get and set attributes.


Validity Checking

You can use the properties notebook to specify that an entry field should accept only data that meets criteria you specify. To ensure that the data matches certain values, set the Compare values. To ensure that the data falls within a range of predefined values, set the Range values.

Note: VisualAge RPG uses the ASCII collating sequence for validity checking. This differs from the server, which uses the EBCDIC collating sequence. Therefore, results may vary between systems.

To have validity checking performed, you must have at least one push button or graphic push button on the window that has the Validate attribute set. When the push button is pressed, validity checking is performed for each entry field that has validity checking criteria defined. If any of the entry fields fail this validity check, a message window is displayed and no press event is signaled to your program.

You can also use the field at the top of the properties notebook to set the message that is displayed if the validity check fails. Either enter the text of the message to be displayed or the named message file (such as *MSG0001) to have the corresponding message appear. If this field is left blank, the default system message is displayed whenever there is a validity check failure.

Note: There is a 15 character limit on this field.
Preventing User Input

To prevent the user from entering data in an entry field, do one of the following:

• Set the **ReadOnly** attribute to 1. After you set this attribute, you can still change the value of the entry field in your program.

• Set the **Enabled** attribute to 0. After you set this attribute, the entry field does not respond to events and the user cannot tab to it.

Masking Sensitive Data

If the entry field contains sensitive data, such as a password or an account number, set the **Masked** attribute to 1. When this attribute is set, the asterisk character ( * ) appears in the entry field for each character typed. This does not affect the actual data read from the entry field.
The graph part allows you to create and design a graph in your project. At runtime, you can send data to the graph and change graph attributes and the graph type. The graph part supports Pie, Line, Bar, and Line and Bar graph types.

The Bar and Line graph types support the ToolTip text control. When enabled, moving the mouse over a data point displays the tooltip text control. To use this support in your program logic, set the value for the point into the TipText attribute and set the ShowTips attribute on for the window that contains the graph part.

**Part Attributes**

| AutoInc | BarLabel | Bottom | Color | DataGroup | DataPoint | ColorArea | ColorMix | DataValue | Enabled | FillStyle* | FontArea | FontBold | FontName | FontSize | FontStyle* | FontUnder* | GroupLabel | GrpHiLite | Handle* | GnEqGrpCol | GnEqPntCol | Height | GnEqPntCol | LabelPlace | Left | LegendType | Refresh | StartNew | TipText | PartName | PartName | ParentName | PartType | Popups | Title | Top | TipText | UseData | UserData | Visible | UnderPoint* | Width |

* Note: See the attribute description for restrictions.

**Applicable Events**

- Click
- MouseDown
- MouseUp
- Create
- MouseEnter
- MouseExit
- MouseMove
- DblClick
- Destroy
- Popup
- MouseMove

**Sending data to the Graph**

Before you send data to the graph, you must indicate which DataGroup and which DataPoint is to receive the value. The DataPoint attribute represents the positional element of the graph that represents the value. For a Bar chart, this would be a Bar. For a line graph, a point, and for a Pie graph, which slice. The DataGroup is optional. This attribute indicates that there are groups of data to be graphed. The default DataGroup is one. Setting the DataValue attribute sets the value of the selected element.

As an example of a DataGroup, assume you wish to plot the high and low temperature of each month for a given year. This graph would have two data groups. The first would represent the low temperature for a month and the second the high temperature for the month. For this graph, you first set the DataGroup attribute to 1. Then, in a loop, you would set the DataValue for each DataPoint to the low temperature value. To complete the graph repeat the same steps for the high temperatures by setting the DataGroup to 2.
Sending data to the graph does not update the graph. You must set the **UseData** attribute to display the data.

The following code fragment shows how this may be done:

```c
*C
C  'Graph1'  Do  2  Group  'DataGroup'
*C
C  'Graph1'  Do  12  Point  'DataPoint'
*C
C  'Graph1'  If  Group = 1
C  'Graph1'  Setatr  Low(Point)  'DataValue'
*C
C  'Graph1'  Else
C  'Graph1'  Setatr  High(Point)  'DataValue'
*C
C  'Graph1'  EndIf
*C
C  'Graph1'  EndDo
*C
C  'Graph1'  Setatr  1  'UseData'
```

*Figure 18. Sending data to the graph*

If the graph is a pie graph, each group is represented as a separate pie.
Graphic Push Button

Use graphic push buttons to provide convenient access to frequently used actions.

A graphic push button provides the same function as a push button. It indicates an action that will be initiated when the user selects it, but instead of displaying a label to describe its function, it displays an image. The FileName attribute specifies the name of the image to use.

Valid Windows image formats include:
• Windows and OS/2® Bitmaps (BMP, VGA, BGA, RLE, DIB, RL4, RL8)
• Icon (ICO)
• Microsoft/Aldus Tagged Image File Format (TIF, TIFF)
• CompuServe Graphics Interchange Format (GIF)
• ZSoft PC Paintbrush Image File Format (PCX)
• Truevision Targa/Vista Bitmap (TGA, VST, AFI)
• Amiga Interleaved Bitmap Format (IFF, ILBM)
• X Windows Bitmap (XBM)
• IBM Printer Page Segment (PSE, PSEG, PSEG38PP, PSEG3820)
• Joint Photographic Experts Group format (JPG, JPEG)

Note: This product’s support for the JPG/JPEG format is based in part on the work of the Independent JPEG Group.

Valid Java image formats include:
• CompuServe Graphics Interchange Format (GIF)
• Joint Photographic Experts Group format (JPG, JPEG)

For related information, see “Push Button” on page 140.
Part Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Bottom</th>
<th>Handle*</th>
<th>Left</th>
<th>Refresh</th>
<th>UserData</th>
<th>Enabled</th>
<th>FileName</th>
<th>Focus</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Height</td>
<td>HelpEnable</td>
<td>ParentName</td>
<td>ShowTips</td>
<td>Validate</td>
<td></td>
<td>HighLight</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>TipText</td>
<td>Top</td>
<td></td>
<td></td>
<td>PartType</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Visible</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Top</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Width</td>
</tr>
</tbody>
</table>

* Note: See the attribute description for restrictions.

Applicable Events

<table>
<thead>
<tr>
<th>Event</th>
<th>Create</th>
<th>Destroy</th>
<th>GotFocus</th>
<th>LostFocus</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MouseEnter</td>
<td>MouseExit</td>
<td>MouseMove</td>
<td>Popup</td>
</tr>
</tbody>
</table>

Setting the Image

To set the image that is displayed on a graphic push button, use the FileName attribute, and specify a valid bitmap (.BMP) or icon (.ICO) file name. You must store system-specific bitmap and icon files in the appropriate runtime directory. For more information, see Chapter 12, “Using Picture, Sound, and Video Files,” on page 243.

Assigning Command Keys

You can assign a command key to a graphic push button. To do this, open the properties notebook and select one of the command keys from the available list. When the user presses the command key at run time, it has the same effect as pressing the mouse button or a key on the keyboard. A Press event is signaled to your program.

Signaling Events

When the push button is pressed, a Press event is signaled to your program.
Group Box

Use a group box to visually distinguish a group of parts in a window.

A group box is a rectangular box that is drawn around a group of parts to indicate that they are related. It is generally advisable to label a group box. If a label is not needed, you can use an outline box.

**Part Attributes**

| Attribute       | Bottom | Enabled | FontBold | Fontitalic | FontStrike* | FontUnder* | ForeColor | ForeMix | Handle* | Height | Label | Left | ParentName | PartName | PartType | Refresh | Visible | Width |
|-----------------|--------|---------|----------|------------|-------------|------------|-----------|---------|---------|--------|-------|-----|------------|----------|----------|---------|---------|-------|-------|

**Note:** See the attribute description for restrictions.

**Applicable Events**

Create   Destroy

**Labeling a Group Box**

Use the **Label** attribute to specify what string is to be used for the group box label.

**Grouping Radio Buttons**

See "Grouping Radio Buttons" on page 142 for related information.
## Horizontal Scroll Bar

Use the horizontal scroll bar part to allow users to scroll through a pane of information from left-to-right, or right-to-left. The information can be a list of files, records in a database, columns in a document, and so on. You can use the **Range** attribute to represent the total number of objects to be scrolled through and the **PageSize** attribute to determine the number of objects that can be displayed on a page.

### Part Attributes

<table>
<thead>
<tr>
<th>Bottom</th>
<th>Enabled</th>
<th>Focus</th>
<th>Handle*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Height</td>
<td>Left</td>
<td>NextLine</td>
<td>NextPage</td>
</tr>
<tr>
<td>PageSize</td>
<td>ParentName</td>
<td>PartName</td>
<td>PartType</td>
</tr>
<tr>
<td>Position</td>
<td>PrevLine</td>
<td>PrevPage</td>
<td>Range</td>
</tr>
<tr>
<td>Top</td>
<td>UserData</td>
<td>Visible</td>
<td>Width</td>
</tr>
</tbody>
</table>

* **Note:** See the attribute description for restrictions.

### Applicable Events

- Create
- Destroy
- Scroll
Image

Use the image part to display a picture. The **FileName** attribute specifies the name of the image to use.

Valid **Windows** image formats include:
- Windows and OS/2 Bitmaps (BMP, VGA, BGA, RLE, DIB, RL4, RL8)
- Icon (ICO)
- Microsoft/Aldus Tagged Image File Format (TIF, TIFF)
- CompuServe Graphics Interchange Format (GIF)
- ZSoft PC Paintbrush Image File Format (PCX)
- Truevision Targa/Vista Bitmap (TGA, VST, AFI)
- Amiga Interleaved Bitmap Format (IFF, ILBM)
- X Windows Bitmap (XBM)
- IBM Printer Page Segment (PSE, PSEG, PSEG38PP, PSEG3820)
- Joint Photographic Experts Group format (JPG, JPEG)

**Note:** This product’s support for the JPG/JPEG format is based in part on the work of the Independent JPEG Group.

Valid **Java** image formats include:
- CompuServe Graphics Interchange Format (GIF)
- Joint Photographic Experts Group format (JPG, JPEG)

These files reside on the programmable workstation (PWS), not on the host. You should store system-specific bitmap and icon files in the appropriate runtime directory (**RT_JAVA**, or **RT_WIN32**) so that the packaging utility includes them when you package your application.

**Note:** The image part can only be dropped on a notebook page with canvas or window with canvas.
Creating the Image Part

The image part can be created only on a canvas part.

Setting the File Name

To display a picture in the image part, set the **FileName** attribute with the name of the file containing the image. In Windows applications, the file must contain a valid bitmap or icon image. In Java applications, the file must contain a valid GIF or JPG image. The image may appear differently on individual workstations depending on the display device driver. If you specify a file name that is not valid, no picture will be displayed. You can clear the image part by setting the **FileName** attribute to blanks. If you get an error (such as *File not found*) when using the SETATR operation code to set a file name, the error indicator is turned on. For more information, see Chapter 12, “Using Picture, Sound, and Video Files,” on page 243.

Controlling the Magnification Panel

By default, the image part is created with a magnification panel. You can remove the magnification panel by disabling it in the properties notebook of the image part.

You can also enable or disable the magnification panel by using the **Panel** attribute in your program. If you set the **Panel** attribute to 0, the magnification panel is not displayed, and more of the picture can be shown. If you set it to 1, the magnification panel is displayed, and there is less room for the picture to be shown.

You can set the amount of magnification in your program. The magnification value is represented by a percentage between 25 and 200. Specifying a value of 0 will result in the best fit, where the image will just fit in the image window while keeping the horizontal to vertical ratio constant.

Image Example

In this example, a file is read from the iSeries 400 server. Each record in the file contains a part number field, which is inserted into the list box as each record is
read. When the user selects a part number in the list box and presses the View push button, the part number is concatenated with the string .BMP to form a file name. That file name is then used to set the FileName attribute of the image part to display the picture. The Label attribute of the PINFO static text part is updated to indicate the result of setting the file name attribute. Press the Close push button to terminate the program.

---

H
* Define the PARTS file
* FPARTS IF E DISK REMOTE
* DPath C "'
dnopic C 'Picture not available'

Figure 19. Sample Using the Image Part (Part 1 of 3)
Figure 19. Sample Using the Image Part (Part 2 of 3)
*********************************************************************
* Window . . . : WIN1
* Part . . . : VIEW
* Event . . : PRESS
* Description: Display the image for the selected part
*
*********************************************************************
* VIEW BEGACT PRESS WIN1
* Get index of selected item
C 'LB1' getatr 'FirstSel' x 4 0
* If an item was selected, build the bitmap file name
C x ifgt *zero
C 'LB1' setatr x 'Index'
C 'LB1' getatr 'GETITEM' tmp20 20
C movel tmp20 part 5
C endif
*
C move *blanks fullpath 64
C move *blanks tmp64 64
C path cat part:0 tmp64
C tmp64 cat '.gif':0 fullpath
*
* Set the file name in the image FILENAME attribute
* to display the image
C 'IMG1' setatr fullpath 'FILENAME' 80
*
* If indicator 80 is on, set the image file name
* failed, i.e. the file was not found.
* Set the Label attribute for the static text part PINFO to
* indicate status
C *in80 ifeq *on
C 'PINFO' setatr nopic 'Label'
C else
C 'PINFO' setatr *BLANKS 'Label'
C endif
*
C ENDACT
*

Figure 19. Sample Using the Image Part (Part 3 of 3)
Java Bean

*Restriction:* This part is unsupported in Windows applications.

Use the Java Bean part to add JavaBeans® to your project. You can use JavaBeans by calling Java methods, directly. For more information on calling Java methods, see Chapter 18, “Calling Java Methods from VisualAge RPG Programs,” on page 279.

To develop applications that use the Java Bean part, you must have Sun Microsystem’s Java 2 Software Development Kit (J2SDK), Standard Edition, version 1.2 or higher, installed on your workstation. If you do not have the J2SDK, you can download it from Sun Microsystems at the following URL:

http://java.sun.com/products/

After installing the J2SDK, set the PATH environment variable to point to the location of the Java compiler. For example, if your home directory for the J2SDK is x:\jdk1.2, add the following path statement:

x:\jdk1.2\bin

Also, set the JVM_DIR environment variable to point to the location of the jvm.dll, which is part of the J2SDK and JRE (Java Runtime Environment). For example, if your home directory for the J2SDK is x:\jdk1.2, set JVM_DIR to the following statements:

x:\jdk1.2\jre\bin\classic

**Part Attributes**

<table>
<thead>
<tr>
<th>AddEvent</th>
<th>Bottom</th>
<th>Enabled</th>
<th>Height</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>ParentName</td>
<td>PartName</td>
<td>PartType</td>
</tr>
<tr>
<td>RmvEvent</td>
<td>ShowProp</td>
<td>Top</td>
<td>UserData</td>
</tr>
<tr>
<td>Visible</td>
<td>Width</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Applicable Events**

Create  Destroy  BeanEvent

**Adding Beans to your Project**

To add a bean to your project, click on the Java Bean part in the parts palette. Click the mouse pointer onto the design window where you want the bean placed. A file Open dialog appears. Select the JAR file containing the bean (or beans) you want to work with. A window appears listing all the beans available in the JAR file. After you select a bean from the list, the bean will be instantiated. It will be shown in a separate window together with the associated property-sheet dialog and the bean-customizer, if available. (You can change the properties of the bean through the property-sheet dialog and the bean customizer.)

To show the properties, methods, and events for a bean, open the Java Bean Part Properties notebook from the design window. Right-click on the Java Bean icon in
the design window and select **Properties**. The properties, methods, and events of a bean are on the Information page. Select the appropriate radio button to view what is available.

Not all bean events are supported. VARPG-supported events are prefixed with an asterisk (*) in the Events list.

**Location of Bean JAR Files**

All bean-related JAR files for the project should be in the internal bean directory, \(x:\ldots\WDSC\beans\), where \(x:\ldots\WDSC\) is the home directory where VARPG is installed. This directory should include any bean-dependent JAR files, as well. For example, if BeanA is found in BeanA.jar and requires class files in beanclass.jar, both BeanA.jar and beanclass.jar must be copied to the internal bean directory.

While editing a project, you may select a bean from a JAR file that is not in the internal bean directory. This JAR file will be copied to the internal bean directory after you build the project. However, you still need to copy any bean-dependent JAR files to this directory.

Remove any unused JAR files from the internal bean directory. Doing this avoids the loading of unnecessary, non-project JARs.

**Setting the JAR Classpath**

The VARPG packaging utility does not handle the classpath setup for JAR files. You must set the classpath variable to include all the JAR files used by the beans in your project. For example, if BeanA uses BeanA.jar and depends on beanclasses.jar, set the classpath as follows:

```
SET CLASSPATH=x:\beandir\BeanA.jar;x:\beandir\beanclasses.jar;%CLASSPATH%;
```

where \(x:\beandir\) is the path for bean’s the JAR files.
Setting/Getting JavaBean Properties and Invoking Methods

VARPG supports invoking Java methods directly. (See Chapter 18, “Calling Java Methods from VisualAge RPG Programs,” on page 279 for details.) The VARPG run time provides a Java accessor class to access the JavaBeans objects in your projects. The accessor class, com.ibm.varpg.parts.VarpgBeanPart, is in the varpg.jar file. This class allows you to retrieve the bean objects that are instantiated by the VARPG run time. These bean objects are parts of the VARPG project. The method to get the bean object is:

public static Object getBeanObject(String strComponentName,
String strParentName,
String strPartName);

where strComponentName is the name of component containing the bean part, strParentName is the name of the window containing the bean part, and strPartName is the name of the bean part. The caller must check for any null references returned by the method to make sure the call is successful.

To find the actual method specification for setting or getting properties and invoking methods, refer to the Bean’s documentation as provided by the vendor, or view the Information page of the Java Bean Part Properties Notebook.
List Box

Use the list box part to provide the user with a list of items from which one or more items can be selected. A list box consists of read-only items. An item in a list box is a string of characters.

Horizontal and vertical scroll bars allow the user to view sections of the list that are not currently displayed. You can configure the list box so that the user can select either just one item or multiple items. You can use the Search, SearchType, and Case attributes to easily search for a particular item in the list.

Part Attributes

<table>
<thead>
<tr>
<th>AddItemEnd</th>
<th>AddLink*</th>
<th>AllowLink*</th>
<th>BackColor</th>
</tr>
</thead>
<tbody>
<tr>
<td>BackMix</td>
<td>Bottom</td>
<td>Case*</td>
<td>Count</td>
</tr>
<tr>
<td>DelimChar</td>
<td>DeSelect</td>
<td>DragEnable*</td>
<td>DropEnable*</td>
</tr>
<tr>
<td>Enabled</td>
<td>ExtSelect*</td>
<td>FirstSel</td>
<td>Focus</td>
</tr>
<tr>
<td>FontBold</td>
<td>FontItalic</td>
<td>FontName</td>
<td>FontSize</td>
</tr>
<tr>
<td>FontStrike*</td>
<td>FontUnder*</td>
<td>ForeColor</td>
<td>ForeMix</td>
</tr>
<tr>
<td>GetItem</td>
<td>Handle*</td>
<td>Height</td>
<td>Index</td>
</tr>
<tr>
<td>InsertItem*</td>
<td>ItemKey</td>
<td>Left</td>
<td>MultiSelect</td>
</tr>
<tr>
<td>NbrOfSel</td>
<td>ParentName</td>
<td>PartName</td>
<td>PartType</td>
</tr>
<tr>
<td>Refresh</td>
<td>RemoveItem</td>
<td>RemoveLink*</td>
<td>Search*</td>
</tr>
<tr>
<td>SearchType*</td>
<td>Selected</td>
<td>SelectItem</td>
<td>SelectList</td>
</tr>
<tr>
<td>Sequence*</td>
<td>SetItem</td>
<td>SetTop</td>
<td>ShowTips</td>
</tr>
<tr>
<td>SizeToFit</td>
<td>TipText</td>
<td>Top</td>
<td>TopItem</td>
</tr>
<tr>
<td>UseDelim</td>
<td>UserData</td>
<td>Visible</td>
<td>Width</td>
</tr>
</tbody>
</table>

* Note: See the attribute description for restrictions.

Applicable Events

Create    Destroy    Drop*    Enter*
GotFocus   KeyPress   LostFocus MouseEnter
MouseExit  MouseMove  Popup    Select
VKeyPress

* Note: See the event description for restrictions.

Adding and Setting the Sequence of Items

By default, items are displayed in the list box in the order in which you added them. If you want them displayed in a more precise order, then before you start adding them, set the Sequence attribute to either ascending, descending, or index. This sorts the items in ASCII collating sequence as they are added.

You cannot use the Sequence attribute to change the order of items that are already in the list box.
Adding Items at Run Time
You can insert items into a list box at run time by using the InsertItem attribute. The order in which items are displayed is determined by the Sequence attribute.

Updating Items in a List
You can change items that are already in the list. Use the Index attribute to indicate which item you want to change, and the SetItem attribute to specify the changed data.

Note: When you change an item using the SetItem attribute, the item remains in its original location, regardless of the value of the Sequence attribute. For example, if you had set the Sequence attribute to ascending order when you created the list, the items appear in the list box in ascending order; however, if you then retrieve an item, change its value, and use the SetItem attribute to replace it in the list box, the item is inserted in the same position it was in before. Therefore, the list may or may not be in ascending sequence after the change.

Setting the Top of the List
Use the SetTop attribute to specify which list item should appear at the top of the list box. Setting this item scrolls the list. It changes the display, but it does not reorder the items in the list.

Removing Items
Use the RemoveItem attribute to remove items from the list. Use the index value to specify the item that is to be removed. Index values start at 1. When an item is removed from the list, all items following the removed item are moved up one position in the list.

To remove all items in the list, specify an index value of 0.

Selecting and Deselecting Items
The user can select or deselect items by using the mouse or the keyboard. You can select and deselect items by setting the Selected and DeSelect attributes in your program. To use these attributes, first set the Index attribute.

Types of Selection
You can use attributes to specify how items are selected in a list box. Single, multiple, and extended selection are available.

Single selection
Single selection (the default) permits only one item in a list to be selected at one time. If an item is currently selected, it will be deselected when another item is selected.

Multiple selection
The user can select any number of objects, or not select any.

Extended selection
This type of selection is optimized for the selection of a single object, but the user can extend selection to more than one object, if required.
Retrieving Items from the List

To retrieve an item from a list box, use the GetItem attribute. First set the Index attribute to indicate which item you want to retrieve.

You typically retrieve items that have been selected by the user. To determine which items in a list box have been selected, use the FirstSel or Selected attribute. The FirstSel attribute returns the index of the first selected item in the list. If you need to check for additional selected items, be sure to use the DeSelect attribute to deselect this item; otherwise the FirstSel attribute returns the same item.

To determine if a specific item has been selected, use the Selected attribute. The Selected attribute uses the Index attribute value to determine if that item has been selected.

You can use the Count attribute to determine if there are any items to retrieve.

Using Keys

Both the list box and the combination box allow you to add items to the list that consist of a ‘key’ portion and a ‘data’ portion. When items are added to the list, only the data portion of the item is displayed. When the user selects an item you can programatically retrieve the key portion of the item.

See the Using Keys section in the combination box part description for more information.

Signaling Events

The Select event is signaled when:
• The user selects an item that is in a list box.
• You select an item in the list in your program.
• The user selects an item that is already selected.

The Enter event is signaled when:
• The user double-clicks over an item that is in the list box
• The user presses the Enter key when the list box has focus and an item has been selected.

In your action subroutine for these events, you can use the Selected or FirstSel attribute to determine which item was selected.

List Box Example

Press the Add push button to insert the text value from the entry field part into the list. Press the Clear push button to clear the list, the Select push button to select an item from the list and the Remove push button to remove the selected item from the list. Press Close to end the program.
Chapter 7. Using Parts

![Image of a GUI application](image-url)

The GUI shows a list of planets: Jupiter, Mercury, Venus, Earth, and Mars. There are buttons labeled Add, Remove, Clear, and Select.
<table>
<thead>
<tr>
<th>Program ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LISTBOX</td>
<td>Sample program to demonstrate the Listbox part.</td>
</tr>
</tbody>
</table>

```
****** Program ID . : LISTBOX ******
* Description . : Sample program to demonstrate the Listbox part. *
* ****** Program ID . : LISTBOX ******
* Window . : MAIN *
* Part . : CLEAR *
* Event . : PRESS *
* Description: Clear the listbox and the entry field. Give focus to the entry field part. *

C CLEAR BEGACT PRESS MAIN *
C 'LB1' setatr 0 'RemoveItem'
C 'EF1' setatr *blanks 'Text'
C 'EF1' setatr 1 'Focus'

C ENDACT

****** Program ID . : LISTBOX ******
* Window . : FRA0000B *
* Part . : CLOSE *
* Event . : PRESS *
* Description: Terminate the program *

C CLOSE BEGACT PRESS MAIN *
C move *on *INLR *
C ENDACT
```

*Figure 20. Coding Example Using the List Box Part (Part 1 of 3)*


Figure 20. Coding Example Using the List Box Part (Part 2 of 3)
You can use the Search, SearchType, and Case attributes to search for a particular item in the list. Using them is faster than reading each item in your program and comparing for specific values.

The following example shows how to locate a customer name in a list box as the user types a name in the entry field. The window is named MAIN, the list box is LB1, and the entry field is EF1. The user interface follows:

```plaintext
Figure 20. Coding Example Using the List Box Part (Part 3 of 3)

Search Example

You can use the Search, SearchType, and Case attributes to search for a particular item in the list. Using them is faster than reading each item in your program and comparing for specific values.

The following example shows how to locate a customer name in a list box as the user types a name in the entry field. The window is named MAIN, the list box is LB1, and the entry field is EF1. The user interface follows:

```
In the window’s Create event, we set the Case attribute of the list box to 0 to indicate that the search is not case sensitive. The SearchType attribute is set to 1 indicating we only want to compare the number of characters in the search string with the first characters of the list item. The rest of the code is filling the list box with records from the iSeries database.

```c
C MAIN BEGACT CREATE MAIN
* 'LB1' Setatr 0 'Case'
C 'LB1' Setatr 1 'SearchType'
* Read Custom01
* DoW NOT *in99
C 'LB1' Setatr CustNa 'AddItemEnd'
C Read Custom01
C EndDo
* 'LB1' Setatr 1 'SelectItem'
* ENDACT
```

The following code is the action subroutine for the Change event of the entry field EF1. Each time a character is typed in the entry field, this action subroutine is invoked.

The value of the Text attribute of the entry field is retrieved, and if it is not blank, that value is used as the search string for the Search attribute of the list box. If a match is found (Index attribute is greater than 0), the found item is selected and then moved to the top of the list box with the Settop attribute.

```c
C EF1 BEGACT CHANGE MAIN
* 'EF1' Getatr 'Text' Search 40
* If Search <> *Blanks
C 'LB1' Setatr Search 'Search'
* If %Getatr('Main':'LB1':'Index')<>0
```
If the entry field is blank, the first item in the list box is moved to the top, and is selected. The INDEX attribute is set to 1 so that subsequent searches begin at the top of the list.
Use the media part to play or record audio information or to play video files.

The media part gives your programs the ability to process wave (.WAV), MIDI (.MID), and QuickTime Movie (.MOV) files. If you want to use audio files, the computer must be equipped with a sound card capable of processing these files. To record a sound file, you will need a microphone or some other supported input device for the sound card. MIDI files cannot be recorded with the media part.

Java applications require the Java Media Framework (JMF) API. The media part only supports the playback of audio and video files in the Java environment.

The video file formats that can be processed are: MPEG (*.mpg) files, QUICKTIME Movie (*.mov) files, *.dat files, Microsoft® Video for Windows *.avi files are supported for Windows. To play these video files, the computer must have the appropriate drivers.

**Part Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddLink*</td>
<td>Add link to part</td>
</tr>
<tr>
<td>AllowLink*</td>
<td>Allow link to part</td>
</tr>
<tr>
<td>AudioMode</td>
<td>Audio mode</td>
</tr>
<tr>
<td>Bass*</td>
<td>Bass</td>
</tr>
<tr>
<td>Bottom</td>
<td>Bottom</td>
</tr>
<tr>
<td>FileName</td>
<td>File name</td>
</tr>
<tr>
<td>Handle*</td>
<td>Handle</td>
</tr>
<tr>
<td>InPlace</td>
<td>In-place</td>
</tr>
<tr>
<td>Length</td>
<td>Length</td>
</tr>
<tr>
<td>Panel</td>
<td>Panel</td>
</tr>
<tr>
<td>PartMode</td>
<td>Part mode</td>
</tr>
<tr>
<td>Position</td>
<td>Position</td>
</tr>
<tr>
<td>ParentName</td>
<td>Parent name</td>
</tr>
<tr>
<td>RemoveLink*</td>
<td>Remove link to part</td>
</tr>
<tr>
<td>ShowPlyBar*</td>
<td>Show play bar</td>
</tr>
<tr>
<td>Top</td>
<td>Top</td>
</tr>
<tr>
<td>Treble*</td>
<td>Treble</td>
</tr>
<tr>
<td>UserData</td>
<td>User data</td>
</tr>
<tr>
<td>Visible</td>
<td>Visible</td>
</tr>
<tr>
<td>Volume</td>
<td>Volume</td>
</tr>
</tbody>
</table>

* Note: See the attribute description for restrictions.

**Applicable Events**

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete</td>
<td>Complete</td>
</tr>
<tr>
<td>Create</td>
<td>Create</td>
</tr>
<tr>
<td>Destroy</td>
<td>Destroy</td>
</tr>
<tr>
<td>Link*</td>
<td>Link</td>
</tr>
</tbody>
</table>

* Note: See the event description for restrictions.

**Specifying a File Name**

Use the `FileName` attribute value to specify the name of the file you want to process. For more information, see Chapter 12, “Using Picture, Sound, and Video Files,” on page 243.

Note: Some wave files are shipped in compressed format. The media part processes only noncompressed wave files.

**Setting AudioMode**

To process a file, set the `AudioMode` attribute to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Pause — Suspend processing the file</td>
</tr>
</tbody>
</table>
2 Play — Plays the file
3 Record — Records a file
4 Stop — Stops processing the file

Setting the Volume
Use the Volume attribute to set the volume for the media part and the system’s waveout and synthesizer.

Setting the Position
Use the Position attribute to determine the start position in the file to be processed. Express the attribute value in milliseconds.

Using the Media Panel Part
You can use the media panel part to control the media part. In the media panel part’s properties notebook, set the media part name in the AddLink attribute, and enable the AllowLink attribute. This allows the user to control the media part simply by pressing the appropriate button on the media panel. See “Media Panel” on page 103 for more information.

Signaling Events
When the media part has completely processed a file, a Complete event is signaled.
Media Panel

Note: This part is not supported in Java applications.

Use the media panel part to provide convenient access to frequently used actions.

You can also use it to give the user control over other parts without your having to write any program logic. For example, you can use it to create push buttons or slider controls that control the volume or mode of a media part.

In the properties notebook for the media panel part, you can determine:
- Which buttons, from a defined set of buttons, the media panel will contain
- Whether or not the position and volume slider controls will be visible

Note: The media panel part can only be dropped on a notebook page with canvas or window with canvas.

Part Attributes

<table>
<thead>
<tr>
<th>AddLink</th>
<th>AllowLink</th>
<th>BackColor</th>
<th>BackMix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bottom</td>
<td>Enabled</td>
<td>Handle</td>
<td>Height</td>
</tr>
<tr>
<td>Left</td>
<td>PanelItem</td>
<td>PanelMode</td>
<td>ParentName</td>
</tr>
<tr>
<td>PartName</td>
<td>PartType</td>
<td>Position</td>
<td>RemoveLink</td>
</tr>
<tr>
<td>Top</td>
<td>UserData</td>
<td>Visible</td>
<td>Volume</td>
</tr>
<tr>
<td>Width</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Applicable Events

<table>
<thead>
<tr>
<th>Change</th>
<th>Create</th>
<th>Destroy</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>MouseEnter</td>
<td>MouseExit</td>
<td>MouseMove</td>
<td>Popup</td>
</tr>
</tbody>
</table>

Creating a Media Panel Part

A media panel part can be created only on a canvas part.

Linking Other Parts

There are two methods for linking a media panel part to another part: one involves using the properties notebook and the other involves writing program logic. The first method is the simplest. The only time you need to write program logic is if you want the link to be set during run time, and then you would set the AddLink and AllowLink attributes. A typical example would be to link the media panel to a media part. When a control is changed on the media panel, the link mechanism automatically affects the media part.

When you create a link from the media panel part to another part, only certain buttons are enabled on the media panel part. To make all the buttons enabled, you must also create a link from the other part back to the media panel part.
Refer to the AddLink description in VisualAge RPG Parts Reference, SC09-2450-05 for more information about the parts you can link to a media panel part.

**Signaling Events**

When the volume slider or the position slider is moved, a Change event is signaled. Use the PanelItem attribute to determine which slider was changed. Use the Volume attribute to determine the value of the volume slider, and the Position attribute to determine the value of the position slider.

When a push button on the media panel is pressed, a Press event is signaled. Use the numeric value returned by the PanelItem attribute to determine which button caused the Press event. Refer to VisualAge RPG Parts Reference, SC09-2450-05 for a list of possible values.
Menu Bar

Use the menu bar part to give users access to pull-down menus. You can add submenu parts and menu item parts to the menu bar.

A menu bar appears near the top of the window frame, just below the title bar. When the user selects a menu item from it, a pull-down menu appears, showing the items on that menu. Selecting a menu item immediately initiates the action it describes.

**Note:** You can manipulate this part’s properties, events, and so on, only from its pop-up menu in the project tree view.

For related information, see:
- “Menu Item” on page 106
- “Submenu” on page 171
- “Pop-up Menu” on page 138

**Part Attributes**

<table>
<thead>
<tr>
<th>PartType</th>
<th>PartName</th>
<th>ParentName</th>
<th>UserData</th>
</tr>
</thead>
</table>

**Applicable Events**

Create Destroy

**Creating Pull-down Menus**

You cannot open the properties notebook of a menu bar, submenu, or menu item by double clicking on it or via pop-up menus. You must manipulate it in the tree view.
Menu Item

Use menu items to construct pull-down or pop-up menus.

A menu item describes an action that is initiated when the user selects that item.

To construct a menu:
1. Drop a submenu part onto a menu bar or pop-up menu.
2. Drop menu items onto the submenu.

Note: You can manipulate this part’s properties, events, and so on, only from its pop-up menu in the project tree view.

For related information, see:
- “Menu Bar” on page 105
- “Pop-up Menu” on page 138
- “Submenu” on page 171

Part Attributes

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checked</td>
<td>Enabled</td>
</tr>
<tr>
<td>ParentName</td>
<td>PartName</td>
</tr>
<tr>
<td>Visable</td>
<td>PartType</td>
</tr>
<tr>
<td>FileName</td>
<td>Label</td>
</tr>
<tr>
<td>UserData</td>
<td></td>
</tr>
</tbody>
</table>

Applicable Events

<table>
<thead>
<tr>
<th>Event</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>Destroy</td>
</tr>
<tr>
<td></td>
<td>MenuSelect</td>
</tr>
</tbody>
</table>

Placing a Check Mark beside a Menu Item

A check mark symbol next to a menu item informs the user that the action represented by the menu item is selected. For example, if a check mark appears next to a Show Grid menu item, a grid is displayed.

To display a check mark next to a menu item, set the Checked attribute to 1. To remove a check mark, set the attribute to 0.

Setting Menu Text

Use the Label attribute to set the text for a menu item.

Setting a Mnemonic

Note: Mnemonics are not supported in Java applications.

To specify a mnemonic key for the menu item, place the mnemonic identifier in front of a character in the text of the Label attribute. For Windows, use an ampersand (&). The designated character is displayed on the interface with an underscore, for example, Display. The underscore informs users that they can select the menu item by pressing the underlined character on the keyboard.
Enabling Menu Items

You can control whether or not a MenuSelect event is issued when the user selects a menu item.

By default, the menu item is enabled when you create it. An enabled menu item generates a MenuSelect event when selected.

Set the Enabled attribute to 0 if you do not want a menu item enabled. When a menu item is not enabled, it is dimmed on the display, and it does not generate a MenuSelect event when selected.

Signaling Events

When the user selects a menu item, a MenuSelect event is signaled.

Note: Only menu items signal a MenuSelect event. Submenus (such as cascaded menus), which are attached to other menu items do not.
Message Subfile

Use the message subfile part to display predefined messages or to display text that you supply in your program logic: for example, error or status information.

This part is always positioned at the bottom of the window frame and runs the width of the window. You cannot resize its width; you can, however, resize its height so that it shows more messages. At run time, users can use scroll bars to view all of the messages.

**Part Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddMsgID</td>
<td>AddMsgText, Count, DragEnable*</td>
</tr>
<tr>
<td>DropEnable*</td>
<td>Enabled, FirstSel, FontBold</td>
</tr>
<tr>
<td>FontItalic</td>
<td>FontName, FontSize, FontStrike*</td>
</tr>
<tr>
<td>FontUnder*</td>
<td>ForeColor, ForeMix, GetItem</td>
</tr>
<tr>
<td>Handle*</td>
<td>Height, Index, MsgSubText</td>
</tr>
<tr>
<td>NbrOfSel</td>
<td>ParentName, PartName, PartType</td>
</tr>
<tr>
<td>RemoveMsg</td>
<td>Selected, ShowTips, TipText</td>
</tr>
<tr>
<td>UserData</td>
<td>Visible</td>
</tr>
</tbody>
</table>

* Note: See the attribute description for restrictions.

**Applicable Events**

- Create
- Destroy
- Drop
- Enter
- MouseEnter
- MouseExit
- MouseMove
- Popup
- Select

**Displaying Predefined Messages**

To display messages that are already defined in the GUI Designer, set the `AddMsgId` attribute to the ID number of the message you want to display. Use the numeric portion of the message identifier.

**Displaying Text Supplied in Your Program**

To display text that is not part of a predefined message, use the `AddMsgText` attribute in your program and supply a text string or literal as the message value.

**Using Substitution Variables**

The message subfile part supports substitution variables. A substitution variable is defined when you create the message by typing an percent (%) character followed by a numeric value (for example, %123). This substitution variable is replaced by data your program provides before you add the message. Message substitution data applies to the `AddMsgID` and `AddMsgText` attributes.

Message substitution data is a series of words separated by blanks. Each substitution word replaces the corresponding substitution variable before the
message is added to the message subfile part. To set the message substitution data, use the \texttt{MsgSubText} attribute \textbf{before} you set the \texttt{AddMsgID} attribute.

\textbf{Note:} The substitution data remains in effect until another \texttt{MsgSubText} attribute is used.

\section*{Removing Messages}

Use the \texttt{RemoveMsg} attribute to remove a message from the message subfile part. Specify the index number of the message to be removed. To remove all messages, use an index value of 0.
Message Subfile Example

In this example, the user is prompted to enter a part number for processing. The part number must be greater than zero and less than 2000. When the OK push button is pressed, the program checks that the value is in the required range. If the value is not in the range, a message is added to the message subfile part.

Figure 21. Coding Example Using the Message Subfile Part (Part 1 of 2)
* Description: Check that the value entered is allowed. If not, add a message to Message subfile part.
* The value entered is used as substitution text in the message.

**Figure 21. Coding Example Using the Message Subfile Part (Part 2 of 2)**
Multiline Edit

Use the multiline edit part if you want the user to be able to type in several lines of text.

The multiline edit part has defined boundaries. Sometimes not all of its text is visible. The user can scroll up, down, left, or right to view text that is currently not visible.

Part Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddLineEnd</td>
<td>AddOffset</td>
</tr>
<tr>
<td>Bottom</td>
<td>CanUndo</td>
</tr>
<tr>
<td>CsrLine</td>
<td>CsrPos</td>
</tr>
<tr>
<td>DragEnable*</td>
<td>DropEnable*</td>
</tr>
<tr>
<td>FontBold</td>
<td>FontItalic</td>
</tr>
<tr>
<td>FontStrike*</td>
<td>FontUnder*</td>
</tr>
<tr>
<td>Handle*</td>
<td>Height</td>
</tr>
<tr>
<td>Left</td>
<td>LineNumber</td>
</tr>
<tr>
<td>ParentName</td>
<td>PartName</td>
</tr>
<tr>
<td>ReadOnly</td>
<td>Refresh</td>
</tr>
<tr>
<td>TextEnd</td>
<td>TextLength</td>
</tr>
<tr>
<td>TextString</td>
<td>TipText</td>
</tr>
<tr>
<td>Undo</td>
<td>UserData</td>
</tr>
</tbody>
</table>

*Note: See the attribute description for restrictions.

Applicable Events

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change</td>
<td>Click</td>
</tr>
<tr>
<td>Destroy</td>
<td>Drop</td>
</tr>
<tr>
<td>LostFocus</td>
<td>MouseDown</td>
</tr>
<tr>
<td>MouseMove</td>
<td>MouseUp</td>
</tr>
<tr>
<td></td>
<td>Create</td>
</tr>
<tr>
<td></td>
<td>GotFocus</td>
</tr>
<tr>
<td></td>
<td>MouseEnter</td>
</tr>
<tr>
<td></td>
<td>Popup</td>
</tr>
<tr>
<td></td>
<td>DblClick</td>
</tr>
<tr>
<td></td>
<td>KeyPress</td>
</tr>
<tr>
<td></td>
<td>MouseExit</td>
</tr>
<tr>
<td></td>
<td>VKeyPress</td>
</tr>
</tbody>
</table>

Getting and Setting the Text

Use the Text attribute to get or set the text of the multiline edit part.

Note: The default text entered in the properties notebook for the multiline edit part is not saved. Text for a multiline edit part can only be set at run time.

Manipulating Lines of Text in a Multiline Edit Part

To insert new lines into a multiline edit part:
1. Set the LineNumber attribute to the line number after which you want to insert text.
2. Use the InsertLine attribute.
   Your text is inserted after the line you specify. Any lines that are below the line you specified are moved down to make room for the inserted text.
Manipulating Characters in a Multiline Edit Part

To insert a string of characters into a multiline edit part:
1. Set the CharOffset attribute to specify where you want the new text inserted. Text following the CharOffset value will be replaced with the new text.
2. Use the AddOffset attribute to add text at CharOffset.

Manipulating Selected Portions of Text in a Multiline Edit Part

You can use several attributes to manipulate selected portions of text in a multiline edit part.

To return just the selected text, use the TextSelect attribute. If no text is selected, the TextSelect attribute returns a null string, and the result field that is to receive the text remains unchanged.

Use the TextStart and TextEnd attributes to return the starting and ending character positions of the selected text.

Changing Color

If a multiline edit part exists on a canvas part whose background color is set to the system default, changes to the background color of the canvas will be inherited by the multiline edit part. Additional multiline edit parts added to the canvas will not inherit this color. To correct this, defer setting the background color of the canvas until you have placed all multiline edit parts on it. Alternatively, you can make the multiline edit parts inherit the color by setting the color of the canvas to the system default and then back to your predefined RGB color setting.

If you drag and drop a color onto the scroll bar of a multiline edit part, that color is not saved. The multiline edit part will be changed to the new color, but when you close and reopen the window, the color will be changed back to the original.

Choosing Fonts

Not all fonts are supported by the multiline edit part. After you select a font for this part, it will adjust to display the closest match for the selected font.

Preventing User Input

You can prevent users from entering text in the multiline edit part by doing one of the following:
- Set the ReadOnly attribute to 0.
- Set the Enabled attribute to 0. (This also prevents the multiline edit part from responding to events such as Change and GotFocus.)

You can still change the value of the multiline edit part in your program.

Multiline Edit Example

In this example, pressing the Copy push button copies the selected text from the multiline edit to the entry field. Pressing the Close push button ends the program.
This example demonstrates how user-selected text in a Multiline Edit part can be manipulated.
Figure 22. Coding Example Using the Multiline Edit Part (Part 1 of 2)
Figure 22. Coding Example Using the Multiline Edit Part (Part 2 of 2)
Notebook

Use the notebook part to present data that can be logically grouped by topic: for example, customer information divided into categories such as Name, Shipping Address, Orders, and Credits.

A notebook part is a graphical representation of a bound notebook. (In Windows applications, this is known as a tab control.) You can add pages to the notebook, and you can group the pages into sections separated by tabbed dividers. If the notebook page has a canvas, you can add more than one part to it. If it does not have a canvas, you can add only one part to it.

The user can turn the pages of the notebook to move from one page to the next, or go straight to a section by clicking on its divider tab.

You can add notebook pages by:
• Using the properties notebook for the notebook part
• Pointing-and-clicking (or dragging-and-dropping) a properties tab or notebook page with canvas onto the notebook part

For related information, see:
• “Notebook Page” on page 118
• “Notebook Page with Canvas” on page 119

Part Attributes

<table>
<thead>
<tr>
<th>BackColor</th>
<th>BackMix</th>
<th>Bottom</th>
<th>Count</th>
<th>Enabled</th>
<th>Focus</th>
<th>FontBold</th>
<th>FontItalic</th>
<th>FontName</th>
<th>FontSize</th>
<th>FontStrike*</th>
<th>FontUnder*</th>
<th>ForeColor</th>
<th>ForeMix</th>
<th>Handle*</th>
<th>Height</th>
<th>Left</th>
<th>PageNumber</th>
<th>ParentName</th>
<th>PartName</th>
<th>Refresh</th>
<th>ShowTabs*</th>
<th>Visible</th>
<th>Width</th>
</tr>
</thead>
</table>

* Note: See the attribute description for restrictions.

Applicable Events

Create Destroy

Changing Font Emphasis

Changing the font emphasis for a notebook part to underscore or strikeout causes the status text to take on the new emphasis but not the tab text.
Use the notebook page part to add pages to a notebook.

You can add only one part to a notebook page; that part will be automatically sized to fit the entire page. If you want to add more than one part on a page, you must point-and-click a canvas part onto the notebook page. Alternatively, you can use the notebook page with canvas part to save a step.

**Note:** You can manipulate this part’s properties, events, and so on, only from its pop-up menu in the project tree view.

The user can press the left and right arrow keys to move from one page to the next.

For related information, see
- [“Notebook” on page 117](#)
- [“Notebook Page with Canvas” on page 119](#)

### Part Attributes

<table>
<thead>
<tr>
<th>Enabled</th>
<th>OnTop</th>
<th>ParentName</th>
<th>PartName</th>
</tr>
</thead>
<tbody>
<tr>
<td>PartType</td>
<td>Refresh</td>
<td>TabImage</td>
<td>TabLabel</td>
</tr>
<tr>
<td>UserData</td>
<td>Visible</td>
<td>[TabImage]</td>
<td>[TabLabel]</td>
</tr>
</tbody>
</table>

### Applicable Events

| Create | Destroy | PageSelect | SelPending* |

* **Note:** See the event description for restrictions.

### Showing Tab Text

On a DBCS machine, the tab of a notebook page may not show all its text when the MINCHO Proportional font is used. Changing the font to another style, such as MINCHO Normal or MINCHO System, will fix this.

### Setting a Mnemonic

To specify a mnemonic key for the notebook page, place the mnemonic identifier in front of a character in the text of the **Label** attribute. This designated character is displayed on the interface with an underscore (for example, **Display**). Note that for Windows, mnemonics are displayed, but do not function on notebook pages.

**Note:** Mnemonics are not supported in Java applications.
Notebook Page with Canvas

Use the notebook page with canvas to add pages to a notebook part.

The canvas part occupies the client area of a notebook page part. By adding parts to the canvas part, you can create a graphical user interface.

If you want to add only one part to the page, you can use the notebook page part instead of the notebook page with canvas part. Because the notebook page part does not have a canvas on it, the part you add will be sized automatically.

For related information, see:
- “Notebook” on page 117
- “Notebook Page” on page 118

Part Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Enabled</th>
<th>OnTop</th>
<th>ParentName</th>
<th>PartName</th>
<th>PartType</th>
<th>Refresh</th>
<th>TabImage</th>
<th>TabLabel</th>
<th>UserData</th>
<th>Visible</th>
</tr>
</thead>
</table>

Applicable Events

<table>
<thead>
<tr>
<th>Event</th>
<th>Create</th>
<th>Destroy</th>
<th>PageSelect</th>
<th>SelPending</th>
</tr>
</thead>
</table>

The ODBC/JDBC Interface part provides the ability to process database files that support the Windows ODBC API or Sun Microsystem’s JDBC API. Examples of these database file types include Foxpro, Access, and Paradox.

To develop applications that can use the ODBC/JDBC Interface part, you must be familiar with SQL and have either the Windows ODBC SDK or Sun Microsystem’s Java 2 Software Development Kit (J2SDK), Standard Edition, installed on your workstation.

If you do not have the ODBC SDK, you can download it from Microsoft at the following URL:
http://www.microsoft.com/odbc/download.htm

The JDBC support is part of the Java™ 2 Software Development Kit (J2SDK) Version 1.2 for Windows. If you do not have the J2SDK, you can download it from Sun Microsystems at the following URL:
http://java.sun.com/products/

Applications that access and manipulate data in a JDBC database require the appropriate JDBC 2.0 compliance driver. You can find JDBC driver and other information at the following URL:
http://java.sun.com/products/jdbc/

**Note:** JDBC is not supported in applets.

An ODBC or JDBC database consists of one or more tables. Data is stored in a table as a series of rows. Each row, or record, contains a number of columns with data. Your program can submit SQL statements along with ODBC/JDBC Interface part attributes to manipulate rows, or to move data between program fields and table columns.

Before you can process an existing database, your VARPG program must first connect to the database and indicate which table to reference. To manipulate the rows in a table, your program must create a record set that identifies the records to be returned and maintained by the ODBC/JDBC Interface part. To access the data in a row, you must bind each column used in the table row with a program field in your program. In Java applications, pointers are not supported. A column is bound to a part; only the static text and entry field parts can be used for binding.

If you are creating a Java application that uses the ODBC/JDBC interface part, end users running your application must install the varpgjdb.jar file on their workstation and add its location to their classpath statement. The packaging utility does include this JAR file. The JAR file is located in the WDSC\java subdirectory.

**Part Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllowChg*</td>
<td></td>
</tr>
<tr>
<td>BufferLen*</td>
<td></td>
</tr>
<tr>
<td>BindPart</td>
<td></td>
</tr>
<tr>
<td>Bottom</td>
<td></td>
</tr>
<tr>
<td>BufferDec*</td>
<td></td>
</tr>
<tr>
<td>BufferPtr*</td>
<td></td>
</tr>
<tr>
<td>BufferType*</td>
<td></td>
</tr>
<tr>
<td>CharData</td>
<td></td>
</tr>
</tbody>
</table>
Connecting to an ODBC Database

Before you can process an existing database, your VARPG program must first connect to the database and indicate which table to reference.

Note: The VARPG ODBC/JDBC Interface part can only connect to one table at a time. If that table has dependencies or relations with other tables, your program cannot update or delete records in the database.

To connect to a database, first set the ConnectStr attribute to the required connection string for the database. Then, use the Connect attribute to do the connection. In Windows, if you set the ConnectStr attribute to "BLANKS, the ODBC Manager will display the Select Data Source dialog from which you can select the table to connect to. Once the connection is made, the Connected attribute will be set to 1. If the connection fails, the Connected attribute is set to 0.

When the following code fragment is executed, the Select Data Source dialog will be displayed. If a table was selected, the ConnectStr attribute will contain the connection string. The Connect attribute is set to make the connection.

```c
C  'ODBC'  Setatr  'ConnectStr'  *Blanks
C  'ODBC'  Setatr  1  'Connect'
C  If  %Getatr('Main':'ODBC':'Connected')=1
C    ...  
C  Else
C    ...  
C  EndIf
```

Creating a Record Set

Once you have connected to a database, you must create a record set before you can access any data in the database. To create a record set, submit a SELECT statement to the ODBC/JDBC Interface part using the SQLQuery and ExecuteSQL attributes. The SELECT statement identifies which table in the database is being accessed and which group of records in the table is to be processed.

The following code segment is an example of creating a record set of all records in table CUSTOMERS:
D SelAll
C 'Select * From "Customers"
* 
C 'ODBC' Setatr SelAll 'SQLQuery'
C 'ODBC' Setatr 1 'ExecuteSQL'

Accessing Table Data

Data is stored in a table as a series of rows. Each row contains a number of columns with data. You can manipulate the rows in a table by using the ODBC/JDBC Interface part attributes such as FetchNext, FetchPrior, UpdateRow, and so on. However, to access the data in a row, you must bind each column in the table row with a program field in your program. Once this binding is set, the ODBC/JDBC Interface part can move data between the program fields and the table columns.

To bind the program field, you use the following ODBC/JDBC Interface part attributes:

Column
Establishes which column in the table is to be bound.

BufferPtr
Contains the address of the program field to bind to the column.

BufferDec
Specifies the number of decimal places for the buffer column.

BufferLen
Specifies the length of the program field.

BufferType
Indicates the data type of the program field.

In the following example, 2 fields defined in the D specifications are being bound to columns 1 and 2 in a table:

D first $ 20
D last $ 30
* *
D fptr $ * INZ(%Addr(first))
D lptr $ * INZ(%Addr(last))
* *

C 'ODBC1' Setatr 1 'Column'
C 'ODBC1' Setatr 20 'BufferLen'
C 'ODBC1' Setatr fptr 'BufferPtr'
C 'ODBC1' Setatr 1 'BufferType'
* *
C 'ODBC1' Setatr 2 'Column'
C 'ODBC1' Setatr 30 'BufferLen'
C 'ODBC1' Setatr lptr 'BufferPtr'
C 'ODBC1' Setatr 1 'BufferType'

You can also use the %ADDR built-in directly on the C specifications to avoid coding the D specifications to define the pointers:

C Eval %Setatr('Main':'ODBC1':'BufferPtr')=%Addr(first)

Data Types

Use the BufferType attribute to indicate the data type of the program field referenced by the BufferPtr attribute. The ODBC/JDBC Interface part uses the BufferType attribute to perform the correct data translation when moving data
between the program field and table column. It is important to set this attribute correctly, as there is no checking for proper field types.

Set the Column attribute before using the BufferType attribute. If the program field is associated with a part on the interface, you can use the DataType attribute to get the buffer type.

Use the following chart to set the VARPG data type for the corresponding, supported SQL data type. Specify the BufferLen and BufferDec attributes only as listed in the chart.

For character, decimal, integer, or small integer data types, specify only the BufferLen attribute.

Note that Double, Float, and Real data types can be defined, in VARPG, as either Float(F) or Zoned. If you define these as Zoned, the VARPG run time will only use the number of decimal places specified by the BufferDec attribute when moving data from the column. This can result in a loss of precision if the data source has more decimal places than is specified by the BufferDec attribute. If you define these fields as Float(F), do NOT specify the BufferLen or BufferDec attribute.

<table>
<thead>
<tr>
<th>SQL Data Type</th>
<th>VARPG Data Type</th>
<th>Specify Program Field Length (use BufferLen)</th>
<th>Specify Decimal Places for Buffer Column (use BufferDec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>CHAR</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Decimal</td>
<td>Zoned</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Integer</td>
<td>Zoned</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Small Integer</td>
<td>Zoned</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Double</td>
<td>8F</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Double</td>
<td>Zoned</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Float</td>
<td>4F</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Float</td>
<td>Zoned</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Real</td>
<td>4F</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Real</td>
<td>Zoned</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

If a column contains a data type that is not supported by the ODBC/JDBC Interface part, set the AllowChg attribute to 0 for that column. The ODBC/JDBC Interface part will not move data between any program field and the column. The data remains unchanged.

**Retrieving Table Rows**

To process rows in a table, you must first create a record set. A record set is a group of records returned and maintained by the ODBC/JDBC Interface part. Your program submits an SQL statement to the ODBC/JDBC Interface part using the SQLQuery and ExecuteSQL attributes. First, the SQLQuery attribute is set to the SQL statement to execute. Then, the ExecuteSQL attribute is set to 1 to execute the query.

In the following example, all records are being selected from the table Customers:

```
D SelAll C 'Select * From "Customers"'
* 'ODBC1' Setatr SelAll 'SQLQuery'
C 'ODBC1' Setatr 1 'ExecuteSQL'
```
To determine the number of rows that were returned as the result of an SQLQuery,
you can check the value of the Rows attribute.

Once a record set has been returned, you can process each row using the
FetchNext and FetchPrior attributes. Set the FetchNext attribute to 1 to return the
next row in the record set. Set the FetchPrior attribute to 1 to return the previous
row in the record set. To determine if a FetchNext or FetchPrior successfully
returned a row, check the value of the IsData attribute. A value of 1 indicates that
data was returned. Otherwise, the IsData value is set to 0.

In the following example, all of the records in a record set are read and the value
of column 1 (field first) is added to list box LB1.

C  'ODBC1'  Setatr  1       'FetchNext'
C  'ODBC1'  Getatr  'IsData'  Temp   1  0
*  
C  DoW              Temp = 1
C  'LB1'  Setatr  first     'AddItemEnd'
C  'ODBC1'  Getatr  'IsData'  Temp
C  EndDo

**Updating Row Data**

To update data in a row, use the UpdateRow attribute to specify the row to be
updated. Be aware that UpdateRow will cause any row to be updated. You do not
need to fetch the row first. Typically however, you will update a row that has just
been fetched. In this case, you would use the CurrentRow attribute. This attribute
contains the row number of a row just fetched.

In the following code segment, assume a row has been read and information has
been displayed on a window. The user presses the update button after making
changes.

C  PB_Update  BEGACT  PRESS  Main
*  
C  Read  'Main'
C  'ODBC1'  Getatr  'CurrentRow'  Row   1  0
C  'ODBC1'  Setatr  Row  'UpdateRow'
*  
C  ENDACT

**Deleting a Row**

Deleting a row is similar to updating one. (See "Updating Row Data.") Use the
DeleteRow attribute to specify the row to be deleted. As with the UpdateRow
attribute, DeleteRow will cause any row in the row set to be deleted. It is not
necessary to fetch the row first.

In the following example, the user has pressed the Delete push button to delete a
record that has just been fetched and is currently being displayed on a window.

C  PB_Delete  BEGACT  PRESS  Main
*  
C  'ODBC1'  Getatr  'CurrentRow'  Row   1  0
C  'ODBC1'  Setatr  Row  'DeleteRow'
*  
C  ENDACT

**ODBC/JDBC Interface Part Example**

The following example uses a database created with Microsoft Access. The
database has one table named CUSTOMERS. This simple inquiry program displays
a window containing details about a customer. Push buttons are provided that allow the user to go to the next and previous records, and to update and delete the current record.

The following figure shows the inquiry window:

![Customer Inquiry Window]

The code for this example follows.
* Define connect string

* Define connect string

D ConnectStr  C  'DSN=MS Access 97 Database;DBQ=-
D CelDial.mdb;-
D DriverId=25;FIL=-
D MS Access;MaxBufferSize'

* Working variables

DClr  S  2 0
DCol  S  10
DSQL  S  255A
DColNumber  S  2 0
DPart  S  10
DCharacter  S  2
D AppStart  C  'HourGlas.ANI'

* DDel  M  MsgNbr(+MSG0003)
D  MsgData(CustNo:CustNa)
D  J  S  4 0
D  I  S  4 0

* CSENDINFO  S  15 0

* Define pointers to field buffers

* DCBalance  S  18S 3  Balance (Numeric-Double)
DP_001  S  *  Number
DP_002  S  *  Name
DP_003  S  *  Rep number
DP_004  S  *  Contact
DP_005  S  *  Phone
DP_006  S  *  Fax
DP_007  S  *  Address
DP_008  S  *  City
DP_009  S  *  Country
DP_010  S  *  Zip Postal Code
DP_011  S  *  Zip location
DP_012  S  *  Balance
DP_013  S  *  Send Market Info to?

* Select ALL records

* DSelAll  C  'Select * From "Customers"'

* Figure 23. Code for ODBC/JDBC Inquiry Example (Part 1 of 11)
Bind program fields to columns and connect to the database table CUSTOMERS

Figure 23. Code for ODBC/JDBC Inquiry Example (Part 2 of 11)
* Bind column: Phone
C 'ODBC' SetAtr 5 'Column'
C 'ODBC' SetAtr 17 'BufferLen'
C 'ODBC' SetAtr 1 'BufferType'
C Eval P_005=%Addr(CPhone)
C 'ODBC' SetAtr P_005 'BufferPtr'
*
* Bind column: Fax
C 'ODBC' SetAtr 6 'Column'
C 'ODBC' SetAtr 17 'BufferLen'
C 'ODBC' SetAtr 1 'BufferType'
C Eval P_006=%Addr(CFax)
C 'ODBC' SetAtr P_006 'BufferPtr'
*
* Bind column: Address
C 'ODBC' SetAtr 7 'Column'
C 'ODBC' SetAtr 40 'BufferLen'
C 'ODBC' SetAtr 1 'BufferType'
C Eval P_007=%Addr(CAddr)
C 'ODBC' SetAtr P_007 'BufferPtr'
*
* Bind column: City
C 'ODBC' SetAtr 8 'Column'
C 'ODBC' SetAtr 30 'BufferLen'
C 'ODBC' SetAtr 1 'BufferType'
C Eval P_008=%Addr(CCity)
C 'ODBC' SetAtr P_008 'BufferPtr'
*
* Bind column: Country
C 'ODBC' SetAtr 9 'Column'
C 'ODBC' SetAtr 20 'BufferLen'
C 'ODBC' SetAtr 1 'BufferType'
C Eval P_009=%Addr(Count)
C 'ODBC' SetAtr P_009 'BufferPtr'
*
* Bind column: Zip Postal Code
C 'ODBC' SetAtr 10 'Column'
C 'ODBC' SetAtr 10 'BufferLen'
C 'ODBC' SetAtr 1 'BufferType'
C Eval P_010=%Addr(CZip)
C 'ODBC' SetAtr P_010 'BufferPtr'
*
* Bind column: Zip location
C 'ODBC' SetAtr 11 'Column'
C 'ODBC' SetAtr 1 'BufferLen'
C 'ODBC' SetAtr 1 'BufferType'
C Eval P_011=%Addr(CZiplo)
C 'ODBC' SetAtr P_011 'BufferPtr'
*

Figure 23. Code for ODBC/JDBC Inquiry Example (Part 3 of 11)
* Bind column: Balance
C 'OODBC' SetAtr 12 'Column'
C 'OODBC' SetAtr 18 'BufferLen'
C 'OODBC' SetAtr 0 'BufferType'
C 'OODBC' SetAtr 3 'BufferDec'
C Eval P_012=%Addr(CBalance)
C 'OODBC' SetAtr P_012 'BufferPtr'
C 'OODBC' SetAtr ConnectStr 'ConnectStr'
 *
* Bind column: Send Info
C 'OODBC' SetAtr 13 'Column'
C 'OODBC' SetAtr 1 'BufferLen'
C 'OODBC' SetAtr 0 'BufferType'
C 'OODBC' SetAtr 0 'BufferDec'
C Eval P_013=%Addr(CSendInfo)
C 'OODBC' SetAtr P_013 'BufferPtr'
 *
* Connect to the database and select all records
C 'OODBC' SetAtr 1 'Connect'
C 'OODBC' SetAtr SelAll 'SQLQuery'
C 'OODBC' SetAtr 1 'ExecuteSQL'
 *
C 'Main' Setatr 1 'ProgressBar'
C 'OODBC' Getatr 'Rows' Rows 4 0
C 'Main' Setatr Rows 'PBRange'
C Eval %Setatr('Main':'Main':'PBStepSize')=110
 *
C Z-Add 22 MaxRows 2 0
C Exsr Fill
 *
C ENDACT
******************************************************************************
* Window/Part/Event: Main/PB_Close/Press
* Description: End the program
******************************************************************************
* PB_CLOSE BEGACT PRESS Main
* Move *ON *INLR
* ENDACT

Figure 23. Code for ODBC/JDBC Inquiry Example (Part 4 of 11)
Figure 23. Code for ODBC/JDBC Inquiry Example (Part 5 of 11)
Figure 23. Code for ODBC/JDBC Inquiry Example (Part 6 of 11)
Figure 23. Code for ODBC/JDBC Inquiry Example (Part 7 of 11)
Figure 23. Code for ODBC/JDBC Inquiry Example (Part 8 of 11)
Figure 23. Code for ODBC/JDBC Inquiry Example (Part 9 of 11)
Figure 23. Code for ODBC/JDBC Inquiry Example (Part 10 of 11)
* Window/Part/Event: Detail/Detail/Create
* Description: Set all fields to read only
*********************************************************************
C   DETAIL  BEGACT  CREATE  DETAIL
C   'CAN0000037'  Setatr  1  'ReadOnly'
C   ENDACT
*********************************************************************
* Window/Part/Event: Main/MI_Tips/MenuSelect
* Description: Toggle display of tip text
*********************************************************************
C   MI_TIPS  BEGACT  MENUSELECT  MAIN
C   If  %Getatr('Main':'MI_Tips':'Checked')=0
C   Eval  %Setatr('Main':'MI_Tips':'Checked')=1
C   Else
C   Eval  %Setatr('Main':'MI_Tips':'Checked')=0
C   EndIf
C   Eval  %Setatr('Main':'Main':'ShowTips')=
         %Getatr('Main':'MI_Tips':'Checked')
C   ENDACT

Figure 23. Code for ODBC/JDBC Inquiry Example (Part 11 of 11)
Outline Box

Use an outline box around a group of parts to indicate that they are related.

An outline box is a rectangular, unlabeled box. If you need a label on the box, use the group box part instead.

For related information, see “Group Box” on page 82.

**Part Attributes**

<table>
<thead>
<tr>
<th>Bottom</th>
<th>Handle*</th>
<th>Height</th>
<th>Left</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParentName</td>
<td>PartName</td>
<td>PartType</td>
<td>Refresh</td>
</tr>
<tr>
<td>Top</td>
<td>UserData</td>
<td>Visible</td>
<td>Width</td>
</tr>
</tbody>
</table>

**Note:** See the attribute description for restrictions.

**Applicable Events**

Create

Destroy

**Special Height and Width Settings**

You can create lines using two outline box attributes. Set the **Width** attribute to 1 to create a vertical line, or set the **Height** attribute to 1 to create a horizontal line.
Pop-up Menu

Use the pop-up menu part to display a number of choices that pertain to a particular part on your interface. You can add menu item parts and submenu parts to the pop-up menu part.

The menu is called a “pop-up” because it appears when the user presses the appropriate key or mouse button.

Note: You can manipulate this part’s properties, events, and so on, only from its pop-up menu in the project tree view.

For related information, see:
• “Menu Bar” on page 105
• “Menu Item” on page 106
• “Submenu” on page 171

Part Attributes

Handle*    InvName    InvPName    ParentName
PartName    PartType    UserData    Visible*
X           Y

* Note: See the attribute description for restrictions.

Applicable Events

There are no events for this part.
Use the progress bar part to indicate graphically the progress of a process, such as copying files, loading a database, and so on.

For example, to show the progress of copying 100 files, you could set the `PBRange` attribute to 100 and the `PBStepSize` attribute to 10. Your code could then monitor the copyfile process and move the progress bar indicator forward in steps for every ten files copied.

In Java applications, if the progress bar’s width is smaller than its height, the progress bar will have a vertical orientation.

**Part Attributes**

<table>
<thead>
<tr>
<th>Bottom</th>
<th>Handle*</th>
<th>Height</th>
<th>Left</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParentName</td>
<td>PartName</td>
<td>PartType</td>
<td>PBRange</td>
</tr>
<tr>
<td>PBSSetPos</td>
<td>PBStep</td>
<td>PBStepSize</td>
<td>Top</td>
</tr>
<tr>
<td>UserData</td>
<td>Visible</td>
<td>Width</td>
<td></td>
</tr>
</tbody>
</table>

* Note: See the attribute description for restrictions.

**Applicable Events**

Create  Destroy

**Progress Bar Example**

In the following example, the progress bar indicator is updated every time a read operation occurs:

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq---- *
```

```
C EVAL %setatr('win01':'WIN1':'PBRange')=10
C EVAL %setatr('win01':'WIN1':'PBStepSize')=1
C DO 10
C Read Input
C EVAL %setatr('win01':'WIN1':'PBStep')=1
C EndDo
```

Chapter 7. Using Parts  139
Push Button

Use push buttons to provide convenient access to frequently used actions.

Each push button part controls a specific action. When the user clicks on a push button, its action is initiated immediately. The text label on the push button describes its action.

Compare with "Graphic Push Button" on page 80.

Part Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>BackColor</th>
<th>BackMix</th>
<th>Border*</th>
<th>Bottom</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enabled</td>
<td>Focus</td>
<td>FontBold</td>
<td>FontItalic</td>
<td>FontUnder*</td>
</tr>
<tr>
<td>FontName</td>
<td>FontSize</td>
<td>FontStrike*</td>
<td>Height</td>
<td></td>
</tr>
<tr>
<td>ForeColor</td>
<td>ForeMix</td>
<td>Handle*</td>
<td>Left</td>
<td></td>
</tr>
<tr>
<td>HelpEnable</td>
<td>HighLight</td>
<td>Label</td>
<td>Refresh</td>
<td></td>
</tr>
<tr>
<td>ParentName</td>
<td>PartName</td>
<td>PartType</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ShowTips</td>
<td>TipText</td>
<td>Top</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Validate</td>
<td>Visible</td>
<td>Width</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Note: See the attribute description for restrictions.

Applicable Events

<table>
<thead>
<tr>
<th>Event</th>
<th>Create</th>
<th>Destroy</th>
<th>GotFocus</th>
<th>LostFocus</th>
</tr>
</thead>
<tbody>
<tr>
<td>MouseEnter</td>
<td>MouseExit</td>
<td>MouseMove</td>
<td>Popup</td>
<td></td>
</tr>
</tbody>
</table>

Setting a Default Push Button

In the properties notebook of a push button part, you can specify that you want the push button to be the default push button for the window you are designing. The default push button is displayed with a heavy black border, and the action associated with it is performed when the user presses the Enter key.

Note: You can define only one default push button per window. If you define more than one, VisualAge RPG will choose one.

Setting a Mnemonic

Note: Mnemonics are not supported in Java applications.

For each push button, use the Label attribute to associate text with a specific push button. That text appears on the button.

To specify a mnemonic key for the push button, place the mnemonic identifier in front of a character in the text of the Label attribute. For Windows, use an ampersand (&). This designated character is displayed on the interface with an
underscore (for example, Cancel). The underscore informs users that they can select the push button by pressing the Alt key and the underlined character on the keyboard.

**Assigning Command Keys**

You can assign a command key to a push button. To do this, open the part’s properties notebook and select one of the command keys from the available list.

When the user presses the command key at run time, it has the same effect as pressing the mouse button or a key on the keyboard. A **Press** event is signaled to your program.

**Signaling Events**

A **Press** event is signaled to your program when:

- the user selects a push button.
- the user presses the Enter key if a default push button is defined.
- the user presses a command key that is assigned to a push button.
Radio Button

Use radio buttons if you want the user to select only one of a group of related but mutually exclusive choices. When the user makes a selection, the previously selected choice in the group is deselected.

A radio button appears as a raised circular button that is labeled with text beside it. When selected, the circular button displays a dot.

Do not use radio buttons if you want the user to be able to select more than one choice at a time. In that case, see “Check Box” on page 58.

Part Attributes

<table>
<thead>
<tr>
<th>BackColor</th>
<th>BackMix</th>
<th>Bottom</th>
<th>Checked</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enabled</td>
<td>Focus</td>
<td>FontBold</td>
<td>FontItalic</td>
</tr>
<tr>
<td>FontName</td>
<td>FontSize</td>
<td>FontStrike*</td>
<td>FontUnder*</td>
</tr>
<tr>
<td>ForeColor</td>
<td>ForeMix</td>
<td>Handle*</td>
<td>Height</td>
</tr>
<tr>
<td>Highlight*</td>
<td>Label</td>
<td>Left</td>
<td>ParentName</td>
</tr>
<tr>
<td>PartName</td>
<td>PartType</td>
<td>Refresh</td>
<td>SelectIdx</td>
</tr>
<tr>
<td>ShowTips</td>
<td>TipText</td>
<td>Top</td>
<td>UserData</td>
</tr>
<tr>
<td>Visible</td>
<td>Width</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Note: See the attribute description for restrictions.

Applicable Events

<table>
<thead>
<tr>
<th>Create</th>
<th>Destroy</th>
<th>Enter</th>
<th>MouseEnter</th>
</tr>
</thead>
<tbody>
<tr>
<td>MouseExit</td>
<td>MouseMove</td>
<td>Popup</td>
<td>Select</td>
</tr>
</tbody>
</table>

Setting a Mnemonic

To specify a mnemonic key for the radio button, place the mnemonic identifier in front of a character in the text of the Label attribute. For Windows, use an ampersand (&). This designated character is displayed on the interface with an underscore (for example, Blue). The underscore informs users that they can select the radio button by pressing the underlined character on the keyboard.

Note: Mnemonics are not supported in Java applications.

Grouping Radio Buttons

When you create radio buttons, group them logically, so that selecting a button will affect only the state of buttons in its own group.

For example, assume that you have four radio buttons on a design window. RB1 and RB2 are mutually exclusive to each other, and RB3 and RB4 are mutually exclusive to each other. You must group these buttons into two logical groups. The following figure illustrates how these radio buttons can be grouped on the design window:
To arrange radio buttons in logical groups:
1. Arrange the radio buttons as desired, and optionally place a group box around each group. (See "Group Box" on page 82.)
2. Select the canvas part in the design window and press mouse button 2.
3. From the pop-up menu, select **Tabs and Groups**.

   The Customize Tabs and Groups window appears, listing all of the parts on the design window. Resize this window, if necessary, to see all the parts.
4. Click mouse button 1 to select the radio button that will be the first button in the first group. In this example, RB1 is the first radio button in group 1.
5. Click mouse button 2 to get the pop-up menu for this radio button part, and select **Group mark**.

   An X mark symbol appears next to the radio button under the **Group Mark** column.

   **Note:** You can also set the group mark in the properties notebook for the part.

6. Use Ctrl+Up Arrow and Ctrl+Down Arrow to position RB2 after RB1. Note that this positioning can also be done within the tree view and can also be done using **move up** and **move down** menu items. Ensure that the **Group mark** attribute is not set for RB2.

   This specifies that RB2 is the second radio button in group 1.
7. Press **OK** to close the window.

   **Note:** Do **not** close the window using the system menu, or your changes will not be saved.

RB1 and RB2 are now considered to be part of one group, so selecting either will only affect the other. Repeat the same process for RB3 and RB4. The following figure shows the Customize Tabs and Groups window after parts have been grouped:
Note: Tab stops and group marks can also be set for individual parts from within a part’s properties notebook.

Setting the State of a Radio Button

In the radio button’s properties notebook, you can indicate if the radio button is to be initially selected or not. Only one radio button in a group may be selected at one time. If you select more, only the last one in the group will be selected.

By default, when you create a radio button, the Checked attribute is set to 0. This means that the radio button is not set and the state is turned off. The radio button is displayed with the circle empty.

If you want to create a radio button that is set and the state is turned on, you must set the Checked attribute to 1. In this case, the radio button is displayed with the circle partially filled.

You can set the Checked attribute in the properties notebook or in your program.

Signaling Events

When the user selects a radio button, a Select event is signaled.
Use the slider part if you want the user to be able to display, set, or modify a value by moving a slider arm along a slider shaft.

Sliders are typically used for values that have familiar increments, such as seconds or degrees, or to show the percentage of a task that has been completed.

By default, a slider is placed horizontally in the center of a box with the slider shaft on the left side. A scale can be displayed to show the units of measure for the shaft.

Use the properties notebook for the slider part to:
• Set the range of values that a slider can return
• Position the slider vertically or horizontally in a window
• Provide a scale to indicate the units of measure represented by the slider

**Part Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddLink*</td>
<td>Add link to the slider</td>
</tr>
<tr>
<td>Bottom</td>
<td>Enable the slider</td>
</tr>
<tr>
<td>FontItalic</td>
<td>Set or get font italic</td>
</tr>
<tr>
<td>FontUnder*</td>
<td>Set or get font under</td>
</tr>
<tr>
<td>Height</td>
<td>Set or get height</td>
</tr>
<tr>
<td>ParentName</td>
<td>Set or get parent name</td>
</tr>
<tr>
<td>RemoveLink*</td>
<td>Remove link from the slider</td>
</tr>
<tr>
<td>TipText</td>
<td>Set or get tip text</td>
</tr>
<tr>
<td>Visible</td>
<td>Set or get visibility</td>
</tr>
<tr>
<td>Value</td>
<td>Set or get value</td>
</tr>
</tbody>
</table>

*Note: See the attribute description for restrictions.*

**Applicable Events**

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change</td>
<td>Signal when the slider value changes</td>
</tr>
<tr>
<td>Create</td>
<td>Create event</td>
</tr>
<tr>
<td>Destroy</td>
<td>Destroy event</td>
</tr>
<tr>
<td>GotFocus</td>
<td>Got focus event</td>
</tr>
<tr>
<td>Link*</td>
<td>Link event</td>
</tr>
<tr>
<td>LostFocus</td>
<td>Lost focus event</td>
</tr>
<tr>
<td>MouseEnter</td>
<td>Mouse enter event</td>
</tr>
<tr>
<td>MouseExit</td>
<td>Mouse exit event</td>
</tr>
<tr>
<td>Popup</td>
<td>Popup event</td>
</tr>
</tbody>
</table>

*Note: See the event description for restrictions.*

**Getting and Setting the Slider Value**

You can get or set the value of the slider by using the **Value** attribute.

When you get the **Value** attribute, make sure that you have defined a large enough result field to contain the returned value.

**Signaling Events**

The **Change** event is signaled when the position of the slider arm changes.
If you use increment buttons to move the slider arm, the Change event is signaled continuously as long as the buttons are pressed.

If you use the mouse to move the slider arm, the Change event occurs when the mouse button is released.

**Slider Example**

This example illustrates how the slider part can be used to control the color of other parts by using the BackMix attribute. As each slider is moved, its value is used to determine the background color mix of its corresponding static text part to show the intensity of that color. The background color of the static text labeled Sample is updated to show the combined color mix of all three colors.
Program ID . .: Slider

Description .: Sample program to illustrate the slider part.

As each slider arm is moved, a CHANGE event is signalled for that slider.

The CHANGE action subroutine retrieves the value of the slider, and updates the background colour mix of its corresponding static text part to show the intensity of that colour.

The background colour mix of static text part 'SAMPLE' is also updated to show the result of mixing all the colour values.

*********************************************************************************

H

*********************************************************************************

* Window . .: MAIN
* * Part . .: PB_EXIT
* * * Event . .: PRESS
* * * Description: Terminate the program.
* * *
*********************************************************************************

C   PB_EXIT   BEGACT   PRESS   MAIN
C   move   *on      *inlr
C   ENDACT

Figure 24. Coding Example Using the Slider Part (Part 1 of 4)
Figure 24. Coding Example Using the Slider Part (Part 2 of 4)
Subroutine UPDATE

Description: Updates the background colour mix of the static text part 'Sample' to show the results of combining the colour values from the three sliders.

******************************************************************************
* UPDATE BEGSR
* move *blanks smpmix 11
* smpmix cat ':':0 smpmix
* smpmix cat grnmix:0 smpmix
* smpmix cat ':':0 smpmix
* smpmix cat blumix:0 smpmix
* 'Sample' setatr smpmix 'BackMix'
* ENDSR
******************************************************************************
* Window . .: MAIN
* Part . .: RED
* Event . .: CHANGE
* Description: Update Red colour value.
******************************************************************************
* RED BEGACT CHANGE FRA0000B
* 'red' getatr 'Value' val
* move val redmix 3
* move *blanks mix
* movel redmix mix
* mix cat ':000:000':0 mix
* 'STRed' setatr mix 'BackMix'
* exsr update
* ENDACT

Figure 24. Coding Example Using the Slider Part (Part 3 of 4)
Figure 24. Coding Example Using the Slider Part (Part 4 of 4)
Spin Button

Use the spin button part to display, in sequence, a group of related but mutually exclusive choices that have a logical consecutive order; for example, months of the year. The choices are displayed as though they were arranged in a ring. The user can move (or “spin”) through the choices by pressing the up arrow to go to the next higher value, or the down arrow to go to the next lower one. Alternatively, one of the choices can be typed directly into the entry field for the spin button.

Part Attributes

| AddItemEnd | Alignment* | BackColor | BackMix |
| Bottom | Enabled | Focus | FontBold |
| FontItalic | FontName | FontSize | FontStrike* |
| FontUnder* | ForeColor | ForeMix | Handle* |
| Height | Left | Maximum | Minimum |
| ParentName | PartName | PartType | ReadOnly |
| Refresh | RemoveItem | ShowTips | Text |
| TipText | Top | UserData | Value |
| Visible | Width |

* Note: See the attribute description for restrictions.

Applicable Events

| Change | Create | Destroy | GotFocus |
| Link* | LostFocus | MouseEnter | MouseExit |
| MouseMove | Popup | SpinDown | SpinEnd |
| SpinUp |

* Note: See the event description for restrictions.

Setting Spin Button Values

The data type of the spin button determines the method used to set the spin list values.

To specify the allowed values for a numeric spin button, set the **Maximum** and **Minimum** attributes.

To set the initial spin list values of a character spin button, set the **AddItemEnd** attribute for each item you want to add. Add the items in the order in which you want them to appear because the character spin button items are not sorted automatically.

Getting the Spin Button Value

The attribute you use to retrieve the value that is selected in a spin button depends on the type of spin button.

- For character spin buttons, use the **Text** attribute.
• For numeric spin buttons, use the **Value** attribute. This attribute returns a value ranging from the minimum to the maximum value specified for the spin button.

**Preventing User Input**

You can prevent the user from typing a value directly into the field associated with the spin button by setting the **ReadOnly** attribute in the spin button properties notebook or by setting the **ReadOnly** attribute to 1 in your program.

**Spin Button Example**

This example illustrates how to set and get the values for a numeric and a character spin button. When you start the program, an initial list is inserted into each spin button. When you select the **Copy** push button, the value of each spin button is copied to the associated entry field part.

Press the **Close** push button to end the program.
SPIN

Description:
Sample program to demonstrate the Spin button part.
A Character, and Numeric spin button are used to show how they are initialized, and how their values are retrieved.

********************************************************************************

** Program ID .: SPIN
** Description .: Sample program to demonstrate the Spin button part.
** A Character, and Numeric spin button are used to show how they are initialized, and how their values are retrieved.
**
**********************************************************************************

H

DDAY S 10A DIM(7) PERRCD(1) CTDATA

********************************************************************************

** Window .: MAIN
** Part .: PB_COPY
** Event .: PRESS
** Description: Copy the value from each Spin button to its corresponding entry field part.
**
**********************************************************************************

C PB_COPY BEGACT PRESS MAIN

C 'SPB1' Getatr 'Value' tmp2N 2 0
C 'EF1' Setatr tmp2N 'Text'

C 'SPB2' Getatr 'Text' tmp 10
C 'EF2' Setatr tmp 'Text'

C ENDACT

Figure 25. Coding Example Using the Spin Button Part (Part 1 of 2)
Figure 25. Coding Example Using the Spin Button Part (Part 2 of 2)
Use the static text part as a label for other parts, such as a prompt for an entry field part. Static text parts do not accept end user input. In Java applications, static text can be displayed only on a single line.

### Part Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alignment</td>
<td>BackColor</td>
</tr>
<tr>
<td>Data Type</td>
<td>DragEnable*</td>
</tr>
<tr>
<td>Enabled</td>
<td>FontBold</td>
</tr>
<tr>
<td>Font Size</td>
<td>FontStrike*</td>
</tr>
<tr>
<td>Fore Mix</td>
<td>Handle*</td>
</tr>
<tr>
<td>Left</td>
<td>ParentName</td>
</tr>
<tr>
<td>Refresh</td>
<td>ShowTips</td>
</tr>
<tr>
<td>UserData</td>
<td>Visible</td>
</tr>
<tr>
<td></td>
<td>BackMix</td>
</tr>
<tr>
<td></td>
<td>DropEnable*</td>
</tr>
<tr>
<td></td>
<td>FontItalic</td>
</tr>
<tr>
<td></td>
<td>FontUnder*</td>
</tr>
<tr>
<td></td>
<td>Height</td>
</tr>
<tr>
<td></td>
<td>PartName</td>
</tr>
<tr>
<td></td>
<td>TipText</td>
</tr>
<tr>
<td></td>
<td>Width</td>
</tr>
<tr>
<td></td>
<td>Bottom</td>
</tr>
<tr>
<td></td>
<td>DropValue*</td>
</tr>
<tr>
<td></td>
<td>FontName</td>
</tr>
<tr>
<td></td>
<td>ForeColor</td>
</tr>
<tr>
<td></td>
<td>Label</td>
</tr>
<tr>
<td></td>
<td>PartType</td>
</tr>
<tr>
<td></td>
<td>Top</td>
</tr>
</tbody>
</table>

*Note: See the attribute description for restrictions.

### Applicable Events

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Click</td>
<td>Create</td>
</tr>
<tr>
<td>Drop</td>
<td>Link*</td>
</tr>
<tr>
<td>Mouse Exit</td>
<td>MouseMove</td>
</tr>
<tr>
<td></td>
<td>DblClick</td>
</tr>
<tr>
<td></td>
<td>MouseDown</td>
</tr>
<tr>
<td></td>
<td>Mouse Up</td>
</tr>
<tr>
<td></td>
<td>Destroy</td>
</tr>
<tr>
<td></td>
<td>Mouse Enter</td>
</tr>
<tr>
<td></td>
<td>Popup</td>
</tr>
</tbody>
</table>

*Note: See the event description for restrictions.

### Changing the Text of a Static Text Part

The static text part is a rectangular area into which text is placed. Use the **Label** attribute to change the text of a static text part.

If you change the text so that it is longer than the original text, the new text will be clipped at the borders of the enclosing rectangle. The text will also be clipped if you change the **Font Name** and **FontSize** attributes to a larger font or size.

When you change text in your program, make sure that the static text part in the GUI Designer is large enough to show the new text.

### Getting Static Text Values

To get the value of a static text part, you must specify the **Label** attribute. If you are getting the value of a numeric static text part, the field that receives the value must also be defined as numeric.

### Getting and Setting Information for a Window

During compilation, the compiler implicitly defines fields in your program with the same name as the static text part, and with the same data type and length. By using the READ and WRITE operation codes with a window name specified in factor 2, the **Label** attribute value is automatically copied to or from these fields.
The READ and WRITE operation codes are most useful if you have many static text parts in your user interface because you do not have to execute a series of get and set attributes.


**Editing Output**

You can edit the contents of a static text part if the data type is numeric. See Chapter 11, “Editing Output,” on page 239 for a description of editing.
Use the status bar part to provide additional information about a process or action for your window. You can create up to five panes for the status bar. The status bar part provides more flexibility than the `StatusBar` attribute for the window part.

By default, a status bar is created at the bottom of the window. However, you can use the properties notebook to reposition it to the top. You can also set the border style, number of panes, and text alignment.

**Part Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handle</td>
<td></td>
</tr>
<tr>
<td>ParentName</td>
<td></td>
</tr>
<tr>
<td>PartName</td>
<td></td>
</tr>
<tr>
<td>PartType</td>
<td></td>
</tr>
<tr>
<td>SBIndex</td>
<td></td>
</tr>
<tr>
<td>SBLabel</td>
<td></td>
</tr>
<tr>
<td>SBPanes</td>
<td></td>
</tr>
<tr>
<td>UserData</td>
<td></td>
</tr>
<tr>
<td>Visible</td>
<td></td>
</tr>
</tbody>
</table>

*Note: See the attribute description for restrictions.

**Applicable Events**

Create    Destroy

**Status Bar Example**

In the following example, the status bar label is updated while some initial processing occurs:

```c
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result+++++++Len++D+HiLoEq----

* C STBAR BEGACT CREATE MAIN
 C 'STBAR' SETATR 'Wait...' 'SBLABEL'

* Do some processing.
 *
 C DO
 C ...
 C ENDDO

* Clear the status bar label.
 *
 C 'STBAR' SETATR 'BLANKS' 'SBLABEL'
 C ENDATR
```

Chapter 7. Using Parts  157
Subfile

Use the subfile part to display a list of records, each consisting of one or more fields.

The subfile part has similar function to an iSeries™ subfile. The user can scroll horizontally or vertically through the list using the subfile’s scroll bars.

To create a subfile entry field, point-and-click on a field from the Define Reference Fields window or the parts palette and click it onto the subfile part. You can also add fields using the properties notebook.

Note: The subfile part can only be point-and-clicked onto a notebook page with canvas or window with canvas.

Part Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddItemEnd</td>
<td>AllowEdit, AutoSelect, BackColor</td>
</tr>
<tr>
<td>BackMix</td>
<td>Bottom, ButtonIdx, Buttons</td>
</tr>
<tr>
<td>ButtonTip</td>
<td>ByteComp, CapsLock, CellBGClr</td>
</tr>
<tr>
<td>ColBGMix</td>
<td>CellFGClr, CellFGMix, ColBGClr</td>
</tr>
<tr>
<td>ColBMix</td>
<td>ColFGMix, ColNumber</td>
</tr>
<tr>
<td>ColWidth</td>
<td>Count, DColFRVCol, DeSelect</td>
</tr>
<tr>
<td>EditColumn</td>
<td>EditIndex, EditText, EnableBtn</td>
</tr>
<tr>
<td>Enabled</td>
<td>ExtSelect*, FirstSel, Focus</td>
</tr>
<tr>
<td>FontArea</td>
<td>FontBold, FontItalic, FontName</td>
</tr>
<tr>
<td>FontSize</td>
<td>FontStrike*, FontUnder*, ForeColor</td>
</tr>
<tr>
<td>ForeMix</td>
<td>Handle*, HdgBGClr, HdgBGMix</td>
</tr>
<tr>
<td>HdgFGClr</td>
<td>HdgFgMix, HdgIdx, HdgText</td>
</tr>
<tr>
<td>Height</td>
<td>Hidden, HRule, Index</td>
</tr>
<tr>
<td>ItemCount</td>
<td>Left, MapViewCol, MultiSelect</td>
</tr>
<tr>
<td>NbrOfSel</td>
<td>OpenEdit, PageSize, ParentName</td>
</tr>
<tr>
<td>PartName</td>
<td>PartType, RemoveItem, RowBGClr</td>
</tr>
<tr>
<td>RowBMix</td>
<td>RowFGClr, RowFGMix, Scale</td>
</tr>
<tr>
<td>Selected</td>
<td>SelectItem, SelectList, SetTop</td>
</tr>
<tr>
<td>SflNxtChg</td>
<td>ShowTips, SizeToFit, SortAsc</td>
</tr>
<tr>
<td>SortDesc</td>
<td>StartAt, TipText, Top</td>
</tr>
<tr>
<td>TopRecord</td>
<td>UserData, VColFRDCol, ViewColumn</td>
</tr>
<tr>
<td>Visible</td>
<td>VRule, Width</td>
</tr>
</tbody>
</table>

* Note: See the attribute description for restrictions.

Applicable Events

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change</td>
<td>ColSelect, Create, Destroy</td>
</tr>
<tr>
<td>Enter</td>
<td>FirstRec, GotFocus, KeyPress</td>
</tr>
<tr>
<td>LastRec</td>
<td>LostFocus, MouseEnter, MouseExit</td>
</tr>
<tr>
<td>MouseMove</td>
<td>NextRec, PageEnter, PageDown</td>
</tr>
<tr>
<td>PageTop</td>
<td>PageUp, Popup, PageEnd</td>
</tr>
<tr>
<td>Select</td>
<td>VKeyPress, PrevRec</td>
</tr>
</tbody>
</table>
Creating a Subfile Part
You can create a subfile part only on a canvas part.

Maximum Number of Fields per Subfile
You can define a maximum of 99 fields for a subfile.

Operation Codes for Manipulating Subfile Parts
In addition to using attributes to control a subfile, you can use several operation codes to affect the subfile part. Specify the name of the subfile in factor 2. Do not enclose it in quotation marks.

The following operation codes are supported. For a complete description of each, refer to VisualAge RPG Language Reference, SC09-2451-04, or to the language-sensitive help.

<table>
<thead>
<tr>
<th>Code</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAIN</td>
<td>Reads a record from a subfile by specifying an index.</td>
</tr>
<tr>
<td>CLEAR</td>
<td>Clears all records from the subfile.</td>
</tr>
<tr>
<td>DELETE</td>
<td>Deletes a record from the subfile. All records following the deleted record are moved up one position.</td>
</tr>
<tr>
<td>READC</td>
<td>Reads a record if the value of any of the entry fields in the record has changed.</td>
</tr>
<tr>
<td>READS</td>
<td>Reads a selected record from the subfile. Users can select a record with either the mouse or the keyboard. After the record has been read, it is deselected.</td>
</tr>
<tr>
<td>UPDATE</td>
<td>Updates an existing subfile record. A record must have been read before this operation code can be used.</td>
</tr>
<tr>
<td>WRITE</td>
<td>Adds a new record to the subfile.</td>
</tr>
</tbody>
</table>

Loading a Subfile
To display information in a subfile part, the information is written one record at a time to the subfile part. Subfile fields that were defined in the GUI Designer for the subfile part are set to the desired values, and the WRITE operation is performed on the subfile record format.

Determining the Subfile Size
Unlike the iSeries 400 subfile, the subfile part does not have subfile or subfile page sizes. The number of records a subfile can hold is limited by the amount of memory on your workstation. The subfile page size (that is, the number of records shown at one time) is determined when you create the subfile in the GUI Designer.

Getting the Record Count
To determine how many records are currently in a subfile, use the Count attribute.
Reading and Updating Records

Records in a subfile part can be updated or deleted. To update records, you must first position the subfile to the record that you want to update. You can position the subfile by a CHAIN, READC, or READS operation. These operations cause the field values from the retrieved record to be assigned to the corresponding program fields for the subfile record format. Your program can then modify the field values.

An UPDATE operation that is run on the subfile part then sends the current values from the associated fields back out to the subfile. Use CHAIN to select records by relative position within a subfile, READC to select records that the user has changed on the subfile display, and READS for the records that the user selected.

The following example shows a READS operation do a loop to obtain all the selected records in a subfile, process them, and update them one record at a time. This is coded in an action subroutine for the **Press** event for a push button called Report.

```
*: 
  C REPORT BEGACT PRESS WIN1 
  C READS SUBF1 99 
  * *IN99 DONEQ *OFF 
  * * For the selected record, process it, and mark it 
  * as 'Reported' in the subfile display. 
  * 
  C MOVE ' (Reported)' SFNAME 
  * 
  C UPDATE SUBF1 
  C READS SUBF1 99 
  C END 
  C ENDACT
```

*Figure 26. Coding example of reading and modifying records*

Changing Subfile Fields

**Note:** A subfile field cannot be changed by the user if it is set as read-only.

Before a field in a subfile can be changed by the user, it must be opened for editing, either by the user of your application or by you in your program:

1. The user selects the field with the mouse pointer then clicks mouse button 1 while holding down the Alt key. The user can then use the tab and back-tab keys to move to different fields on the same record, and use the up and down arrow keys to move to different records.

2. To open a field for editing in your program:
   a. Use the **Index** attribute to indicate which record contains the field to be edited.
   b. Set the **ColNumber** attribute to indicate the column number of the field to be edited.
   c. Set the **OpenEdit** attribute value to 1 to open the field for editing. (You can set this attribute value to 0 to close any fields that are currently open for editing.)
Use the READC operation to determine if the user has changed any field in the subfile.

**Hidden Fields**

In the subfile part’s properties notebook, you can set subfile fields to be Hidden so that they are not displayed. For example, a subfile record can contain record key information in a hidden field. You cannot see the hidden field, but the field is returned to the program with the subfile record.

**Formatting Subfile Fields**

The fields in a subfile can be highlighted in several ways. Foreground and background colors can be set for both subfile headings and individual subfile fields. You can place horizontal or vertical line separators within a subfile.

See [Chapter 11, “Editing Output,” on page 239](#) for information.

**Enabling Tabbing**

Use the Customize Tabs and Groups dialog to enable tabbing to a subfile part. Right-click on the canvas part of the window containing the subfile part. Select Tabs and Groups from the pop-up menu. The Customize Tabs and Groups dialog appears. To set or clear the Tab stop setting, right-click on the part name and select the Tab stop menu item.

**Subfile Example**

In the following example, a subfile part is used to display records from a database file on an iSeries 400 server. Rather than filling the subfile with all records from the database, navigation push buttons (FirstRec, LastRec, PageTop, PageUp, PageDown, PrevPage, NextPage) are provided to control scrolling through the records in the subfile.

When you press the Select push button, the READS operation code is used to determine which record was selected, and the value of the CUSTNO field is displayed in the static text part. Also, the first field in the record is opened for editing.

Select the Exit menu item to end the program.
Figure 27. Coding Example Using the Subfile Part (Part 1 of 10)

Figure 27. Coding Example Using the Subfile Part (Part 2 of 10)
Subroutine . . : PREVPAGE
* Description . : Return the previous page of records from the database.

*********************************************************************
* Subroutine . . : PREVPAGE
* Description . : Return the previous page of records from the database.
*********************************************************************

C PREVPAGE BEGSR
C Sub PageSize CurRec
C CurRec IfLe *zero Add PageSize CurRec
C Else Exsr FillPage
C 'SFL1' setatr 5 'BUTTONIDX'
C 'SFL1' setatr 1 'ENABLEBTN'
C EndIf
C ENDSR

*********************************************************************
* Subroutine . . : FILLPAGE
* Description . : Fill the subfile part with a page of records from the database.
*********************************************************************

C FILLPAGE BEGSR
C Clear SFL1
C CurRec Setll customer Z-Add 1 count 2 0
C Read customer 9999
C *in99 DoWeq *off
C count AndLE PageSize
C Write SFL1
C If %Getatr('Main':'HILITE':'Checked')=1
C 'SFL1' Setatr Count 'Index'
C 'SFL1' Setatr 1 'ColNumber'
C 'SFL1' Setatr +DarkGreen 'CellFGClr'
C 'SFL1' Setatr 2 'ColNumber'
C 'SFL1' Setatr +DarkPink 'CellFGClr'
C 'SFL1' Setatr 3 'ColNumber'
C 'SFL1' Setatr +DarkBlue 'CellFGClr'
C EndIf

Figure 27. Coding Example Using the Subfile Part (Part 3 of 10)
Figure 27. Coding Example Using the Subfile Part (Part 4 of 10)
Figure 27. Coding Example Using the Subfile Part (Part 5 of 10)
Figure 27. Coding Example Using the Subfile Part (Part 6 of 10)
Figure 27. Coding Example Using the Subfile Part (Part 7 of 10)
Figure 27. Coding Example Using the Subfile Part (Part 8 of 10)
Figure 27. Coding Example Using the Subfile Part (Part 9 of 10)
Signaling Events

The Select event is signaled when:
- The user selects an item that is in a subfile
- You select an item in the list in your program
- The user selects an item that is already selected

The Enter event is signaled when:
- The user double-clicks over an item that is in the subfile
- The user presses the Enter key when the subfile has focus, and an item has been selected

In your action subroutine for these events, you can use the READS operation code to determine which item was selected.

Figure 27. Coding Example Using the Subfile Part (Part 10 of 10)
Submenu

Use a submenu to:
• Start a new cascaded menu from a menu item on an existing menu.
• Start a pull-down menu from a menu item on the menu bar.

After creating a submenu, you can add menu items to it by pointing-and-clicking (or dragging-and-dropping) the menu item part onto the submenu part in the tree view only.

Note: You can manipulate this part’s properties, events, and so on, only from its pop-up menu in the project tree view.
For related information, see “Menu Item” on page 106.

Part Attributes

<table>
<thead>
<tr>
<th>ParentName</th>
<th>PartName</th>
<th>PartType</th>
<th>UserData</th>
</tr>
</thead>
</table>

Applicable Events

Create
Destroy
Timer

Use the timer part if your program must perform certain operations at preset time intervals. For example, you can use it to close a window, or perhaps end an application, after a certain period of inactivity.

A timer part counts units of time and tracks the preset time interval between two events, triggering the second event once the interval has passed.

When you create a timer part in the GUI builder, the part is represented as an icon on the design window. However, in the properties notebook for a timer part, you can specify that you do not want the icon displayed while the program is executing.

Note: Do not use the timer part when precise timing is required. Due to other programs running on your system, the Tick event may not necessarily occur at the exact interval you specify.

Part Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddLink*</td>
<td></td>
</tr>
<tr>
<td>Left</td>
<td></td>
</tr>
<tr>
<td>PartType</td>
<td></td>
</tr>
<tr>
<td>Top</td>
<td></td>
</tr>
<tr>
<td>AllowLink*</td>
<td></td>
</tr>
<tr>
<td>Multiplier</td>
<td></td>
</tr>
<tr>
<td>RemoveLink*</td>
<td></td>
</tr>
<tr>
<td>UserData</td>
<td></td>
</tr>
<tr>
<td>Bottom</td>
<td></td>
</tr>
<tr>
<td>ParentName</td>
<td></td>
</tr>
<tr>
<td>TimerMode</td>
<td></td>
</tr>
<tr>
<td>Interval</td>
<td></td>
</tr>
<tr>
<td>PartName</td>
<td></td>
</tr>
<tr>
<td>TimerTicks</td>
<td></td>
</tr>
<tr>
<td>Visible</td>
<td></td>
</tr>
</tbody>
</table>

* Note: See the attribute description for restrictions.

Applicable Events

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td></td>
</tr>
<tr>
<td>Destroy</td>
<td></td>
</tr>
<tr>
<td>Link*</td>
<td></td>
</tr>
<tr>
<td>Tick</td>
<td></td>
</tr>
</tbody>
</table>

* Note: See the event description for restrictions.

Displaying the Timer Icon

By default, the Visible attribute is set to 1 so that the timer icon is displayed while the program is executing. If you do not want this icon displayed, set this attribute to 0.

Setting the Interval

The timer interval is expressed in milliseconds. When the interval elapses, a timer Tick event is signaled. You can set this interval in the timer part’s properties notebook. You can also set it in your program by using the Interval attribute.

Note: The minimum timer interval is 100 milliseconds.

The timer part has a Multiplier attribute. By setting this attribute you can determine how many times the interval value elapses before a timer Tick event is generated. The default multiplier value is set to 1, so that the timer generates a Tick event at the end of each interval.
Generating Tick Events
When a timer is started, its interval value is reset to zero. When the interval value is reached, the timer generates a Tick event and updates the interval value.

Getting the Timer Value
Each time the timer generates a Tick event, its value is incremented by one. Use the Value attribute to get the current value of the timer. You can set the timer value in the properties notebook or in your program.

Controlling the Timer Using Timer Modes
Use the TimerMode attribute to control the timer.

Set TimerMode to 1 to start the timer. Starting the timer causes it to begin generating Tick events, and its Value attribute is incremented when the interval value is reached.

Set TimerMode to 2 to stop the timer. When the timer stops, it ceases generating Tick events, and its value is not updated.

Timer Example
In this example, a static text part is moved in the window for each timer Tick event.

When you press the Start push button, the timer mode is set to 1. This starts the timer and generates Tick events. During the processing of the Tick event, new coordinates are calculated for the static text part, and the part is set to the new location.

When you press the Stop push button, the TimerMode is set to 2. This stops the timer.

Press the Close push button to terminate the program.
**VARPG**

**Figure 28. Coding Example Using the Timer Part (Part 1 of 6)**
Window . . : FRA0000B
Part . . : FRA0000B
Event . . : CREATE

Description: Center the window on the display.

Calculate starting values.
Since the height attribute of the window part includes the title bar, we subtract the height of the title bar so the static text part remains within the window frame.

For SVGA, this value is about 20 pixels. It could be adjusted for other resolutions.

C    FRA0000B        BEGACT    CREATE    FRA0000B
Get beginning window height and width
C    'FRA0000B'    getatr    'Height'    winHeight    4 0
C    'FRA0000B'    getatr    'Width'    winWidth    4 0

Center the window on the display
C    eval    %setatr('FRA0000B';
C    'FRA0000B';
C    'Left')=(%DspWidth-winWidth)/2
C    eval    %setatr('FRA0000B';
C    'FRA0000B';
C    'Bottom')=(%DspHeight-winHeight)/2

Get beginning coordinates of static text part
C    'ST1'    getatr    'Left'    picX    4 0
C    'ST1'    getatr    'Bottom'    picY    4 0

Get dimensions of static text part
C    'ST1'    getatr    'Height'    picHeight    4 0
C    'ST1'    getatr    'Width'    picWidth    4 0

Calculate minimum and maximum Y coordinates
C    'Start'    getatr    'Height'    startH    4 0
C    'Start'    getatr    'Bottom'    startB    4 0
C    eval    minY = startB + startH
C    eval    maxY = winHeight - picHeight - 20

Figure 28. Coding Example Using the Timer Part (Part 2 of 6)
* Calculate maximum X coordinate
C       eval     maxX = winWidth - picWidth
*
C       ENDACT
*********************************************************************
*
* Window . . : FRA0000B
* * *
* Part . . : START
* *
* Event . . : PRESS
* *
* Description: Start the timer.
* *
*********************************************************************
*
C       START   BEGACT   PRESS   FRA0000B
*
C       'Timer1'   setatr  1   'TimerMode'
*
C       ENDACT
*********************************************************************
*
* Window . . : FRA0000B
* * *
* Part . . : STOP
* *
* Event . . : PRESS
* *
* Description: Stop the timer.
* *
*********************************************************************
*
C       STOP    BEGACT   PRESS   FRA0000B
*
C       'Timer1'   setatr  2   'TimerMode'
*
C       ENDACT

Figure 28. Coding Example Using the Timer Part (Part 3 of 6)
Figure 28. Coding Example Using the Timer Part (Part 4 of 6)
Figure 28. Coding Example Using the Timer Part (Part 5 of 6)

*********************************************************************
* Window . . : FRA0000B
* Part . . : FRA0000B
* Event . . : RESIZE
* Description: Get the size of the window after it has been resized
so static part uses entire window.
* *
*********************************************************************

C FRA0000B BEGACT RESIZE FRA0000B
C eval maxY = %NewHeight - picHeight - 20
C eval maxX = %NewWidth - picWidth
C ENDACT

Figure 28. Coding Example Using the Timer Part (Part 6 of 6)
Use the vertical scroll bar part to allow users to scroll through a pane of information vertically. The information can be a list of files, records in a database, columns in a document, and so on. You can use the `Range` attribute to represent the total number of objects to be scrolled through and the `PageSize` attribute to determine the number of objects that can be displayed on a page.

**Part Attributes**

<table>
<thead>
<tr>
<th>Bottom</th>
<th>Enabled</th>
<th>Focus</th>
<th>Handle*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Height</td>
<td>Left</td>
<td>NextLine</td>
<td>NextPage</td>
</tr>
<tr>
<td>PageSize</td>
<td>ParentName</td>
<td>PartName</td>
<td>PartType</td>
</tr>
<tr>
<td>Position</td>
<td>PrevLine</td>
<td>PrevPage</td>
<td>Range</td>
</tr>
<tr>
<td>Top</td>
<td>UserData</td>
<td>Visible</td>
<td>Width</td>
</tr>
</tbody>
</table>

* Note: See the attribute description for restrictions.

**Applicable Events**

| Create | Destroy | Scroll |
Window

Windows are the user’s primary means of interacting with your program. Your application must contain at least one window.

You can add only one part to the client area of a window, except for parts that are extensions to the window frame, such as menu bars, pop-up menus and message subfiles. The part you add is automatically sized to fit the client area.

If you want a window to contain more than one part, you must add a canvas part to it. Or, use the window with canvas part to save a step.

**Note:** The window part is located in the *Frames* section of the parts catalog, not on the parts palette.

For related information, see:
- “Canvas” on page 56
- “Window with Canvas” on page 181

**Part Attributes**

<table>
<thead>
<tr>
<th>Bottom</th>
<th>Center</th>
<th>Enabled</th>
<th>FileName*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Focus*</td>
<td>FontBold*</td>
<td>FontItalic*</td>
<td>FontName*</td>
</tr>
<tr>
<td>FontSize*</td>
<td>FontStrike*</td>
<td>FontUnder*</td>
<td>Handle*</td>
</tr>
<tr>
<td>Height</td>
<td>IconHandle*</td>
<td>Label</td>
<td>Left</td>
</tr>
<tr>
<td>MouseIcon*</td>
<td>MouseShape*</td>
<td>ParentName</td>
<td>PartName</td>
</tr>
<tr>
<td>PartType</td>
<td>PBRange</td>
<td>PBSetPos</td>
<td>PBStep</td>
</tr>
<tr>
<td>PBStepSize</td>
<td>Print</td>
<td>PrintAsIs</td>
<td>ProgressBar</td>
</tr>
<tr>
<td>Refresh</td>
<td>SLabel</td>
<td>SBPosition</td>
<td>SBStyle</td>
</tr>
<tr>
<td>ShowTips</td>
<td>StatusBar</td>
<td>Top</td>
<td>UserData</td>
</tr>
<tr>
<td>Visible</td>
<td>Width</td>
<td>WindowMode*</td>
<td></td>
</tr>
</tbody>
</table>

* **Note:** See the attribute description for restrictions.

**Applicable Events**

- Activate
- Close
- Create
- DeActivate
- Destroy
- LClickTray
- Moved
- RClickTray
- ReSize
- ShutDown
- ShutDown
- ShutDown

180 Programming with VisualAge RPG
Window with Canvas

Windows are the end user’s primary means of interacting with your program. The canvas, on a window with canvas part, allows you to add many parts to the window.

You can point and click various parts onto the canvas portion, position them, and organize them to produce a graphical user interface. You can also add parts that are extensions of the window’s frame, such as menu bars, pop-up menus and message subfiles.

If you need to put only one part on the client area of the window, you do not need the window with canvas part: you should use the window part instead (found in the Frames section of the parts catalog). Without a canvas, the part you add will be automatically sized to fit the client area.

For related information, see:
• “Canvas” on page 56
• “Window” on page 180

Part Attributes

<table>
<thead>
<tr>
<th>Bottom</th>
<th>Center</th>
<th>Enabled</th>
<th>FileName*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Focus*</td>
<td>FontBold*</td>
<td>FontItalic*</td>
<td>FontName*</td>
</tr>
<tr>
<td>FontSize*</td>
<td>FontStrike*</td>
<td>FontUnder*</td>
<td>Handle*</td>
</tr>
<tr>
<td>Height</td>
<td>IconHandle*</td>
<td>Label</td>
<td>Left</td>
</tr>
<tr>
<td>MouseIcon*</td>
<td>MouseShape*</td>
<td>ParentName</td>
<td>PartName</td>
</tr>
<tr>
<td>PartType</td>
<td>PBRange</td>
<td>PBSethPos</td>
<td>PBStep</td>
</tr>
<tr>
<td>PBStepSize</td>
<td>Print</td>
<td>PrintAsIs</td>
<td>ProgresBar</td>
</tr>
<tr>
<td>Refresh</td>
<td>SBLabel</td>
<td>SBPosition</td>
<td>SBSyle</td>
</tr>
<tr>
<td>ShowTips</td>
<td>StatusBar</td>
<td>Top</td>
<td>UserData</td>
</tr>
<tr>
<td>Visible</td>
<td>Width</td>
<td>WindowMode*</td>
<td></td>
</tr>
</tbody>
</table>

* Note: See the attribute description for restrictions.

Applicable Events

<table>
<thead>
<tr>
<th>Activate</th>
<th>Close</th>
<th>Create</th>
<th>DeActivate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Destroy</td>
<td>LClickTray</td>
<td>Moved</td>
<td>RClickTray</td>
</tr>
<tr>
<td>ReSize</td>
<td>ShutDown</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Displaying a Window

By default, all windows are marked as Visible and Open Immediately when they are created in the GUI Designer.

Decide which window you want the user to see first. That window is called the main or primary window and you must set the Visible and Open Immediately attributes accordingly for it. If you do not change the default settings, all the windows will appear when the user starts your application.
Setting the Open Immediately attribute
Set this attribute at design time if you want the window to be created when the application starts. Creating a window loads it into memory; because there is an overhead associated with this, you should decide which windows need to be loaded when the application starts. (You can have the other windows loaded later on.) You can use the SHOWWIN operation code to display windows that are not displayed very often (such as a window that displays a product copyright), instead of setting them so that they open immediately.

Note: The Open Immediately attribute does not control whether a window is actually displayed on the screen. To display a window, you must set its Visible attribute to 1 in your program, or mark it as Visible in its properties notebook.

Using the SHOWWIN operation code
You can load a window in your program by specifying the window name in Factor 2 of the SHOWWIN operation code. This operation code loads the window into memory.

Note: The SHOWWIN operation does not control whether a window is actually displayed on the screen. To display a window, you must set its Visible attribute to 1 in your program or mark it Visible in its properties notebook.

You can set a window’s attributes only after it has been loaded. To load a window, either select the Open Immediately check box on the Startup page of the part’s notebook, or use the SHOWWIN operation code in your program.

If a window is defined as Open Immediately, and you issue the SHOWWIN operation code for that window in your program, you will receive a runtime error indicating that the window has already been loaded. You can avoid this error by coding an error indicator on the SHOWWIN operation code and checking the error indicator in your program. If the indicator is turned on, then the window is already up and you should set the Visible attribute on. This will display the window, and the error will not be issued.

Referencing
The parts on a window are created when the window is created. Therefore, if you attempt to reference any part on a window that has not been loaded, or to reference a window attribute before the window is created, you will receive a Part not found message.

---

Hint
If a window is displayed and you cannot click on its title bar, use this method to move the window:
1. Position the mouse cursor somewhere on the visible portion of the window.
2. Click and release mouse button 1.
3. Press the Alt-space key combination. Then press M.
4. Use the arrow keys to reposition the window.
5. When the window is in the desired position, press Enter.
Resizing a Window

There are two things you can do to create your application so that the user has one or more ways to resize a window:

- In the GUI Designer, set the border of a window as **Sizeable**. This setting allows the user to select the window border with the mouse button, and resize the border while keeping the mouse button pressed. When the mouse button is released, the ReSize event is signaled.
- Add a **Maximize** and a **Minimize** button to the window. The user can then change the size of the window by selecting one of these buttons.

You can position parts on the window so that they maintain their relative position and size within the window’s boundaries after the window is resized. To do this, use the ReSize event with the %NewHeight and %NewWidth event attributes.

In the following coding example, a push button part labeled PB1 is located in the upper right corner of a window. When the window is resized, the ReSize action subroutine calculates new **Left** and **Bottom** attribute values to ensure that the push button remains within the window’s boundaries.

```
H
* Declare display size System attributes
D%DspHeight  S   4  0
D%DspWidth   S   4  0
*

Figure 29. Ensuring parts are displayed correctly after a window is resized (Part 1 of 3)
```
* Declare %NewHeight, and %NewWidth event attributes. These will contain the width and height of the window after it has been resized.

```c
D%NewHeight S 4 0
D%NewWidth S 4 0
```

*********************************************************************
* Window . . : FRA0000B
* *
* Part . . : FRA0000B
* *
* Event . . : RESIZE
* *
* Description: Ensure the push button part 'PB1' remains visible. *
* *
*********************************************************************

```c
C FRA0000B BEGACT RESIZE FRA0000B
C %NewWidth sub HOffset NewLeft 4 0
C %NewHeight sub VOffset NewBottom 4 0
C 'PB1' setatr NewLeft 'Left'
C 'PB1' setatr NewBottom 'Bottom'
C ENDACT
```

*********************************************************************
* Window . . : FRA0000B
* *
* Part . . : PSB0000D
* *
* Event . . : PRESS
* *
* Description: Terminate the program.
* *
*********************************************************************

```c
C PSB0000D BEGACT PRESS FRA0000B
C move *on *inlr
C ENDACT
```

Figure 29. Ensuring parts are displayed correctly after a window is resized (Part 2 of 3)
### Setting the Focus

Determine which window you want the user to work with first, and use the **Focus** attribute to give that window focus. If you do not, VisualAge RPG determines which window has focus when your application is loaded. By default, it will be the last window created that has the **Visible** attribute set.

### Window List

In the properties notebook for a window part, you can indicate if the window should appear in the window list. This list appears when you press the Ctrl+Alt+Delete in Windows. By default, window parts do not appear in the window list. You should set at least the main window to appear in the window list. You can use the task list to redisplay the window.
**Terminating a Program**

If the user selects the Close option from the system menu on a window, the operating system closes the window but does not necessarily terminate your program. To prevent this from happening, you can do one of the following:

- Select the **Terminate on close** check box in the second Style page in the window’s properties notebook. This will terminate your program when the user closes the window.
- In the first Style page of the window’s properties notebook, deselect the **System Menu** check box so that your windows are created without a System Menu. (By default, all windows are created with a system menu.)
- Use the **Close** event. This event is signaled when the user selects Close from the system menu. In the **Close** event action subroutine, you could set the LR indicator on, or prompt the user to confirm that this window should be closed, and set the **ENDACT** return point accordingly. For example, by setting the return value to "*NODEFAULT" the close request is ignored and the window is not closed.

```
* * Define message box variables
Dstyle M button(*yesbutton: *nobutton)
D Dmsg M style(*WARN)
Dmsg M msgtext('Are sure you want to exit?')
*
*********************************************************************
* * Window . . : FRA0000B
* * Part . . . : FRA0000B
* * Event . . : CLOSE
* * Description: Handle Close event from system menu to verify user
* * wants to close this window.
* *
*********************************************************************
* C FRA0000B BEGACT CLOSE FRA0000B
* * Prompt for close
C msg dply style rc 9 0
* * If Yes, terminate program, allow close to occur
C rc ifeq *YESBUTTON
C move *on *inlr
C movel '*DEFAULT 'return 12
* * Else, do not close this window
C else
C movel '*NODEFAULT 'return
C endif
* C ENDACT return
```
Clearing Fields on a Window

If you have several entry fields on a window, you can use the CLEAR operation code. This will clear all entry field values to their default values. Numeric fields are cleared with zeros and character fields are cleared with blanks.

Example of a Window Part

The window part shown below has a System menu, a Minimize button, and a Maximize button.

![Window with canvas](image)
The *component part allows programmers to access and use component- and system-wide attributes.

A *component part is the "part representation" of the component. One *component part is created for each component automatically; it is invisible and not on the palette.

**Part Attributes**

<table>
<thead>
<tr>
<th>Active*</th>
<th>Alarm</th>
<th>AppData</th>
<th>Button</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clipboard</td>
<td>CurrentDir</td>
<td>Dialog</td>
<td>DIRName*</td>
</tr>
<tr>
<td>DlgOwner</td>
<td>DlgPrompt*</td>
<td>DoEvents*</td>
<td>DspHeight</td>
</tr>
<tr>
<td>DspWidth</td>
<td>FileName</td>
<td>FocusPart*</td>
<td>FolderName*</td>
</tr>
<tr>
<td>HelpWindow</td>
<td>HostName*</td>
<td>LookNFeel*</td>
<td>MsgData</td>
</tr>
<tr>
<td>MsgFile*</td>
<td>MsgID</td>
<td>MsgText</td>
<td>Name</td>
</tr>
<tr>
<td>OS</td>
<td>Parent</td>
<td>PartCount</td>
<td>PartList</td>
</tr>
<tr>
<td>Platform</td>
<td>PlugCmd*</td>
<td>PlugDLL*</td>
<td>PlugID*</td>
</tr>
<tr>
<td>PlugRC*</td>
<td>PlugResult*</td>
<td>Printer*</td>
<td>PrtDevmode</td>
</tr>
<tr>
<td>SelFolder*</td>
<td>SelPrinter*</td>
<td>ShData</td>
<td>ShDataLen</td>
</tr>
<tr>
<td>ShDataName</td>
<td>ShDataPos</td>
<td>ShowMsgID</td>
<td>SwitchTo*</td>
</tr>
<tr>
<td>WrkStnName*</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Note: See the attribute description for restrictions.

**Applicable Events**

There are no events associated with this part.

**Using the *component part**

The *component part allows programmers to access and use component- and system-wide attributes. A *component part is the ‘part representation’ of the component. One *component part is created for each component automatically; it is not visible and is not on the parts palette.

**Displaying a File Open/Save As dialog.**

The **Button**, **FileName**, **Dialog**, and **DlgOwner** attributes of the *component part are used to display the Windows common File Open or Save As dialog. The dialog attribute determines which type of dialog to display. Set it to 1 to display an Open dialog or to 2 to display a Save As dialog. The **DlgOwner** attribute specify which part is the ‘owner’ of the dialog. When this attribute is set, the owner is ‘modal’ to the dialog. That is it can not respond to events until the dialog is dismissed.

Setting the **FileName** attribute displays the file open dialog. To determine which button the user used to dismiss the dialog, retrieve the value of the **Button** attribute.

In the following example, a File Open dialog is displayed. Notice that **FileName** attribute can be set to display only files with a certain extension:
The following example shows how to use a File Open dialog to select multiple files.

```c
C      eval %setatr('*Component':'*Component'
C      :'MulSel') = 1
C      eval %setatr('*Component':'*Component'
C      :'Dialog') = 1
C      eval %setatr('*Component':'*Component'
C      :'Filename') = '*.jpg'
C*Get the number of selected files.
C      z-add 0 num
C      '*component' getatr 'NumOfSel' num
C*Get the path of the selected files.
C      eval str=%getatr('*Component':'*Component':'
C      folderName')
C*Retrieve the name of the selected files.
C      1 do num idx 4 0
C      eval %setatr('*Component':'*Component'
C      :'FileIndex') = idx
C      eval str=%getatr('*Component':'*Component'
C      :'filename')
C      enddo
```

Figure 30. Displaying a File Open dialog

The following example shows how to use a File Open dialog to select multiple files.

Figure 31. Use a File Open dialog to select multiple files

Selecting a printer

If your application prints to a printer attached to the workstation you can use the SelPrinter and Printer attributes to allow the user to select to which printer the output is to be sent. Setting the SelPrinter attribute to 1 displays the Windows Print dialog to be displayed. When the user selects a printer from that dialog, the printed output from your application will be sent to that printer.

Initial settings shown in the dialog can be set by the PrtDevmode attribute.
Using Plugins

The PlugDLL, PlugID, PlugCmd, PlugRC, and PlugResult attributes give you the ability to extend the functionality of the GUI Designer. You provide the additional functionality in a program that you have developed. Once your application is registered to the GUI Builder by using the Vendor menu, your application can interact with the GUI Designer. See chapter 20 for more details on creating plugins.

Querying the Parts in a Component

The Parent, PartCount and PartList attributes can be used at runtime to query the part names in a component. For example, you could use these attributes to resize and reposition parts on a window if the window has been resized.
Part 3. Working with iSeries Data

Chapter 8, “iSeries Connectivity,” on page 193
Describes how to set up a connection between your application and an iSeries server.

Chapter 9, “Reusing iSeries Applications,” on page 213
Describes how to import existing display files, UIM help, and RPG source from existing iSeries 400 applications.
Chapter 8. iSeries Connectivity

If you are using an iSeries server while you are developing your application (for example, importing display files) or while you are running it (for example, accessing iSeries database files for I/O), you must define the iSeries information used by the application. This information is stored separately from the application, so that it can be updated without changing the application itself.

This section discusses the following topics:
• Defining iSeries information
• Setting up an iSeries server at design time and at run time
• Using data areas
• Using iSeries database files
• Database I/O considerations
• Controlling server connections at run time
• Using the security file for applets

Defining iSeries Information

During the development of your application, you can use the Define iSeries Information properties notebook to define aliases (override names) for the following iSeries information:
• Servers
• Files
• Programs
• Data areas
• Lock level

Once you have developed an application and are ready to install it on your user’s workstation, you need to ensure that either:
• For SNA communications the following is configured:
  A router must be defined using Client Access. This router name is also used as the Remote Location Name.
• For TCP/IP communications the following is configured:
  Use the host name defined for your iSeries server as the Remote Location Name.

Additionally, refer to the online help for the steps you must take to define iSeries information.

Notebook Considerations

If the Define iSeries Information properties notebook pages do not contain the override name for the program, the data area, or the database file, then the following occurs:
1. The name of the program, data area, or database file in the program is used.
2. If the program name, data area, or database file is library-qualified in the program, then this library is used.
3. If the program name or database file is not library-qualified in the program, the library list (*LIBL) on the iSeries server is searched.
4. The first server listed on the server page is used.
Setting Up a Server

You must set up a server when you are developing your application, so that you can access it while you edit, compile, and debug your application. When you package and distribute your application to other workstations, you also have to set up a server if the running application accesses a different server than the one used during design time.

Whenever you set up a server, ensure that the library list of the service job contains the remote resource that you want to work with.

Setting a Server at Design Time

If you need to use a server while you are developing your application, you must define server information using the Define Server Logon window and the Define iSeries Information notebook. See the online help for more information.

You must also define an iSeries job description to set up the library list. You can associate a library list with a job description on the iSeries server. This job description can then be associated with a user profile. Use the user ID from this user profile when you are prompted by the VisualAge RPG to logon to a server. The iSeries service job contains the correct library list.

Setting a Server at Run Time

If you need to access a server while you are running your application, you must verify that the iSeries information points to the correct server. Use the Define iSeries Information utility to invoke the Define iSeries Information notebook.

You must also set up the library list, either by changing the job description or by using the CL commands QCMDDDM or QCMDEXC.

Defining a job description to set up a library list

You can associate a library list with a job description on the iSeries server. This job description can then be associated with a user profile. Use the user ID from this user profile when you are prompted by the VisualAge RPG to logon to a server. The iSeries service job contains the correct library list.

Changing the library list

If a VisualAge RPG program calls CL commands:
- Specify a CALL to QCMDDDM if the CL command issues commands for iSeries files.
- Specify a CALL to QCMDEXC if the CL command issues commands to server programs or data areas.

CL commands can be issued to be run in the DDM service job using the CALL operation code. A special program must be called in order for the CL command to be run in the DDM service job. The special program is QCMDDDM. This interface is the same as the interface for calling QCMDEXC. The difference between QCMDEXC and QCMDDDM is that QCMDEXC runs in a separate job that is used to service remote call requests and data area requests.

Note: The first server listed in the server page of the Define iSeries Information notebook is known as the default server. At least one server is required for every program that makes use of an iSeries server.
QCMDDDM can be used to change the library list of the DDM service job to ensure that the library containing the database files is present in the DDM job’s library list.

**Using Data Areas**

Before your application can use data areas, you must set up the server.

If your application accesses a data area, the name of this data area can be either the name of the data area or an override name. You can define the override name in the GUI Designer using the Data area page of the Define iSeriesInformation notebook.

See “Notebook Considerations” on page 193 if the notebook page does not contain an override name for the data area.

Table 5 and Figure 32 illustrate how to access a data area using an override name.

Table 5. Enter this information on the Data area page of the Define iSeriesInformation notebook

<table>
<thead>
<tr>
<th>Data area override name:</th>
<th>DTAARA (this must be entered in uppercase)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remote data area name:</td>
<td>REMDTAARA</td>
</tr>
<tr>
<td>Server alias name:</td>
<td>SERVER01</td>
</tr>
</tbody>
</table>

Be sure the data area has been initialized before you attempt to use it. A runtime exception is issued if a data area on the server does not contain a valid packed decimal value when attempting to retrieve it into a data area data structure with a packed decimal subfield in a VisualAge RPG program.

```
*********************************************************************
* Program ID . : dtaaraex.vpg                                      *
* Description . : Code segment to get the contents of an AS/400    *
* data area.                                                     *
*********************************************************************
D dtaara S 6P 0 DTAARA
*********************************************************************
* Window . : WIN1                                                *
* Part . : PSB00000C                                             *
* Event . : PRESS                                               *
* Description: Get the contents of the AS/400 data area.         *
*********************************************************************
C PSB00000C BEGACT PRESS WIN1
C IN dtaara
C EDACT
```

Figure 32. Accessing a data area
Using iSeries 400 Database Files

Before your application can access iSeries 400 database files, you must set up the server.

Remote DISK file names used in your VisualAge RPG programs can be either the iSeries 400 file name or a file alias name. You can define a file alias name using the File page of the Define iSeriesInformation notebook. See "Notebook Considerations" on page 193 for information about what happens if the notebook page does not contain a file alias for the iSeries 400 file.

Database file overrides issued in the remote server DDM job are ignored by open requests issued by the VisualAge RPG application. Open requests made by server programs that run in the DDM service job may elect to either ignore or apply the file overrides.

The VisualAge RPG supports overriding the server's library name, file name, and member name using the File page in the Define iSeriesInformation notebook.

The Define iSeriesInformation notebook is used when the application is being built and while the application is running. At build time, the File page is used during file extracts to find the external descriptions of the files. It is also used for an externally described data structure when so specified in a definition specification.

At application run time, the File page is used to locate the actual remote iSeries 400 database files being used. The file alias name used in the VisualAge RPG program is used to find an appropriate entry in the File page.

If no entry exists in the File page, then the library list of the first server defined in the server page is used to find a file with the same name as the file in the VisualAge RPG program.

Keeping the actual file name separate from the file name used in the VisualAge RPG program allows you to retarget the actual file. You can direct it to a different file on the same iSeries server or a different iSeries server without changing the VisualAge RPG program.

Figure 33 on page 197 contains an example that illustrates:
- The association of file names with file entries in the File page of the Define iSeriesInformation notebook.
- Matching part names with fields.

Note: The NAME and ADDRESS information must be entered on the application's window. The information is entered on the database when the OK push button is pressed.
Program ID .: ioex.vpg

Description .: Create Database records using data from window.

Files . . . : FILE1

*****************************************************************************

FILE1  UF A E       DISK REMOTE USROPN
*****************************************************************************

*****************************************************************************

Window .: WIN1

Part . . . : *INZSR

Event . . : Initialization routine

Description: Open Database file (FILE1).

*****************************************************************************

C   *INZSR  BEGSR
C   OPEN     FILE1
C   ENDSR

*****************************************************************************

C   PSB0000D  BEGACT  PRESS  WIN1
C   SETON  LR
C   ENDACT

*****************************************************************************

Figure 33. Database file example (Part 1 of 2)
FILE1 in the file specification is used as a file alias, since an entry exists in the File page of the associated Define iSeriesInformation notebook.

The first member of FILE1 in library LIB1 on server TORAS180 is used during file open. FILE1 in the remote name does not have to match the override name in the file entry. The override name represents a link between the file entry in the File page and the file name used in the VisualAge RPG program.

The part names of the two entry fields are NAME and ADDRESS. The VisualAge RPG creates fields with the same names and the same attributes. In this example, NAME and ADDRESS are 20-character fields. The database file also contains two fields named NAME and ADDRESS, both 20 characters. The following is the DDS for these fields:

```plaintext
R RECORD100
A  NAME      20A
A  ADDRESS   20A
```

When field names and their attributes match, only one field is created. This example reads the data from the window.

```plaintext
C* NO1Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..
C  READ    'WIN1'
```

When this READ is performed, data is moved automatically from the screen into the two fields NAME and ADDRESS. Since the data is now in the appropriate fields, it can be written directly to the database without any further field movement.

```plaintext
C* NO1Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..
C  WRITE   FORMAT1
```

In this example, the data in the two fields NAME and ADDRESS is moved to the output buffer automatically before the write command is issued to the iSeries 400 database.
Level Checking

The VisualAge RPG supports level checking between a VisualAge RPG program and the iSeries 400 database files being used.

The compiler always provides the information required by level checking. Level checking occurs on a record-format basis when the file is opened, unless you specify LVLCHK(*NO) when creating or changing the database file.

Note: If a level check occurs, it is handled as an I/O error. For more information, see VisualAge RPG Language Reference.

Locking Database Files

The OS/400 system allows a lock state (exclusive, exclusive allow read, shared for update, shared no update, or shared for read) to be placed on a file used during the execution of a job. Programs within a job are not affected by file lock states. A file lock state applies only when a program in another job tries to use the file concurrently. The file lock state can be allocated with the CL command ALCOBJ (Allocate Object). For more information on allocating resources and lock states, see the CL and APIs section of the Programming category in the Information Center at this Web site - http://www.ibm.com/eserver/iseries/infocenter.

The OS/400 system places the following lock states on database files when it opens the files:

- Opened for INPUT: Lock state of Shared for read
- Opened for UPDATE: Lock state of Shared for update
- Opened for ADD: Lock state of Shared for update
- Opened for OUTPUT: Lock state of Shared for update

Overriding Database Files

To override the library name or file name of a database file, use the QCMDDDM command as shown in the following example:

```
D QCMDDDM C 
C 'QCMDDDM' Linkage(*Server) 
C OvrMenufl BEGSR 
C Eval QCMDDDM_Parm1 = 'OVRDBF FILE(MENUFL)' + 
C 'TOFILE(SYSLIBT/MENUFL)' + 
C 'MBR(' + MemberName + ')' + 
C 'OVRSCOPE(*JOB)' + 
C 'OPNSCOPE(*JOB)' 
C Exsr CallExecDDM 
C ENDSR 
C CallExecDDM BEGSR 
C EVAL QCMDDDM_Parm2 = %LEN(QCMDDDM_Parm1) 
C Call QCMDDDM 
C Parm QCMDDDM_Parm1 
C Parm QCMDDDM_Parm2 
C ENDSR 
```
iSeries 400 Database I/O Considerations

In general, all VisualAge RPG database I/O operations available in the ILE RPG/400 language are also available in the VisualAge RPG language, and are semantically equivalent. See VisualAge RPG Language Reference for more information, including which operation codes support local access, remote access, or both.

Using Record Blocking to Improve Performance

If your application reads data from an iSeries 400 server, you can improve the performance of your application by using record blocking. Record blocking means that file I/O operations are done on multiple sequential records (on blocks of records) instead of on one record at a time.

VisualAge RPG offers default record blocking if any of the following are true:
- The file is output-only and contains only one record format.
- The file is a combined file.
- The file is input-only, contains only one record format, and uses only OPEN, CLOSE, FEOD, and READ operation codes.
- The RECNO keyword is not specified on the file description specification.

In addition, you can perform explicit record blocking on files that meet the following criteria:
- The File Addition entry (position 20) is blank.
- The RECNO keyword is not used on the file.
- The file has only one record format.
- The CHAIN, SETLL or SETGT operation codes are used on the file.
- The READE, READP, or READT operation codes are not used on the file.

If a file meets the above criteria, you can enable record blocking by updating your program with BLOCK(*YES) on the file description and recompiling the program. Figure 34 shows an example that uses the BLOCK keyword option.

```
FILE1 IF E
F DISK BLOCK(*YES) REMOTE
:
C FLD2 SETLL REC1
:
C READ REC1 10
```

Figure 34. Example using BLOCK keyword option

If you use BLOCK(*NO) on a file description and recompile the program, no record blocking will take place, not even the default record blocking that VisualAge RPG supports.

iSeries 400 Servers Used

If you use TCP/IP, the Optimized Central server and the Remote Command server must be activated. Use the STRHOSTSVR command to start these servers. Go to the iSeries Information Center at URL http://www.ibm.com/eserver/iseries/infocenter for more information on this command.
These servers are required when you develop your applications. In addition, when you run your applications, you should have the TCP/IP DDM server active as well. Use the STRTCPNSV command to start this server.

**Controlling Server Connections at Run Time**

VARPG has a default way to establish connections to remote servers, for most applications this default way of connecting an application to a remote server is sufficient. In some instances requirements surfaced to allow a more dynamic way of connecting to remote servers. These APIs are made available for VARPG programmers to use, in case the application they write has a need to use a non default connection setup.

The default VARPG runtime controlled connection startup works as follows:

1. The VARPG runtime gets the server name from the project’s Remote Server Table (.RST) file,
2. The VARPG runtime gets the userid/password information from the VARPG security file if available. If this information is not available the user gets prompted for userid/password.
3. The VARPG runtime then establishes the connection.

If an application has requirements that can’t be handled thru this default connection startup the programmer is encouraged to use these VARPG connection APIs.

VARPG provides two types of connection APIs:

1. The Set server APIs to set the server name (Remote Location) to connect to
2. The Connect APIs to control the initiation of the connection with the pre-set server, including authentication.

Here are two scenarios why a programmer would want to implement a non default connection start up:

1. The application needs to dynamically switch servers without re-starting it
2. Signon information needs to be handled without using the VARPG security file, but signon information needs to be available for multiple VARPG applications to avoid multiple prompts for authentication.

Programmers can use the Set server and Connect APIs individually or in conjunction with each other. When used together, one of the Set server APIs has to be used first to set the server name, then one of the Connect APIs can be used to establish the connection to this server.

Both API types are described in this section, first the Set server APIs and then the Connect APIs.

In the VARPG environment, connections normally remain active during the lifetime of an application. Connections are shared by components running inside the application’s process. This behavior is true for user controlled VARPG connections, too.

**Setting the Remote Location name**

As described before, by default the remote location name of the server is specified in the .RST file and the connection gets set to the remote location name at start up of the application. If the programmer wants to supply the remote location name dynamically instead of using the .RST file he can use one of the two Set server
APIs supplied in VARPG. These APIs will set the connection environment, they will not establish a connection. The connection can then be established by accessing the server (opening a file or invoking a remote program) or by using the Connect APIs described in the second half of this chapter.

VARPG provides these two Set server APIs:
• The Set Server API allows the programmer to set a server to a new Remote Location name regardless of which remote server a current session is already connected to. In the case an active connection is already established this connection will be terminated.
• The Change Server API allows a programmer to specify a new Remote Location name for a specific remote server connection already in use. It will stop the communications session already in place and will set the connection to the new Remote Location name so a subsequent request to connect to the server alias will result in establishing a session with this new server.

The only difference to the Set server API is the fact that the programmer can qualify the server name to be terminated and changed. The parameters for new remote location name and keep job are the same as in the Set Server function.

These two APIs allow the VARPG programmer to easily write VARPG programs that can dynamically change the target server, and allow the application to connect to different servers without ending the VARPG application.

For the user controlled connection startup, the programmer uses the APIs to pass the existing server name, new server name, and information how to deal with jobs on an existing connection. The functions whose interface is described by these APIs are located in the FVDCWVC9.DLL. This DLL is part of the VARPG run time and is located in the path; there is no need to rearrange the path environment for applications using this API.

The Set remote location function, VARPG_Set_Remote_Location, accepts the following parameters and provides a numeric return code indicating the success or failure of the change server process:
• New Remote Location name
• Keep job

The parameters are null terminated character variables passed by reference. The following example shows the C signature and the RPG IV prototype for the API:

```c
* Set server prototype
* extern "C" VARPG_ENTRY int VARPG_Set_Remote_Location(char *
* newRmtLocation, char * keepServerJob);
```

```rpg
D setsrv pr 5i 0 dll('FVDCWVC9') extproc
D ('VARPG_Set_Remote_Location')
D newrmtl * value options(*string)
D keepjob * value options(*string)
```

If you need to set the remote location name back to the default, specify "" (an empty string) in the new remote location name parameter.

When terminating an existing connection, if the programmer wants to end the active VARPG jobs running on the iSeries server for this session, he needs to specify the literal ‘OFF’ for the keepServerJob parameter. Only if the jobs are supposed to continue running, after the VARPG communications session terminated, specify ‘ON’. The programmer has to manage non terminated VARPG
jobs himself, and has to make sure they get terminated eventually. The VARPG runtime doesn’t have access to these jobs after a server has been switched.

The Change remote location function, VARPG_Chg_Remote_Location, accepts the following parameters and provides a numeric return code indicating success or failure of the change server process:
* Old Remote location name
* New Remote Location name
* Keep job

The parameters are null terminated character variables passed by reference.

The following example shows the C signature and the RPG IV prototype for the API:
```c
* Change server prototype
* extern "C" VARPG_ENTRY int VARPG_Chg_Remote_Location( char *
*          oldRmtLocation, char * newRmtLocation,
*          char * keepServerJob );
*
D changesrv pr 5 i 0 dll('FVDCWVC9') extproc
D oldrmt1 * value options(*string)
D newrmt1 * value options(*string)
D keepjob * value options(*string)
```

The return codes for these APIs are listed in Figure 35 on page 205

### Connecting to a remote location

A user-controlled connection differs from a default VARPG connection in the way the signon information gets provided. For a user controlled connection the programmer provides the authentication information through the connection API. For a default connection the authentication information is provided in the VARPG security file and if not specified there the default signon dialog is displayed to gather this information.

VARPG provides two APIs to control connection startup at run time. These APIs can be used directly in VARPG programs.
1. The Signon API allows programmers to connect to an iSeries server by providing their own signon information.
2. The Change Password API allows programmers to handle changes to the signon password.

For the user controlled connection startup, the programmer uses the Signon API to identify the server name, user Id, and password so the VARPG run time can establish the connection. The functions whose interface is described by these APIs are located in the FVDCWVC9.DLL. This DLL is part of the VARPG runtime and is located in the path; there is no need to rearrange the path environment for applications using this API.

It is important to note the programmer can’t switch servers by just using the Connect APIs and specifying a new server name, in order to switch a server, one of the Set server APIs has to be used first to re-set the VARPG communications environment, then the Set Signon function which is part of the Connect APIs can be used to establish the connection to the new server.
The Set Signon function, VARPG_Set_Signon_Info, accepts the following parameters and provides a numeric return code indicating its success or failure:

- Server name
- User Id
- Password

The parameters are null terminated character variables passed by reference.

The following example shows the C signature and the RPG IV prototype for the API:

```c
* Signon prototype
* extern "C" int VARPG_Set_Signon_Info(char * server, char * userid,
 * char * password);
```

```rpg
D signon pr 5I 0 dll('FVDCWVC9')
D signon pr extproc('VARPG_Set_Signon_Info')
D system   *   VALUE options(*string)
D userid    *   VALUE options(*string)
D password  *   VALUE options(*string)
```

The Change Password function, VARPG_Change_Password, has one additional parameter - the new password to be used. The function also returns a numeric value indicating success or failure of the API execution. Its parameters are:

- Server name
- User Id
- Old password
- New password

These parameters are null terminated character variables passed by reference.

The following example shows the C API and the corresponding RPG prototype:

```c
* New password prototype
* VARPG_ENTRY int cdecl VARPG_Change_Password(char * server,
 * char * userid, char * password, char * newpassword);
```

```rpg
D newpassword pr 5I 0 dll('FVDCWVC9')
D newpassword pr extproc('VARPG_Change_Password')
D system       *   value options(*string)
D userid       *   VALUE options(*string)
D oldpassword  *   VALUE options(*string)
D newpassword  *   VALUE options(*string)
```

There are many different ways to gather the server/user Id information. The sample program provided in VARPG uses its own signon dialog written in VARPG. Remember that the connection is established for the VARPG application and its components. If you start another VARPG application, by default the VARPG run time will use its usual connection startup mechanism again the same way it does for the first VARPG application. The programmer has two options to change this behavior.

1. Use the control specification option INHERITSIGNON in the application to re-use the previously specified authentication information
2. Use these Connect APIs to control how to deal with server authentication

When using the Signon API in a VARPG application with remote server files, make sure to specify the USROPN keyword in the file specifications for the remote files. If USROPN is not specified, the server connection will be established at application
startup before you have a chance to invoke the SIGNON function. In components started after the communication session has been established, you can use the RPG implicit opening of files by not specifying the USROPN keyword in these components. The components will reuse the existing connection of the application.

The return codes for these Connection and Signon APIs are:

```
OK            0
INVALID_PARAMETER  1
INTERNAL_ERROR      2
FUNCTION_NOT_SUPPORTED  3
COMMUNICATIONS_ERROR  4
SERVER_INVALID      101
USER_ID_UNKNOWN    201
USER_ID_REVOKED    202
NEW_PWD_LENGTH_LONGER_THAN_MAX  301
NEW_PWD_LENGTH_SHORTER_THAN_MIN    302
NEW_PWD_CONTAINS_CHAR_USED_THAN_ONCE  303
NEW_PWD_CONTAINS_ADJACENT_DIGIT    304
NEW_PWD_CONTAINS_REPEATED_CONSECUTIVELY  305
NEW_PWD_PREVIOUSLY_USED  306
NEW_PWD_MUST_CONTAIN_ONE_NUMERIC  307
NEW_PWD_CONTAINS_INVALID_CHAR    308
NEW_PWD_CONTAINS_DISALLOWED_WORD    309
NEW_PWD_CONTAINS_USERID      310
PASSWORD_INCORRECT       311
PASSWORD_DISABLED_NEXT_INVALID_ATTEMPT  312
PASSWORD_EXPIRED        313
NEW_PWD_CONTAINS_CHAR_SAME_POSITION_AS_LAST  315
```

*Figure 35. Return codes for the Connection and Signon APIs*

**Sample Program Using the Signon API**

The sample application presents a window with a "Signon to server" push button. Pressing this button starts the Signon component, which gathers the userID/password information. The component will signal back whether or not the connection is successful. This application uses the component reference part to monitor the Signon component for completion.
Files have been specified with the USROPN keyword, so no connection request is made by the VARPG run time. Pressing "Sign on to server" will start the component.

Figure 36. Initial Window with Signon Push Button
The user can now specify the server and userID/password information. The component will use this data to start the signon API and establish the connection to the server. When the initial window gets notified that a connection has been established successfully, the program accesses the customer database on the iSeries server.

![Image of customer detail](image)

A variation that is implemented in this sample is to show a Change password dialog if the user’s password has expired. This condition can be detected by checking the return code of the SIGNON function.
The program uses this dialog to get the new password and then send it to the VARPG communications layer by invoking the VARPG_Change_Password function. The code for all of these functions is included in the Runtime_control_of_server_connections sample program.

Handling Server Sign-On Errors

The first time an application requires server access (to open a file or perform a remote program call) it will seek to establish a server connection. A sign-on dialog may then be displayed to prompt for a userid and password, if these settings were not pre-defined. If this dialog is cancelled by the user, then a runtime error is issued. (1421 - Logon cancelled by user) If the error is passed to the RPG default error handler, a runtime error dialog is shown.

To avoid showing this runtime error, the application can instead handle the error by:
- Handling file open errors
- Explicitly connecting to the server
- Setting up a general program error handler

Explicitly Handling File Open Errors

To explicitly handle file open errors, the file must first be opened directly by the application code, so it must be specified with File keyword USROPN. Then, open it with OPEN(E) or with a result error indicator, and check the results.

Note:

Note: All files in the application must be changed to USROPN, to prevent the otherwise implicit file opens that are done at application startup trying to connect first.
Explicitly connecting to the server
Another approach is to control the establishing of the server connection directly. Refer to the sample project runtime_control_of_server_connection.

This must be done before the application requires the connection for implicit file opens, file access, or program calls. (Make all files USROPN, or move the files to a sub-component started only after the connection is established)

Setting up a general program error handler
Another approach is to use a *PSSR, which has the benefit that files can still be implicitly opened. (Instead of using USROPN.)

However, this receives control for other errors as well, so care should be taken to handle the errors specifically; allowing for unexpected errors.

*D*A program exception/error subroutine can be specified by coding *PSSR in factor 1 of D*a BEGSR operation.
C  *PSSR  BEGSR
C  ENDSR  '*CANCL'

If you would like to determine the cause of the exception/error in *PSSR subroutine, you can use a program status data structure:

D*Information regarding the program exception/error is made D*available through a program status data structure that is specified with an S in D*position 23 of the data structure statement on the definition specifications.
DMYPSDS  SDS
D  PROC_NAME  *PROC
D  PGM_STATUS  *STATUS
D  PRV_STATUS  16  205  0
D  LINE_NUM  21  28
D  ROUTINE  *ROUTINE
D  PARMS  *PARMS
D  EXCP_TYPE  40  42
D  EXCP_NUM  43  46
D*
D  EXCP_DATA  91  170
D* The following is a general error handler. If any error occur, it D*will get called. You can then determine the cause by checking PGM_STATUS D*in the above PSDS.
Using the Security File for Applets

When a VARPG application needs an iSeries resource, such as a database file, a valid user ID and password is required to connect to the iSeries server. For VARPG, the user ID and password are stored in a security file on the client workstation. However, applets by default, cannot access any files on the client workstation. This results in the user being prompted for a user ID and password each time the applet is run. If you want to use the security file when running applets and avoid this prompting, you need to give each applet permission to read the security file. You do this by using the Policy Tool utility that is part of Sun Microsystems’s J2SDK. You can find more information on how to use Policy Tool in the Tool Documentation section of the J2SDK 1.2 documentation.

Note: The procedure described here only works for Windows clients.

1. Create the security file on the client workstation.
   - If not already done, install the VARPG runtime on the client workstation. From the Start menu, choose Programs>VisualAge RPG Runtime>Define Server Logon. The Define Server Logon dialog appears. Add the user ID and password for a specific iSeries server to store them in the security file.

2. Create the required permissions.
   - Permissions are required so that the Java VM can read the security file. To create the permissions, do the following:
     a. From an MS DOS prompt, type Policy Tool and press enter. The Policy Tool dialog box appears.
        - If this is the first time you have used Policy Tool, a message indicating that the policy file cannot be found in a certain directory is also displayed. Make a note of the name and directory of the policy file in the message. This will be the location you will save your policy file to in a later step.
     b. From the Policy Tool dialog, click the Add Policy Entry button. The Policy Entry dialog appears. In the CodeBase entry field, type:
        "http://xxxxx/"

        where xxxxx is the URL and directory that contains the applet that is to be given permission to the security file.
     c. Click the Add Permission button. Complete the Permissions dialog as follows:
        • From the Permissions combination box, choose RuntimePermission.
        • From the Target combination box, choose loadLibrary.<library name>.
        • In the entry field immediately to the right of the Target combination box, change loadLibrary.<library name> to loadLibrary.fvdcjava.
        • Press OK to return to the Policy Entry dialog.
     d. From the Policy Entry dialog, again click the Add Permission button. Complete the Permissions dialog as follows:
        • From the Permissions combination box, choose FilePermission.
        • In the entry field immediately to the right of the Target combination box, type the name of the security file. This file is located in the ibmcom subdirectory of your Windows directory. For example:
          c:\windows\ibmcom\fvdcsec.txt

          where your Windows directory is c:\windows.
        • From the Actions drop down combination box, choose read.
• Press **OK** to return to the Policy Entry dialog.
• Press **Done** to return to the Policy Tool dialog.

e. From the Policy Tool dialog, choose **File** then **Save as** from the menu. Save the policy file you have just created to the file name and directory you noted in [2a on page 210](#). Now choose **File**, then **Exit** to exit the Policy Tool.
If you are still prompted for the user ID and password, use Policy Tool again to verify that you have specified all the parameters correctly.
Chapter 9. Reusing iSeries Applications

When you develop your VisualAge RPG application, you may want to reuse an existing iSeries 400 application or various pieces of it. This section describes some of the things you must consider when reusing iSeries 400 applications.

Reuse Scenario

You can use VisualAge RPG to modify applications that run on the server so that they run on a PWS, access data on the host, and have a graphical user interface. This section provides an overview of the steps involved.

Importing display files: The Import utility converts existing display files to a graphical user interface on a PWS. After you import a display file, the record formats are converted to user-defined parts and stored on the Imported page of the parts catalog. You can move the parts to the parts palette as you work on the application, and then store them in the parts catalog when you finish working on the application until you need them again.

For example, importing the 5250 screen shown in Figure 37 results in the GUI shown in Figure 38 on page 214. Records are converted to a group of parts, fields are converted to entry field parts, and constants are converted to static text parts. All command keys are converted to push button parts, and the push button labels reflect the original command key keyword.

Figure 37. Sample 5250 screen from a Purchase Order application

© Copyright IBM Corp. 1994, 2005

213
Note: The new parts inherit the original field names, but you can rename them if you like. Retaining the same field name improves productivity when manipulating program logic for reuse.

You may want to customize the imported parts to take advantage of the basic design issues discussed in Chapter 2, “Planning Your Application,” on page 19.

Customizing the GUI:
Figure 39 shows how the imported window would look if you made the following changes to the interface:

- Replace the command keys with a menu bar and associated menu items. For example, a **Vendor** menu contains actions that were originally converted to push buttons. This gives users easy access to frequently performed actions, and launches related windows.
- Group related information, using group box parts to provide a visual cue for users. For example, a group box labeled **Shipment Information** contains entry fields that pertain to shipment information.
- Use grouped radio buttons for information that requires a user to select from a number of known choices. For example, only three methods of payment are possible (prepaid, collect, charge); therefore users need to select only one radio button to indicate the method being used. The radio buttons are in a group box labeled **Payment**.

These changes take advantage of the graphical capabilities that VisualAge RPG offers.

**Reusing online help:** When you reuse an application, you may want to reuse the existing User Interface Manager (UIM) help, too. You will have to modify it somewhat to reflect the look of the new GUI; however, it could save you the effort required to create help from scratch.

You can customize the converted UIM help and add new types of help using the Information Presentation Facility (IPF). You can write help for each window and context-sensitive help for each part, and you can link the help information by
creating hypertext links in the help source. See “Reusing UIM Help” on page 222, and Chapter 13, “Tips for Creating Online Help with IPF,” on page 245, for more information.

**Writing program logic:** You can reuse logic written in RPG IV because the compiler is based on that language. Simply cut-and-paste existing code for reuse.

You also have to write some additional program logic, using event-driven programming. For every event associated with a part, there is an action subroutine which describes how the program responds to an event. Procedural operation codes for program control are not required; program control is implicit. Some of the VisualAge RPG operation codes unique to VisualAge RPG applications are:

- **BEGACT**
  - Begins an action subroutine

- **ENDACT**
  - Ends an action subroutine

- **SETATR**
  - Sets the value of a part attribute

- **GETATR**
  - Retrieves the value of a part attribute

- **SHOWWIN**
  - Displays a window

- **CLSWIN**
  - Closes a window

[Figure 40 on page 217](#) contains an action subroutine from a sample purchase order application. When SHOWWIN is called from a particular window to display the PUR570R2 window, an action subroutine is coded for the **Create** event to prepare the window for the user’s next action.

If this is a new purchase order (PONUM = 0), the menu items **change**, **delete**, **print**, and **fax** are set to not respond to the MENUSELECT event. For each of the menu items, the %setatr function is used to set the **enabled** attribute to 0. The BEGACT operation code indicates the beginning of the action subroutine, and ENDACT indicates the end of it.
Connecting to the Host: If your application uses iSeries 400 database fields or imports iSeries 400 display files when you are building your application, you must define the iSeries 400 server it uses. You can communicate with the host at any time, as long as the server logon information is appropriately defined. See the online help for more information about defining server information.

Importing Display Files

If you have a display file that contains record formats you wish to include on a design window, you can import these record formats from an iSeries server to the VisualAge RPG environment. After the import is completed, the record is converted to a user-defined part and stored on the Imported page of the parts catalog. Before you can import a display file:

- define the server that you will be accessing. See the online help.
- You must have a minimum of *USE authority for these display files.

To import a display file:

1. Choose the Server Import display file from the GUI Designer.
2. Select a display file using one of these methods:
   - Type the full name of the display file (<server>library/file) in the File name entry field, and press Enter while the cursor is still in the field.
   - Do the following:
     a. Select one of the servers that is displayed. A list of libraries is displayed under the server name.
     b. Select one of these library names. A list of display file objects is displayed.
     c. Select one of these display file objects. A list of records in the file is displayed.
3. Select the record that you want to import.
4. If you do not want to use the default part name, type a part name in the Part name entry field.
5. If you do not want to use the default icon, you can press the Find push button to use the Find an Icon window.
6. Select one of the following to indicate where you want the part to reside:
Catalog only
The part is added to the Imported page in the catalog.

Catalog and palette
The part is added to the Imported page in the catalog, and the icon for the imported part appears on the palette.

7. Choose Import.

Converting Display Files
When you import a display file, the record formats, fields, and keywords that have equivalent parts or attributes are converted, and you can use these parts as you would any other part. The record formats, fields, and keywords that do not have an equivalent part are not converted.

In general,
• Record formats are converted to a VisualAge RPG window or a group of parts, depending on the record type.
• Fields are converted to entry fields.
• Constants are converted to static text parts.
• Conditioning indicator options are ignored.

Note: Fields with a length greater than 64 bytes are sized to 64 bytes on the GUI; however, the length attribute in the properties notebook is set to the original length.

Record Formats
The following list describes how display record formats are converted to parts.

MNUBAR
The MNUBAR record format is converted to a menu bar part that you can drop onto a window with canvas.

PULLDOWN
The PULLDOWN record format is used with the MNUBAR record format to create a submenu part. The PULLDOWN record format is converted to menu item parts for the MNUBAR record format that it references.

RECORD
The RECORD record format is converted to a group of parts that you can drop onto a window with canvas.

SFL, SFLCTL
These record formats are converted to a subfile part that you can drop onto a window with canvas.

Constants in the SFL records are not converted.

WINDOW
The WINDOW definition record format is converted to a window with canvas part. The WINDOW reference record format is converted to a group of parts that you can drop onto a window with canvas.

These record formats are not converted: ERRSFL, SFLMSG, and USRDYN. The PULLDOWN and WINDOW keywords within a subfile are not converted.

Positional Entries
The following table describes how positional entries in the DDS used to create a display file determine how formats and fields are converted.
### Table 6. Positional entries and conversion

<table>
<thead>
<tr>
<th>Columns</th>
<th>Meaning</th>
<th>Entry and Conversion Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-16</td>
<td>Indicators</td>
<td>Not converted</td>
</tr>
<tr>
<td>17</td>
<td>Record type</td>
<td>R Converted as described in Record Formats</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H Not converted</td>
</tr>
<tr>
<td>19-28</td>
<td>Name</td>
<td>If a named field, used as the name of the part</td>
</tr>
<tr>
<td>29</td>
<td>Reference</td>
<td>R Not converted</td>
</tr>
<tr>
<td>30-34</td>
<td>Length</td>
<td>Sets the data length</td>
</tr>
<tr>
<td>35</td>
<td>Keyboard shift</td>
<td>A Converted to character</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E Data type set to DBCS Either</td>
</tr>
<tr>
<td></td>
<td></td>
<td>G Data type set to DBCS Only</td>
</tr>
<tr>
<td></td>
<td></td>
<td>I Read-only, Disabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td>J Data type set to DBCS Only</td>
</tr>
<tr>
<td></td>
<td></td>
<td>O Data type set to DBCS Mixed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D F M N S W X Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Not converted</td>
</tr>
<tr>
<td>36-37</td>
<td>Decimals</td>
<td>Determines the number of decimal positions in the converted part. If specified, sets data type to Numeric.</td>
</tr>
<tr>
<td>38</td>
<td>Usage</td>
<td>I B Enabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td>M P Not converted</td>
</tr>
<tr>
<td></td>
<td></td>
<td>O Read-only, Enabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H Converted if the field is a subfile field</td>
</tr>
<tr>
<td>39-41</td>
<td>Location</td>
<td>Determines the position of the part on the window</td>
</tr>
<tr>
<td>45-80</td>
<td>Keywords</td>
<td>Constant Creates a static text part</td>
</tr>
</tbody>
</table>

### Display File Keywords

The following list describes how display file keywords are converted to parts. Options used with these keywords can determine whether or not the keywords are converted.

**CFnn, CAnn**

These keywords are converted to push buttons that have a label identical to the name of the keyword.

**CHOICE**

See MLTCHCFLD and SNGCHCFLD.
CNTFLD
The CNTFLD keyword is converted to a multiline edit part.

COLOR
The COLOR keyword determines the foreground color attribute of the part. If there is more than one COLOR keyword, the last one is used.

COMP
The COMP keyword is used to set the comparison attributes in the part’s properties notebook.

DATE
The DATE keyword is converted to a static text part.

DFT
The text associated with DFT becomes the label for the part.
- If the field that is being converted is a constant field, the DFT value is used on the label.
- If the field that is being converted is specified on a named field, the DFT keyword value is converted to the default text of the part.

DFTVAL
The DFTVAL keyword value is converted to the default value of the part.

DSPATR
If the DSPATR display attribute is:
- HI, the foreground color is made brighter.
- ND, the converted field is set to not visible.
- PR, the converted field is set to disabled.

Any other display attributes are not converted.

HELP
The HELP keyword is converted to a push button with the label HELP.
The help function is not converted.

MLTCHCFLD
If the MLTCHCFLD keyword is used inside a PULLDOWN record, each CHOICE keyword associated with it is converted to a menu item part on a submenu part. The converted menu item part has a check mark next to it to indicate that it is active.

If the MLTCHCFLD keyword is used outside a PULLDOWN record, each CHOICE keyword associated with it is converted to a check box part.
Check boxes are positioned horizontally with the same default space between them. They are not grouped.

MNUBAR
The MNUBAR record format is converted to a menu part.

MNUBARCHC
Each MNUBARCHC is converted to a menu item.

PRINT
The PRINT keyword is converted to a push button with the label PRINT.
The print function is not converted.

PSHBTNCHC, PSHBTNFLD
The PSHBTNFLD is converted to a push button part. The text associated with the PSHBTNCHC keyword is converted to the label on a push button part.

PULLDOWN
The PULLDOWN record format is used with the MNUBAR record format to create a submenu part. The PULLDOWN record format is converted to the menu item parts for the MNUBAR record format that it references.
RANGE
The RANGE keyword is converted to the range attribute for the part.

SFL, SFLCTL
These record formats are converted to a subfile part.

SFLPAG
Influences the initial height of the subfile part.

SNCHCFLD
If the SNCHCFLD keyword is used inside a PULLDOWN record, each
CHOICE keyword associated with it is converted to a menu item part on a
submenu part.
If the SNCHCFLD keyword is used outside a PULLDOWN record, each
CHOICE keyword associated with it is converted to a radio button part.
The radio buttons are arranged horizontally with the same default space
between them. They are not grouped.

SYSNAME
The SYSNAME keyword is converted to a static text part.

TIME
The TIME keyword is converted to a static text part.

USER
The USER keyword is converted to a static text part.

VALUES
The VALUES keyword causes the field to be converted to a drop-down
combination box part. The values associated with the VALUES keyword
are used on the drop-down list.

WDWTITLE
The WDWTITLE keyword is used to determine the label and attributes for
a window with canvas part.
• If the title text is assigned to a program-to-system field, it is not
converted.
• If the title text is assigned to a literal field, the label for the window with
canvas part is set to this text.

WINDOW
The WINDOW definition record format is converted to a window with
canvas part. The WINDOW reference record format is converted to a group
of parts that you can drop onto a window with canvas.

No other keywords are converted.

Converting Color
Character-based computer screen entry fields are converted to entry field parts that
are color-coded.

Note: Not all displays support these colors. On VGA displays, for example, the
converted entry fields will be white.

The color of each converted entry field depends on the type and attributes defined
in the display file:

Table 7. Original and Converted Field Attributes

<table>
<thead>
<tr>
<th>Field Type</th>
<th>Field Attributes</th>
<th>GUI Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O*</td>
<td></td>
<td>ReadOnly: Off</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Enabled: On</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Color: Light Yellow</td>
</tr>
</tbody>
</table>
Table 7. Original and Converted Field Attributes (continued)

<table>
<thead>
<tr>
<th>Field Type</th>
<th>Original Attributes</th>
<th>Converted Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>ReadOnly: On</td>
<td>Enabled: On</td>
</tr>
<tr>
<td></td>
<td>Color: Light Green</td>
<td></td>
</tr>
<tr>
<td>Input</td>
<td>ReadOnly: Off</td>
<td>Enabled: On</td>
</tr>
<tr>
<td></td>
<td>Color: Light Blue</td>
<td></td>
</tr>
<tr>
<td>Input or I/O</td>
<td>Protected</td>
<td>ReadOnly: Off</td>
</tr>
<tr>
<td></td>
<td>Enabled: Off</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Color: Light Red</td>
<td></td>
</tr>
<tr>
<td>Input or I/O</td>
<td>Inhibited keyboard</td>
<td>ReadOnly: On</td>
</tr>
<tr>
<td></td>
<td>Enabled: On</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Color: Medium Red</td>
<td></td>
</tr>
<tr>
<td>Input or I/O</td>
<td>Inhibited keyboard Protected</td>
<td>ReadOnly: On</td>
</tr>
<tr>
<td></td>
<td>Enabled: Off</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Color: Deep Pink</td>
<td></td>
</tr>
</tbody>
</table>

**Note:** I/O = Input and output

The color-coding allows you to visually determine the attributes that are set for the entry field part. For example, if a light-green entry field is displayed, you know that it is used by the program to display data and cannot receive user input. If a light-red entry field is displayed, you know that it can receive user input, but that it is not enabled because it was a protected field in the original application.

**Reusing UIM Help**

You can reuse the iSeries 400 help that was written using User Interface Manager (UIM) even though VisualAge RPG help files are written using the Information Presentation Facility (IPF). Both UIM and IPF formats use General Markup Language (GML) principles and are highly inter-changeable. For detailed information about using IPF, see Information Presentation Facility Guide and Reference(available online). You should also see the online document entitled IPF Restrictions. This document provides details on the subset of IPF tags that you are restricted to in a Windows environment.

To reuse UIM help files:
1. Use the editor to copy and paste the members containing the UIM help.
2. Change the UIM tags to the appropriate IPF tags.

The following sections compare some of the UIM and IPF tags.

**UIM and IPF functions that use the same tags**

There are functions in UIM and IPF that are equivalent and are tagged exactly the same way. In these cases you can use your UIM tags verbatim.

Table 8. Identical UIM and IPF Tags

<table>
<thead>
<tr>
<th>UIM Tag</th>
<th>Tag Function</th>
<th>IPF Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>:DL.</td>
<td>Definition List</td>
<td>:dl.</td>
</tr>
<tr>
<td>:FIG.</td>
<td>Figure</td>
<td>:fig.</td>
</tr>
</tbody>
</table>
Equivalent UIM and IPF functions that use different tags

There are functions in UIM and IPF that are equivalent but are tagged differently. In this situation, change the UIM tagging to its equivalent IPF tagging.

Table 9. Equivalent UIM and IPF Tags

<table>
<thead>
<tr>
<th>UIM Tag</th>
<th>Function</th>
<th>IPF Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>:CIT.</td>
<td>Citation</td>
<td>:hp5.</td>
</tr>
<tr>
<td>:HELP.</td>
<td>Heading</td>
<td>:help.</td>
</tr>
<tr>
<td>:ISCH.</td>
<td>Index item</td>
<td>:i1.</td>
</tr>
<tr>
<td>:ISCHSYN.</td>
<td>Index synonym</td>
<td>:isyn.</td>
</tr>
<tr>
<td>:PK.</td>
<td>Programming keyword</td>
<td>:hp2.</td>
</tr>
</tbody>
</table>
Table 9. Equivalent UIM and IPF Tags (continued)

<table>
<thead>
<tr>
<th>UIM Tag</th>
<th>Function</th>
<th>IPF Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>:PV</td>
<td>Programming variable</td>
<td>:hp5</td>
</tr>
</tbody>
</table>

**UIM Functions with no IPF equivalents**

There are functions available in UIM that are not available in IPF. In this situation, either delete the function or find another way to implement the function using IPF tagging.

Table 10. UIM Tags with No IPF Equivalents

<table>
<thead>
<tr>
<th>UIM Tag</th>
<th>Function</th>
<th>Suggested IPF Substitutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>:HP0</td>
<td>No highlighting</td>
<td>Use no :hp tag around the text.</td>
</tr>
<tr>
<td>:PC</td>
<td>Paragraph continuation</td>
<td>Use no tag. Continue with the text of the paragraph.</td>
</tr>
<tr>
<td>:RT</td>
<td>Reverse text</td>
<td>Use a different type of highlighting using a :hp tag.</td>
</tr>
<tr>
<td>:XH1</td>
<td>Extended help heading</td>
<td>There is no extended help in IPF. Use a :link tag to create a hypertext link to another help window (:h1) where you provide extended help.</td>
</tr>
<tr>
<td>:XH2</td>
<td>Extended help heading</td>
<td>There is no extended help in IPF. Use a :link tag to create a hypertext link to another help window (:h1) where you provide extended help.</td>
</tr>
<tr>
<td>:XH3</td>
<td>Extended help heading</td>
<td>There is no extended help in IPF. Use a :link tag to create a hypertext link to another help window (:h1) where you provide extended help.</td>
</tr>
<tr>
<td>:XH4</td>
<td>Extended help heading</td>
<td>There is no extended help in IPF. Use a :link tag to create a hypertext link to another help window (:h1) where you provide extended help.</td>
</tr>
</tbody>
</table>

**Reusing RPG Source**

To reuse RPG source code on a server:
1. If the source code is not RPG IV syntax, convert it to RPG IV syntax using the ILE RPG conversion tool (CVTRPGSRC) on the server.
2. Use the editor to copy and paste the members containing the RPG source and commonly used subroutines.
3. The syntax checker highlights any operation codes that are not supported by the compiler. See VisualAge RPG Language Reference for a description of the supported operation codes.

**Note:** In addition to differences between operation codes, there are a number of other differences between the RPG IV language and the VisualAge RPG compiler you must be aware of prior to reusing RPG source. For a description of the differences between the RPG IV language and the VisualAge RPG language, see VisualAge RPG Language Reference.
Part 4. Advanced Topics

Chapter 10, “Debugging Your Application,” on page 227
Describes how to debug an application.

Chapter 11, “Editing Output,” on page 239
Describes how to format output.

Chapter 12, “Using Picture, Sound, and Video Files,” on page 243
Describes the use of picture and sound files in your application.

Chapter 13, “Tips for Creating Online Help with IPF,” on page 245
Describes how to create and use online help in your application.

Chapter 14, “Tips for Creating and Using Windows Help,” on page 249
Describes how to create and use Windows help in your application.

Chapter 15, “Tips for Creating JavaHelp,” on page 253
Describes how to create and use JavaHelp in your application.

Chapter 16, “Working with Messages,” on page 259
Describes how to create and use message files in your application.

Chapter 17, “Communicating Between Objects,” on page 265
Describes how to communicate between objects in your application.

Chapter 20, “Creating and Running VisualAge RPG Applets,” on page 293
Describes how to create and run Java applets.

Chapter 18, “Calling Java Methods from VisualAge RPG Programs,” on page 279
Describes how to call Java methods.

Chapter 19, “Considerations When Compiling for Java,” on page 287
Describes RPG source restrictions, possible required source changes, and runtime differences for Java applications.

Chapter 21, “Calling System Functions when Compiling for Java,” on page 299
Describes how to call external procedures through the Java Native Interface.

Chapter 22, “Creating Non-GUI VisualAge RPG Programs,” on page 375
Describes how to create non GUI applications.

Chapter 23, “DBCS Considerations,” on page 381
Describes how to prepare your application for translation.

Chapter 24, “Merging Code in Your Application,” on page 385
Describes how to merge pieces of code into your application.
Chapter 10. Debugging Your Application

The debugger provided with VisualAge RPG helps you detect and diagnose any errors in your application. It can be used to debug multiple-language applications. You can:

- Manage execution of applications and DLLs
- Set and control breakpoints
- Display and modify program states by using storage, registers, variables, and call stack windows.

This section illustrates some of these features using a VisualAge RPG program.

To debug Java code generated by VARPG, you need Windows NT and the Distributed Debugger.

Starting the Debugger

To start the debugger, select the Debug menu item from the Project menu. Two windows appear. The Debug Session Control window and the VisualAge RPG Source window. Figure 41 illustrates these two windows.

![Figure 41. The VisualAge RPG Source and Debug Session Control Windows](image)

When the debugger starts, it searches for the VisualAge RPG source member and then displays it in the Source window. Once the source is displayed, you can perform debugging tasks.

**Note:** The current position of the executing program is indicated by a highlighted line number. Here, the first line of the source member is highlighted.
Displaying the Assembly Code

If the debugger does not find the VisualAge RPG source, it loads the program and displays the assembly source code instead of the VisualAge RPG source code. The window that is displayed is similar to the one displayed in Figure 42.

Figure 42. The Disassembly Window.

To correct this problem, make sure that the VisualAge RPG source code (.VPG file) resides on your workstation.

Loading the DLL Occurrence

If the assembly source code is displayed, it means that the VisualAge RPG source member cannot be found. To resolve this, select Set load occurrence from the Breakpoints pull-down menu. The load occurrence breakpoint window is displayed (Figure 43 on page 229). From this window, you can load the DLL occurrence. Type the following information, and then select OK: application_name.DLL

This returns you to the debug session. When the assembly source view is displayed again, press the R key to resume execution of the program. When the DLL is loaded, the system displays a message and the name of your application is displayed in the control window. You can now click on the application to get the source to display. If the source does not display this means you have lost or deleted the source.
If your application uses the START opcode to start another component, you will have to use this procedure to load the other component DLL. This will allow you to set breakpoints within the other components.

**Entering Debug Startup Information**

If the executable file cannot be located, the debugger displays a window similar to the one in Figure 44 below. You can re-enter the program name and parameters on this window.

![Startup Information Window](image)

*Figure 44. Startup Information.*

**Setting a Breakpoint**

You can control how your program executes by setting breakpoints. A breakpoint stops the execution of your program at a specific location or when a specific event occurs. To set a breakpoint, move the cursor to the line number that you would like to break at and double-click mouse button 1. The debugger highlights the line number with a red mark. You can repeat this process as many times to mark all the necessary lines that you would like to break at. Figure 45 on page 230 illustrates the way the screen looks with several breakpoints set. You can view all of your breakpoints in the Breakpoints List window.

![Load Occurrence Breakpoint Window](image)

*Figure 43. Setting the Load Occurrence Breakpoint.*
Figure 45. Setting Several Breakpoints.

Select Breakpoints> List to display this window. The following information is also provided for each breakpoint:
- The enablement state
- The type of breakpoint
- The position of the breakpoint
- The condition under which the breakpoint is activated
Running with Breakpoints

Pressing the R key causes the program to run. It stops at the first breakpoint that it encounters. When the debugger encounters a breakpoint, it stops and highlights the entire line as shown in Figure 47 below. This indicates the position where the executing program has paused.

![Figure 47. Running with Breakpoints.](image)

Using the Mouse or Keyboard to Start Debug Functions

Most debug functions can be started using the mouse or keyboard. For example, to set a breakpoint location, you double-click mouse button 1 on a line number. The same thing can be accomplished by selecting Line from the Breakpoints pull-down menu. When you select Line, the Line Breakpoint window is displayed. You must then enter the line number. When you enter the line number and press Enter, the line you selected is highlighted with red.
You can also resume execution of the program in different ways. Do any of the following:
- Press the letter R
- Move the mouse to the Run pull-down menu, then select Run
- Move the mouse to the run icon on the toolbar and single-click mouse button 1

Selecting Options from the Tool Bar

The following table lists all the options available on the tool bar and briefly explains each one.

<table>
<thead>
<tr>
<th>Icon</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step over</td>
<td>Executes the current (highlighted) line in the program, but does not enter any called function.</td>
</tr>
<tr>
<td>Step into</td>
<td>Executes the current (highlighted) line in the program and enters any called program or function.</td>
</tr>
<tr>
<td>Step debug</td>
<td>Executes the current (highlighted) line in the program. The debugger steps over any function for which debug information is not available, and steps into any function for which debugging information is available.</td>
</tr>
<tr>
<td>Step return</td>
<td>Automatically executes the lines of code up to, and including, the return statement of the current function.</td>
</tr>
<tr>
<td>Run</td>
<td>Begins execution of the program at the current (highlighted) line.</td>
</tr>
</tbody>
</table>
Halt  Stops execution of the program.
Views  Toggles to the next view.
Monitor expression  Displays a variable or expression in a monitor window.
Call stack  Views the active functions of a thread’s call stack.
Registers  Displays the threads registers in the register window.
Storage  Displays the contents of storage in the storage window.
Breakpoints  Lists all the breakpoints that have been set.
Debug session control  Displays the debug session control window.

Displaying and Changing Variables, Arrays, and Structures

Displaying a variable, array or other valid VisualAge RPG structure while debugging is a commonly used function. The easiest way to do this is to move the mouse to any specification where fields are allowed and double-click mouse button 1. For example, move the mouse to the conditioning indicator, factor 1, factor 2, result field, and/or the resulting indicators and double-click. This causes the specification’s contents to be displayed.

Note: If the variable is an operand for the EVAL operation code, select the variable you want to display by highlighting it with the mouse, then double-click.

If the field or structure you would like to display or change is in view, then the simplest way to display its content is to use the mouse and double-click it. However, if you are dealing with a large program, and you cannot locate a certain variable or structure easily, then Monitor expression from the Monitors pull down menu (Note that pressing Ctrl-M accomplishes the same thing).

On the Monitor Expression window, as shown in Figure 50, type the expression, field or structure that you would like to display, then press Enter.

Figure 50. The Monitor Expression Window.

After you press Enter, the VisualAge RPG field or structure is displayed in the Program Monitor window, as shown in Figure 51 on page 234.
Changing the Contents of a Field or Structure

Once you display a field or a VisualAge RPG structure, you can change its contents. To do this, double-click on the value in the Program Monitor window, type the new value, then press Enter.

Changing the Representation

The debugger allows you to change the representation for any displayed variable on the program monitor. Representation types can be decimal, hexadecimal, binary, or string; any valid representation for that variable or structure. To do this, select a variable in the Program Monitor window. Then select a representation from the Edit>Representation menu. The contents of the variable is now displayed in the representation you selected.

Changing the Default Representation

Variables have default representation types. For example, a character field would be displayed in the Program Monitor window as characters, not hexadecimal. The debugger allows you to alter this behavior. You have the option of setting the default representation for each data type. To change the default representation for a field, select Options>Debugger settings>Default data representation>System. The Default Data Representation window is displayed.

Displaying Pointers and Storage

One of the data types supported by VisualAge RPG is pointers. Figure 52 on page 235 illustrates an example of displaying a pointer value by using the Storage window. Select Monitor>Storage to display the Storage window.
Figure 52. Displaying a Pointer Value.
Changing the Debugger Views

Most of the examples in this section illustrate the VisualAge RPG Source view. The debugger also displays other views: disassembly and mixed view. To change views, select an option from the View menu, as shown in Figure 53.

Figure 53. Changing the Debug Views.
Setting Fonts

There are many options available in the debugger that allow you to customize your debug session. For example, you can set your fonts. Figure 54 displays the font window. To display the font window, select Options>Window settings>Fonts. In the font window, select the desired font, style, and size, then select OK. The font changes display in your debugging session.

*Figure 54. Setting fonts.*
Chapter 11. Editing Output

The compiler supports editing capabilities that determine how data is formatted when it is displayed in entry field and static text parts. To edit the output, you can set edit codes or edit words in the properties notebook for these parts.

Edit codes let you format data according to predefined formats, while edit words let you define your own formatting. You can specify either an edit code or an edit word for a part: you cannot specify both.

Edit codes and edit words can be specified only for numeric entry field and numeric static text parts.

When data from a formatted entry field is read into your program, the compiler strips all editing characters before returning the data to your program.

Note: Edit code entries on the control specification in your application are ignored; they have no effect on the output of these edit codes.

Edit Codes

Several edit codes are supported to format the data into predefined formats. These formats insert the proper thousand and decimal separators, and determine how a negative number is displayed by providing a fixed or floating minus sign or the CR (Credit) symbol.

You can optionally specify asterisk protection or floating currency symbol with the edit codes. If you specify asterisk protection, an asterisk is displayed with each zero that is suppressed. If you specify floating currency symbol, the symbol appears to the left of the first significant digit. The symbol does not display on a zero balance when an edit code is used that suppresses the zero balance.

The actual characters to be used for the thousand and decimal separators and currency symbol are determined by the operating system when the application is run.
The following table summarizes the supported edit codes and the editing they provide, and provides examples.

**Note:** The compiler does not support user-defined edit codes. User-defined edit codes are defined and stored on the AS/400 system, and are not available to VisualAge RPG programs.

**Table 11. VisualAge RPG Edit Codes**

<table>
<thead>
<tr>
<th>Edit Code</th>
<th>Com-mas</th>
<th>Decimal Point</th>
<th>Sign for Negative Balance</th>
<th>Positive Number Example</th>
<th>Negative Number Example</th>
<th>Zero balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>0123456789</td>
<td>0123456789-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Yes</td>
<td>Yes</td>
<td>No Sign</td>
<td>124,567.89</td>
<td>124,567.89</td>
<td>.00</td>
</tr>
<tr>
<td>2</td>
<td>Yes</td>
<td>Yes</td>
<td>No Sign</td>
<td>124,567.89</td>
<td>124,567.89</td>
<td>.00</td>
</tr>
<tr>
<td>3</td>
<td>No</td>
<td>Yes</td>
<td>No Sign</td>
<td>124567.89</td>
<td>124567.89</td>
<td>.00</td>
</tr>
<tr>
<td>4</td>
<td>No</td>
<td>Yes</td>
<td>No Sign</td>
<td>124567.89</td>
<td>124567.89</td>
<td>.00</td>
</tr>
<tr>
<td>A</td>
<td>Yes</td>
<td>Yes</td>
<td>CR</td>
<td>124,567.89</td>
<td>124,567.89CR</td>
<td>.00</td>
</tr>
<tr>
<td>B</td>
<td>Yes</td>
<td>Yes</td>
<td>CR</td>
<td>124,567.89</td>
<td>124,567.89CR</td>
<td>.00</td>
</tr>
<tr>
<td>C</td>
<td>No</td>
<td>Yes</td>
<td>CR</td>
<td>124567.89</td>
<td>124567.89CR</td>
<td>.00</td>
</tr>
<tr>
<td>D</td>
<td>No</td>
<td>Yes</td>
<td>CR</td>
<td>124567.89</td>
<td>124567.89CR</td>
<td>.00</td>
</tr>
<tr>
<td>J</td>
<td>Yes</td>
<td>Yes</td>
<td>-(minus)</td>
<td>124,567.89</td>
<td>124,567.89-</td>
<td>.00</td>
</tr>
<tr>
<td>K</td>
<td>Yes</td>
<td>Yes</td>
<td>-(minus)</td>
<td>124,567.89</td>
<td>124,567.89-</td>
<td>.00</td>
</tr>
<tr>
<td>L</td>
<td>No</td>
<td>Yes</td>
<td>-(minus)</td>
<td>124567.89</td>
<td>124567.89-</td>
<td>.00</td>
</tr>
<tr>
<td>M</td>
<td>No</td>
<td>Yes</td>
<td>-(minus)</td>
<td>124567.89</td>
<td>124567.89-</td>
<td>.00</td>
</tr>
<tr>
<td>N</td>
<td>Yes</td>
<td>Yes</td>
<td>-(floating minus)</td>
<td>124,567.89</td>
<td>-124,567.89</td>
<td>.00</td>
</tr>
<tr>
<td>O</td>
<td>Yes</td>
<td>Yes</td>
<td>-(floating minus)</td>
<td>124,567.89</td>
<td>-124,567.89</td>
<td>.00</td>
</tr>
<tr>
<td>P</td>
<td>No</td>
<td>Yes</td>
<td>-(floating minus)</td>
<td>124567.89</td>
<td>-124567.89</td>
<td>.00</td>
</tr>
<tr>
<td>Q</td>
<td>No</td>
<td>Yes</td>
<td>-(floating minus)</td>
<td>124567.89</td>
<td>-124567.89</td>
<td>.00</td>
</tr>
<tr>
<td>Y (2.)</td>
<td></td>
<td></td>
<td></td>
<td>1984-12-25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z (3.)</td>
<td>No</td>
<td>No</td>
<td>No Sign</td>
<td>1234567</td>
<td>1234567</td>
<td></td>
</tr>
</tbody>
</table>

Notes:
1. All edit codes suppress leading zeros
2. The Y edit code is used to date fields. The date field should be defined as a numeric field. The output of this edit code is in the form nnnn-nn-nn. This format cannot be changed. The date separator character is determined by the operating system when the application is run.
3. The Z edit code removes the + or − sign.

**Edit Words**

You can use edit words if none of the supplied edit codes meets your editing requirements. An edit word is a template that is applied to your data before it is placed in the part. With edit words you can specify:

- Suppression of leading zeros
- Leading asterisks
- The fixed/floating currency symbol
- The position of thousands and decimal separators.
**Note:** When you use edit words, make sure that you specify the currency, decimal, and thousands symbols correctly. If the symbols do not match the edit word, you will get improperly formatted output but no runtime error. These symbols are replaced by the runtime operating system values when the application is run.

**Parts of an Edit Word**

An edit word consists of the body, the status and the expansion. These parts are shown in the following example:

```
x x x , x x $ 0 . x x & C R * x T O T A L
```

where

- **BLANK** = x
- **CURRENCY SYMBOL** = $
- **THOUSAND SEPARATOR** = ,
- **DECIMAL Symbol** = .

**Body of an edit word**

The body is the space for the digits that are transferred from the data field to the part. It begins at the farthest left position of the edit word. It contains a number of blanks plus one zero or asterisk, the total equals the number of digits in the data field to be edited.

The following characters have special meaning when used in the body of an edit word:

- **Blank** A blank is replaced with the digit from the corresponding position of the data field.
- **Ampersand** An ampersand causes a blank in the edited display.
- **Zero** A zero stops zero suppression. The zero is itself a digit position. Any zeros in the data field to the right of the stop-zero-suppression character are displayed. Each zero that is suppressed is replaced by a blank.
- **Asterisk** An asterisk instead of a zero can be used as a stop-zero-suppression character. This is called asterisk protection, and each zero that is suppressed is replaced by an asterisk. Any asterisks or zeros to the right of the stop-zero-suppression character are constants, and will be displayed as-is.
- **Currency Symbol** If you code a currency symbol immediately to the left of the stop-zero-suppression character, a currency symbol is inserted in the position to the left of the first significant digit. It is called a floating currency symbol when it is used in this manner. If you code a currency symbol in the farthest left position of the edit word, it is fixed and is displayed in the same location. It is called a fixed currency symbol.

**Thousand Separator and Decimal Separators**

Thousand and decimal separators are displayed in the same relative positions in which they are coded in the edit word. All other characters are displayed if they are to the right of significant digits in the edit word. If they are to the left of the high-order significant digit in the edit word, they are blanked out or replaced by an asterisk if asterisk protection is being used.
Status of an edit word
The status positions display the sign of the data. The status continues to the right of the body to either a credit (CR) or minus (-) symbol. These two symbols display only when the field is negative. An ampersand (&) is used for displaying a blank.

Expansion of an edit word
The expansion positions are not changed by the edit operations. The expansion position starts at the first position to the right of the status (or body, if status is not specified). The expansion cannot contain blanks. If a blank is required, use an ampersand in the edit word.
Chapter 12. Using Picture, Sound, and Video Files

The animation control, canvas, graphic push button, image, media, and menu item parts allow you to display images on your windows by specifying a valid image name in the **FileName** attribute.

Valid **Windows** image formats include:

**OS/2 and Windows Bitmaps**
The file extensions .BMP, .VGA, .BGA, .RL3, .RL4, and .RL8 are recognized as OS/2 or Windows bitmaps.

**Icon Format**
The .ICO file extension is recognized as an icon file.

**CompuServe Graphics Interchange Format**
The .GIF file extension is recognized as a GIF file.

**ZSoft PC Paintbrush Image File Format**
The .PCX file extension is recognized as a Paintbrush file.

**Microsoft/Aldus Tagged Image File Format**
The .TIFF and .TIFF file extensions are recognized as TIFF files.

**Truevision Targa/Vista Bitmap**
The file extensions .TGA, .VST, and .AFI are recognized as Targa/Vista files. This class only supports 8 bit-per-plane and 24 bit-per-plane images.

**Amiga IFF/ILBM Interleaved Bitmap Format**
The file extensions .IFF and .LBM are recognized as interleaved bitmap files.

**X Windows Bitmap**
The .XBM file extension is recognized as a X Bitmap file. This class supports X10 and X11 1bpp bitmaps. Some .XBM files with text strings inside look to be sprites or icons and are not supported.

**IBM Printer Page Segment**
The following file extensions .PSE, .PSEG, .PSEG38PP and .PSEG3820 are recognized as PSEG files. PSEG files are used to include image data in BookMaster documents. PSEG files only contain 1 bit-per-plane, which is always ink on paper, that is, black on white.

Valid **Java** image formats include:
- CompuServe Graphics Interchange Format (GIF)
- Joint Photographic Experts Group format (JPG, JPEG)

In addition, you can add sound and video by using the media part, which supports WAV, .MID, .MPG, .MOV, .DAT, and .AVI files.

When developing a VisualAge RPG application that includes pictures, sound, or video, avoid hard coding the **FileName** attribute for the parts. The user will probably install your application in a different directory than the one in which it was developed.
To be sure that these files are found at run time, use the **Current directory** string (.\): a dot and a backslash followed by the file name. At run time, the file is found in the current directory from which the application is run.

For example, in the properties notebook for a graphic push button, specify the following as the file name for an icon named EXIT.ICO, so that it will be found at run time in the current directory.

```
.\EXIT.ICO
```

**Note:** For applications running in Windows, the current directory must be specified in the AUTOEXEC.BAT file.

During build time, you must copy your picture files to the build-time directory to access these files.

Before packaging your application for distribution, copy all associated picture and sound files to the appropriate runtime subdirectory (RT_JAVA or RT_WIN32) of your project, because it is this directory that is packaged and distributed to users. See [Part 5, “Distributing Your Application,” on page 413](#) for instructions on packaging your application.

### Creating Icons for Windows

If you have VisualAge for C++ for Windows, you can use the Resource Workshop utility to create Windows icons.

### Converting OS/2 Icons to Windows Format

VisualAge RPG includes a utility program to convert OS/2 icons and bitmaps into Windows versions. For details on this utility and its parameters, go to a DOS prompt and type `IBMPCNV -H`.
Chapter 13. Tips for Creating Online Help with IPF

The Information Presentation Facility (IPF) lets you create and manage online help files for your application. You can also use IPF to create tutorials and online documentation. With VisualAge RPG, the online help that you create will be native Windows help.

This section introduces IPF, and gives you some tips for creating online help for your application. For detailed information about using IPF, see *Information Presentation Facility Guide and Reference* (available online). You should also see the online document entitled *IPF Restrictions*. This document provides details on the subset of IPF tags that you are restricted to in a Windows environment.

You can reuse UIM help source from an AS/400 system, as well. See “Reusing UIM Help” on page 222.

Creating Online Help

To add online help for a part in your application:
1. Display the part’s pop-up menu.
2. Choose Help text. An edit session opens.
3. Type the contextual help text for the part.
4. Tag the help text using the IPF tag language.
5. Save the help by choosing Save from the File menu.

Using IPF

The source for VisualAge RPG application help modules is in IPF format. IPF enables you to create online information, specify how it will appear on the screen, connect various parts of the information, and provide help information that can be requested by the user. IPF features include:
- A tagging language that formats text, provides ways to connect information units, and customizes windows
- A compiler that creates online documents and help windows
- A viewing program that displays formatted online documents

Supporting Help for Other Languages

You can copy and manually edit the .VPF file using any text editor. Do not, however, do either of the following:
- Modify or remove the number that appears after the res= text. That number is the resource identifier, and it is generated by the GUI Designer when you create the help for a part. The resource identifier is used to locate the appropriate help text. If you delete or change a resource identifier, the help text pertaining to it will not be located.
- Remove the heading information. You can replace the heading information with the translated text.

Adding Graphics to Your Online Help

Use the :artwork. tag to imbed graphics inside the source files, as required. The graphics must be in bitmap format (.BMP files).
Deciding What Type of Help to Provide

Users can access help in three different ways within your VisualAge RPG application:

**Context-sensitive help**
Help information that is adapted to the current context of a choice or part. A user accesses this help by pressing F1 when a choice or part is in focus. You can provide this help via the part’s pop-up menu.

**Window-level help**
Information about the purpose of a window. A user accesses this help by pressing on a help push button. You can provide this help by creating a Help push button and adding the associated help information.

**Task help**
Information about tasks the user can perform with your application. A user accesses this help by hypertext linking to other help information from within a help panel. You can supply this information by creating online information and creating a hypertext link to it from the window-level or context-sensitive help. A hypertext link allows a user to jump from one help panel to another, or from selected text within a help panel to related information.

**ToolTip help**
Hover help style information about the tools that are available for use. To create this help, go to the ‘general’ page of a part’s properties notebook and type a description of the tool (up to 15 characters) in the entry field. You can also use a message identifier, for example *MSG0001, to specify the help text.

Adding Context-Sensitive Help
To add context-sensitive help for a part, select Help text from the part’s pop-up menu. This starts an edit session that contains information similar to that shown in Figure 55.

```
:h1 res=01.PSB0000C
:p.Help
```

*Figure 55. Edit session for adding online help*

The :h1 res=01. is a resource identifier that is automatically generated. Do not edit this text. Type a heading after this tag that identifies the purpose of the help panel, and type the help text after the :p. tag.

Creating a Help Push Button
To create a Help push button, select a push button from the parts palette with the right mouse button, move the mouse pointer onto the design window, and right-click again. Select Help text from the push button’s pop-up menu to edit the help information. Set the Help Enable attribute for that push button, and set the Label attribute to the word Help.

Creating Hypertext Links
To link related pieces of help information so that your users can find the appropriate information quickly and easily, use a link tag in your help text. You can create links to a help panel using a resid or a refid.
For linking to help panel defined with a id=

For linking to help panel defined with a res=
:link reftype=hd res=15433.Search push button:elink.
Chapter 14. Tips for Creating and Using Windows Help

One of the features of VisualAge RPG is the ability to create cursor-sensitive help for your applications. You create the help by right clicking the mouse on a part in the design window and choosing Help text. This starts the editor. You write the help text using the Information Presentation Facility (IPF) tag language. During the build process, the help source is compiled to create the HLP file. The IPF tag language results in a help file with a distinctive OS/2 look. This section explains how you can create true Windows help for your application.

What you Need

You need two tools to create a Windows help file:

• A word processor capable of saving files in Rich Text Format (RTF) format
• The Windows help compiler

The help compiler uses a help source file saved in RTF as input. Several word processors, including Lotus WordPro, Microsoft Word, and WordPerfect are capable of saving files in RTF format. Note that the Windows WordPad editor can save files in RTF format. However, this particular RTF format cannot be used to create help files. It does not retain many of the formatting options required by the help compiler to create a help file.

The Help Compiler Workshop is a tool available from Microsoft that consists of an IDE for managing your help files, as well as the help compiler. It can be downloaded from the Microsoft help compiler FTP at URL:


There are many tools on the market available, commercially and as shareware, that provide complete help authoring environments. In addition, there a several books available that describe how to use the Help Compiler workshop. Many of these books include a CD-ROM with the help compiler workshop, such as the Microsoft Windows 95 Help Authoring Kit ISBN1-55615-892-0.

Steps to creating Windows help

The basic steps to follow to use Windows help in your application are:

1. Establish the resource id for each part that will have help.
2. Write the help text.
3. Create the help project file.

Establishing the Resource ID

Every part, such as an entry field, push button, or window, has an identifier assigned to it usually referred to as a resource ID. VisualAge RPG assigns resource IDs for you and they cannot be changed. To see the resource ID for a part, right-click on the part in the design window. Select Properties to show its properties notebook. The resource Id is the number at the top of the General page. In the following example, it is the number 12 next to the Part ID:
Figure 56. Displaying the Resource ID

During the build process, VisualAge RPG generates a resource ID table entry for each part that you have created help for using the Help text menu item from the part’s popup menu. The Windows help engine uses this table to determine the resource ID for a part so it can display the correct help. You must create the help text for each part in this way for the part to have Windows help. Currently, VisualAge RPG does not create this table entry automatically for you. If you do follow this process, no help is displayed and no error message is generated.

Writing the Help Text

Before writing the help, you need to know a few Windows help terms. The following files are needed before you can create a Windows help file (HLP extension):

**Topic file**
This file contains your help text. Your help project can consist of one or more topic files. Topic files contain one or more topics. You create the topic file with your word processor and save it in RTF format (RTF extension).

**Project file**
The project file contains information about your help file. It contains such things as which topic files are to be included. The project file is maintained by the Help Workshop IDE. Typically, you do not modify it directly.

**Contents file**
If you want a Contents tab when the help file is displayed, you must have a contents file. The Contents file is also created and maintained by the Help Workshop IDE.

The following example outlines the basic steps for creating a topic file with one topic. It has the help text for the entry field part. Lotus WordPro is used to create the topic file. When the new document is opened in your word processor, type a title at the top of the page such as Help for Entry Field. Following the title, type the body of the help text.

Each topic must have a topic ID. A topic ID is a footnote with the # symbol.

Here are the steps for creating the required footnote in WordPro. Follow the steps in your word processor for creating footnotes with the # symbol:
1. Position the cursor just before the topic heading.
2. Select Create-Footnote/EndNote...
3. On the Footnotes dialog press OK.
4. The cursor will be positioned at the bottom of the document in the footnote section.
5. Type the topic Id: HelpForEF.
6. Position the cursor at the beginning of the topic Id.
7. Right click the mouse and select Text properties from the pop-up menu.
8. On the Properties dialog, select the Bullet and number tab.
9. Check the Edit on page checkbox.
10. Type a # character before the topic Id.
11. Close the Text properties dialog.

Your document should look similar to the following. The data following the line is the footnote:

```
#Help for Entry field

This is help for the entry field part.

more stuff ...
```

________________________________________

#HelpForEF

You can have several topics in a single topic file. Each topic must begin on a new page. Once you complete typing the help text, save your topic file in RTF format.
Creating the Help Project File

Following are the basic steps to create a minimum project file. Start the Microsoft Help Workshop and do the following:

1. Create a new project file by choosing File-New and select New project. A new project is created.
2. Press the Files push button.
3. On the Topic files dialog, press Add... and add the topic file you just created. Press OK to close the Topic Files dialog.
4. Press the Windows push button.
5. On the Window Properties dialog, press Add to display the Add a New Window Type dialog.
6. Create a window named main, and close all dialogs to return to the Help Workshop.
7. Map the topic Id( HelpForEF) in the topic file to the resource Id for the entry field part(12).
8. Press the Map push button.
10. When the Add Map Entry dialog appears, type HelpForEF in the Topic ID field, and 12 in the Mapped numeric value field. Press OK.
11. Press OK to close the Map dialog.
12. Save and compile the project file. This will create the help (HLP) file.

Copy the new HLP file to the RT_WIN32 directory of the VARPG project.

Compiling the VARPG Program

During the build process, VisualAge RPG creates a HLP file in the RT_WIN32 directory. This will, of course, overwrite the HLP file you just copied. Also, a RTF file with will be created in the project’s source directory. If you have saved your topic file here with the same name, it will be overwritten. To prevent this, open the project’s Build Options properties notebook and go to the Help file page. Clear the Create RTF Help file check box. Now, VisualAge RPG will not build the help or create the RTF and HLP files.

Each time you add help to a part, you must recompile the VARPG program.

Testing the Help

Start the VARPG application. When the window appears, tab to the entry field and press F1. The help should be displayed in a Windows help window.

You can also display the help as What's this? help. To do this, open the properties notebook for the window. On the Style page, check the Context check box. The Minimize and Maximize button check boxes must be cleared.

To have the help displayed in a pop-window rather than a help window, check the Popup choice.

Creating a Contents File

If you want your help to have a Help Topics dialog box, you need to create a Contents file. A Contents file is created in the Help Workshop IDE when you select File-New and New contents file. Name the contents file the same name as the help file, and save it in the same directory.
Chapter 15. Tips for Creating JavaHelp

One of the features of VisualAge RPG is the ability to serve context-sensitive JavaHelp for your VARPG Java applications. (VisualAge RPG currently supports the JavaHelp 1.1 release.) To build and run VARPG Java applications that include JavaHelp, you need:

- A basic knowledge of the HTML 3.2 tags.
- JavaHelp metadata files for your application:
  - Navigational data - table-of-contents file (TOC)
  - HelpSet data - HelpSet and Map files
  - HTML topic files
- A copy of the Java 2 Software Development Kit, Standard Edition (J2SDK) version 1.2, or higher, installed on your workstation. (The J2SDK is available from Sun at URL http://java.sun.com/products/)

This section summarizes how to create basic, context-sensitive JavaHelp for your VARPG Java applications. For complete information on the JavaHelp System, see the JavaHelp System User’s Guide. All JavaHelp documentation is available with the JavaHelp System, which you can download at URL http://java.sun.com/products/javahelp.

The following steps summarize how to create and incorporate JavaHelp into your application:

1. Create a HelpSet:
   - Create the HTML topics.
   - Create a HelpSet file.
   - Create a map file.
   - Create a table-of-contents (TOC) file.

   Optionally, you can create an index file and a full-text search database. Refer to the JavaHelp System User’s Guide for details on these topics and the tools needed to implement search.

   - Compress and encapsulate the help files into a JAR file.

2. After creating your JavaHelp with all required files and packaging them in a JAR file, copy the JAR file to the RT_JAVA subdirectory of your project.

3. Build and run your project.

The JavaHelp system is file based - topics are contained in files that are displayed in a suitable viewer, one file at a time. It is a good idea to group related topics together to keep them organized and to make it as easy as possible to link the topics together. It is also important to organize topics so they can be easily packaged into a compressed JAR for your application. It is usually best to organize your topics in a folder hierarchy that you can “tear off” and place in the JAR file.

The Video Store Catalog application contains sample JavaHelp files. They are located in the javahelp and help subdirectories of the WDSC\samples\vidcust directory. Use these files as templates for developing your own JavaHelp.

Note: In Java, file and folder names are case-sensitive. Type names exactly as shown in the samples provided.
Creating a HelpSet File

When JavaHelp is started by your application, the first thing it does is read the HelpSet file. The HelpSet file defines the HelpSet for your application: the set of data that comprises your help system. The HelpSet file includes the following information:

Map file
The map file associates topic IDs with the URL or path name of your HTML topic files.

View information
Describes the navigators used in the HelpSet. The standard navigators are: table of contents, index, and full-text search.

HelpSet title
The name of the top-level TOC folder.

Home ID
The name of the (default) ID that is displayed when the help viewer is called without specifying an ID.

The HelpSet file (filename.hs) is coded in Extended Markup Language (XML) format. The following is an example of a HelpSet file:

```xml
<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE helpset PUBLIC "-//Sun Microsystems Inc./DTD JavaHelp HelpSet Version 1.0//EN" "http://java.sun.com/products/javahelp/helpset_1_0.dtd">

<helpset version="1.0">
  <!-- title -->
  <title>Video Store Catalog - Help</title>

  <!-- maps -->
  <maps>
    <homeID>11</homeID>
    <mapref location="Map.jhm"/>
  </maps>

  <!-- views -->
  <view>
    <name>TOC</name>
    <label>Table Of Contents</label>
    <type>javax.help.TOCView</type>
    <data>VIDCTOC.xml</data>
  </view>

</helpset>
```

Where:

<title> Names the HelpSet. This corresponds to the title of the help window.

<homeID> Specifies the name of the (default) ID that is displayed when the help is called if an ID is not explicitly specified.

<data> Specifies the path to the data used by the navigator. In our example, this is the TOC view. The TOC file name is uppercase and the xml extension is lowercase. The TOC file must exist in your help directory.
Creating the Map File

When your application activates JavaHelp, the first thing it does is read the application’s HelpSet file. The next thing it does is read the map file listed in the HelpSet file. The map file associates topic IDs with URLs (paths to HTML topic files). By convention, map file names include the \textit{jhm} suffix. The Map file is in XML format.

Following is an example of a map file:

```xml
<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE map PUBLIC "-//Sun Microsystems Inc.//DTD JavaHelp Map Version 1.0//EN" "http://java.sun.com/products/javahelp/map_1_0.dtd">

<map version="1.0">
  <mapID target="11" url="help/welcome.htm" />
  <mapID target="18" url="help/catalog.htm" />
  <mapID target="14" url="help/browse.htm" />
  <mapID target="15" url="help/new.htm" />
  <mapID target="16" url="help/top10.htm" />
  <mapID target="17" url="help/search.htm" />
</map>
```

\textbf{target} Specifies the part ID for the VARPG part. The part ID is automatically assigned to the part by the GUI Designer. You can retrieve it from the part’s properties notebook.

\textbf{url} Specifies the path to the HTML topic file containing the help text. The path can be relative or absolute.

Creating the TOC File

The table of contents (TOC) file describes to the TOC navigator the content and layout of the TOC. The TOC file is in XML format. Following is a small example of a TOC file:

```xml
<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE toc PUBLIC "-//Sun Microsystems Inc.//DTD JavaHelp TOC Version 1.0//EN" "http://java.sun.com/products/javahelp/toc_1_0.dtd">

<toc version="1.0">
  <tocitem text="Video Store Catalog - help">
    <tocitem text="Welcome" target="11"/>
    <tocitem text="Help" target="22"/>
    <tocitem text="Browse" target="14"/>
    <tocitem text="New" target="19"/>
    <tocitem text="Top 10" target="20"/>
    <tocitem text="Search" target="21"/>
  </tocitem>
</toc>
```

Where:

\textbf{tocitem} The first TOC entry specifies the title for your table of contents. (You can nest TOC entries within a higher-level entry.)

\textbf{text} Specifies the text to use for subsequent TOC entries.
target  Specifies the ID of the HTML topic to display when the user chooses this entry in the TOC. The ID corresponds to the part ID identified in the map file.

Creating the JAR File

Once all necessary help files are created, use the jar command to encapsulate and compress your files. Your jar file name must be as follows:

```
SOURCE_FILE_NAMEHS
```

Where the SOURCE_FILE_NAME part is the name specified in the Source file field on the Save as Application - VisualAge RPG window. The file name must end with HS and be in uppercase. The jar extension is lowercase.
Issue the command from the top most directory containing your help hierarchy. For example, if your help directory structure is as follows:

```
javahelp (directory)
  Map.jhm
  CATALOG.hs
  VIDCTOC.xml

  help (subdirectory)
    browse.htm
    catalog.htm
    new.htm
    search.htm
    top10.htm
    welcome.htm
```

Issue the jar command from the javahelp directory as follows:

```
jar -cf VIDCUSTHS.jar *.*
```

Copy the resultant jar file into the RT_JAVA subdirectory for your project. Build and run the project with the Java option (Build>Java or Run>Java, respectively).
Chapter 16. Working with Messages

You can create, view, edit, and delete messages for your VARPG application.

You can view and delete existing messages directly from the Define Messages window. Use the Define Messages window to access the Edit Message window, from which you can create a new message or modify an existing one.

Messages fall into two groups in VARPG: those that you cannot reference in your code at run time, and those that you can.

The first group consists of a label-type message that is used to replace a substitution label; for example, on a push button or a window.

The second group contains four types of messages: Action, Critical, Information, and Warning. These messages can be displayed on a message window or in a message subfile part. They can be used to dynamically update text in your interface at run time; for example, to display installation progress messages.

Defining Text for Substitution Labels

To associate text with a substitution label:
1. Ensure that you have defined a substitution label on the part. Follow the procedure described in the online help.
2. Choose Project→Define messages from the GUI Designer. The Define Messages window opens.
3. Select a label-type message from the list that is displayed.
4. Choose the Edit push button. The Edit Message window opens displaying the label you selected.
5. In the Message field, type the text to be substituted for the label
6. Select Save to keep your changes, or Cancel (or double click in the window’s system menu) to discard them.

Note: When sizing a part in the GUI Designer that has a substitution label, keep in mind that translated text may be longer than the original.
Creating a New Message

To create a new message:
1. Choose Project→Define messages from the GUI Designer. The Define Messages window opens.
2. Select Create. The Edit Message window opens.
3. In the Message Alias field, type a string up to 10 characters long. It must not contain blanks. Your code can use the message alias instead of the message ID to display the message.
4. Select a message type from the Type drop-down box. There are four types to choose from:
   
   **Message Type**
   **Meaning**
   
   **Action**
   Use this type of message for situations in which the user must take some action to correct the situation or choose an alternative action.
   
   **Critical**
   Use this type of message for situations in which the user must take immediate action to correct the situation or choose an alternative action.
   
   **Information**
   Use this type of message for situations in which you simply want to inform the user about something; but the user does not have to perform any action.
   
   **Warning**
   Use this type of message when the user can continue the original request without modification, but should be aware of the existence of some situation.
   
5. Type the message text in the Message field.
6. If you want to provide help for the message, type it in the Message Help field.

   When you create message help and use the DSPLY operation code to display the message, a Help push button will appear at the bottom of the message window. When the user clicks on this push button, the help text will be displayed as additional information.
7. Select the Moveable check box if you want the user to be able to move the message to the background and continue with other tasks before taking action with the message.
8. From the Buttons drop-down box, select what combination of push buttons you want to appear at the bottom of the message window:

   **Choice Buttons That Will Appear**
   
   **AbortRetryIgnoreButton**
   Abort, Retry and Ignore
   
   **okButton**
   OK
   
   **okCancelButton**
   OK and Cancel
   
   **retryCancelButton**
   Retry and Cancel
9. Select a default push button by selecting the **Button 1**, **Button 2**, or **Button 3** radio button. When the message window is displayed and the user presses the Enter key, the action associated with the default push button is performed. For example, if you selected enterCancelButton from the **Buttons** drop-down and you want the default push button to be Cancel, you would select the **Button 2** radio button.

10. Select **Save** to keep the message, or **Cancel** to discard it.

**Note:** Message identifiers (message IDs) range from MSG0001 to MSG9999, and are assigned by VisualAge RPG. When all message IDs in the range are used, VisualAge RPG posts an error when you try to create a new message, and no new message can be created until you delete one. After you delete a message, you can create a new message that uses the ID of the deleted one.

---

**Editing a Message**

To edit a message:
1. Select **Project**→**Define messages** from the GUI Designer. The Define Messages window appears.
2. Select a message from the list that is displayed. If you cannot find the message you want, follow the instructions in "Finding a Message."
3. Choose **Edit** from the Define Messages window. The Edit Message window opens, displaying the message you selected.
4. Change the message alias, type, text, help or message window information.
5. Select **Save** to keep your changes, or **Cancel** to discard them.

---

**Deleting a Message**

To delete a message:
1. Choose **Project**→**Define messages** from the GUI Designer. The Define Messages window opens.
2. Select a message from the list that is displayed. If you cannot find the message you want, follow the instructions in "Finding a Message."
3. Choose the **Delete** push button.

---

**Finding a Message**

Here are some tips for finding a message:
- If you know what the exact message ID is, use the **Sort by Message ID** feature of the Define Messages window. The messages appear in ascending order of message ID.
- If you know what type of message you are looking for, use the **Sort by Type** feature of the Define Messages window. The messages are sorted in ascending order of message ID within the following groups:
  1. Messages you can set at run time:
     a. Information
     b. Warning
     c. Action
     d. Critical
2. Messages you cannot set at run time (substitution labels).

You can move through the list of messages using either the arrow keys or the scroll bars. If the list is long, scrolling is the fastest way to find what you are looking for.

Using Messages with Logic

It is common practice to display messages in message windows at run time. Once a message is created, one way to display it is to use the DSPLY operation code and the message subfile part’s AddMsgID attribute.

For information on the AddMsgID attribute, see the VisualAge RPG Parts Reference, SC09-2450-05.

You can use the MSGDATA and MSGNBR keywords on the definition specification to define messages with substitution variables. A substitution variable is defined when you create the message by typing a percent (%) character followed by a numeric value (for example, %1 %2 %3). The substitution variable is replaced by the corresponding field defined in the MSGDATA keyword. For example, %1 would be replaced by the first field defined in MSGDATA, %2 by the second field defined in MSGDATA, and so on. The MSGNBR keyword must contain an 8-character message identifier; for example, *MSG0001.

To use message substitution on the DSPLY operation code, define a message data type on the D specification. For example:

```
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++
  *
D notFound      M     MSGNBR(*MSG0001)
D                 MSGDATA(cusno: file)
  *
```

The fields CUSNO and FILE are defined elsewhere in the program. Assume that the message text for message *MSG0001 is:

Customer number %1 was not found in file %2.

To display the message with the DSPLY operation and have substitution done, code the following on the C specification:

```
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq
C     notFound    DSPLY        rc     9 0
```

For more information on the DSPLY operation code, see the VisualAge RPG Language Reference, SC09-2451-04.

Translating Message Files

You do not have to recompile your application to incorporate translated messages.

You can have more than one message file in a runtime directory, by assigning different file extensions to each. For example, an English version of the compiled message file could be named SAMPLE.ENG and a German version could be named SAMPLE.GER. You can instruct the user to rename the appropriate message file to SAMPLE.MSG before running the application.
**Manually Changing Message Files**

You can manually edit the .TXM ASCII file for translation purposes. This file contains the messages you created for your application. It is created in the source directory that is specified when the application is created.

An example of the record layout of the file is shown in Figure 57.

```
MSG
MSG0001I: The file was saved to your current working directory.
MSG0002W: Another user already has this file open for editing.
```

*Figure 57. Sample record layout for a TXM file*

Make sure that you edit only the text that appears after the colon (:) in the record layout.

The first record identifies the message prefix, and the following records each represent a message in the application.

Each message has a message prefix, MSG; a four-digit identifier or ID number; and a letter describing the type of message. In our example, message number 1 is an information message, and message number 2 is a warning message.

Do **not** do any of the following:

- **Modify** the message ID. Changing message IDs will cause unpredictable results. Without a message ID, your message cannot be displayed.
- **Add** a message. The message style will not be defined, and the message will never display in the Define Messages window.
- **Delete** a message. The Define Messages window will still display everything about the message except its message text.

---

**Using Messages as Labels**

You can set the label for any part that has a LABEL attribute from the message text in a message file. Any label with the prefix "MSG" indicates the message text from a message file. In the example in Figure 58 the label for the push button PB1 is set with the text from message number 0001 in the message file.

```
C 'PB1' SETATR '*MSG0001' 'Label'
```

*Figure 58. Dynamically setting a part label from a message file*

If the message number cannot be found in the component message file, then the application searches the message file indicated by the *component MsgFile attribute for the message number. If the message number does not exist in either message file, then the message identifier (in this example, MSG0001) appears as the label text.
Chapter 17. Communicating Between Objects

With VisualAge RPG, you can perform various kinds of communications between objects.

**Part to Part**
You can link parts in VisualAge RPG so that one part notifies another that it has changed, and the recipient part issues an event when it is notified of this change.

**VisualAge RPG application to other PWS applications**
You can enable your application to exchange information with another application that supports the DDE protocol. A VisualAge RPG application can be either the client or the server in the exchange. For information on the client function, see "DDE Client" on page 74. The server function is described in this section.

**Component to Component**
You can enable one component to communicate with another.

You can also use operation codes to do the following:
- Call local functions
- Call local programs
- Start and stop components
- Call remote programs

This section provides helpful tips for each type of communication, and gives examples.

**Linking Parts**

The following parts can be linked using VisualAge RPG:
- Check box
- Entry field
- Image
- List box
- Media
- Media panel
- Slider
- Timer

A part that notifies another part when it changes is called the *source* part, and the part that is notified of this change is called the *target* part.

One way to set up communication between a source part and a target part is to use the Link page of the source part’s properties notebook. In the fields provided, type the name of the target part and the name of the window in which it resides. If you want the target to issue a Link event when it is notified by the source part, select the Enable notify target check box.

Alternatively, you can set up the communication link by setting the AddLink attribute and the target in the form WindowName|PartName. If you want the target to issue a Link event, set the AllowLink attribute to 1. 

© Copyright IBM Corp. 1994, 2005
shows sample code used to link a media panel part, MMP1, to a media part, AUDIO1.

```
*  'MMP1'   SETATR  'WIN2|AUDIO1'  'AddLink'
C  'MMP1'   SETATR   1       'AllowLink'
*  
```

Figure 59. Sample code showing one part linked to another

**Note:** You can set only one link for a source part in the GUI Designer, but you can set multiple links in your code.

### Using a VisualAge RPG Application as a DDE Server

Any VisualAge RPG application can act as a server in a dynamic data exchange (DDE) conversation.

Parts that can be a source of a LINK event can produce DDE data. A DDE client part can obtain data from a component of the same application or a different application. For more information about the DDE client part, see "DDE Client" on page 74.

For example, assume that you are building an application called CLIENT. It consists of a window called WINDOW_C, a DDE client part called DDECLI_C, and a static text part called STTEXT_C.

Suppose the application needs data from a server application called SERVER. This server application has a window called WINDOW_S and an entry field part called ENTRY_S. Whenever the value in the entry field of the server application is changed, the static text part of the client application is updated to reflect the change.

To establish a hot link between the client and server applications, you would specify the following attributes of the DDECLI_C DDE client part in the client application:

**AppName**

This is the name of the server application: SERVER.EXE.

**Topic**

This is the name of the server component, followed by a vertical bar, followed by the component instance name. For VisualAge RPG, in most cases the component name is the same as the component instance name, and also the same as the executable name. For this example, the component name is SERVER|SERVER.

**Item**

This is the name of the server part. For VisualAge RPG programs, this is the window name, followed by a vertical bar, followed by the part name. In this example, the item attribute value is WINDOW_S|ENTRY_S.
DDEAddLink

This is the name of the client part. It consists of the window name, followed by a vertical bar, followed by the part name. In this example, the DDEAddLink attribute is WINDOW_C|STTEXT_C.

DDEMode

Set DDEMode to 1 to begin the conversation and initiate the hot link between the server and the client. To terminate the conversation, set DDEMode to 2. This signals the Terminate event to the client application.

Communicating Between Components

Components are projects in VisualAge RPG. They represent one or more application windows that were created with the GUI Designer. An example is a window that prompts a user to enter the name of an image file, and then displays the image. To enable one VisualAge RPG component to communicate with another, use a component reference part. For more information, see "Component Reference" on page 65.

Making Local Calls

This section discusses local calls you can make using these operation codes:

<table>
<thead>
<tr>
<th>Operation Code</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALLB</td>
<td>Calls a local function. The function can be in an object code file (OBJ) or exported from a dynamic link library (DLL).</td>
</tr>
<tr>
<td>CALLP</td>
<td>Calls a local program or function (procedure) . The function must be exported from a dynamic link library (DLL). For more information, see &quot;Using Multiple Procedures&quot; on page 274. Using CALLP is preferable to using CALLB.</td>
</tr>
<tr>
<td>START</td>
<td>Starts a new component in the application or calls a local program.</td>
</tr>
</tbody>
</table>

Using the CALLB Operation

Use the CALLB operation code to call a function from your VisualAge RPG application. If you are linking to an OBJ that was compiled in a language other than RPG, make sure that the runtime environment for the compiler is correctly initialized and terminated (see the compiler documentation for more information).

The following examples illustrate the different ways that you can call a C function using CALLB. Figure 60 on page 268 contains the sample C function that is called.
#include <stdio.h>

/*The following two lines are required only if you compile */
/*the OBJ with the IBM C/C++ compiler. These lines */
/*are not required if the function is exported from a DLL. */
int _CRT_init(void);
void _CRT_term(void);
*
/* print the str and age parameters to a file */
void MYFUNC(char *str, int *age) {
    FILE *fp;
    int j;
    *
    /*The following line is required only if you compile */
    /*the OBJ with the IBM C/C++ compiler. This line */
    /*is not required if the function is exported from a DLL. */
    _CRT_init();
    *
    fp=fopen("myfunc.log", "a");
    *
    /* print the character data to a file*/
    for (j=0; j<10; ++j) {
        fprintf(fp, "%c", str[j]);
    }
    *
    /* if an age is given, print the age */
    if ( age == NULL ) {
        fprintf(fp, "no age is given\n");
    } else {
        fprintf(fp, "num = \%d\n", *age);
    }
    *
    fclose(fp);
    *
    /*The following line is required only if you compile */
    /*the OBJ with the IBM C/C++ compiler. This line */
    /*is not required if the function is exported from a DLL. */
    _CRT_term();
}

Figure 60. Sample C function, MYFUNC

Calling functions using named constants or literals

The following examples illustrate how to call a function using a named constant or literal:

```
DConst1 C CONST('MYFUNC')
Dwilma s 80a inz('mydata')
Dage s 9b 0 inz(32)
```

```
C *inzsr begsr
C***********************************************************************
C*********** *** CALLB in VRPG with a PLIST *** *********************
C***********************************************************************
C myplist plist
C parm wilma
C parm age
C CALLB Const1 myplist
C seton Const1
C endlr
```

Figure 61. Calling functions using a named constant
Calling functions using a procedure pointer

The following example illustrates how to call a function using a procedure pointer. If a procedure pointer is used with CALLB, then the *ROUTINE field in the program status data structure (PSDS) is not updated with the name of the function being called. The field is set to blanks.

```
* Dwilma s 80a inz('mydata')
  Dage s 9b 0 inz(32)
  C *inzsr begsr
  C callb MYFUNC
  C parm wilma
  C parm age
  C seton 1r
  C endsr

Figure 62. Calling library functions using a literal
```

Calling functions using a procedure pointer

```
Dp2 s * procptr inz(%paddr('MYFUNC'))
Dwilma s 80a inz('mydata')
  Dage s 9b 0 inz(32)
  C *inzsr begsr
  C callb p2
  C parm wilma
  C parm age
  C seton 1r
  C endsr

Figure 63. Calling functions using a procedure pointer
```

Calling functions without the required parameters

The following example illustrates how to call a function with less than the required number of parameters. Use the *OMIT parameter which maps to a NULL pointer.

```
* Dp2 s * procptr inz(%paddr('MYFUNC'))
Dwilma s 80a inz('mydata')
  Dage s 9b 0 inz(32)
  C *inzsr begsr
  C callb p2
  C parm wilma
  C parm *OMIT
  C seton 1r
  C endsr

Figure 64. Calling functions without the required parameters
```

Calling Local Programs using CALLP

Use CALLP to make calls to local programs synchronously. This means that the called program completes execution before the VisualAge RPG statement following CALLP is executed.

Each program that you call using CALLP requires a prototype. The prototype defines the system name of the called program and the number and types of parameters that the program is expecting. Specify this prototype using the PR type definition specification. This specification consists of:
Columns

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
</tr>
<tr>
<td>7-21</td>
</tr>
<tr>
<td>24-25</td>
</tr>
<tr>
<td>44-80</td>
</tr>
</tbody>
</table>

Use the CLTPGM keyword with the system name of the program as a parameter.

If the program expects parameters, use one definition specification for each parameter immediately after the PR definition specification. These definition specifications should consist of the name, length, and type of parameter. Specify the precision of numeric parameters. Always specify the VALUE keyword. You can also specify the ASC, DATFMT, DESC, DIM, LIKE, NOOPT, OPTIONS, and TIMFMT keywords on your parameter definitions.

**Figure 65** defines pgm1 to VisualAge RPG. One parameter can be passed to the program.

```
D pgm1
D parm1
PR
20A
CLTPGM('testprog')
VALUE
```

**Figure 65. Specifying definition specification parameters when calling local programs**

In **Figure 66** the CALLP operation code calls pgm1 with parameters f1d1 and 22.4.

```
CALLP pgm1(f1d1:22.4)
```

**Figure 66. Calling a local program using CALLP**

For more information on procedures, see “Using Multiple Procedures” on page 274.

**Calling Local Programs using START**

When you use the START operation code to call a program, VisualAge RPG does not wait for the called program to finish executing, but makes the call and then continues. From that point on, the called program executes independently of the VisualAge RPG program that called it.

When using START, you do not have to prototype local programs.

F2 can be a character literal, a named constant, or a variable name.

If F2 is a character literal, it is assumed to be a component. If it is a constant name and you specify LINKAGE(CLIENT) on the definition of the constant, it is assumed to be a local program. See **Figure 67 on page 271**.
If F2 is a variable name, it is assumed to be the component name unless you define the variable on a definition specification with LINKAGE(*CLIENT) specified. Any variable defined in this way can be used like any other RPG field. In Figure 68, the first START operation code will attempt to start a component, and the second START operation code will attempt to start a local program.

```
D test1 C 'component'
D test2 C 'testprog' LINKAGE(*CLIENT)
* To start a component:
C START 'xxx'
* To start a component:
C START test1
* Starts local program testprog.exe:
C START test2
```

**Figure 67. Example using START to call local programs**

START can still have a PLIST specified in the result field, or it can be followed by a list of PARMS. These PARMS are passed to the component or local program.

**Restrictions for CALLP and START**

Note these restrictions when using CALLP and START operation codes with local programs:

- The PATH environment variable is used to find the local program if the program name is not specified with a full path name.
- The program can normally have a maximum of 20 parameters. In some cases, this maximum is less than 20 because the command string must not exceed 1024 bytes. (The command string consists of the program name and the parameters converted to characters.)
- Pointers and procedure pointers are not allowed as parameters. Everything must be passed by value.
- When you use START with an error indicator to call local programs, the error indicator is set to ON if the local program cannot be started.
- LINKAGE(*SERVER) is not valid with the START operation code.
- When specifying the name of a program to call, include the extension if it is other than EXE. If you do not provide an extension, EXE is assumed. For example,

  CLTPGM('superc2')
  Calls SUPERC2.EXE

  CLTPGM('rexpgm')
  Calls REXXPDM.EXE

  CLTPGM('rexpgm.cmd')
  Calls REXXPDM.CMD

This applies when specifying the program name as a named constant for START, or when passing the program name as a variable.

**Figure 68. Defining variable names for the START operation code**

D name1 S 20A
D name2 S 20A LINKAGE(*CLIENT)
* C START name1
C START name2
Starting Components using START

Use the START operation code to start a new component in the application, and the STOP operation code to terminate its execution. For a detailed description of the syntax for these two operation codes, see VisualAge RPG Language Reference.

The following section describes the behavior of START and STOP with your application’s components.

Starting a Component

The START operation code starts a new component in the application. When the operation is performed, both the starting and the started components, together with any other active components in the application, are ready to receive user actions on all the parts currently enabled by all the components.

The START operation code is similar to the CALL operation code in the following ways:

- Parameters can be passed to a component.
- Parameters are mapped to the parameters in the *ENTRY PLIST of the target component.
- In the source component, factor 2 of the PARM operation code is copied to the result field of the same PARM operation code. When control returns to the source component, the result field is copied to factor 1.
- In the target component, the result field is copied to factor 1. When control returns to the source component, factor 2 is copied to the result field if the target component completes a successful startup.
- No checks or conversions are performed on the parameters.

The START operation code is different from the CALL operation code in the following ways:

- The terms called and calling are used with the CALL operation code. A called program is a program whose execution is requested by another program. A calling program is a program that requests the execution of another program. With the START operation code, the terms target (called) and source (calling) are used.
- CALL invokes a program, executes it, then returns back to the calling program with factor 1, factor 2, and the result field copied as described above. START initializes a component, executes its *INZSR, and returns to the source component with factor 1, factor 2, and the result field copied as described above. The difference is that with the START operation code, factor 2 in the target program is copied to the result field at the end of the *INZSR (if *INZSR is successful), not at the end of the program.
- Once the START operation has finished initializing the target component, the action subroutine in the source component continues executing, and the target component remains active with its action subroutines enabled to receive events.
- Since parameters are passed by address, any parameters that are passed can be accessed by both the source and target components after the initial START has ended. This means that both the source and target components can continue to share information using the parameter fields.

Terminating a Component

The STOP operation code terminates the execution of a component. If you do not specify the component name in factor 2, the component that is currently running is terminated. When a component is terminated, any child components that it may have started are terminated first.
When a STOP operation is performed that affects the currently executing component, operations following the STOP are not executed. In other words, the result of a STOP is immediate. For example, if COMPA starts COMPB, and COMPB is the component that is currently executing and it issues a STOP for COMPA, COMPB terminates first, followed by COMPA. No operations following the STOP are performed.

Terminating a component with a STOP is considered normal termination, and the *TERMSR is invoked for any final user processing.

Calling Remote Programs

This section discusses how your VisualAge RPG application can call an iSeries 400 program, and how an RPG application running on an iSeries 400 server can call a VisualAge RPG application.

Calling iSeries 400 Programs

Before your application can call an iSeries server program, you must set up the server.

The name of the called program can be either the iSeries server program name (optionally library-qualified) or an override name. You can define the program override using the Program page of the Define iSeries Information notebook. See "Notebook Considerations" on page 193 for information about what happens if the notebook pages do not contain an override name for the data area.

Table 12 and Figure 69 on page 274 illustrate how to call an iSeries program using an override name. The program in Figure 69 on page 274 calls MYLIB/LOOKUP on SERVER01.

Table 12. Enter this information on the Program page

<table>
<thead>
<tr>
<th>Program override name:</th>
<th>REMPGM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remote program name:</td>
<td>MYLIB/LOOKUP</td>
</tr>
<tr>
<td>Server alias name:</td>
<td>SERVER01</td>
</tr>
</tbody>
</table>
Note: If the program on the iSeries server contains a workstation file, it will fail when the system attempts to open it. Since the remote call command is done through the DDM server, the display device is unknown to workstation data management. A technique you can use is to create the workstation file on the iSeries server with the Display Device value set to the name of the session (OMXxxxx) and set the Maximum Number of Devices parameter to a value greater than 1. This will allow parameters to be passed to the iSeries server program. Do not try to explicitly acquire the session with an ACQ statement. This will cause a conflict to occur which results in an error. You still cannot acquire any 5250 emulator display device on your workstation, because it will result in a deadlock that can only be ended by rebooting the workstation.

Starting Workstation Programs from the iSeries server

If you have an RPG application running on the server and would like to start a VisualAge RPG application on a Windows workstation, use the STRPCCMD command.

Using Multiple Procedures

The ability to code more than one procedure greatly enhances your ability to code a modular application.
A VisualAge RPG program consists of one or more modules. A procedure is any piece of code that can be called with a bound call. VisualAge RPG has two kinds of procedures: a main procedure and a subprocedure. A main procedure is a procedure that can be specified as the program entry procedure and receives control when it is first called. Note that a main procedure is only produced when creating an EXE.

A subprocedure is a procedure that is specified after the main source section. A subprocedure differs from a main procedure primarily in that:

- Names that are defined within subprocedure are not accessible outside the subprocedure.
- The call interface must be prototyped.
- Calls to subprocedures must be bound procedure calls.
- Only P, D, and C specifications can be used.

Subprocedures can provide independence from other procedures because the data items are local. Local data items are normally stored in automatic storage, which means that the value of a local variable is not preserved between calls to the procedure.

Subprocedures offer another feature. You can pass parameters to a subprocedure by value, and you can call a subprocedure in an expression to return a value.

**Prototyped Calls**

To call a subprocedure, you must use a prototyped call. You can also call any program or procedure that is written in any language by using a prototyped call. A prototyped call is one where the call interface is checked at compile time through the use of a prototype. A prototype is a definition of the call interface. It includes the following information:

- Whether the call is bound (procedure) or dynamic (program)
- How to find the program or procedure (the external name)
- The number and nature of the parameters
- Which parameters must be passed, and which are optionally passed
- The data type of the return value, if any (for a procedure)

The prototype is used by the compiler to call the program or procedure correctly, and to ensure that the caller passes the correct parameters. Figure 70 shows a prototype for a procedure FmtCust, which formats various fields of a record into readable form. It has two output parameters.

```
* Prototype for procedure FmtCust (Note the PR on definition
  * specification.) It has two parameters.
D FmtCust PR
D Name 100A
D Address 100A
```

*Figure 70. Prototype for FmtCust Procedure*

To produce the formatted output fields, FmtCust calls a procedure NumToChar. NumToChar has a numeric input parameter that is passed by value, and returns a character field. Figure 71 on page 276 shows the prototype for NumToChar.
* Prototype for procedure NumToChar

* The returned value is a character field of length 31.

D NumToChar
   PR 31A
   * The input parameter is packed with 30 digits and 0 decimal
   * positions, passed by value.
   D NUMPARM 30P 0 VALUE

Figure 71. Prototype for NumToChar Procedure

If the program or procedure is prototyped, you call it with CALLP or within an
expression if you want to use the return value. You pass parameters in a list that
follows the name of the prototype, for example, name (parm1 : parm2 : ...).

Figure 72 shows a call to FmtCust. Note that the names of the parameters, shown
in Figure 70 on page 275 do not match those in the call statement. The parameter
names in a prototype are for documentation purposes only. The prototype serves to
describe the attributes of the call interface. The actual definition of call parameters
takes place inside the procedure itself.

C CALLP FmtCust(RPTNAME : RPTADDR)

Figure 72. Calling the FmtCust Procedure

Using prototyped calls you can call (with the same syntax):
• Programs that are on the system at run time
• Exported procedures in other modules
• Subprocedures in the same module

In order to format the name and address properly, FmtCust calls NumToChar to
convert the customer number to a character field. Because FmtCust wants to use
the return value, the call to NumToChar is made within an expression. Figure 73
shows the call.

*----------------------------------------------------------------------
* CUSTNAME and CUSTNUM are formatted to look like this:
* A&P Electronics (Customer number 157)
*----------------------------------------------------------------------
C EVAL Name = CUSTNAME + ' '
C + ' (Customer number ' + %trimr(NumToChar(CUSTNUM)) + ')' 

Figure 73. Calling the NumToChar Procedure

The use of procedures to return values, as in the above figure, allows you to write
any user-defined function you require. In addition, the use of a prototyped call
interface opens up a number of options for parameter passing.
• Prototyped parameters can be passed in several ways: by reference, by value (for
procedures only), or by read-only reference. The default method for RPG is to
pass by reference. However, passing by value or by read-only reference gives
you more options for passing parameters.
• If the prototype indicates that it is allowed for a given parameter, you may be
able to do one or more of the following:
  – Pass *OMIT
  – Leave out a parameter entirely
  – Pass a shorter parameter than is specified (for character and graphic
    parameters, and for array parameters)
Procedure Considerations

- You cannot define return values for a main procedure. Parameters must be passed by value.
- A main procedure is only contained within an EXE. you specify that its parameters be passed by value.
- Any of the calculation operations may be coded in a subprocedure. However, all files must be defined globally, so all input and output specifications must be defined in the main source section. Similarly, all data areas must be defined in the main procedure, although they can be used in a subprocedure.
- The control specification can only be coded in the main source section since it controls the entire module.
- A subprocedure can be called recursively. Each recursive call causes a new invocation of the procedure to be placed on the call stack. The new invocation has new storage for all data items in automatic storage, and that storage is unavailable to other invocations because it is local. (A data item that is defined in a subprocedure uses automatic storage unless the STATIC keyword is specified for the definition.)

The automatic storage that is associated with earlier invocations is unaffected by later invocations. All invocations share the same static storage, so later invocations can affect the value held by a variable in static storage.
- Exception handling within a subprocedure differs from that in a main procedure primarily because there is no default exception handler for subprocedures. Situations where the default handler would be called for a main procedure result in the abnormal end of the subprocedure.
- VisualAge RPG procedure names are in uppercase. When calling these procedures, make sure that the case matches that of the procedure.

Procedure Implications

As a programmer, you have the option of producing three possible target objects:
- A VisualAge RPG DLL (contains GUI operation codes)
- A utility DLL which contains only RPG subprocedures that do not include any GUI operation codes
- An RPG EXE which does not contain any GUI operation codes.

VisualAge RPG DLL Considerations

- VisualAge RPG DLL subprocedures are not externalized.

These subprocedures are designated as internal only by the compiler. Entry points are not externalized to other modules. Any attempt to link to these subprocedures will cause the link step to fail.
- The EXPORT keyword is not allowed on procedure specifications, since procedures cannot be exported from VisualAge RPG DLLs.

Utility DLL Considerations

This DLL is built when the keyword NOMAIN is provided on the control specification.

The compiler will produce both a DLL and LIB file as a result of the compilation. The LIB file will contain all the procedures that have the EXPORT keyword on their Begin P-specification. The LIB file allows you to link to the subprocedures that the DLL contains.
- The DLL consists of procedures only.
  - All subroutines (BEGSR) must be local to a procedure.
  - There are no GUI operation codes allowed in the source.
This includes START, STOP, SETATR, GETATR, %SETATR, %GETATR, SHOWWIN, CLSWIN and READS. DSPLY can be used, but if the procedure containing it is called from a VisualAge RPG DLL, then the DSPLY operation code does nothing.

- *inzsr and *termsr are not permitted.
- *ENTRY parms are not permitted.

Exception handling differs from the VisualAge RPG DLL in the following way:
- No information about the exception is communicated back to the caller if the caller does not reside in the utility DLL.
- The recommended way for a user to handle exceptions in a utility DLL is to have an error indicator, or a local *PSSR for each routine which returns an appropriate return code to the caller.
- The default exception handler is never invoked from a utility DLL, since the default exception handler is not invoked when an exception occurs in a procedure. If an exception occurs in the utility DLL and there is no error indicator or *PSSR, an exit() is performed and information about the exception is written to the FVDCERRS.LOG file.

**EXE Considerations**

- An EXE is built when the keyword EXE is provided on the control specification.
- The EXE consists of procedures only.
  
  All subroutines (BEGSR) must be local to a procedure. The EXE must contain a procedure whose name matches the name of the source file. This will be the main entry point for the EXE (i.e. the main procedure).
- There are no GUI operation codes allowed in the source.
  
  This includes START, STOP, SETATR, GETATR, %SETATR, %GETATR, SHOWWIN, CLSWIN and READS. DSPLY can be used.
- *inzsr and *termsr are not permitted.
- *ENTRY parms are not permitted.
  
  If there are entry parameters, they are specified on the parameter definition for the main procedure, and they must be passed in by VALUE (the VALUE keyword must be specified for each parameter).
- The EXPORT keyword is not allowed on the Begin P specification.
- Exception handling differs from the VRPG DLL. The default exception handler is never invoked from an EXE. If an exception occurs in the EXE, and there is no error indicator or *PSSR, an exit() is performed and information about the exception is written to the FVDCERRS.LOG file.
Chapter 18. Calling Java Methods from VisualAge RPG Programs

This section describes how to call Java methods from VARPG programs that have been converted to Java source code, and the additions to the VARPG language to support this. In order to call Java methods, the VARPG compiler needs the following information:

- The name of the method
- The class that contains the method
- The class of the returned object if the method returns an object
- Whether or not the method is a static method
- The data types of the parameters passed to the method

In addition, if the method is not a static method, then an object must be instantiated in order to call the method. If the method returns an object, then the compiler must have somewhere to store that object. If the method accepts an object as a parameter, then there must be some way to create that object.

These requirements have led to the following additions to the VARPG language:

- The Object data type
- The CLASS keyword
- Extension of the EXTPROC keyword

The Object Data Type and CLASS Keyword

Fields that can store objects are declared using the O data type. To declare a field of type O, code O in column 40 of the D-specification and use the CLASS keyword to provide the class of the object. The CLASS keyword accepts two parameters:

CLASS(*JAVA:class_name)

*JAVA identifies the object as a Java object. Class_name specifies the class of the object. It must be a character literal, and the class name must be fully qualified. The class name is case sensitive.

For example, to declare a field that will hold an object of type BigDecimal:

D bdnum S O CLASS(*JAVA:'java.math.BigDecimal')

To declare a field that will hold an object of type String:

D string S O CLASS(*JAVA:'java.lang.String')

Note that both class names are fully qualified and that their case exactly matches that of the java class.

Fields of type O cannot be defined as subfields of data structures. It is possible to have arrays of type O fields, but tables of type O are not allowed because they have to be preloaded at run time.

The following keywords cannot be used with the CLASS keyword:
Prototyping a Java Method

Like subprocedures, Java methods must be prototyped in order to call them correctly. The VARPG compiler must know the name of the method, the class it belongs to, the data types of the parameters and the data type of the returned value (if any), and whether or not the method is a static method.

The extended EXTPROC keyword can be used to specify the name of the method and the class it belongs to. When prototyping a Java method, the expected format of the EXTPROC keyword is:

```
EXTPROC(*JAVA:class_name:method_name | *JAVARPG:class_name:method_name)
```

*JAVARPG identifies the method as a VARPG-generated Java method. *JAVA identifies the method as a Java method that was generated from code originally written in Java, and not VARPG-generated. This distinction is important because methods generated from *JAVARPG will allow certain data types to be passed by reference that normally cannot be passed by reference in Java. This allows the same source code to be used when targeting Windows and when generating Java source code.

Both the class name and the method name must be character literals. The class name must be a fully qualified Java class name and is case sensitive. The method name must be the name of the method to be called, and is case sensitive.

The extended form of the EXTPROC keyword can only be used when calling Java methods. If targeting Windows, using this form of the EXTPROC keyword will result in a compiler error.

The data types of the parameters and the returned value of the method are specified in the same way as they are when prototyping a subprocedure. The only twist on this is that the data types actually map to Java data types. The compiler maps VARPG data types to Java data types as follows:

<table>
<thead>
<tr>
<th>Java Data Type</th>
<th>VARPG Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>char[]</td>
<td>graphic or unicode</td>
</tr>
<tr>
<td>boolean</td>
<td>indicator (N)</td>
</tr>
<tr>
<td>byte[]</td>
<td>alpha (A of any length)</td>
</tr>
<tr>
<td>byte</td>
<td>integer (3I)</td>
</tr>
<tr>
<td>int</td>
<td>integer (10I)</td>
</tr>
<tr>
<td>short</td>
<td>integer (5I)</td>
</tr>
<tr>
<td>long</td>
<td>integer (20I)</td>
</tr>
<tr>
<td>float</td>
<td>float (4F)</td>
</tr>
<tr>
<td>double</td>
<td>float (8F)</td>
</tr>
<tr>
<td>any object</td>
<td>object (O)</td>
</tr>
</tbody>
</table>
Zoned, Packed, Binary, and Unsigned data types are not available in Java. If you pass a Zoned, Packed, Binary, or Unsigned field as a parameter, the compiler will do the appropriate conversion, but this will most likely result in truncation and/or loss of precision.

If the method you are calling is a VARPG-generated method, meaning that "JAVARPG has been specified as the first parameter of the EXTPROC keyword, then Packed, Zoned, Binary, and Unsigned data types can be specified as the data type of parameters and returned values. Methods generated from code originally written in Java cannot use Packed, Zoned, Binary, and Unsigned data types on the prototype for parameters or return values.

When calling a method, the compiler will accept arrays as parameters if the parameter is prototyped using the DIM keyword. Otherwise, only scalar fields, data structures, and tables will be accepted.

Currently, you cannot call methods which expect the following Java data types or which return values of these types: byte, char, and long.

If the return value of a method is an object, then you must provide the class of the object by coding the CLASS keyword on the prototype. The class name specified will be that of the object being returned. Use the EXTPROC keyword to specify the class of the method being called.

If the method being called is a static method, then you must be specify the STATIC keyword on the prototype.

In Java, the following data types can only be passed by value:
byte
int
short
long
float
double

Parameters of these types must have the VALUE keyword specified for them on the prototype.

If the method you are calling is a VARPG-generated method, meaning that "JAVARPG has been specified as the first parameter of the EXTPROC keyword, then these data types can be passed by reference and the VALUE keyword is not required.

Note that objects can only be passed by reference. The VALUE keyword cannot be specified with type O. Since arrays are seen by Java as objects, parameters mapping to arrays must also be passed by reference. This includes byte arrays.

Examples of Prototyping Java Methods
This section presents some examples of prototyping Java methods.

Example 1
The Java Integer class contains a static method called toString, which accepts an int parameter, and returns a String object. It is declared in Java as follows:

\[ \text{String Integer.toString(int)} \]

This method would be prototyped as follows:
The EXTPROC keyword identifies the method as a non VARPG-generated method. It also indicates that the method name is ‘toString’, and that it is found in class ‘java.lang.Integer’.

The O in column 40 and the CLASS keyword tell the compiler that the method returns an object, and the class of that object is ‘java.lang.String’.

The STATIC keyword indicates that the method is a static method, meaning that an Integer object is not required to call the method.

The data type of the parameter is specified as 10I, which maps to the Java int data type. Because the parameter is an int, it must be passed by value, and the VALUE keyword is required.

**Example 2**
The Java Integer class contains a static method called `getInteger`, which accepts `String` and `Integer` objects as parameters, and returns an `Integer` object. It is declared in Java as follows:

```java
Integer Integer.getInteger(String, Integer)
```

This method would be prototyped as follows:

```D
toint string PR 0 EXTPROC(*JAVA:
'D 'java.lang.Integer':
'D 'toString')
D CLASS(*JAVA:'java.lang.String')
D STATIC
D num 10I 0 VALUE
```

This method accepts two objects as parameters. O is coded in column 40 of the D-specification and the CLASS keyword specifies the class of each object parameter.

**Example 3**
The Java Integer class contains a method called `shortValue`, which returns the short representation of the Integer object used to invoke the method. It is declared in Java as follows:

```java
short shortValue()
```

This method would be prototyped as follows:

```D
tostring PR 0 EXTPROC(*JAVA:
'D 'java.lang.Integer':
'D 'toString')
D CLASS(*JAVA:'java.lang.String')
D STATIC
D num 10I 0 VALUE
```

The STATIC keyword is not specified because the method is not a static method. The method takes no parameters, so none are coded.

The returned value is specified as 5I, which maps to the Java short data type.
Example 4

The Java Integer class contains a method called *equals*, which accepts an Object as parameter and returns a boolean. It is declared in Java as follows:

```java
boolean equals(Object)
```

This method would be prototyped as follows:

```
D equals PR N EXTPROC(*JAVA:
D 'java.lang.Integer':
D 'equals')
D obj 0 CLASS(*JAVA:'java.lang.Object')
```

The returned value is specified as N, which maps to the Java boolean data type.

Creating Objects

In order to call a non-static method, an object is required. The class of the object must be the same as the class containing the method. Objects are instantiated, or created, by calling the class constructor. The class constructor is not a static method, but it does not require an object to call it. The special method name *CONSTRUCTOR* is used when prototyping a constructor.

For example, in order to construct a *BigDecimal* object from a float value, the constructor that expects a float parameter must be called as follows:

```java
BigDecimal(float) returns a new BigDecimal object
```

The constructor would be prototyped as follows:

```
D bdcreate PR 0 EXTPROC(*JAVA:
D 'java.math.BigDecimal':
D *CONSTRUCTOR)
D Dnum 4F VALUE
```

Note that the parameter must be passed by value because it maps to the Java float data type.

Calling Java Methods

Java methods can be called using existing operation codes CALLP (when no return value is expected) and EVAL (when a return value is expected). No new syntax is required.

When calling a static method, an object is not required in order to make the call. When calling a non-static method, an object is required. The object to be used must be coded as the first parameter in the call. This parameter is not specified on the prototype, but is implied for all methods that are not static. This means that whenever a method that is not static is called, a minimum of one parameter must be specified.

Example 1

In this example, the goal is to add two BigDecimal values together. In order to do this, two BigDecimal objects must be instantiated by calling the constructor for the BigDecimal class, fields must be declared to store the BigDecimal objects, and the add() method in the BigDecimal class must be called.
* Prototype the BigDecimal constructor that accepts a String parameter. It returns a new BigDecimal object.

D bdcreate1 PR 0 EXTPROC(*JAVA:
 'java.math.BigDecimal':
 *CONSTRUCTOR)
D str CLASS(*JAVA:'java.math.BigDecimal')

* Define fields to store the BigDecimal objects.

D bdnum1 S 0 CLASS(*JAVA:'java.math.BigDecimal')
D bdnum2 S 0 CLASS(*JAVA:'java.math.BigDecimal')

* Since one of the constructors we are using requires a String object, we will also need to construct one of those. Prototype the String constructor that accepts a byte array as a parameter. It returns a String object.

D makestring PR 0 EXTPROC(*JAVA:
 'java.lang.String':
 *CONSTRUCTOR)
D bytes CLASS(*JAVA:'java.lang.String')

* Define a field to store the String object.

D string S 0 CLASS(*JAVA:'java.lang.String')

* Prototype the BigDecimal add method. It accepts a BigDecimal object as a parameter, and returns a BigDecimal object (the sum of the parameter and of the BigDecimal object used to make the call).

D add PR 0 EXTPROC(*JAVA:
 'java.math.BigDecimal':
 'add')
D bd1 CLASS(*JAVA:'java.math.BigDecimal')

* Define a field to store the sum.

D sum S 0 CLASS(*JAVA:'java.math.BigDecimal')
D double S 8F INZ(1.1)
D fld1 S 10A

Here is the code that does the call.
C MOVEL 'mystring' fld1 10
C* Call the constructor for the String class, to create a String object from fld1. Since we are calling the constructor, we
do not need to pass a String object as the first parameter.

Call the BigDecimal constructor that accepts a String parameter, using the String object we just instantiated.

Call the BigDecimal constructor that accepts a double as a parameter.

Add the two BigDecimal objects together by calling the add method. The prototype indicates that add accepts one parameter, but since add is not a static method, we must also pass a BigDecimal object in order to make the call, and it must be passed as the first parameter. bdnum1 is the object we are using to make the call, and bdnum2 is the parameter.

sum now contains a BigDecimal object with the value bdnum1 + bdnum2.

Example 2

This example shows how to perform a TRIM in Java by using the trim() method as an alternative to the VARPG %TRIM built-in function. The trim() method in the String class is not a static method, so a String object is needed in order to call it.

* Define a field to store the String object we wish to trim

* Prototype the constructor for the String class. The constructor expects a byte array.

* Prototype the String method getBytes which converts a String to a byte array. We can then store this byte array in an alpha field.

* Prototype the String method trim. It doesn't take any parameters, but since it is not a static method, must be called using a String object.

*
The call is coded as follows:

```
C* C* Call the String constructor
C* EVAL str = makestring(fld)
C* C* Trim the string by calling the String trim() method.
C* We will reuse the str field to store the result.
C* EVAL str = trimstring(str)
C* C* Convert the string back to a byte array and store it
C* in fld.
C* EVAL fld = makealpha(str)
```

Static methods are called in the same way, except that an object is not required to make a call. If the makealpha() method above was static, the call would look like:
```
C EVAL fld = makealpha()
```

If the method does not return a value, use the CALLP operation code.

**Additional Considerations**

The compiler will not attempt to resolve classes at compile time. If a class cannot be located at run time, a runtime error will occur. It will indicate that an UnresolvedLinkException object was received from the Java environment.

The compiler does no type checking of parameters at compile time. If there is a conflict between the prototype and the method being called, an error will be received at run time.

It is very important that *JAVARPG be specified as the first parameter of EXTPROC if the method being called is a non VARPG-generated method. If this is not done, it is likely that one of the above two error situations will occur.
Chapter 19. Considerations When Compiling for Java

This section describes VARPG source restrictions, possible changes required in your VARPG source, and runtime behaviour differences when using the Java build option to create Java source.

Project File Name Convention

The project file name for a Java application must follow Java naming conventions. The first character must be alphabetic. If your project’s name is incorrect, you can use the Rename Project utility to rename it. (Select Rename project from the pop-up menu of the project’s icon.)

Conditional Compile Directives

Two conditional compiler directives are defined by the compiler to help maintain a single source file that can be used to create both Windows components and Java source code. These directives are:
- **COMPILE_WINDOWS** is defined by the compiler when a Windows build is requested.
- **COMPILE_JAVA** will be defined by the compiler when a Java build is requested.

Since the compiler defines these two names, it is not necessary to define them using the /DEFINE directive.

Java Source Code Restrictions

The following language elements are not supported when generating Java source code:

**Keywords:**
- ALIGN
- EXPROPTS
- STATIC on field definitions. STATIC is supported on Java method prototypes.

**Operation codes:**
- ALLOC
- CABxx
- CALLB
- DEALLOC
- DSPLY (only for NOMAIN and EXE; otherwise supported)
- GOTO
- REALLOC
- TAG

**Operation extenders:**
- M
- R

**Language Elements:**
- Embedded SQL

**Data types:**
• Pointer data type

File types:
• SPECIAL

File operations:
• Writing records by relative record number

Possible VARPG Source Changes

This section summarizes the changes that may be required to your VARPG source in order to generate the Java source code.

1. The to/from notation must be used when defining subfields of the PSDS and INFDS in order to allow the compiler to validate the subfield definitions. The definition of subfields in the INFDS and PSDS must match the definitions specified in the VARPG Language Reference. A compile time error will be issued if they do not.

2. There cannot be an unconditional LEAVE or ITER operation as anything other than the last operation in a loop, otherwise the Java compiler will issue an error. If there is an unconditional LEAVE or ITER operation in a loop, all operations occurring after it in the loop should be deleted, as they will never be executed.

3. When adding and subtracting date/time/timestamp durations, only values between maxint (2 147 483 647) and -maxint (-2 147 483 648) can be used.

4. Because Java does not allow int (10I), short (5I), float (4F), and double (8F) values to be passed by reference, Java code has to be generated by VARPG to retain this functionality for subprocedures being converted to Java. The code generated to accomplish this can cause java compiler errors when the VARPG source contains subprocedures with multiple return points and receives integer or float parameters passed by reference.

Sample code that may cause Java compile errors:

```c
C IF x = 1
C ...
C RETURN 1
C ELSE
C ...
C RETURN 0
C ENDIF
```

The preceding code should be changed to:

```c
C IF x = 1
C ...
C RETURN 1
C ELSE
C ...
C ENDIF
C RETURN 0
```

5. The characters ‘*’, ‘#’, and ‘@’ cannot be used in Java identifiers. Because of this, all occurrences of ‘*’, ‘#’, and ‘@’ in VARPG names will be changed to ‘_’. It is possible that this conversion will result in duplicate names.

6. If a COMMIT or ROLBK operation is coded within an application that has no files, a severity 30 message (RNF7833) will be issued.

7. Due to the way that a local *PSSR is converted to Java, it is not possible to call a local *PSSR. Also note that since GOTO is not supported, the only way to leave a local *PSSR and avoid the default handler is to code a RETURN operation.

8. There is no short circuiting of logical expressions. This means that the order in which a compound logical expression is executed cannot be relied upon.
9. Varying length fields are implemented as a class when converting to Java. This means that they are not stored as documented in the VARPG Language Reference. Code that depends on them being stored a certain way will not work.

10. Data structure subfields will not be initialized to blanks if there is no initial value provided, but will be initialized to a default value depending on the datatype of the subfield. The default value is 0 for numerics, blanks for character, and *LOVAL for date, time, and timestamp. Varying length fields will have their length set to 0.

11. The *HIVAL and *LOVAL values are not allowed for graphic and UCS-2 fields.

12. If a length is specified for a data structure, it must match the total length of the subfields it contains, otherwise the compiler will issue a severity 30 diagnostic message.

13. Subroutines cannot be defined within subprocedures. The only exception to this is that a *PSSR can be defined within a subprocedure. Any subroutines within subprocedures should be moved outside the subprocedure. If the subroutine accesses local fields within the subprocedure, then either the fields need to be changed to global fields, or the subroutine should be changed to a subroutine that accepts the local fields as parameters.

14. Unconditional LEAVE statements within DO loops are not supported. A Java compiler error will occur if this situation exists. Since an unconditional LEAVE within a DO loop means that the loop will only ever be executed once, the LEAVE should be removed and the code changed to remove the loop operation codes.

15. Using event attributes in fixed compound conditional statements currently causes Java compile errors. The equivalent free form expression should be used instead.

Sample code that may cause Java compile errors:

```c
C %mousex IF EQ x
C %mousey AND EQ y
C ...
C ENDIF
```

The preceding code should be changed to:

```c
C IF %mousex = x AND %mousey = y
C ...
C ENDIF
```

16. An unconditional RETURN operation cannot be coded unless it is the last statement in a user subroutine, action subroutine, or subprocedure. Otherwise, the Java compiler may report errors.

17. An unconditional LEAVESR operation cannot be coded unless it is the last statement in a user subroutine or action subroutine. Otherwise, the Java compiler may report errors.

18. SELECT statements can cause Java compile errors when they occur in subprocedures, contain RETURN operations, and no RETURN is coded within the main body of the subprocedure.

Sample code that may cause Java compile errors:

```c
C SELECT
C x WHEN EQ y
C RETURN 1
C x WHEN EQ z
C RETURN 2
C OTHER
C RETURN 0
C ENDSL
```

The preceding code should be changed to:
Runtime Differences

Because of differences between the Windows and Java environments, an application may run differently under Java than it does under Windows. The following areas are affected:

1. The %SCAN builtin function will return an integer result. In Windows, it returns an unsigned result.
2. The truncate numeric build option is unreliable and should not be depended upon.
3. When an I/O exception occurs, the user will not be given the option to retry the operation.
4. Data structures are not treated as one large character field when the Java application is running. This may cause unexpected results if they are used as such.
5. The format of binary, integer, and unsigned datatypes is handled differently for local files. When reading and writing local files, the Java format is assumed, which means that the high order bytes are leftmost, whereas when running as a Windows application, they are rightmost.
6. Exception handling for subprocedures will behave the same as for action subroutines. The default error handler will be called if there is no local *PSSR or INFSSR and no error indicator on the operation.
7. If an invalid date, time, or timestamp value is encountered when reading or writing a field to/from a file, the field will be set to the default value (*LOVAL). No error is reported.
8. Java can only handle a 3 digit millisecond portion in timestamps. When doing calculations with timestamps that use all 6 digits of the millisecond portion (meaning they do not have milliseconds in the form 000xxx), the results might not be as expected.
9. Intermediate results in expressions are not limited to 30 digits. In fact, when running in the VARPG environment, no attention is paid to the precision of intermediate results.
10. Memory cannot be shared between components. If a component is started by another component via START, changes made to passed parameters are not reflected across components.
11. Integer overflow or underflow will not be reported. Float overflow or underflow will be reported as status 9999.
12. If an error occurs while a subprocedure in a NOMAIN or EXE application is being executed, and there is no error indicator or *PSSR, then the error will be reported back to the caller and handled by the caller. When running under Windows, the application would terminate.
13. A status 50 error will never be issued when running Java applications. Java gives no diagnostic messages for character conversions it cannot handle. Java may issue a status 100 for an unsuccessful conversion or an ArrayIndexOutOfBoundsException when the converted string is used.
14. Positioning a host file to Null-Valued Records when ALWNULL(*NO) has been specified results in CPF5035.
Applet Restrictions

The following language elements are not supported when running a VARPG applet and will result in Java errors at run time:
- Printer files
- Local files
- Calling C functions, external subprocedures, EXEs.
- NOMAIN and EXE applications cannot be run as applets.

J2SDK 1.2 Printing Problems

The Java 2 Software Development Kit (J2SDK), version 1.2 or higher, is currently experiencing problems when text is sent to a printer device. One workaround for this problem is to run the Java application as follows:

```
java -Djava2d.font.usePlatformFont=true -ms32m -mx32m <classname>
```

However, the printed text may not appear as expected. This problem will be resolved when the existing problem with J2SDK 1.2 is fixed, without requiring a VARPG update.
Chapter 20. Creating and Running VisualAge RPG Applets

Once you have created a visual interface and the associated VARPG logic on your workstation, you can build and deploy your application as a Java applet that runs in any Web browser with an appropriate Java Virtual Machine (JVM). This gives you the extra flexibility of making your application widely available over the internet. Many users with browser access to your Web site can run the applet inside their browser and also communicate with data on the iSeries server.

This section describes how to build and deploy such VARPG applets.

Creating Applets

Note: In order to build the Java version of a project, the Java 2 Software Development Kit (J2SDK), version 1.2 or greater, must be installed on the workstation. The J2SDK is available from Sun Microsystems at URL http://java.sun.com/products/

To run applets, the international version of the Java 2 Run Time Environment (J2RE) must be installed.

Applets are Java applications that run inside the context of a Web page. When a Web page containing an applet is loaded, the applet’s code is downloaded from the HTTP server to the workstation and the Java applet is started. Typically, the applet is embedded in the main Web page and runs when the Web page is displayed in a browser. The applet can also be displayed in a separate window.

There are no special design or development steps required to create VARPG applets. Designing and coding your VARPG project is the same for applets as for Windows applications. However, you may want to consider writing thin clients when targeting as applets to avoid long download times. (See Appendix B, “Writing Thin Client Applications,” on page 431 for more information.)

There are security restrictions for applets that you should be aware of. These security restrictions are not VisualAge RPG restrictions, but are part of the Java language run time specification for applets. Applets cannot:

• Access local resources on the client, such as, the file system and printers
• Open a socket connection to a different host from the one where the applet resides. This means that you cannot load an applet from one iSeries 400 server and access files that are on a different server.

You can, however, set up a policy file to relax some of these security restrictions. See “Using the Security File for Applets” on page 210 and the Java documentation for more information on setting up policy files.

Use the same steps to design and code your applet as you would to design and code a typical Windows application. However, be aware of the restrictions that apply when coding for the Java environment. Once you complete your applet’s visual interface and the associated VARPG logic, build the application.
You can control your applet’s build options on the Java: Build options notebook. Select Project>Build Options>Java from the VisualAge RPG design window to display the Java Build Options for your project:

The majority of the settings are similar to those used for building a Windows application, with the following exceptions:

**SSL**
Select SSL if you want all TCP/IP connections between your iSeries server and Java applet or application to be encrypted using Secure Sockets Layer technology. (See Appendix D, “Secure Sockets Layer (SSL) Setup,” on page 449 for information on SSL setup.)

**Java Compiler**
VisualAge RPG generates a Java source file (.java) from your project and relies on an external Java compiler to create the class file (.class) from the source. If you are not using the IBM or Sun Microsystem’s Java 2 SDK, then you will need to specify the Java compiler you are using here.

**Options**
Pass any command line options you want to the Java compiler.

**Applet - Embedded**
Determines if your applet is run when the HTML page it is embedded in is displayed in a Web browser, or if your applet starts in an external window.

**Applet - Directory for HTML and JAR files**
You can specify a directory where you want all of the required runtime files for your applet to be placed. These files are generated when you build your project for Java. By default, these files are placed in the project’s source directory on your workstation.

**Hint:** Map a network drive to your iSeries server and enter the IFS directory where you would like to deploy the applet from.

Now that you have configured the options for your applet, you are ready to build the project. From the project’s design window, select Project>Build>Java. Upon a
successful build, the runtime files for the applet are created in the project’s source
directory or the directory specified in the Directory for HTML and JAR files Java
build option.

For example, if you build the Listbox sample to create a Java application and
specify c:\applet\Listbox as the directory to contain the applet’s runtime files,
you should see the following files in this directory:

listbox.htm
listbox_applet.htm
LISTBOX.jar
vapplet.jar

These files are used to deploy your applet from the Web server, as follows:

listbox.htm
   Launches your applet using the Java Plug-in.

listbox_applet.htm
   Checks the user’s workstation for the required VisualAge RPG Java
   runtime (varpg.jar) file. If the workstation has the correct version of the
   run time, then the browser opens the listbox.htm page. Otherwise, the user
   is prompted to download and install the correct runtime file.

LISTBOX.jar
   Contains the LISTBOX.class, LISTBOXApplet.class, LISTBOX.ODX,
   LISTBOX.RST and any *Resources.properties (if you defined messages for
   the application) files used by your project.

vapplet.jar
   Contains a small subset of the VisualAge RPG Java run time that is
   required on the Web server.

---

Testing Your Applet

This section describes the set up required for testing your VisualAge RPG applet.

1. Install the IBM or Sun Microsystem’s Java 2 Runtime Environment (J2RE).
   VisualAge RPG generated applets require the international version of the Java
   2 Run Time Environment (J2RE) to execute properly. When you install the IBM
   or Sun Microsystem's Java 2 SDK (or JRE), the Java Plug-in is automatically
   installed. If you are running on your development machine, then the J2SDK
   you installed for developing the applet is sufficient.

2. Add the VisualAge RPG Java run time (varpg.jar) file to your JRE’s extension
directory.
   The VisualAge RPG Java run time file should be copied to each client machine
   and added to the local JRE’s extension directory. Typically this is a subdirectory
   named jre\lib\ext\. This avoids having to download the run time from the
   HTTP server every time an applet is run from the Web page. The varpg.jar file
   is located in the JAVA subdirectory under the WDSC install directory. For
   example c:\wdsc\java\varpg.jar.

3. Copy your applet’s runtime files to the iSeries IFS directory where you will be
   serving the applet from. In the Listbox sample, these files would be:

listbox.htm
listbox_applet.htm
LISTBOX.jar
vapplet.jar

Chapter 20. Creating and Running VisualAge RPG Applets  295
Hint: Map a network drive to your iSeries server and enter the IFS directory in the Directory for HTML and JAR files Java build option before you create your applet.

4. Set up the iSeries HTTP server to allow access to the directory containing your applet.

You will need to start and configure the IBM HTTP Server if you have not already done so. See HTTP Server for iSeries Webmaster’s Guide for information on configuring the HTTP server.

Add a PASS statement to your iSeries HTTP configuration file that allows access to the IFS directory where you placed the applet runtime files. For this example the applet files are in the IFS directory /applets. So, add the following PASS statement:

```
Pass /applets/* /applets/*
```

5. Run the applet from your Web browser. For example, open your Web browser with the following URL:

```
http://Toras14m:999/Listbox.htm
```

where Toras14m is your iSeries server name, 999 is your HTTP port number, and Listbox.htm is the Web page containing your applet.

Here is the Listbox applet running inside the Windows browser:

![Listbox Applet Running inside Internet Explorer](image)

**Troubleshooting**

The following is a list of common configuration problems that can cause applets not to run:

- The proper J2RE is not installed. Ensure that the Java 2 SDK or international version of the J2RE is installed along with the Java Plug-in.
• The applet’s required runtime files are not in the correct directory on the server. The files AppName.htm, AppName_applet.htm, vapplet.jar, and APPNAME.jar need to be in the directory referenced in the second parameter to the HTTP server’s Pass statement.

• Java file names are case sensitive. This causes the majority of configuration issues. Make sure that all .jar files are in the correct case. Windows Explorer does not always show file names in their actual case. Use the OS/400 wrklnk command to check the case of the file names stored in the IFS.

If your applet is still not running, try enabling the Java Plug-in console to see if any error messages are being displayed. Start the Plug-in control pannel by selecting Start>Programs>Java Plug-in Control Panel. From the basic tab, select Show Java Console and click Apply. Close all open Web browser windows and restart the Web browser for the changes to take affect. The next time you run the applet, the Java Console window appears with messages.

Running One Applet from Another

You can run one VisualAge RPG-generated applet from another, by modifying the calling applet’s main .htm file to include the called applet. For example, to start AppletB from AppletA, modify the AppletA.htm file as follows:

• Find the lines that include APPLETA.jar and varpg.jar. Typically, the following lines:

```xml
<PARAM NAME = "archive" VALUE = "APPLETA.jar , varpg.jar">
```

... archive = "APPLETA.jar , varpg.jar"

• Insert, APPLETB.jar between the varpg.jar and closing quotes so the VALUE and archive values are now:

```xml
<PARAM NAME = "archive" VALUE = "APPLETA.jar , varpg.jar , APPLETB.jar">
```

... archive = "APPLETA.jar , varpg.jar , APPLETB.jar"

Be sure to include a blank character before and after the comma delimiter.

When you display the AppletA.htm page in your Web browser, it should now run AppletB, too.

If the called applet (AppletB, in this example), also accesses data on the server, you need to give each applet permission to read the security file that resides on the workstation. Otherwise, the user will be prompted to enter a valid user Id and password each time that AppletB is run. See "Using the Security File for Applets" on page 210 for more information.

In this example, you need to modify the local policy file by adding the following lines:

```java
permission java.lang.RuntimePermission "modifyThreadGroup";
permission java.lang.RuntimePermission "modifyThread";
```

The policy file (java.policy) is located in the Java run time lib\security subdirectory.

For socket permission, you may need to add a line like the following:

```java
permission java.net.SocketPermission
"server_name:port_number", "connect,resolve";
```
If you need to change the name or location of this policy file, modify the *java.security* file in the same directory.
Chapter 21. Calling System Functions when Compiling for Java

The VisualAge RPG compiler includes support for calling external procedures implemented as function entry points in Dynamic Link Libraries (DLLs) on the Windows platform through the Java Native Interface (JNI). This section discusses how to use this support.

Refer to the Java Native Interface (JNI) section of the Java 2 Software Development Kit (J2SDK) documentation as prerequisite reading.

A Simple Call

The first code example shows a simple call to an external procedure with no parameters and no return value. The simple VisualAge RPG application calls an external procedure in a sample dynamic link library. The JNI specification dictates the function name and interface to the native function being called. A function will be coded and compiled into a new DLL to be the target of the call. The following samples will only demonstrate native functions coded in the C language, but the illustrated coding principles apply equally to other language implementations. Once the native function has control, it is free to call other native functions, such as system APIs.

Code a procedure prototype for the procedure in VisualAge RPG, specifying the DLL keyword to provide the name of the DLL which will contain the native function being called. The EXTPROC keyword may optionally be coded to specify a function name different from the procedure name within the VisualAge RPG program.

Note: The value of the EXTPROC keyword is case sensitive. Code a call to the procedure in the VisualAge RPG source.
The native function is coded to use the StdCall program linkage. The function is coded as an exported function in the DLL. The exported function name must match the name dictated by the JNI specification. The format of the JNI Specification is:

Java_VARPGComponentName_ExternalProcedureName_OverloadedNativeMethods

The full native function names are 'Java_VCOMP1_proc1' and 'Java_VCOMP1_SUB2' in this sample. The overloaded native method are not needed for this sample.

The JNI interface dictates the first two parameters: an interface pointer, and a this object pointer. Additional parameters correspond to the procedures declared parameters. The jni.h header file is included in the C language source program to provide interface definitions. This file is provided with the J2SDK. You may need to update your INCLUDE environment variable in your C compiler to include the directory containing the C language header files for J2SDK.

Lastly, the sample C source file is compiled into a DLL.
Passing and Receiving Parameters

The JNI specification passes Java primitive data types directly, but VisualAge RPG processes all the VARPG data types through classes. This means that VARPG calls to native functions will always involve passing objects. The JNI specification provides interface functions for the native function to access the values of passed objects. Due to the different class methods for each VARPG data type, each type will be discussed individually.

Parameter Types

Character

VisualAge RPG implements character fields as byte arrays in Java including a character field of length one. The JNI interface function GetByteArrayElements returns the value of the byte array parameter. The value can be changed and returned to the calling function through the ReleaseByteArrayElements interface function.

Note: The value should not be used in the native function after calling the release function.

The first parameter for the native function in the C language source has been changed from a void pointer to a JNIEnv pointer. It points to a table of function pointers for JNI interface functions. The prototyped parameters for the external
procedure are added to the native function’s parameters, after the two standard JNI pointers. The character parameters are declared as jbyteArray types in the native function.

The GetByteArrayElements interface function is used to obtain the value of the Java byte array for the VARPG character field.

The obtained value can be changed and returned to the Java caller with the ReleaseByteArrayElements interface function.

The obtained value should not be accessed after releasing it.

**Note:** It is possible the value obtained is the actual value in the Java object, and not a copy in memory. The changes made to it might be reflected in the Java caller even without calling the release function. Refer to the GetByteArrayElements function in the JNI documentation for more information.
**Source File: VCOMPI.VPG**

* Demonstrate calling an external procedure thru JNI.

* This declares a procedure named 'sub1c' which refers to
  a function named 'proc1c' in a dynamic Load Library 'VSUBC.DLL'

* With 1 character parameter
  d sub1c pr dll('VSUBC') extproc('proc1c')
  d 1

* Without the EXTPROC keyword
* With 2 character parameters
  d sub2c pr dll('VSUBC')
  d 4
d 10
d c1 s 1 inz('J')
d c4 s 4 inz('blue')
d c10 s 10 inz('abcdefghij')

d mb1 m style(*info) button(*OK)

d rc s 9 0

C  *INZSR  BEGSR

C    callp  sub1c(c1)
C    callp  sub2c(c4:c10)

* Display the changed values from the calls
  c  c4  dsply  mb1  rc
  c  c10  dsply  mb1  rc

C    seton
C  ENDSR

* This action subroutine is linked to a Create event for the Window.
* It causes the component to end after running the INZSR.

C  CREATE1  BEGACT
C    seton  LR
C  ENDACT

Figure 78. Sample file VCOMPC.VPG
// Source File: VSUBC.C
// Native function with Character parameters

// Compiled with: IBM VisualAge(TM) for C++ for Windows(R), Version 3.5
// Compile command: icc /q /ss /ge- /fe vsubc.dll vsubc.c

#include <stdio.h>
#include <string.h>
#include <jni.h>

//---------------------------------------------------------------

void _Export __stdcall Java_VCOMPC_proc1c(JNIEnv *je, void *jc, jbyteArray p1)
{
    char *c1;
    printf("proc1c called successfully.\n");
    c1 = (char *) (*je)->GetByteArrayElements(je, p1, NULL);
    printf("c1 = \"%c\"\n", c1[0]);
} //-------------------------------------------------------------

void _Export __stdcall Java_VCOMPC_SUB2C(JNIEnv *je, void *jc, jbyteArray p1, jbyteArray p2)
{
    char *c4;
    char *c10;
    printf("SUB2C called successfully.\n");
    c4 = (char *) (*je)->GetByteArrayElements(je, p1, NULL);
    c10 = (char *) (*je)->GetByteArrayElements(je, p2, NULL);
    printf("c4 = %.4s\", c4);
    printf("c10 = %.10s\", c10);
}

Figure 79. Sample file VSUBC.C (Part 1 of 2)

// Now change the values
memcpy(c4,"Gray",4);
memcpy(c10,">Received<",10);

// Update the values back to the Java Caller

// Fourth Parameter = 0 also causes the variable's storage to be freed,
// so can not access the variables after this function call.

(*je)->ReleaseByteArrayElements(je, p1, (signed char *) c4, 0);
(*je)->ReleaseByteArrayElements(je, p2, (signed char *) c10, 0);

Figure 79. Sample file VSUBC.C (Part 2 of 2)
Zoned Numeric

* Source File: VCOMPN.VPG
* Demonstrate calling an external procedure thru JNI.

* With a Zoned(4,0) parameter
  d subz pr dll('VSUBN')
  d 4S 0

* With a Packed(9,2) parameter
  d subp pr dll('VSUBN')
  d 9P 2

* With Binary(4,0), Binary(9,0) parameter
  d subb pr dll('VSUBN')
  d 4B 0
  d 9B 0
  d z4 s 4S 0 inz(1234)
  d p92 s 9P 2 inz(1234567.89)
  d b4 s 4B 0 inz(1234)
  d b9 s 9B 0 inz(123456789)

Figure 80. Sample File VCOMPN.VPG (Part 1 of 2)

* This action subroutine is linked to a Create event for the Window.
* It causes the component to end after running the INZSR.

Figure 80. Sample File VCOMPN.VPG (Part 2 of 2)
// Source File: VSUBN.C

// Native function with Character parameters

// Add (d:\jdk12\include;d:\jdk12\include\win32) to the INCLUDE setting
// in order to find jni.h when compiling.

// Compiled with: IBM VisualAge(TM) for C++ for Windows(R), Version 3.5
// Compile command: icc /q /ss /ge -fe vsubn.dll vsubn.c

#include <stdio.h>
#include <string.h>
#include <jni.h>

static void SwapBin2( short *b2);
static void SwapBin4( int *b4);

//---------------------------------------------------------------
void _Export __stdcall Java_VCOMPN_SUBZ( JNIEnv *je, void *jc, jobject p1)
{
    jclass cls;
    jmethodID mid;
    jobject aryobj;
    char *zd;

    printf(" SUBZ called successfully.\n");
    // p1: Zoned
    // Call the method to get the zoned value
    cls = (*je)->GetObjectClass(je, p1);

    mid = (*je)->GetMethodID( je, cls, "zonedValue", "()[B";
    if ( mid == NULL)
    {
        printf(" ERROR:GetMethod.\n");
        return;
    }

    aryobj = (*je)->CallObjectMethod( je, p1, mid);
    zd = (char *) (*je)->GetByteArrayElements( je, aryobj, NULL);
    printf(" zd = %.4s.\n", zd);

    Figure 81. Sample File VSUBN.C (Part 1 of 2)
// Now change the values
memcpy( zd, "9876", 4);

// Returning the Zoned parameter

// 1. Update the Byte array object with the changed value.

(*je)->ReleaseByteArrayElements( je, aryobj, (signed char *) zd, 0);

// 2. Prepare to call the method from the RpgNumeric class which
// takes a byte array object and assigns it's value into the
// RpgNumeric object. Obtain the method ID.

// cls = (*je)->GetObjectClass(je, p1);
// (cls) still identifies the second parameter. Re-use value

mid = (*je)->GetMethodID( je, cls, "assignFromNative", "([BII)V");

if ( mid == NULL)
{
    printf(" ERROR 2: GetMethod.\n");
    return;
}

(*je)->CallVoidMethod( je, p1, mid,
    aryobj,
    (int) 1,   // = Component.ZONED_TYPE
    0,        // precision
    );

---

**Figure 81. Sample File VSUBN.C (Part 2 of 2)**

An RpgZoned object is passed as the parameter. The RpgZoned::zonedValue
method is used to obtain a byte array containing the numeric value of the object in
zoned decimal format. Once the byte array is obtained on the Java side of the
interface, the GetByteArrayElements JNI interface function is called to access the
byte array on the native side of the interface.

To invoke the zonedValue method on the object, the GetObjectClass, GetMethodID,
and CallObjectMethod interface functions are used. Care must be taken to specify
the correct method signature on the GetMethodID call. (No parameters, and
returning a byte array in this case.)

To alter the value on the Java side, the byte array value is changed, the
ReleaseByteArrayElements interface function is invoked to set the byte array
change on the Java side, and the appropriate RpgZoned class method is invoked
on the Java side to set the RpgZoned object’s value with the value represented in
the byte array. In this case, it’s the assignFromNative method which takes a byte
array and two integers as parameters. (The RpgNumeric class referred to in the
sample code is a parent class to the RpgZoned class.)
Packed Numeric

```c
void _Export __stdcall Java_VCOMPN_SUBP( JNIEnv *je , void *jc,
            jobject p1) // P(9,2)
{
    jclass cls;
    jmethodID mid;
    jobject aryobj;
    char *packednum;

    printf(" SUBP called successfully.\n");

    // p1: Packed 9,2
    // Call the method to get the zoned value
    cls = (*je)->GetObjectClass(je, p1);
    mid = (*je)->GetMethodID( je, cls, "packedValue", "()[B");

    if ( mid == NULL)
    {
        printf(" ERROR: GetMethod.\n");
        return;
    }

    aryobj = (*je)->CallObjectMethod( je, p1, mid);
    packednum = (char *) (*je)->GetByteArrayElements( je, aryobj, NULL);

    // Now change the values
    memcpy( packednum, "\x98\x76\x54\x32\x1C", 5);
}
```

Figure 82. Sample File VSUBN.C continued (Part 1 of 2)
// Returning the Packed parameter

// 1. Update the Byte array object with the changed value.

// Fourth Parameter = 0 also causes the variable's storage to be freed,
// so can not access the variables after this function call.

(*je)->ReleaseByteArrayElements( je, aryobj, (signed char *) packednum, 0);

// 2. Prepare to call the method from the RpgNumeric class which
// takes a byte array object and assigns it's value into the
// RpgNumeric object. Obtain the method ID.

// cls = (*je)->GetObjectClass(je, p1);
// (cls) still identifies the second parameter. Re-use value

mid = (*je)->GetMethodID( je, cls, "assignFromNative", "([BII)V");

if ( mid == NULL)
{
    printf(" ERROR 2: GetMethod.\n");
    return;
}

(*je)->CallVoidMethod( je, p1, mid, aryobj,
                        (int) 2,       // = Component.PACKED_TYPE
                        2                // precision (Number of decimal places)
                     );

Figure 82. Sample File VSUBN.C continued (Part 2 of 2)

The packed decimal parameter case is similar to the zoned decimal. Only the
appropriate methods for converting an RpgPacked object to and from a byte array
value are used.

The RpgPacked::packedValue method is used to obtain a byte array containing the
numeric value of the RpgPacked parameter object in native packed decimal format,
and then the JNI interface functions are invoked to make the Java byte array
accessible from the native side of the interface.

After altering the byte array on the native side and invoking the
ReleaseByteArrayElements interface function to return the byte array to the Java
side, the assignFromNative method is once again invoked to set the RpgPacked
object's value from the Java byte array.
Binary

void _Export __stdcall Java_VCOMPN_SUBB(JNIEnv *je, void *jc,
    jobject p1 // B(4,0)
    ,jobject p2) // B(9,0)
{

    jclass cls;
    jmethodID mid;
    jobject aryobj;
    jobject aryobj2;
    char *binarynum;
    char *b9;
    short binary2;
    int binary4;
    printf("SUBB called successfully.\n");

    // p1: Binary 4,0
    // Call the method to get the binary value
    cls = (*je)->GetObjectClass(je, p1);
    mid = (*je)->GetMethodID(je, cls, "binaryValue", "()[B");
    if ( mid == NULL )
    {
        printf("ERROR: GetMethod.\n");
        return;
    }
    aryobj = (*je)->CallObjectMethod(je, p1, mid);
    binarynum = (char *) (*je)->GetByteArrayElements(je, aryobj, NULL);
    // Must reverse the byte order of the value received
    memcpy( &binary2, binarynum, 2);
    SwapBin2( &binary2);

Figure 83. Sample File VSUBN.C continued (Part 1 of 5)
printf(" binary = \%hd\n", (short ) binary2);

// p2: Binary 9,0
// Call the method to get the binary value

cls = (*je)->GetObjectClass(je, p2);

mid = (*je)->GetMethodID( je, cls, "binaryValue", "()[B");

if ( mid == NULL)
{
    printf(" ERROR: GetMethod.\n");
    return;
}

aryobj2 = (*je)->CallObjectMethod( je, p2, mid);

b9 = (char *) (*je)->GetByteArrayElements( je, aryobj2, NULL);

// Must reverse the byte order of the value received

memcpy( &binary4, b9, 4);
SwapBin4( &binary4);

printf(" binary = \%d\n", (int ) binary4 );

Figure 83. Sample File VSUBN.C continued (Part 2 of 5)
// Now change the values
binary2 = 5;

// Swap it back for returning to the Java value
SwapBin2( &binary2);
memcpy( binarynum, &binary2, 2);

// Returning the parameter

// 1. Update the Byte array object with the changed value.

// Fourth Parameter = 0 also causes the variable's storage to be freed,
// so can not access the variables after this function call.
(*je)->ReleaseByteArrayElements( je, aryobj, (signed char *) binarynum, 0);

// 2. Prepare to call the method from the RpgNumeric class which
// takes a byte array object and assigns it's value into the
// RpgNumeric object. Obtain the method ID.
c1s = (*je)->GetObjectClass(je, pl);

// (cls) still identifies the second parameter. Re-use value
mid = (*je)->GetMethodID( je, cls, "assignFromNative", "([II)V");

if ( mid == NULL)
{
    printf(" ERROR 2: GetMethod.\n");
    return;
}

Figure 83. Sample File VSUBN.C continued (Part 3 of 5)
(*je)->CallVoidMethod( je, p1, mid,
    aryobj,
    (int) 3,  // = Component.BINARY_TYPE
    0       // precision (Number of decimal places)
);  

// Now change the values
binary4 = 19981999;

// Swap it back for returning to the Java value
SwapBin4( &binary4);
memcpy( b9, &binary4, 4);

// Returning the parameter
// 1. Update the Byte array object with the changed value.
// Fourth Parameter = 0 also causes the variable's storage to be freed,
// so can not access the variables after this function call.
(*je)->ReleaseByteArrayElements( je, aryobj2, (signed char *) b9, 0);

// 2. Prepare to call the method from the RpgNumeric class which
//    takes a byte array object and assigns it's value into the
//    RpgNumeric object. Obtain the method ID.
cls = (*je)->GetObjectClass(je, p2);
// (cls) still identifies the second parameter. Re-use value
mid = (*je)->GetMethodID( je, cls, "assignFromNative", "([BI]V");
if ( mid == NULL)
{
    printf(" ERROR 2: GetMethod.\n");
    return;
}

Figure 83. Sample File VSUBN.C continued (Part 4 of 5)
This case is similar to the zoned decimal. Only the appropriate methods for RpgBinary objects are used. The only added complication is that the native Intel architecture platform stores binary integers in a low-order-bytes-leftmost format, but the Java side works with them in a low-order-bytes-rightmost format. The SwapBin2 and SwapBin4 functions are employed to reverse the byte order when converting between the two sides for two- and four-byte binary integers.

The RpgBinary::binaryValue method is used to obtain a byte array containing the numeric value of the RpgBinary parameter object in native binary format. Then the JNI interface functions are invoked to make the Java byte array accessible from the native side of the interface. After altering the byte array on the native side and invoking the ReleaseByteArrayElements interface function to return the byte array to the Java side, the assignFromNative method is once again invoked to set the RpgBinary object’s value from the Java byte array.
Integer, Unsigned

* With Parameters: Integer, unsigned
  
  d subiu     pr       d11('VSUBO')
  d           5i 0     
  d           10i 0    
  d           5u 0     
  d           10u 0    

Figure 84. Sample VJNIO.VPG

static void SwapBin2(char *);
static void SwapBin4(char *);

void _Export __stdcall Java_VJNIO_SUBIU(JNIEnv *je, void *jc,
                                     jobject pl, jobject p2, jobject p3, jobject p4)
{
  jclass cls, cls2;
  jmethodID mid;
  jshort i2;
  jint i4;
  jobject aryobj3, aryobj4;
  unsigned short *u2;
  unsigned int  *u4;

  printf(" SUBIU called successfully.\n");

  // p1: Integer, 2 byte
  // Call the method to get the value

  cls = (*je)->GetObjectClass(je, pl);
  mid = (*je)->GetMethodID( je, cls, "getValue", "()S");
  if (mid == NULL)
  {
    printf(" ERROR: GetMethod.\n");
    return;
  }

  i2 = (*je)->CallShortMethod( je, p1, mid);
  printf(" i2 = %hd\n", (short) i2);

  // p2: Integer, 4 byte
  // Call the method to get the value

  cls = (*je)->GetObjectClass(je, p2);
  mid = (*je)->GetMethodID( je, cls, "getValue", "()I");

  // p2: Integer, 4 byte
  // Call the method to get the value

  cls = (*je)->GetObjectClass(je, p2);
  mid = (*je)->GetMethodID( je, cls, "getValue", "()I");

Figure 85. Sample VSUBO.C (Part 1 of 6)
if ( mid == NULL)
{
    printf(" ERROR: GetMethod.\n");
    return;
}

I4 = (*je)->CallIntMethod( je, p2, mid);
printf(" I4 = %d\n", (short) I4);

// Call the method to get the double value
cls = (*je)->GetObjectClass( je, p3);
mid = (*je)->GetMethodID( je, cls, "unsignedValue", "()[B");
if ( mid == NULL)
{
    printf(" ERROR: GetMethod.\n");
    return;
}
aryobj3 = (*je)->CallObjectMethod( je, p3, mid);
u2 = (unsigned short *) (*je)->GetByteArrayElements( je, aryobj3, NULL);
// Must reverse the byte order of the value received
SwapBin2( (char *) u2);
printf(" u2 = %hu\n", *u2 );

Figure 85. Sample VSUBO.C (Part 2 of 6)
// p4: Unsigned 4-byte.  
// Call the method to get the double value

c1s = (*je)->GetObjectClass(je, p4);
mid = (*je)->GetMethodID( je, cls, "unsignedValue", "()[B");

if ( mid == NULL)
    {
        printf(" ERROR: GetMethod.\n");
        return;
    }
aryobj4 = (*je)->CallObjectMethod( je, p4, mid);
u4 = (unsigned int *) (*je)->GetByteArrayElements( je, aryobj4, NULL);
// Must reverse the byte order of the value received
SwapBin4( (char *) u4);
printf(" u4 = %u\n", *u4);

// Now change the values
i2 = 99;
i4 = 88;
*u2 = 77;
*u4 = 66;

// Must reverse the byte order of the value being returned
SwapBin2( (char *) u2);
SwapBin4( (char *) u4);

// Return the array memory to Java. Used later to set return
// values for parameters
(*je)->ReleaseByteArrayElements( je, p3, (signed char *) u2, 0);
(*je)->ReleaseByteArrayElements( je, p4, (signed char *) u4, 0);

Figure 85. Sample VSUBO.C (Part 3 of 6)
// Returning P1: Integer 2-byte
// Invoke the RpgShortRef::setValue method to set the object
// value with a short parameter value
// Obtain the method ID so it can be invoked.
c1s = (*je)->GetObjectClass(je, p1);
mid = (*je)->GetMethodID( je, c1s, "setValue", "(S)V");
if ( mid == NULL)
{
    printf(" ERROR 5: GetMethod.\n");
    return;
}
(*je)->CallVoidMethod( je, p1, mid, i2);

// Returning P2: Integer 4-byte
// Invoke the RpgIntRef::setValue method to set the object
// value with an integer parameter value
// Obtain the method ID so it can be invoked.
c1s = (*je)->GetObjectClass(je, p2);
mid = (*je)->GetMethodID( je, c1s, "setValue", "(I)V");
if ( mid == NULL)
{
    printf(" ERROR 6: GetMethod.\n");
    return;
}
(*je)->CallVoidMethod( je, p2, mid, i4);

Figure 85. Sample VSUBO.C (Part 4 of 6)
// Returning P3: Unsigned 2-byte
// Invoke the RpgNumeric::assignFromNative method to set the object
// value with an unsigned parameter value
// Obtain the method ID so it can be invoked.
cls = (*je)->GetObjectClass(je, p3);
mid = (*je)->GetMethodID(je, cls, "assignFromNative", "([BI)V");
if (mid == NULL)
{
    printf("ERROR 7: GetMethod.\n");
    return;
}
(*je)->CallVoidMethod(je, p3, mid,
    aryobj3,
    (int) 5, // Component.UNSIGNED_TYPE
    (int) 0); // 0 decimal places
}

// Returning P4: Unsigned 4-byte
// Invoke the RpgNumeric::assignFromNative method to set the object
// value with an unsigned parameter value
// Obtain the method ID so it can be invoked.
cls = (*je)->GetObjectClass(je, p4);
mid = (*je)->GetMethodID(je, cls, "assignFromNative", "([BI)V");
if (mid == NULL)
{
    printf("ERROR 8: GetMethod.\n");
    return;
}
(*je)->CallVoidMethod(je, p4, mid, aryobj4,
    (int) 5, // Component.UNSIGNED_TYPE
    (int) 0); // 0 decimal places
}

Figure 85. Sample VSUBO.C (Part 5 of 6)
Two-byte integers use the `RpgShortRef::getValue` and `setValue` methods to access their values into short values on the native side. Similarly, four-byte integers use `RpgIntRef::getValue` and `setValue` methods to pass between native-side `int` values.

Unsigned parameters are complicated by the lack of a Java primitive matching an unsigned value. The unsigned object value is accessed through byte array primitives. The parameter access invokes the method to get the byte array representing the unsigned value then invokes the `GetByteArrayElements` interface function to access the array elements on the native side. Furthermore, on a native Intel/Windows platform, the byte value must first be byte-reversed to change it into the low-order-bytes-leftmost format. Returning the parameter follows a reverse process.

**Float (4/8)**

```java
static void SwapBin2(char *p)
{
    char tmp;
    tmp = p[0];
    p[0] = p[1];
    p[1] = tmp;
}

static void SwapBin4(char *p)
{
    char tmp;
    tmp = p[0];
    p[0] = p[3];

    tmp = p[1];
    p[1] = p[2];
}
```

*With Parameters: Float 4, Float 8.*

d subf pr d11('VSUBO')
d 4f
d 8f

Figure 85. Sample VSUBO.C (Part 6 of 6)

Two-byte integers use the `RpgShortRef::getValue` and `setValue` methods to access their values into short values on the native side. Similarly, four-byte integers use `RpgIntRef::getValue` and `setValue` methods to pass between native-side `int` values.

Figure 86. Sample VJNI0.VPG
void _Export __stdcall Java_VJNIO_SUBF ( JNIEnv *je , void *jc ,
   jobject p1 , jobject p2 )
{
    jclass cls , cls2 ;
    jmethodID mid ;
    jfloat f4 ;
    jdouble f8 ;

    // p1: Float
    // Call the method to get the float value

    cls = (*je)->GetObjectClass ( je , p1 ) ;
    mid = (*je)->GetMethodID ( je , cls , "getValue" , "()F" ) ;
    if ( mid == NULL )
    {
      printf("ERROR: GetMethod.\n") ;
      return ;
    }
    f4 = (*je)->CallFloatMethod ( je , p1 , mid ) ;
    printf("f4 = \%f\n", (float) f4 ) ;

    // p2: Double
    // Call the method to get the double value

    cls2 = (*je)->GetObjectClass ( je , p2 ) ;
    mid = (*je)->GetMethodID ( je , cls2 , "getValue" , "()D" ) ;
    if ( mid == NULL )
    {
      printf("ERROR: GetMethod.\n") ;
      return ;
    }
    f8 = (*je)->CallDoubleMethod ( je , p2 , mid ) ;
    printf("f8 = \%lf\n", (double) f8 ) ;

  
Figure 87. Sample VSUBO.C (Part 1 of 2)
// Now change the values
f4 = 999.888;
f8 = 98789.65456;

// Returning the Float parameter
// Invoke the method from the RpgFloatRef class which
// assigns a Float parameter value to the object
// Obtain the method ID so it can be invoked.
mid = (*je)->GetMethodID( je, cls, "setValue", "(F)V");
if ( mid == NULL)
{
    printf(" ERROR 2: GetMethod.\n");
    return;
}

(*je)->CallVoidMethod( je, p1, mid, f4);

// Returning the Double parameter
// Invoke the method from the RpgDoubleRef class which
// assigns a Double parameter value to the object
// Obtain the method ID so it can be invoked.
mid = (*je)->GetMethodID( je, cls2, "setValue", "(D)V");
if ( mid == NULL)
{
    printf(" ERROR 2: GetMethod.\n");
    return;
}

(*je)->CallVoidMethod( je, p2, mid, f8);

Figure 87. Sample VSUBO.C (Part 2 of 2)

The float and double parameter cases are similar to the previous data types. Only
the methods to access the parameter values work with Java primitive data types,
which map to corresponding native primitives, instead of the usual byte arrays.
JNI interface functions dealing with these specific primitives are used to invoke the
methods to access the parameter values.

The RpgFloatRef::getValue, setValue, RpgDoubleRef::getValue, and setValue
methods are used.
Date, Time, Timestamp

* With Parameters: Date, Time, Timestamp.

```c
void _Export __stdcall Java_VJNIOSUBDTZ( JNIEnv *je, void *jc,
                                            jByteArray p1, jByteArray p2, jByteArray p3)
{
    char *fd, *ft, *fz;

    fd = (char *) (*je)->GetByteArrayElements( je, p1, NULL);
    ft = (char *) (*je)->GetByteArrayElements( je, p2, NULL);
    fz = (char *) (*je)->GetByteArrayElements( je, p3, NULL);

    printf(" fd = %.10s
", fd);
    printf(" ft = %.8s
", ft);
    printf(" fz = %.26s
", fz);

    // Now change the values

    memcpy( fd, "2000-01-01",10);
    memcpy( ft, "17.00.00", 8);
    memcpy( fz, "2222-22-22-02.02.02", 19);

    // Update the values back to the Java Caller

    // Fourth Parameter = 0 also causes the variable's storage to be freed,
    // so can not access the variables after this function call.

    (*je)->ReleaseByteArrayElements( je, p1, (signed char *) fd, 0);
    (*je)->ReleaseByteArrayElements( je, p2, (signed char *) ft, 0);
    (*je)->ReleaseByteArrayElements( je, p3, (signed char *) fz, 0);
}
```

**Figure 89. Sample VSUBO.C**

Date, time, and timestamp parameters work the same as character parameters, since they are implemented on the Java side as byte arrays.

**Passing Arrays**

Handling arrayed parameters is done one of two ways depending on the datatype. Invoke the GetObjectArrayElement interface function to get an address to an individual object element in the array, then process it like the scalar parameter methods. Or in the case of an array of Java primitives, there are specific interface functions to access them as an array of native primitives, and then release them back to Java.
d subca  pr  dll('VSUBA')
  d  4
  d  10 dim(4)

  d c1  s  1  inz('J')
  d c4  s  4  inz('blue')
  d c10 s  10 inz('abcdefghij') dim(4)

  d subz pr  dll('VSUBA')
  d  4S 0 dim(4)

  d subp pr  dll('VSUBA')

  d  9P 2 dim(4)

  d subb pr  dll('VSUBA')
  d  4B 0 dim(4)
  d  9B 0 dim(4)

  d z4  s  4S 0 dim(4)
  d p92 s  9P 2 dim(4)
  d b4  s  4B 0 dim(4)
  d b9  s  9B 0 dim(4)

  d subf pr  dll('VSUBA')

  d  4f  dim(4)
  d  8f  dim(4)

*Figure 90. Sample VJNI.A.VPG (Part 1 of 2)*
Figure 90. Sample VJNI.A.VPG (Part 2 of 2)
// Source File: VSUBA.C
// Native function with Character parameters
// Add (d:\jdk12\include; d:\jdk12\include\win32) to the INCLUDE setting
// in order to find jni.h when compiling.
// Compiled with: IBM VisualAge(TM) for C++ for Windows(R), Version 3.5
// Compile command: icc /q /ss /ge - fe vsuba.dll vsuba.c
#include <stdio.h>
#include <string.h>
#include <jni.h>
static void SwapBin2( char *);
static void SwapBin4( char *);

void _Export __stdcall Java_VJNIA_SUBCA( JNIEnv *je, void *jc,

jbyteArray p1, jobjectArray p2)
{
    char  *c4;
    char  *c10;
    jobject p2e;

    // Resolve to 2nd element of array parameter
    p2e = (*je)->GetObjectArrayElement( je, p2,
        1); /* Array index, first element = 0. */

    c4 = (char  *)(*je)->GetByteArrayElements( je, p1, NULL);
    c10 = (char  *)(*je)->GetByteArrayElements( je, p2e, NULL);

    printf(" c4 = %.4s.\n", c4);
    printf(" c10 = %.10s.\n", c10);

    // Now change the values
    memcpy( c4, "Gray", 4);
    memcpy(c10, "Changed ", 10);

    // Update the values back to the Java Caller
    (*je)->ReleaseByteArrayElements( je, p1, (signed char  *) c4, 0);
    (*je)->ReleaseByteArrayElements( je, p2e, (signed char  *) c10, 0);
}

void _Export __stdcall Java_VJNIA_SUBZ( JNIEnv *je, void *jc,

jobject p1)

Figure 91. Sample VSUBA.C (Part 1 of 14)
jclass cls;
jmethodID mid;
jobject aryobj;
char *zd;
jobject pe;

// Resolve to element of array parameter
pe = (*(je))->GetObjectArrayElement(je, p1, 0); /* Array index, first element = 0. */

// p1: Zoned
// Call the method to get the zoned value
cls = (*(je))->GetObjectClass(je, pe);
mid = (*(je))->GetMethodID(je, cls, "zonedValue", "()[B");
aryobj = (*(je))->CallObjectMethod(je, pe, mid);
zd = (char *)(*(je))->GetByteArrayElements(je, aryobj, NULL);
printf("zd = %.4s\n", zd);

// Now change the values
memcpy(zd, "9876", 4);

// Returning the Zoned parameter
// 1. Update the Byte array object with the changed value.
(*(je))->ReleaseByteArrayElements(je, aryobj, (signed char *)zd, 0);

Figure 91. Sample VSUBA.C (Part 2 of 14)
2. Prepare to call the method from the RpgNumeric class which takes a byte array object and assigns its value into the RpgNumeric object. Obtain the method ID.

```c
cls = (*je)->GetObjectClass(je, p1);
// (cls) still identifies the second parameter. Re-use value
mid = (*je)->GetMethodID( je, cls, "assignFromNative", "([BI)V" );
(*je)->CallVoidMethod( je, pe, mid,
  aryobj,
  (int) 1, // Component.ZONED_TYPE
  0 // precision
);
```

```c
void __stdcall Java_VJNIA_SUBP( JNIEnv *je , void *jc,
     jobject p1 // P(9,2)
 )
{
    jclass cls;
    jmethodID mid;
    jobject aryobj;
    char *packednum;
    char tmp[80]; // For tracing
    jobject pe;

    // Resolve to element of array parameter
    pe = (*je)->GetObjectArrayElement( je, p1,
     1); /* Array index, first element = 0. */

    // p1: Packed 9,2
    // Call the method to get the zoned value
    cls = (*je)->GetObjectClass(je, pe);
    mid = (*je)->GetMethodID( je, cls, "packedValue", "()[B" );
    aryobj = (*je)->CallObjectMethod( je, pe, mid);
    packednum = (char *) (*je)->GetByteArrayElements( je, aryobj, NULL);
```

Figure 91. Sample VSUBA.C (Part 3 of 14)
// Now change the values
memcpy( packednum, "{98\x76\x54\x32\x1C}", 5);

// Returning the Packed parameter
// 1. Update the Byte array object with the changed value.
// Fourth Parameter = 0 also causes the variable's storage to be freed,
// so can not access the variables after this function call.
(*je)->ReleaseByteArrayElements( je, aryobj, (signed char *) packednum, 0);

// 2. Prepare to call the method from the RpgNumeric class which
// takes a byte array object and assigns it's value into the
// RpgNumeric object. Obtain the method ID.
// cls = (*je)->GetObjectClass(je, pl);
// (cls) still identifies the second parameter. Re-use value
mid = (*je)->GetMethodID( je, cls, "assignFromNative", "([BII)V");

(*je)->CallVoidMethod( je, pe, mid,
    aryobj,
    (int) 2, // = Component.PACKED_TYPE
    2 // precision (Number of decimal places)
);

//---------------------------------------------------------------

Figure 91. Sample VSUBA.C (Part 4 of 14)
void _Export __stdcall Java_VJNIA_SUBB( JNIEnv *je, void *jc,
   jobject p1 // B(4,0)
   ,jobject p2 // B(9,0)
 )
{
    jclass cls;
    jmethodID mid;
    jobject aryobj;
    jobject aryobj2;
    char *binarynum;
    char *b9;

    short binary2;
    int binary4;
    jobject pe,p2e;

    // Resolve to element of array parameter
    pe = (*je)->GetObjectArrayElement( je, p1, 2); /* Array index, first element = 0. */

    // Resolve to element of array parameter
    p2e = (*je)->GetObjectArrayElement( je, p2, 3); /* Array index, first element = 0. */

    // p1:  Binary 4,0
    // Call the method to get the binary value
    cls = (*je)->GetObjectClass(je, pe);
    mid = (*je)->GetMethodID( je, cls, "binaryValue", "()[B");
    aryobj = (*je)->CallObjectMethod( je, pe, mid);
    binarynum = (char *) (*je)->GetByteArrayElements( je, aryobj, NULL);

    Figure 91. Sample VSUBA.C (Part 5 of 14)
// Must reverse the byte order of the value received
memcpy( &binary2, binarynum, 2);
SwapBin2( (char *) &binary2);

printf(" binary = %hd\n", (short ) binary2);

// p2: Binary 9,0
// Call the method to get the binary value
cls = (*je)->GetObjectClass(je, p2e);
mid = (*je)->GetMethodID( je, cls, "binaryValue", "()[B");
aryobj2 = (*je)->CallObjectMethod( je, p2e, mid);
b9 = (char *) (*je)->GetByteArrayElements( je, aryobj2, NULL);
// Must reverse the byte order of the value received
memcpy( &binary4, b9, 4);
SwapBin4( (char *) &binary4);

printf(" binary = %d.\n", (int ) binary4 );

// Now change the values
binary2 = 5;
// Swap it back for returning to the Java value
SwapBin2( (char *) &binary2);

Figure 91. Sample VSUBA.C (Part 6 of 14)
memcpy( binarynum, &binary2, 2);

// Returning the Packed parameter
// 1. Update the Byte array object with the changed value.
// Fourth Parameter = 0 also causes the variable's storage to be freed,
// so can not access the variables after this function call.
(*je)->ReleaseByteArrayElements(je, aryobj, (signed char *) binarynum, 0);

// 2. Prepare to call the method from the RpgNumeric class which
// takes a byte array object and assigns it's value into the
// RpgNumeric object. Obtain the method ID.

cls = (*je)->GetObjectClass(je, pe);
// (cls) still identifies the second parameter. Re-use value

mid = (*je)->GetMethodID(je, cls, "assignFromNative", "([BII)V");

(*je)->CallVoidMethod(je, pe, mid,
aryobj,
(int) 3, // Component.BINARY_TYPE
0 // precision (Number of decimal places)
);

// Now change the values

binary4 = 19981999;

// Swap it back for returning to the Java value

SwapBin4((char *) &binary4);

memcpy(b9, &binary4, 4);

Figure 91. Sample VSUBA.C (Part 7 of 14)
// Returning the Packed parameter

// 1. Update the Byte array object with the changed value.

// Fourth Parameter = 0 also causes the variable's storage to be freed,
// so can not access the variables after this function call.

(*je)->ReleaseByteArrayElements( je, aryobj2, (signed char *) b9, 0);

// 2. Prepare to call the method from the RpgNumeric class which
// takes a byte array object and assigns it's value into the

// RpgNumeric object. Obtain the method ID.

cls = (*je)->GetObjectClass(je, p2e);
// (cls) still identifies the second parameter. Re-use value

mid = (*je)->GetMethodID( je, cls, "assignFromNative", "([BII)V");

(*je)->CallVoidMethod( je, p2e, mid,
   aryobj2,
   (int) 3, // = Component.BINARY_TYPE
   0,     // precision (Number of decimal places)
);

}-------------------------------

Figure 91. Sample VSUBA.C (Part 8 of 14)
void _Export __stdcall Java_VJNIA_SUBF( JNIEnv *je, void *jc,
            jFloatArray p1, jdoubleArray p2)
{
    jclass cls, cls2;
    jmethodID mid;
    jfloat *f4;
    jdouble *f8;
    jobject p1e,p2e;

    printf("SUBF called successfully.\n");
    f4 = (*je)->GetFloatArrayElements( je, p1, NULL);
    f8 = (*je)->GetDoubleArrayElements( je, p2, NULL);
    printf("f4 = %f\n", (float)f4[0]);

    // p2: Double
    // Call the method to get the double value
    printf("f8 = %f\n", (double)f8[0]);

    // Now change the values
    f4[0] = 999.888;
    f8[1] = 98789.65456;

    // Returning the Float parameter
    (*je)->ReleaseFloatArrayElements( je, p1, f4, 0);
    (*je)->ReleaseDoubleArrayElements( je, p2, f8, 0);
}

Figure 91. Sample VSUBA.C (Part 9 of 14)
//---------------------------------------------------------------

void _Export __stdcall Java_VJNIA_SUBDTZ( JNIEnv *je, void *jc, jbyteArray p1, jbyteArray p2, jbyteArray p3)
{
    char *fd, *ft, *fz;
    jobject p1e, p2e, p3e;

    printf(" SUBDTZ called successfully.\n");

    // Resolve to element of array parameter
    p1e = (*je)->GetObjectArrayElement( je, p1, 2); /* Array index, first element = 0. */
    p2e = (*je)->GetObjectArrayElement( je, p2, 3); /* Array index, first element = 0. */
    p3e = (*je)->GetObjectArrayElement( je, p3, 0); /* Array index, first element = 0. */

    fd = (char *) (*je)->GetByteArrayElements( je, p1e, NULL);
    ft = (char *) (*je)->GetByteArrayElements( je, p2e, NULL);
    fz = (char *) (*je)->GetByteArrayElements( je, p3e, NULL);

    printf(" fd = %.10s.\n", fd);
    printf(" ft = %.8s.\n", ft);
    printf(" fz = %.26s.\n", fz);

    // Now change the values
    memcpy( fd, "2000-01-01", 10);
    memcpy( ft, "17.00.00", 8);
    memcpy( fz, "2222-22-22-02.02.02", 19);

    // Update the values back to the Java Caller
    (*je)->ReleaseByteArrayElements( je, p1e, (signed char *) fd, 0);
    (*je)->ReleaseByteArrayElements( je, p2e, (signed char *) ft, 0);
    (*je)->ReleaseByteArrayElements( je, p3e, (signed char *) fz, 0);
}

Figure 91. Sample VSUBA.C (Part 10 of 14)
// Export __stdcall Java_VJNI_SUBIU( JNIEnv *je , void *jc,
    jshortArray p1, jintArray p2, jobject p3, jobject p4)
{
    jclass cls, cls2;
    jmethodID mid;
    jshort *i2;
    jint *i4;
    jobject aryobj3, aryobj4;
    unsigned short *u2;
    unsigned int *u4;
    jobject p1e, p2e, p3e, p4e;
    printf("SUBIU called successfully.\n");

    // Resolve to element of array parameter
    i2 = (*je)->GetShortArrayElements( je, p1, NULL);
    i4 = (*je)->GetIntArrayElements( je, p2, NULL);
    p3e = (*je)->GetObjectArrayElement( je, p3, 2);
    p4e = (*je)->GetObjectArrayElement( je, p4, 3);

    printf(" i2 = %hd\n", (short) i2[0]);
    printf(" i4 = %d\n", (short) i4[1]);

    // Call the method to get the double value
    cls = (*je)->GetObjectClass( je, p3e);
    mid = (*je)->GetMethodID( je, cls, "unsignedValue", "([B");
    aryobj3 = (*je)->CallObjectMethod( je, p3e, mid);
    u2 = (unsigned short *) (*je)->GetByteArrayElements( je, aryobj3, NULL);

Figure 91. Sample VSUBA.C (Part 11 of 14)
// Must reverse the byte order of the value received
SwapBin2((char *)u2);

printf(" u2 = \%hu\n", *u2);

// p4: Unsigned 4-byte.
// Call the method to get the double value

cls = (*je)->GetObjectClass(je, p4e);
mid = (*je)->GetMethodID( je, cls, "unsignedValue", "([B="");
aryobj4 = (*je)->CallObjectMethod( je, p4e, mid);
u4 = (unsigned int *) (*je)->GetByteArrayElements( je, aryobj4, NULL);

// Must reverse the byte order of the value received
SwapBin4((char *)u4);

printf(" u4 = \%u\n", *u4);

Figure 91. Sample VSUBA.C (Part 12 of 14)
// Now change the values

i2[0] = 99;
i4[1] = 88;

*u2 = 77;
*u4 = 66;

// Must reverse the byte order of the value being returned
SwapBin2((char *) u2);
SwapBin4((char *) u4);

// Return the array memory to Java. Used later to set return
// values for parameters

(*je)->ReleaseByteArrayElements( je, p3e, (signed char *) u2, 0);
(*je)->ReleaseByteArrayElements( je, p4e, (signed char *) u4, 0);

(*je)->ReleaseShortArrayElements( je, p1, i2, 0);
(*je)->ReleaseIntArrayElements( je, p2, i4, 0);

// Returning P3: Unsigned 2-byte

// Invoke the RgpNumeric::assignFromNative method to set the object
// value with an unsigned parameter value

// Obtain the method ID so it can be invoked.
cls = (*je)->GetObjectClass(je, p3e);
mid = (*je)->GetMethodID( je, cls, "assignFromNative", "([BII)V");

// Pass (aryobj3) as first parameter to method because the
// method expects a Java byte array object

(*je)->CallVoidMethod( je, p3e, mid,
aryobj3,
(int) 5, // = Component.UNSIGNED_TYPE
(int) 0); // 0 decimal places

Figure 91. Sample VSUBA.C (Part 13 of 14)
// Returning P4: Unsigned 4-byte
// Invoke the RpgNumeric::assignFromNative method to set the object
// value with an unsigned parameter value
// Obtain the method ID so it can be invoked.

cls = (*je)->GetObjectClass(je, p4e);
mid = (*je)->GetMethodID( je, cls, "assignFromNative", "([BII)V");

(*je)->CallVoidMethod( je, p4e, mid, aryobj4,
(      int) 5, // = Component.UNSIGNED_TYPE
(int) 0); // 0 decimal places

}

 static void SwapBin2( char *p)
{
    char tmp;
    tmp = p[0];
    p[0] = p[1];
    p[1] = tmp;
}

 static void SwapBin4( char *p)
{
    char tmp;
    tmp = p[0];
    p[0] = p[3];
    tmp = p[1];
    p[1] = p[2];


Figure 91. Sample VSUBA.C (Part 14 of 14)

Returning A Char Value

       d     subrc     pr       10     dll('VSUBR')
       d     fc       s        10     inz('ibm varpg ')
       C       eval     fc = subrc

Figure 92. Sample VJNIIR.VPG
Returning a value from the function involves obtaining the appropriate Java object and then returning it. In this sample, a new object (matching a Character(10) field) was created, then its value was assigned. Since the RPG character fields are implemented as Java byte arrays, a Java byte array object of length ten was created, then the GetByteArrayElements interface function was used to access the byte array elements on the native side, then released back to Java, and finally used to return from the function.

If the appropriate Java byte array object was already available from one of the input parameters, then it could have been used instead of creating a new object.

### Returning A Zoned Value

```
d subrs  pr  5s 0  d11('VSUBR')
d fs    s    5s 0
C      eval fc = subrc
```

Figure 94. Sample VJNIR.VPG
jobject _Export __stdcall Java_VJNIR_SUBRS( JNIEnv *je, void *jc) {
    jclass cls;
    jmethodID mid;
    jobject rzo;
    jbyteArray ba;
    char *p;

    printf(" SUBRS called successfully.\n");
    // Create a new RpgZoned object.
    cls = (*je)->FindClass(je, "com/ibm/varpg/rpgruntime/RpgZoned");
    mid = (*je)->GetMethodID(je, cls, "<init>", "(II)V");
    rzo = (*je)->NewObject(je, cls, mid,
                        (int) 5, /* # of digits */
                        (int) 0 /* # of decimal places */
                       );
    // To set the zoned object value, we need a Java byte array to use
    // as an input parameter to the method for setting the zoned object.
    // Could construct a new byte array object, or get one by retrieving
    // the zoned value from the object.
    // Will construct a byte array.
    // Create a new byte array object
    ba = (*je)->NewByteArray(je, 5 /* = byte array length */ );
    // Pin the byte array element memory so native side can access it.
    p = (char *) (*je)->GetByteArrayElements(je, ba, NULL);
    memcpy(p, "55555", 5);

    Figure 95. Sample VSUBR.C (Part 1 of 2)
An RpgZoned object is constructed so it may be returned. Its value is then set through a method call. However, the method to set the value requires a byte array object as an input parameter supplying the value, so the byte array object is constructed first.

An RpgZoned object is constructed by looking up the class, then the constructor method for the class, then invoking the constructor method. A Java byte array object is then constructed and set to a zoned format byte value. A method for setting the RpgZoned object’s value is then resolved and invoked, passing it the byte array object as one of its parameters.

**Returning A Packed Value**

```
  d subrp   pr   5p 0 dll('VSUBR')
  d fp      s    5p 0

  C        eval   fp = subrp
```

*Figure 96. Sample VJNIR.VPG*
object _Export __stdcall Java_VJNIR_SUBRP( JNIEnv *je, void *jc) {
  jclass cls;
  jmethodID mid;
  jobject ro;
  jbyteArray ba;
  char *p;

  printf(" SUBRP called successfully.\n");

  // Create a new RpgPacked object.
  cls = (*je)->FindClass( je, "com/ibm/varpg/rpgruntime/RpgPacked");
  mid = (*je)->GetMethodID( je, cls, "<init>", "(II)V");
  ro = (*je)->NewObject( je, cls, mid,
      (int) 5, /* # of digits */
      (int) 0 /* # of decimal places */
  );

  // To set the packed object value, we need a Java byte array to use
  // as an input parameter to the method for setting the packed object.
  // Could construct a new byte array object, or get one by retrieving
  // the packed value from the object.

  // Create a new byte array object
  ba = (*je)->NewByteArray( je, 3 /* = byte array length */ );

  // Pin the byte array element memory so native side can access it.
  p = (char *) (*je)->GetByteArrayElements( je, ba, NULL);
  memcpy( p, "\x55\x55\x5C", 3);

  // Update the values back to the Java Caller
  (*je)->ReleaseByteArrayElements( je, ba, (signed char *) p, 0);

  // Prepare to call the method from the RpgNumeric class which
  // takes a byte array object and assigns its value into the
  // RpgNumeric object. Obtain the method ID.

  mid = (*je)->GetMethodID( je, cls, "assignFromNative", "([BII)V");
  (*je)->CallVoidMethod( je, ro, mid,
      ba,                               // The byte array object
      (int) 2,                           // = Component.PACKED_TYPE
      0                                 // decimal places
  );

  return ro;
}

Figure 97. Sample VSUBR.C (Part 2 of 2)

Returning a packed value is similar to the zoned case above.
An RpgPacked object is constructed by looking up the class, then the constructor method for the class, then invoking the constructor method. A Java byte array object is then constructed and set to a packed format byte value. A method for setting the RpgPacked object’s value is then resolved to and invoked, passing it the byte array object as one of its parameters.

**Returning A Binary Value**

```c
    d subrb      pr       5b 0 d11('VSUBR')
    d fb        s        5b 0

    C    eval    fb = subrb
```

*Figure 98. Sample VJNI.R.VPG*

```c
jobject _Export __stdcall Java_VJNI_SUBRB( JNIEnv *je, void *jc) {
    jclass    cls;
    jmethodID mid;
    jobject    ro;
    jbyteArray ba;
    char *p;
    printf(" SUBRB called successfully.\n");
    // Create a new RpgPacked object.
    cls = (*je)->FindClass( je, "com/ibm/varpg/rpgruntime/RpgBinary");
    mid = (*je)->GetMethodID( je, cls, "<init>", "(II)V");
    ro = (*je)->NewObject( je, cls, mid,
             (int)5, /* # of digits */
             (int)0   /* # of decimal places */
           );
```

*Figure 99. Sample VSUBR.C (Part 1 of 2)*
Returning a binary value is similar to the zoned or packed cases above, only an RpgBinary object is returned.

Returning An Integer Value

```java
// To set the object value, we need a Java byte array to use // as an input parameter to the method for setting the object.

// Create a new byte array object
ba = (*je)->NewByteArray(je, 4 /* byte array length */);

// Pin the byte array element memory so native side can access it.
p = (char *) (*je)->GetByteArrayElements(je, ba, NULL);

memcpy(p, "\x00\x00\x0D\x03", 4); // 55555 = 0xD903

// Update the values back to the Java Caller
(*je)->ReleaseByteArrayElements(je, ba, (signed char *) p, 0);

// Prepare to call the method from the RpgNumeric class which // takes a byte array object and assigns it's value into the // RpgNumeric object. Obtain the method ID.
mid = (*je)->GetMethodID(je, cls, "assignFromNative", "([BII)V);

(*je)->CallVoidMethod(je, ro, mid, // The byte array object
ba, (int) 3, // Component.Binary_TYPE
0 // decimal places
);

return ro;
}
```

Figure 99. Sample VSUBR.C (Part 2 of 2)

Returning a binary value is similar to the zoned or packed cases above, only an RpgBinary object is returned.

Returning An Integer Value

```java
d subri2 pr 5i 0 def('VSUBR')
d subri4 pr 10i 0 def('VSUBR')
d fi2 s 5i 0
d fi4 s 10i 0
C eval fi2= subri2
C eval fi4= subri4
```

Figure 100. Sample VJNIIR.VPG
Returning a two-byte or four-byte binary integer value is simple. This is due to the types supported as Java primitives.

**Returning An Unsigned Value**

```c
jshort __Export __stdcall Java_VJNIR_SUBRI2( JNIEnv *je, void *jc)
{
    jshort rc;
    rc = -5555;
    return rc;
}

jint __Export __stdcall Java_VJNIR_SUBRI4( JNIEnv *je, void *jc)
{
    return -55555;
}
```

*Figure 101. Sample VSUBR.C*

Returning a two-byte or four-byte binary integer value is simple. This is due to the types supported as Java primitives.

**Figure 102. Sample VJNIR.VPG**
jobject _Export __stdcall Java_VJNIR_SUBRU( JNIEnv *je, void *jc) {
    jclass cls;
    jmethodID mid;
    jobject ro;
    jbyteArray ba;
    char *p;

    printf(" SUBRU called successfully.\n");

    // Create a new RpgUnsigned object.
    cls = (*je)->FindClass( je, "com/ibm/varpg/rpgruntime/RpgUnsigned");
    mid = (*je)->GetMethodID( je, cls, "<init>", "(II)V");
    ro = (*je)->NewObject( je, cls, mid,
                           (int) 5, /* # of digits */
                           (int) 0 /* # of decimal places */
                        );

    // To set the object value, we need a Java byte array to use
    // as an input parameter to the method for setting the object.
    // Create a new byte array object
    ba = (*je)->NewByteArray( je, 4 /* byte array length */ );

    // Pin the byte array element memory so native side can access it.
    p = (char *) (*je)->GetByteArrayElements( je, ba, NULL);

    memcpy( p, "\x00\x00\xD9\x03", 4); // 55555 = 0xD903

    // Update the values back to the Java Caller
    (*je)->ReleaseByteArrayElements( je, ba, (signed char *) p, 0);

    // Prepare to call the method from the RpgNumeric class which
    // takes a byte array object and assigns it's value into the
    // RpgNumeric object. Obtain the method ID.
    mid = (*je)->GetMethodID( je, cls, "assignFromNative", "([BII)V");

    (*je)->CallVoidMethod( je, ro, mid,
                           ba,  // The byte array object
                           (int) 5, /* = Component.UNSIGNED_TYPE */
                           0 /* decimal places */
                        );

    return ro;
}

Figure 103. Sample VSUBR.C (Part 1 of 2)

Returning a two- or four-byte unsigned binary value is similar to the zoned or
packed cases above, only an RpgUnsigned object is used.

Chapter 21. Calling System Functions when Compiling for Java 347
Returning A Date, Time, or Timestamp Value

Date, time, and timestamp values are returned as Java byte arrays of the expected length. This is similar to the character data type.

Returning A Date, Time, or Timestamp Value

Figure 104. Sample VJNI.R.VPG

```
jbyteArray _Export __stdcall Java_VJNI_SUBRD( JNIEnv *je , void *jc)
{
    jbyteArray ba;
    char *p;
    // Create a new byte array object so it can be returned.
    ba = (*je)->NewByteArray( je, 10 /* = byte array length */ );
    // Pin the byte array element memory so native side can access it.
    p = (char *) (*je)->GetByteArrayElements( je, ba, NULL);
    memcpy( p, "2000-01-01",10);
    // Update the values back to the Java Caller
    (*je)->ReleaseByteArrayElements( je, ba, (signed char *) p, 0);
    return ba;
}
```

Figure 105. Sample VSUBR.C

Date, time, and timestamp values are returned as Java byte arrays of the expected length. This is similar to the character data type.

Returning A Float Value

Figure 106. Sample VJNI.R.VPG
Returning a float or double (eight-byte float) value is done directly. This is due to the types supported as Java primitives.

**Returning A Varying-Length Character Value**

A varying-length character value is returned through a Java byte array, where the array length matches the current value length.

**Returning Array Values**

A JNI interface function is called to allocate an array object. If the array elements are Java primitive data types, then the interface functions for allocating these type of array objects is used. (There is a specific function for each primitive type.) These
also allocate the elements of the array. Then it is only a matter of calling the
interface function to map the array elements to native memory and they can be set,
released back to Java, and then returned from the function.

If it is not the case of a Java primitive data type for the array elements, then a Java
object must be allocated for each element of the array, its value set as desired, and
then finally the object is assigned to the specific element of the array. Allocating
the individual objects for the elements is similar to the scalar return value case for
that datatype.

* Source File: VJNIRA.VPG

d subrca pr 10 d11('VSUBRA') dim(4)
d subrsa pr 5s 0 d11('VSUBRA') dim(4)
d subrpa pr 5p 0 d11('VSUBRA') dim(4)
d subrba pr 5b 0 d11('VSUBRA') dim(4)
d subri2a pr 5i 0 d11('VSUBRA') dim(4)
d subri4a pr 10i 0 d11('VSUBRA') dim(4)
d subrua pr 10u 0 d11('VSUBRA') dim(4)

d subrda pr 10d d11('VSUBRA') dim(4)
d subrfa pr 4f d11('VSUBRA') dim(4)
d subrf8a pr 8f d11('VSUBRA') dim(4)
d subrcva pr 10 d11('VSUBRA') varying dim(4)

d fc s 10 dim(4)
d fs s 5s 0 dim(4)
d fp s 5p 0 dim(4)
d fb s 5b 0 dim(4)

d fi2 s 5i 0 dim(4)
d fi4 s 10i 0 dim(4)
d fu s 10u 0 dim(4)
d fd s 10d dim(4)
d ff s 4f dim(4)
d ff8 s 8f dim(4)
d fcv s 10 varying dim(4)

d mb1 m style(*info) button(*OK)
d rc s 9 0

* Figure 110. Sample VJNIRA.VPG (Part 1 of 2)
Figure 110. Sample VJNIRA.VPG (Part 2 of 2)
// Source File: VSUBRA.C
// Native function which returns Array values

// Add (d:\jdk12\include;d:\jdk12\include\win32) to the INCLUDE setting
// in order to find jni.h when compiling.

// Compiled with: IBM VisualAge(TM) for C++ for Windows(R), Version 3.5
// Compile command: icc /q /ss /ge- /fe vsubra.dll vsubra.c

#include <stdio.h>
#include <string.h>
#include <jni.h>

static void SwapBin2( char *);
static void SwapBin4( char *);

//---------------------------------------------------------------
jobjectArray _Export __stdcall Java_VJNIRA_SUBRCA( JNIEnv *je, void *jc)
{
    jobjectArray  oa;
    jclass       cls;
    jbyteArray   ba;
    char         *p;
    int           i;

    printf(" SUBRCA called successfully.\n");

    // Create the object array

    cls = (*je)->FindClass(je, "java/lang/Object");
    if ( cls == NULL)
    {
        printf(" ERROR 1: FindClass.\n");
        return NULL;
    }

Figure 111. Sample VSUBRA.C (Part 1 of 22)
oa = (*je)->NewObjectArray( je, 4 /* array length */, cls, NULL );

if ( oa == NULL )
{
    printf(" ERROR 2: Newobj\n");
    return NULL;
}

// Populate the array
for (i=0; i<4; i++)
{
    // Create a new byte array object so it can be returned.
    ba = (*je)->NewByteArray( je, 10 /* = byte array length */ );

    // Set value of 2nd element
    if ( 1 == i )
    {
        // Pin the byte array element memory so native side can access it.
        p = (char *) (*je)->GetByteArrayElements( je, ba, NULL );
        memcpy( p, "Success ",10 );

        // Update the values back to the Java Caller
        // Fourth Parameter = 0 also causes the variable's storage to be freed,
        // so can not access the variables after this function call.
        (*je)->ReleaseByteArrayElements( je, ba, (signed char *) p, 0 );
    }

    (*je)->SetObjectArrayElement( je, oa, i /* array element, starting at 0 */ , ba );
} // for i

return oa;

//---------------------------------------------------------------

Figure 111. Sample VSUBRA.C (Part 2 of 22)
jobjectArray _Export __stdcall Java_VJNI_RSUBRSA(JNIEnv *je, void *jc) {
    jobjectArray oa;
    int i;
    jclass cls;
    jmethodID mid;
    jobject rzo;
    jbyteArray ba;
    char *p;

    printf(" SUBRSA called successfully.
");  // Create the object array
    cls = (*je)->FindClass(je, "com/ibm/varpg/rpgruntime/RpgZoned");
    if (cls == NULL) {
        printf(" ERROR 1: FindClass.
");
        return NULL;
    }
    oa = (*je)->NewObjectArray(je, 4 /* array length */, cls, NULL);
    if (oa == NULL) {
        printf(" ERROR 2: Newobj
");
        return NULL;
    }
    // Populate the array
    for (i=0; i<4; i++) {
        // Create a new RpgZoned object.
        cls = (*je)->FindClass(je, "com/ibm/varpg/rpgruntime/RpgZoned");
        if (cls == NULL) {
            printf(" ERROR 1: FindClass.
");
            return NULL;
        }
    }

    Figure 111. Sample VSUBRA.C (Part 3 of 22)
mid = (*je)->GetMethodID(je, cls, "<init>", "(II)V");

if (mid == NULL)
{
    printf("ERROR: GetMethod.
");
    return NULL;
}

rzo = (*je)->NewObject(je, cls, mid,
                        (int) 5, /* # of digits */
                        (int) 0 /* # of decimal places */);

if (rzo == NULL)
{
    printf("ERROR3: 
");
    return NULL;
}

// Set value of 2nd element
if (1 == i)
{

    // To set the zoned object value, we need a Java byte array to use
    // as an input parameter to the method for setting the zoned object.
    // Could construct a new byte array object, or get one by retrieving
    // the zoned value from the object.
    // Will construct a byte array.

    // Create a new byte array object
    ba = (*je)->NewByteArray(je, 5 /* byte array length */);

    if (ba == NULL)
    {
        printf("ERROR4: 
");
        return NULL;
    }

Figure 111. Sample VSUBRAC.C (Part 4 of 22)
// Pin the byte array element memory so native side can access it.
p = (char *) (*je)->GetByteArrayElements( je, ba, NULL);
memcpy( p, "55555", 5);

// Update the values back to the Java Caller
// Fourth Parameter = 0 also causes the variable's storage to be freed,
// so can not access the variables after this function call.
(*je)->ReleaseByteArrayElements( je, ba, (signed char *) p, 0);

// Prepare to call the method from the RpgNumeric class which
// takes a byte array object and assigns it's value into the
// RpgNumeric object. Obtain the method ID.

// cls = (*je)->GetObjectClass( je, pl);
// (cls) still identifies the second parameter. Re-use value
mid = (*je)->GetMethodID( je, cls, "assignFromNative", "([BI]V");
if ( mid == NULL) {
    printf(" ERROR 2: GetMethod.\n");
    return NULL;
}

(*je)->CallVoidMethod( je, rzo, mid, ba, 1, 0, 0);  // Component.ZONED_TYPE
}

Figure 111. Sample VSUBRA.C (Part 5 of 22)
(*je)->SetObjectArrayElement(je, oa, i /* array element, starting at 0 */ , rzo);
} // for i
return oa;
}
//---------------------------------------------------------------

jobjectArray _Export __stdcall Java_VJNIRA_SUBRPA(JNIEnv *je, void *jc)
{
    jobjectArray oa;
    int i;
    jclass cls;
    jmethodID mid;
    jobject ro;

    jbyteArray ba;
    char *p;

    printf( "SUBRPA called successfully.\n");

    // Create the object array
    cls = (*je)->FindClass(je, "com/ibm/varpg/rpgruntime/RpgPacked");
    if (cls == NULL)
    {
        printf( "ERROR 1: FindClass.\n");
        return NULL;
    }
    oa = (*je)->NewObjectArray(je, 4 /* array length */, cls, NULL);
    if (oa == NULL)
    {
        printf( "ERROR 2: Newobj\n");
        return NULL;
    }

    // Populate the array
    for (i=0; i<4; i++)
    {


Figure 111. Sample VSUBRA.C (Part 6 of 22)
// Create a new RpgPacked object.
#if 0
  cls = (*je)->FindClass(je, "com/ibm/varpg/rpgruntime/RpgPacked");
  if (cls == NULL)
    { printf("ERROR 1: FindClass.
           return NULL;
    }
#endif

mid = (*je)->GetMethodID(je, cls, "<init>", "(II)V");
if (mid == NULL)
  { printf("ERROR: GetMethod.
           return NULL;
  }

ro = (*je)->NewObject(je, cls, mid,
    (int) 5, /* # of digits */
    (int) 0, /* # of decimal places */
);    
if (ro == NULL)
  { printf("ERROR3: \n");
    return NULL;
  }

// Set value of 2nd element
if (1 == i)
{

  // To set the packed object value, we need a Java byte array to use
  // as an input parameter to the method for setting the packed object.
  // Could construct a new byte array object, or get one by retrieving
  // the packed value from the object.
  // Will construct a byte array.

Figure 111. Sample VSUBRA.C (Part 7 of 22)
// Create a new byte array object
ba = (*je)->NewByteArray( je, 3 /* byte array length */ );

if ( ba == NULL )
{
    printf("ERROR4: \n");
    return NULL;
}

// Pin the byte array element memory so native side can access it.
p = (char *) (*je)->GetByteArrayElements( je, ba, NULL);
memcpy( p, "\x55\x55\x5C", 3);

// Update the values back to the Java Caller
// Fourth Parameter = 0 also causes the variable's storage to be freed,
// so can not access the variables after this function call.
(*je)->ReleaseByteArrayElements( je, ba, (signed char *) p, 0);

// Prepare to call the method from the RpgNumeric class which
// takes a byte array object and assigns it's value into the
// RpgNumeric object. Obtain the method ID.

// cls = (*je)->GetObjectClass( je, pl);
// (cls) still identifies the second parameter. Re-use value
mid = (*je)->GetMethodID( je, cls, "assignFromNative", "([BII)V");
if ( mid == NULL )
{
    printf("ERROR 2: GetMethod.\n");
    return NULL;
}

Figure 111. Sample VSUBRA.C (Part 8 of 22)
(*je)->CallVoidMethod(je, ro, mid, ba, // The byte array object
   (int) 2,   // = Component.PACKED_TYPE
   0         // decimal places
);

(*je)->SetObjectArrayElement(je, oa, i /* array element, starting at 0 */,
   ro);
} // for i

return oa;
} //---------------------------------------

tobjectArray _Export _stdcall Java_VNIRA_SUBRBA(JNIEnv *je, void *jc)
{
    jobjectArray oa;
    int i;
    jclass cls;
    jmethodID mid;
    jobject ro;
    jbyteArray ba;
    char *p;

    printf(" SUBRBA called successfully.
");

    // Create the object array
    cls = (*je)->FindClass(je, "com/ibm/varpg/rpgruntime/RpgBinary");
    if (cls != NULL)
    {
        printf(" ERROR 1: FindClass.
            return NULL;
    }

    Figure 111. Sample VSUBRA.C (Part 9 of 22)
oa = (*je)->NewObjectArray( je, 4 /* array length */, cls, NULL );

if ( oa == NULL )
{
    printf(" ERROR 2: Newobj\n");
    return NULL;
}

// Populate the array
for ( i=0; i<4; i++ )
{
    // Create a new RpgPacked object.
#if 0
    cls = (*je)->FindClass( je, "com/ibm/varpg/rpgruntime/RpgBinary" );
    if ( cls == NULL )
    {
        printf(" ERROR 1: FindClass.\n");
        return NULL;
    }
#endif
    mid = (*je)->GetMethodID( je, cls, "<init>", "(II)V" );
    if ( mid == NULL )
    {
        printf(" ERROR: GetMethod.\n");
        return NULL;
    }
    ro = (*je)->NewObject( je, cls, mid,
        (int) 5, /* # of digits */
        (int) 0 /* # of decimal places */
    );

Figure 111. Sample VSUBRA.C (Part 10 of 22)
if (ro == NULL) {
    printf("ERROR3: \n");
    return NULL;
}

// Set value of 2nd element
if (1 == i) {
    // To set the packed object value, we need a Java byte array to use
    // as an input parameter to the method for setting the packed object.
    // Could construct a new byte array object, or get one by retrieving
    // the packed value from the object.
    // Will construct a byte array.
    // Create a new byte array object
    ba = (*je)->NewByteArray(je, 4 /* byte array length */);

    if (ba == NULL) {
        printf("ERROR4: \n");
        return NULL;
    }

    // Pin the byte array element memory so native side can access it.
    p = (char.*) (*je)->GetByteArrayElements(je, ba, NULL);
    memcpy(p, "\x00\x00\xD9\x03", 4); // 55555 = 0xD903

    // Update the values back to the Java Caller
    // Fourth Parameter = 0 also causes the variable's storage to be freed,
    // so can not access the variables after this function call.
    (*je)->ReleaseByteArrayElements(je, ba, (signed char*) p, 0);

Figure 111. Sample VSUBRA.C (Part 11 of 22)
// Prepare to call the method from the RpgNumeric class which
// takes a byte array object and assigns it's value into the
// RpgNumeric object. Obtain the method ID.

// cls = (*je)->GetObjectClass(je, pl);
// (clS) still identifies the second parameter. Re-use value

mid = (*je)->GetMethodID(je, cls, "assignFromNative", "([BII)V");
if ( mid == NULL)
{
    printf(" ERROR 2: GetMethod.\n");
    return NULL;
}

(*je)->CallVoidMethod(je, ro, mid,
    ba,       // The byte array object
    (int) 3,   // = Component.Binary_TYPE
    0         // decimal places
    );

(*je)->SetObjectArrayElement(je, oa, i /* array element, starting at 0 */
    , ro);
} // for i

return oa;

Figure 111. Sample VSUBRA.C (Part 12 of 22)
JNIEXPORT JNICALL Java_VJNIIRA_SUBRI2A(JNIEnv *je, void *jc)
{
  jshortArray rc;
jshort *n;

  printf(" SUBRI2A called successfully.\n");
  rc = (*je)->NewShortArray( je, 4 /* = array length */ );
  // Pin the array element memory so native side can access it.
  n = (*je)->GetShortArrayElements( je, rc, NULL);
  n[1] = -5555;
  // Update the values back to the Java Caller
  (*je)->ReleaseShortArrayElements( je, rc, n, 0);
  return rc;
}

JNIEXPORT JNICALL Java_VJNIIRA_SUBRI4A(JNIEnv *je, void *jc)
{
  jintArray rc;
jint *n;
  printf(" SUBRI4A called successfully.\n");
  rc = (*je)->NewIntArray( je, 4 /* = array length */ );
  // Pin the array element memory so native side can access it.
  n = (*je)->GetIntArrayElements( je, rc, NULL);
  n[1] = -5555;
  // Update the values back to the Java Caller
  (*je)->ReleaseIntArrayElements( je, rc, n, 0);
  return rc;
}

Figure 111. Sample VSUBRA.C (Part 13 of 22)
jobjectArray _Export __stdcall Java_VJNIRA_SUBRUA( JNIEnv *je, void *jc) {
    jobjectArray ao;
    int i;
    jclass cls;
    jmethodID mid;
    jobject ro;
    jbyteArray ba;
    char *p;

    printf(" SUBRUA called successfully.\n");

    // Create the object array
    cls = (*je)->FindClass( je, "com/ibm/varpg/rpgruntime/RpgUnsigned");
    if ( cls == NULL) {
        printf(" ERROR 1: FindClass.\n");
        return NULL;
    }

    ao = (*je)->NewObjectArray( je, 4 /* array length */, cls, NULL );
    if ( ao == NULL) {
        printf(" ERROR 2: Newobj\n");
        return NULL;
    }

    // Populate the array
    for (i=0; i<4; i++) {
        // Create a new RpgPacked object.
        cls = (*je)->FindClass( je, "com/ibm/varpg/rpgruntime/RpgUnsigned");
        if ( cls == NULL) {
            printf(" ERROR 1: FindClass.\n");
            return NULL;
        }
    }

    printf(" SUBRUA called successfully.\n");

    return ao;
}

Figure 111. Sample VSUBRA.C (Part 14 of 22)
mid = (*je)->GetMethodID( je, cls, "<init>", "(II)V");

if ( mid == NULL) {
    printf(" ERROR: GetMethod.\n");
    return NULL;
}

ro = (*je)->NewObject( je, cls, mid,
                    (int) 5, /* # of digits */
                    (int) 0 /* # of decimal places */
                    );

if ( ro == NULL) {
    printf(" ERROR3: \n");
    return NULL;
}

// Set value of 2nd element
if ( 1 == i) {

    // To set the packed object value, we need a Java byte array to use
    // as an input parameter to the method for setting the packed object.
    // Could construct a new byte array object, or get one by retrieving
    // the packed value from the object.

    // Will construct a byte array.

Figure 111. Sample VSUBRA.C (Part 15 of 22)
// Create a new byte array object

ba = (*je)->NewByteArray( je, 4 /* = byte array length */ );

if ( ba == NULL )
{
  printf(" ERROR4: \n");
  return NULL;
}

// Pin the byte array element memory so native side can access it.
p = (char *) (*je)->GetByteArrayElements( je, ba, NULL);
memcpy( p, "\x00\x00\xD9\x03", 4);  // 55555 = 0xD903

// Update the values back to the Java Caller
// Fourth Parameter = 0 also causes the variable's storage to be freed,
// so can not access the variables after this function call.
(*je)->ReleaseByteArrayElements( je, ba, (signed char *) p, 0);

Figure 111. Sample VSUBRA.C (Part 16 of 22)
Prepare to call the method from the RpgNumeric class which takes a byte array object and assigns it's value into the RpgNumeric object. Obtain the method ID.

cls = (*je)->GetObjectClass(je, p1);
// (cls) still identifies the second parameter. Re-use value
mid = (*je)->GetMethodID(je, cls, "assignFromNative", "([BII)V");

if (mid == NULL)
{
    printf(" ERROR 2: GetMethod.\n");
    return NULL;
}

(*je)->CallVoidMethod(je, ro, mid,
    ba, // The byte array object
    (int) 5, // = Component.UNSIGNED_TYPE
    0, // decimal places
    );

(*je)->SetObjectArrayElement(je, oa, i /* array element, starting at 0 */
    , ro);
} // for i

return oa;

Figure 111. Sample VSUBRA.C (Part 17 of 22)
jobjectArray _Export __stdcall Java_VJNIRA_SUBRDA( JNIEnv *je, void *jc) 
{
    jobjectArray oa;
    jclass cls;
    jbyteArray ba;
    char *p;
    int i;

    printf(" SUBRD called successfully.\n");

    // Create the object array
    cls = (*je)->FindClass( je, "java/lang/Object");
    if ( cls == NULL) 
    {
        printf(" ERROR 1: FindClass.\n");
        return NULL;
    }
    
    oa = (*je)->NewObjectArray( je, 4 /* array length */, cls, NULL );
    if ( oa == NULL) 
    {
        printf(" ERROR 2: Newobj\n");
        return NULL;
    }

    // Populate the array
    for (i=0; i<4; i++)
    {
        // Create a new byte array object so it can be returned.
        ba = (*je)->NewByteArray( je, 10 /* byte array length */ );
    }

    {Figure 111. Sample VSUBRA.C (Part 18 of 22)
// Set all elements to a valid date value.
    // Pin the byte array element memory so native side can access it.
    p = (char *) (*je)->GetByteArrayElements( je, ba, NULL);
    memcpy( p, "2000-01-01", 10);
    // Update the values back to the Java Caller
    // Fourth Parameter = 0 also causes the variable's storage to be freed, 
    // so can not access the variables after this function call.
    (*je)->ReleaseByteArrayElements( je, ba, (signed char *) p, 0);
    
    (*je)->SetObjectArrayElement( je, oa, i /* array element, starting at 0 */
                              , ba);
    
    return oa;
}

java(JNIEnv *je, void *jc)
{
    jfloatArray rc;
    jfloat *n;

    printf("SUBRF called successfully.\n");

    rc = (*je)->NewFloatArray( je, 4 /* = array length */);
    // Pin the array element memory so native side can access it.
    n = (*je)->GetFloatArrayElements( je, rc, NULL);
    n[1] = -4444.4444;
    // Update the values back to the Java Caller
    (*je)->ReleaseFloatArrayElements( je, rc, n, 0);
    return rc;
}

Figure 111. Sample VSUBR.A.C (Part 19 of 22)
//---------------------------------------------------------------
jdoublereal _Export __stdcall Java_VJNIRA_SUBRF8A( JNIEnv *je, void *jc)
{
  jdoublereal rc;
  jdoublereal *n;
  printf(" SUBRF8 called successfully.\n");
  rc = (*je)->NewDoubleArray( je, 4 /* = array length */ );
  // Pin the array element memory so native side can access it.
  n = (*je)->GetDoubleArrayElements( je, rc, NULL);
  n[1] = -777777.55555;
  // Update the values back to the Java Caller
  (*je)->ReleaseDoubleArrayElements( je, rc, n, 0);
  return rc;
}
//---------------------------------------------------------------
jobjectArray _Export __stdcall Java_VJNIRA_SUBRCVA( JNIEnv *je, void *jc)
{
  // This is similar to fixed length character, only the individual
  // array elements can be created as byte arrays of different lengths
  // to reflect the current length of the varying length values.

Figure 111. Sample VSUBRA.C (Part 20 of 22)
jobjectArray oa;
jclass cls;
jbyteArray ba;
char *p;
int i;

printf(" SUBRCVA called successfully.\n");

// Create the object array
cls = (*je)->FindClass( je, "java/lang/Object");
if ( cls == NULL)
{
    printf(" ERROR 1: FindClass.\n");
    return NULL;
}

oa = (*je)->NewObjectArray( je, 4 /* array length */, cls, NULL);
if ( oa == NULL)
{
    printf(" ERROR 2: Newobj\n");
    return NULL;
}

Figure 111. Sample VSUBRA.C (Part 21 of 22)
// Populate the array
for (i=0; i<4; i++)
{
    // Create a new byte array object so it can be returned.
    ba = (*je)->NewByteArray( je,
        /* = byte array length */
        (1==i) ? 4 : 10);

    // Set value of 2nd element
    if ( 1 == i )
    {
        // Pin the byte array element memory so native side can access it.
        p = (char *) (*je)->GetByteArrayElements( je, ba, NULL);
        memcpy( p, "abcd", 4);

        // Update the values back to the Java Caller

        // Fourth Parameter = 0 also causes the variable's storage to be freed,
        // so can not access the variables after this function call.
        (*je)->ReleaseByteArrayElements( je, ba, (signed char *) p, 0);
    }

    (*je)->SetObjectArrayElement( je, oa, i /* array element, starting at 0 */
        , ba);
}

return oa;

Figure 111. Sample VSUBRA.C (Part 22 of 22)
Chapter 22. Creating Non-GUI VisualAge RPG Programs

This section describes how to create standalone VARPG applications and **Dynamic Link Libraries** (DLLs). Standalone VARPG applications have no user interface, but they can access local and AS/400 files, and call AS/400 programs. DLLs are modules that cannot be executed directly; they contain procedures that can be called by other VARPG applications. DLLs can also access local files, as well as AS/400 files and programs. You can think of DLLs as you would of AS/400 service programs.

You can create standalone VARPG applications within the VARPG GUI designer, or by issuing commands in an MS-DOS command prompt. (See Appendix C, “Creating and Compiling Non-GUI Programs from MS-DOS,” on page 447 for the commands.) This section describes how to use the GUI designer to create non-GUI programs.

When creating standalone applications or DLLs, the following restrictions apply:

- They must consist entirely of procedures.
- *ENTRY is not permitted.
- The special subroutines *INZSR and *TERMSR are not allowed.
- All subroutines must be local to a procedure.
- The EXPORT keyword is not allowed when creating standalone applications.
- Because standalone applications and DLLs have no user interface, the %GETATR and %SETATR built-ins, and GUI operation codes are not allowed. These include:
  - CLSWIN, GETATR, SETATR, START, STOP, SHOWWIN, READS
- The DSPLY operation code can be used. However, if the procedure containing it is called from a VisualAge RPG DLL, the DSPLY operation code does nothing. Also, the DSPLY operation code does not support a message data type in factor 1.

Creating Standalone VARPG Programs

A standalone VARPG program is created when the EXE keyword is specified on the control specification.

H  EXE

The program source must contain a procedure whose name matches the name of the source file. This will be the main entry point for the program. If there are parameters to be passed to the program, they must be specified on the parameter definition for the main procedure, and they must be passed in by value. That is, the **VALUE** keyword must be specified for each parameter. When calling an application from the command line, separate parameters by spaces. If more parameters or fewer are passed than are specified, no error message is displayed.

To create a standalone program in the GUI Designer, select **Project > New > Non GUI project** from the project window. The editor opens a new source file that has an H control specification template. Uncomment the H * EXE specification and code your program. When completed, save your project and build the application. You can set any needed build options from the project window, as well.
In the following example, the standalone VARPG program accepts a single parameter. When run, the program will translate the parameter to uppercase and display the result using the DSPLY operation code. Note that the name of the main, and only, procedure is MyPgm. If you want to try this sample, be sure to specify MYPGM as the file name when you save it.

* Sample standalone VARPG program
H EXE
  * Prototype for the main procedure
D MyPgm       PR
  *
  * Procedure definition for MYPGM
PMyPgm        B
  *
D MyPgm       PI
D InString    64A  Value
  *
D OutString   S   64A
  *
D LC          C   'abcdefghijklmnopqrstuvwxyz'
D UC          C   'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
  *
* Translate input parameter to uppercase and display it
C lc:uc      Xlate InString OutString
  *
C OutString   Dspy  1
  *
PMyPgm        E

Creating DLLs

A DLL is created when the keyword NOMAIN is specified on the control specification:
H NOMAIN

To create a DLL in the GUI Designer, select Project > New > Non GUI project from the project window. The editor opens a new source file that has an H control specification template. Uncomment the H * NOMAIN specification and code your program. When completed, save your project and build the DLL. You can set any needed build options from the project window, as well.

When you build a DLL, the compiler produces the DLL and a LIB file. The LIB file is used to link the DLL to other applications. The LIB file will be in the same directory as the source and it will have the same name as the DLL. The LIB file contains all the procedures that have the EXPORT keyword on their Begin P-specification.

The following example shows how to code the part of program MyPGM that translates the lowercase string to uppercase as a procedure in a DLL. The source for the DLL has one procedure named ToUpper in it. Add the Export keyword to the procedure definition so that this procedure can be called from other programs.

* Sample VARPG DLL
H NOMAIN
  *
  * Prototype ToUpper procedure
D ToUpper    PR   64A
D              64A  Value
  *
* The ToUpper procedure

PToupper B Export

* D ToUpper PI 64A
D InString 64A Value

* D OutString S 64A
* D LC C 'abcdefghijklmnopqrstuvwxyz'
D UC C 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

* C lc:uc Xlate InString OutString
C Return OutString

* PToUpper E

When you create and build a DLL, you can give it any name. For this example, we use MyFunc. A successful build will create the following files in the source directory:

- MyFunc.VPG - program source
- MyFunc.DLL - the DLL
- MyFunc.LST - the compiler listing
- MyFunc.LIB - the library file
- MyFunc.EVT - the event file (used by the GUI designer to display the error feedback window; not required to run the program)

Edit and modify the MyPGM source so it calls the ToUpper procedure in the MyFunc.DLL that was just created. The modified source follows:

0000 * Calling a procedure in a VARPG DLL
0001 H EXE
0002 *

0003 D ToUpper PR 64A DLL('MyFunc')
0004 D ExtProc('TOUPPER')
0005 D Value

0007 *
0008 D MyMain PR 64A Value
0009 D *
0010 PMyMain B
0011 *
0012 D MyMain PI
0013 D InString 64A Value
0015 *
0016 D Upper S 64A
0017 *

0018 C Eval Upper=ToUpper(Instring)

0019 C Upper Dsply I 1
0010 *
0011 PMyMain E

Line Description of Change

0003 Define the prototype for the ToUpper procedure and specify that the procedure returns a parameter that is an alpha field, 64 bytes long. The DLL keyword specifies that the procedure is in the DLL named MyFunc.DLL.
The ExtProc keyword specifies the name of the procedure to be called. Since the name is the same name that is used on the Definition specification (line 0003), you can omit the keyword. If the name is specified, however, it must be in uppercase, as shown.

This statement indicates that the procedure expects one parameter - a 64 character alpha field. In this case, the parameter is being passed by VALUE.

This is the call to the procedure.

If the procedure you are calling does not return a value, then you must use the CALLP operation code to invoke it:

```
C     CALLP     SomeFunc(parm1:parm2)
```
Exception Handling

Exception handling differs from GUI VARPG applications in the following ways:

- No information about the exception is communicated back to the caller if the caller does not reside in the utility DLL.
- The default exception handler is never invoked from a DLL, since the default exception handler is not invoked when an exception occurs in a procedure. If an exception occurs in the DLL and there is no error indicator or *PSSR, the DLL ends. Information about the exception is written to the FVDCCERRS.LOG file.
- The recommended way to handle exceptions in a utility DLL is to have an error indicator or a local *PSSR for each routine which returns an appropriate return code to the caller.

Debugging Applications

To debug VARPG programs, be sure to use the debug compiler option when building the application. If the debug option is not set, you can still start the debugger on the program, but you will have to work with the assembler view of your program.

To run the debugger against your source, you must first build your application. From the project view, choose Project > Build > Windows NT/95/98. To start the debugger, select the Debug menu item from the Project menu. See Chapter 10, “Debugging Your Application,” on page 227 for more information on debugging.

Debugging Procedures

If you want to debug code in your DLL, you need to follow some extra steps:

1. Start the debugger for your main application- in our example, MyMain.
2. On the Debugger - Session Control dialog, choose Breakpoints-Set load occurrence....
3. When the Load Occurrence Breakpoint dialog is displayed, type the name of the DLL, MyFunc, in the DLL File Name entry and press OK.
4. Run your program.

When the procedure is called in the DLL, a debugger message dialog is displayed indicating that the DLL is being loaded. Press OK and do the following:

1. Locate the Debug - Session Control dialog and note that there is a new entry in the right side panel with the name of the DLL.
2. Click on the + sign next to the DLL name. It will expand to show the name of the object module, MyFunc.obj.
3. Double-click on the object module name.
4. The source view of the debugger will now show the source for procedure ToUpper in DLL MYFUNC.

You can now add breakpoints and display program variables in the DLL. Also, if you are currently STARTing other VARPG components, or if you are calling your own ‘C’ functions, you can also use the above procedures to debug them.
Chapter 23. DBCS Considerations

If you plan to use VisualAge RPG on a Double Byte Character Set (DBCS) system, you must consider the following:

- The compiler does not allow shift-in and shift-out characters in literals. If you use the VisualAge RPG editor to open an AS/400 member in order to copy source into your VisualAge RPG program, you must remove the shift-in and shift-out characters from all literals. If they are not removed, compile errors occur.
- The compiler removes shift-in and shift-out characters from your VisualAge RPG source members when they are retrieved using the remote /COPY feature.
- DBCS characters are not allowed in the icon file name extension for an application.
- A VisualAge RPG application name that contains non-DBCS characters will cause a build failure.

VisualAge RPG Support for DBCS Data Types

VisualAge RPG supports a number of DBCS data types. When you run your application, certain rules are followed when the DBCS data types are used, in order to ensure that data is correctly transferred between the AS/400 server and the workstation. The following DBCS data types are supported:

- **DBCS Only**
  A field of this data type contains only DBCS data, and should be used when you are using the AS/400 database. It is equivalent to the J data type supported by the AS/400 database.

- **DBCS Either**
  A field of this data type contains all single-byte or all DBCS data. It should be used when you are using the AS/400 database. It is equivalent to the E data type supported by the AS/400 database.

- **DBCS Mixed**
  A field of this data type contains all single-byte or all DBCS data. It should be used when you are interchanging data with the AS/400 database. It is equivalent to the O data type supported by the AS/400 database.

AS/400 J, O and E data types require that DBCS data be bracketed by SO (Shift Out) and SI (Shift In) characters. The workstation fields DBCS Either, DBCS Mixed, and DBCS ONLY fields do not use SO and SI characters. When these fields are used to transfer data to the server, SO and SI characters are added appropriately. When data is being retrieved from the server, SO and SI characters are stripped, and the VisualAge RPG field is padded with two single-byte blanks.

DBCS Either, DBCS Mixed, and DBCS ONLY fields are represented as character fields with the same names as their part names within the VisualAge RPG application.

The following example illustrates how data is converted when DBCS data is transferred to and from the server. In this example, a 10 byte DBCS ONLY field is created using VisualAge RPG. This means that the field can contain four DBCS characters, since each DBCS character requires two bytes. The extra two bytes are
used to insert the SO and SI characters before the field is transferred to the server. Assume that the field contains the following data before being transferred to the server:

```
DBDBDBDb1b1
```

Where DB = 1 Double byte character.

```
bl = 1 Single byte blank character.
```

Before the field is transferred to the server, it is converted so that the DBCS data is bracketed by the SO and SI characters. The single-byte blanks are treated as being insignificant, and they are replaced with the appropriate SO and SI characters. Therefore, the field would appear as follows before being transferred to the server:

```
S0DBDBDDBSI
```

If the same data is retrieved from the server, then the SO and SI characters are stripped and the field is padded with two single byte blanks:

```
DBDBDBDBb1b1
```

Where DB = 1 Double byte character

```
bl = 1 Single byte blank character
```

**Note:** The character fields representing the DBCS ONLY, DBCS Mixed, or DBCS Either data types must be padded with the appropriate number of single-byte blanks in order for the field to be transferred to the server and in order for the data within the field to be displayed in the window correctly.

VisualAge RPG ensures that enough single-byte blanks are present. When setting DBCS fields or retrieving information from DBCS fields using the SETATR and GETATR operation codes, respectively, you must ensure that the length of the field in the SETATR and GETATR operations is the same length as the field in the window. If it is not, it may not be transferred between the server and the workstation.

**DBCS ONLY Data Type**

VisualAge RPG ensures the following when the DBCS ONLY data type is used, regardless of whether the data is added via the field on the window or entered using the SETATR operation code:

- The minimum field length is 2. This ensures that there is enough room for the SO and SI characters that are added when the data is transferred to the server.
- The field contains valid DBCS characters. Each double-byte pair is checked to ensure that a valid DBCS character is used.
- The field is appropriately padded with blank characters. If a smaller value is entered than the field allows, the field is padded to the maximum length of 2 with double-byte blanks. The last two bytes of the field are padded with single-byte blanks.

**DBCS Either Data Type**

The DBCS Either data type must contain either all single-byte data or all double-byte data: mixture of DBCS and single-byte data is not allowed. If single-byte data is used, then the maximum length of the field can be used to hold the single-byte data and the maximum length of the data can be transferred to and from the server.
VisualAge RPG enforces the following rules when the first two bytes of the field represent a DBCS character, regardless of whether the data is added via the field on the window or entered using the SETATR operation code:

- The minimum field length is 2. This ensures that there is enough room for the SO and SI characters that are added when the data is transferred to the server.
- The field contains only valid DBCS characters. Each double-byte pair is checked to ensure that a valid DBCS character is used.
- The field is appropriately padded with blank characters. If a smaller value is entered than the field will allow, then the field is padded to the maximum length of 2 with double-byte blanks. The last two bytes of the field are padded with single-byte blanks.

**DBCS Mixed Data Type**

This field can contain any number of DBCS or single-byte characters interchangeably. VisualAge RPG enforces the following rules:

- This character field is always padded with single-byte blanks.
- For each change in DBCS mode, an SO and SI character must be accounted for.
  Each time the user changes between entering DBCS characters and entering single-byte characters, 2 is subtracted from the maximum length that can be entered. For example, assume a DBCS Mixed field is created with a length of 20 using VisualAge RPG. This field has the following value:

\[ DBsbDBsbDBsbDBsb. \]

where \( DB = 1 \) DBCS character.
\( sb = 1 \) single byte character.

This is the maximum length of the field, since the field is converted to the following before being transferred to the server.

\[ SB0DSISbSB0DSISbSB0DSISbSB0DSISb. \]

where \( SO = 1 \) shift out character.
\( sb = 1 \) shift in character.

All 20 bytes of the field are used.

**Pure DBCS Considerations**

Both the VisualAge RPG language and the AS/400 database support a pure DBCS data type: the G or Graphic data type. Pure DBCS data does not require the SO (Shift Out) or the SI (Shift In) characters on the AS/400 server or the workstation. When Graphic data is converted between the AS/400 server and the workstation, no SO and SI characters are added or removed.

GUI entry fields do not directly map to the Graphic data type supported in the VisualAge RPG language. To use the full extent of the field, it is recommended that you create a character entry on the window. When you do, a VisualAge RPG character field is created with the same name as the GUI Designer part. A separate Graphic field can then be used to interact with the character entry field created, using the GUI Designer. Use the SETATR or GETATR operation code to interact with the entry fields. In this way, the entire length of the entry field can be used to store DBCS characters without concern for the SO and SI characters.
Chapter 24. Merging Code in Your Application

When programming, you may wish to merge two or more parts of a project or component and the associated code together. You can use the Merge function to do this. Select the Merge menu item from the Project pull-down menu. This will bring up the Open Component - VisualAge RPG dialog box, which will allow you to select the project that you would like to merge from.

This dialogue is similar in look and function to the Find Folder/Project dialog box. You can either specify the project in the entry field, including the complete path, or you can use the list box to select a drive and proceed to select folders to find the necessary project. Both methods will open the GUI Objects Tree View window for the specified project. Alternatively, you can select the GUI Objects menu item from the View pull-down menu in the project organizer.

![Figure 114. The Code Merge GUI Objects Tree View](image)

This window shows two views, the one on the left contains the tree view of the project that you selected to merge from, and the one on the right contains all of the children of the part that is selected in the tree-view on the left. You can select multiple parts in the right side of the window much like you can in the Windows Explorer. This view may be used as an additional parts pallette because you can select items from here (either the left or the right pane), and then point-and-click them onto the current project’s tree view or onto the design window. This works in the same way as the parts pallette in that you can only place parts into a frame-based part, and frame-based parts can only be placed into the root of your project tree. When you merge the GUI and the associated code, the builder will force a save of the current project that you are working on, in order to provide you with a backup of your work in the event that you are not satisfied with the merge results.
In addition to the GUI layout, the merge will copy linked action subroutines, help panels, technical descriptions, references to media files, referenced user subroutines and user messages. There are a few rules to keep in mind about merging code in these specific cases.

- All linked action subroutines will be copied.
- Referenced media files are not copied along with the references. It is your responsibility to do this.
- File description specifications and definition specifications are not copied to the current project. Again, it is your responsibility to do this.
- User subroutines, RPG procedures and User messages referenced by the action subroutine being copied will also be copied. This includes all references to user subroutines used by an EXSR or a CASxx operation code, RPG procedures referenced on a CALLP operation code, and user messages referenced with the DSPLY operation code.
- For parts which have been renamed, all action subroutines that refer to the part, and which have names that conform to the standard format will be renamed. For example, the following source code would be renamed as it follows the standard format. The requirement for this format is that the partname and windowname directly correspond to the location in which the part can be found.

```
*....+....2....+....3....+....4....+....5....+....6....+....7....+....8
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq----
```

- User messages which are copied are renamed consecutively beginning with the first message copied. All merged user messages are numbered sequentially starting immediately after the last message in the current project. The message IDs are changed on DSPLY operation codes that reference them.
- If a name conflict is detected for a user subroutine, it will not be renamed and it will be added to a list contained in the merge log file. The log will also be displayed on the Code Merge Results window. The merge log file will be placed in the project directory, with a filename of `projectname.mrg` where `projectname` is the name of your project. This file will be overwritten if more than one merge is performed for the same project. The file is not automatically appended. The following example contains a listing of a sample merge file.

The following parts were copied to the target project:

<table>
<thead>
<tr>
<th>Source Name</th>
<th>Target Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEARCHW:CAN000023</td>
<td>SEARCHW:CAN000023</td>
</tr>
<tr>
<td>SEARCHW:SEARCHW</td>
<td>SEARCHW:SEARCHW</td>
</tr>
<tr>
<td>SEARCHW:SEARCHGB</td>
<td>SEARCHW:SEARCHGB</td>
</tr>
<tr>
<td>SEARCHW:STX00071</td>
<td>SEARCHW:STX00071</td>
</tr>
<tr>
<td>SEARCHW:TITLECB</td>
<td>SEARCHW:TITLECB</td>
</tr>
<tr>
<td>SEARCHW:STX00073</td>
<td>SEARCHW:STX00073</td>
</tr>
<tr>
<td>SEARCHW:STX00074</td>
<td>SEARCHW:STX00074</td>
</tr>
<tr>
<td>SEARCHW:STX00075</td>
<td>SEARCHW:STX00075</td>
</tr>
<tr>
<td>SEARCHW:CATCB</td>
<td>SEARCHW:CATCB</td>
</tr>
<tr>
<td>SEARCHW:DIRCt</td>
<td>SEARCHW:DIRCt</td>
</tr>
<tr>
<td>SEARCHW:ACTORCB</td>
<td>SEARCHW:ACTORCB</td>
</tr>
<tr>
<td>SEARCHW:SEARCHPB</td>
<td>SEARCHW:SEARCHPB</td>
</tr>
<tr>
<td>SEARCHW:CANCELSEPB</td>
<td>SEARCHW:CANCELSEPB</td>
</tr>
<tr>
<td>SEARCHW:HELPBP</td>
<td>SEARCHW:HELPBP</td>
</tr>
<tr>
<td>SEARCHW:STX00082</td>
<td>SEARCHW:STX00082</td>
</tr>
</tbody>
</table>
The following help panels were copied to the target project:

<table>
<thead>
<tr>
<th>Source Name</th>
<th>Target Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>24.SEARCHW</td>
<td>88.SEARCHW</td>
</tr>
<tr>
<td>79.SEARCHPB</td>
<td>99.SEARCHPB</td>
</tr>
<tr>
<td>80.CANCELSEPB</td>
<td>100.CANCELSEPB</td>
</tr>
<tr>
<td>81.HELPPB</td>
<td>101.HELPPB</td>
</tr>
</tbody>
</table>

Merging Source Code:

<table>
<thead>
<tr>
<th>Action Subroutine</th>
<th>CATW+CLOSE+CATW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Renaming To</td>
<td>SEARCHW+CLOSE+SEARCHW</td>
</tr>
<tr>
<td>Action Subroutine</td>
<td>TITLECB+CREATE+SEARCHW</td>
</tr>
<tr>
<td>Action Subroutine</td>
<td>DIRCB+CREATE+SEARCHW</td>
</tr>
<tr>
<td>Action Subroutine</td>
<td>ACTORCB+CREATE+SEARCHW</td>
</tr>
<tr>
<td>Action Subroutine</td>
<td>TITLECB+ENTER+SEARCHW</td>
</tr>
<tr>
<td>Action Subroutine</td>
<td>CATCB+SELECT+SEARCHW</td>
</tr>
<tr>
<td>Action Subroutine</td>
<td>DIRCB+SELECT+SEARCHW</td>
</tr>
<tr>
<td>Action Subroutine</td>
<td>ACTORCB+SELECT+SEARCHW</td>
</tr>
<tr>
<td>Action Subroutine</td>
<td>CANCELSEPB+PRESS+SEARCHW</td>
</tr>
<tr>
<td>Action Subroutine</td>
<td>SEARCHPB+PRESS+SEARCHW</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Message</th>
<th>*MSG0001 -&gt; *MSG0003</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Subroutine</td>
<td>WRTBRSFSR</td>
</tr>
<tr>
<td>User Subroutine</td>
<td>CASECAT</td>
</tr>
<tr>
<td>User Subroutine</td>
<td>CKCRITERIA</td>
</tr>
<tr>
<td>User Subroutine</td>
<td>DSPBROWSE</td>
</tr>
</tbody>
</table>

| Message          | *MSG0001 -> *MSG0003 |
| User Subroutine  | BRACTION              |
| User Subroutine  | BRCHILDREN            |
| User Subroutine  | BRSCIFI               |
| User Subroutine  | BRCOMEDY              |
| User Subroutine  | BRHORROR              |
| User Subroutine  | BRWESTERN             |
| User Subroutine  | BRROMANCE             |
| User Subroutine  | BRCLASSIC             |

The following messages were copied to the target project:

<table>
<thead>
<tr>
<th>Source Message</th>
<th>Target Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

Technical description of the following parts were copied to the target project:

<table>
<thead>
<tr>
<th>Source Name</th>
<th>Target Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEARCHW:SEARCHPB</td>
<td>SEARCHW:SEARCHPB</td>
</tr>
<tr>
<td>SEARCHW:CANCELSEPB</td>
<td>SEARCHW:CANCELSEPB</td>
</tr>
<tr>
<td>SEARCHW:HELPPB</td>
<td>SEARCHW:HELPPB</td>
</tr>
</tbody>
</table>
The following rules apply to the resolution of part name conflicts:

- The merged part or window will be renamed.
- If a window gets renamed all of the parts contained in it will inherit the new window name.
- Action subroutines linked to a renamed window or part will be renamed and relinked, if they follow the standard naming format.
- Calculation Specifications containing a GETATR or SETATR operation code which refer to a renamed part will be changed.
- The merge process tries to correct part references in the code being merged to address part name changes.

User messages that are used in merged parts are also copied. The following rules apply:

- If there is a name conflict, for a user message, a rename will take place.
- References to renamed user messages will be updated.
Chapter 25. Vendor Plugins

Plugins are applications that are written by third party developers and are created to provide additional functionality to VisualAge RPG. A wide variety of tasks could be automated such as the insertion of a line of code to set the value of an attribute selected from a list for a named part; or a procedure which allows programmers to print their RPG source formatted to include page headings and footers.

Adding a Vendor Plugin

To add a plugin that you have either created or obtained from a third party developer, do the following:
1. Click on the Vendor pull-down menu.
2. Highlight the Plugins item to reveal the plugins submenu.
3. Click on the Add plugin... command to open the Add plugin window.
4. Select the vendor plugin that you would like to add from the files displayed in the window. The files that are displayed have a .pi2 extension.

Invoking a Vendor Plugin

When you have added a plugin to the VisualAge RPG GUI designer, you then have to invoke the function. The most simple method of doing this is to select the plugin menu item that was defined by the vendor. The menu item can either be on the Vendor pull-down menu, or on the Selected pull-down menu or from a part’s popup menu. In some cases the plugin will not have a menu item defined, and it will be necessary to invoke this plugin external to the GUI designer (to see an example of this method of invocation, see the LPXSAMP sample plugin). As an example, add one or more of the vendor plugin samples provided with VisualAge RPG. To invoke these plugins, do the following:
1. Click on the Vendor pull-down menu.
2. Highlight the Plugins item to reveal the plugins submenu.
3. Highlight the menu item that is added along with the plugin. The name of this menu item is variable, because it is created by the developer of the plugin. Depending on the developer/vendor this may produce another submenu. If this is the case, go to step 4, otherwise, click the menu item to invoke the plugin.
4. Click on the appropriate menu item from the submenu to invoke the plugin.

Managing Vendor Plugins

If you have vendor plugins functioning in VisualAge RPG, you may want to find out information about a plugin, such as the developer, the developer’s description of what it is that the plugin does, or the dll that is associated with the plugin. This information can be obtained in the Manage plugins window.

To open the Manage plugins window, do the following:
1. Click on the Vendor pull-down menu.
2. Highlight the Plugins item to reveal the plugins submenu.
3. Click on the Manage plugins... command to open the Manage plugins window.
Chapter 26. Creating Plugins

You can create a plugin to address your own specific needs, or as a VisualAge RPG supplement to be shipped to other programmers who may benefit from your plugin. Plugins can be created using one of either VisualAge RPG, VisualAge C++, or REXX. The following steps outline the plugin creation process using VisualAge RPG code. Additional sections will follow that will address the exceptions and additional guidelines for creating plugins with VisualAge C++ or REXX.

Creating Plugins Using VisualAge RPG

There are two components required to create a plugin, the executable file and the plugin informations file. The executable file consists of compiled source code which performs the desired function. The plugin information file (.plg) acts as an interface between the GUI designer and the executable file. It contains important information such as the definitions for the pull-down menu items that are added to the GUI designer, as well as the actual call to the executable file.

It does not matter which of the two files that you create first, as both must be present for the plugin to function. This example starts with the .plg file.

Creating the .plg file

The .plg file acts as the interface between the plugin and the designer. It is an ASCII file which contains organizational information such as the location of the necessary DLL, the location of the associated help files, and the name of the plugin itself and the vendor. The .plg file also contains the information that is needed to interface the plugin with the designer. This includes the command line string, the desired text for the menu items, and the key combination that serves as an accelerator. The following list of keywords explains the various parameters, including the rules which apply, whether the parameter is required or optional, and the information that you need to provide. An example follows this list of keywords which illustrates how this file appears when it is finished.

Note: The spaces between the keyword and the parameter value that is being used for the value in your plugin are arbitrary. The spaces used in the examples are designed for presentation only.

Alternate Paths

This is a string that indicates what relative paths should be used when loading the plugin DLL described below. This field is optional.

The string takes the form:

"&1\relativePath1;&1relativePath2;...;&1\relativePathN;"

where &1\ will be converted to the full path to the .plg file. This means that if you have a suite of plugins that get installed in a directory such as c:\myplugins, and they all link to a common DLL: c:\myplugins\plugutil\plugutil.dll, then if each individual plugin was located in c:\myplugins\plugN\plugN.plg, you could specify the alternate path: &1\..\util1; which would be converted to:

c:\myplugins\plugN..\\util1; and be appended to the PATH before your DLL is loaded. The quotation marks are required if you choose to use this keyword.
Instead of a string, you may also use a resource id to point to a path specified in an external source.

**DLL_Names**
The value for this field is a string that defines the name of the required DLL files for this plugin. This is an optional field, but if used, there must be a DLL named that is used by the plugin. Additionally, an mri.dll may be named. The mri.dll name is optional, but cannot be included without a plugin.dll. If both are included, they are delimited by a space. The name of the DLL files can include the path relative to the location of the plugin files.

plugin.dll  This is the name of the DLL containing the code for the plugin. If the plugin is not a function call within a DLL, then this may be omitted.

mri.dll    If mri (translatable strings) is in a separate DLL, name that dll here. The DLLS are specified first so that any strings to follow can be contained in the DLLs.

Whenever a string is required further on in the .plg file, if a string enclosed in quotation marks is not specified, then the value given is assumed to be a string resource id in either mri.dll or plugin.dll

**Vendor_Name**
This is either the name of the vendor enclosed in double quotation marks, or the resource id of a string. This field is optional, but recommended. An example of this string would be:

Vendor_Name: "Plug-Me-In Inc."

**Plugin_Name**
This is either the name of the plugin enclosed in quotation marks, the resource id of a string. This field is optional, but recommended. An example of this field using a string is:

Plugin_Name: "Who Am I?"

**Help_File**
This field is optional, and identifies the Windows .hlp file used for displaying help for the menu items. It is a string, not enclosed in quotation marks, which includes the relative path to the help file.

**Unloading_Function**
This field is optional. The Unloading_Function field cannot be used in conjunction with the Unloading_Command_Line field. This field is only used if there is information or a display element that needs to be modified or removed after the plugin is finished or removed. This string, enclosed in quotation marks, designates the function which will be used. This function must be contained in the DLL that accompanies the plugin.

The Unloading_Function is the name of the function to be called when the plugin is about to be unloaded. It has the following signature:
unsigned long
unloadFunctionName(
    const char* ppluginPath_,
    const char* ppluginStub_,
    const char* pdllPath_,
    const char* builderId_,
    int remove_)

where the parameters are:

ppluginPath_
The fully qualified path to the plugin being invoked, up to and including the final backslash.

ppluginStub_
The rest of the plugin filename (eg. "myplug.plg").

pdllPath_
The fully qualified path to the DLL containing VARPG’s exposed methods.

builderId_
A string used by VARPG to identify the builder and which is used by the plugin when communicating with the builder.

remove_
  0 The builder is shutting down.
  1 The user has requested that the plugin be removed altogether. If this is the case, the plugin should remove any information it has stored in the registry at this time.

Return value
  0 success
  1 failure, or refusal

If the plugin returns a 1 from this function, the builder may prompt the user with the option to forcibly remove the plugin by unloading its DLL. If this occurs, it is possible that the plugin may then crash the designer. If this occurs, restart the designer.

Unloading_Command_Line
As mentioned above, this field cannot be used at the same time as the Unloading_Fункция field.

When this option is used, you provide a string to be executed as though it were being run from the command line. For example, you could start Netscape by specifying the string: netscape.exe

This method allows you to obtain the same set of parameters that would be available to a function in a DLL. This is accomplished through the definition of substitution variables. Whenever a ’&0’, ’&1’, ’&2’, ’&3’, ’&4’ or ’&5’ is found in the string specified, it is replaced with the following:

&0 ppluginPath_
&1 ppluginStub_
&2 pdllPath_
&3 builderId_
&4 remove_
path to the GUI Designer’s root directory

**IBM_PluginInterface | PluginInterface**

This is an advanced feature that is not required. This field allows you to expose your plugin as a programmable component. You cannot use these two options in the same .plg file. If you do not have a reason to specify either of these interfaces, do not do so.

When either one or the other of these options is used, the function name specified is used when other plugins interact with this plugin through a target/command/parameters interface.

The signature of the function should be:

```c
unsigned long __stdcall IBMtargetCommandFunction(
    const IString& pluginPath_,
    const IString& dllPath_,
    const IString& builderId_,
    const IString& target_,
    const IString& command_,
    IString& arguments_);
```

for the IBM_ style function, where arguments_ is used for input and output.

For the non-IBM style function, the signature should be:

```c
unsigned long __stdcall targetCommandFunction(
    const char* ppluginPath_,
    const char* pdllPath_,
    const char* pbuilderId_,
    const char* ptarget_,
    const char* pcommand_,
    const char* parguments_,
    char** ppreturnString_);
```

In this case, if there is a return string, ppreturnString_ should be allocated memory by the plugin using GlobalAlloc( GMEM_FIXED, [bufferSize]) so that VARPG can deallocate the memory when it is finished with it. If there is no return string required, this parameter can be ignored. The following is an example of this command:

```c
{
    IString returnString = ...;
    ...
    *ppreturnString = GlobalAlloc( returnString. length() + 1);
    strcpy( *ppreturnString, returnString);
}
```

To see an example of a plugin that supports the IBM_PluginInterface feature, look at the LPEXSAMP sample provided in the x:\...\WDSC\samples\vndplugs\lpexsamp directory (where x corresponds to the letter of the drive onto which you installed VisualAge RPG).

**Begin_Details ... End_Details**

Between these tags, enter anything that you wish to show the user when this plugin’s information is displayed in the **Manage plugins** dialog. You may want to give a brief description of the purpose and use of your plugin. You can enter text here, or you can use the String/Resid form described for the mri.dll. This field is optional, but is highly recommended.
**Function Name**

This is the name of the function that should be called in the plugin.dll when the menu item is triggered. You can either use this field or the **Command Line** field. You may not use both. It should have the following signature:

```c
unsigned long
pluginFunctionName(
    const char* ppluginPath_,
    const char* pdllPath_,
    const char* builderId_,
    unsigned long menuContextId_,
    const char* partsIds_);
```

where the parameters are:

- **ppluginPath**
  - Same meaning as for unloadFunctionName().

- **pdllPath**
  - Same meaning as for unloadFunctionName().

- **builderId**
  - Same meaning as for unloadFunctionName().

- **menuContextId**
  - An unsigned long value representing the type of menu from which this plugin is invoked. This value determines the meaning of partsIds. The possible values are:
    1. The plugin was invoked from the menu bar, (that is, this is a project-scoped plugin) and partsIds is an empty string.
    2. The plugin was invoked for a single selected part, (that is, this is a single-selection-scoped plugin) and partsIds contains the identifier of the selected part.
    4. The plugin was invoked for a group of jointly-selected parts (that is, this is a multiple-selection-scoped plugin) and partsIds is a string containing the blank delimited set of part identifiers of the selected parts.
    8. The plugin is invoked when the GUI designer is started.

- **partsIds**
  - This is a string representing the part or parts, to which the function call should apply as indicated by the menuContextId. Within partsIds, each individual part identifier is a sequence of dot-separated unsigned long values (eg. 432.5632.612) which represent the child-parent hierarchy of the given part. In the stated example, 612 is the ID of the part, 5632 is the ID of its parent, and 432 is the ID of the parent’s parent.

**Command Line**

When this option is used, you provide a string to be executed as though it was being run from the command line. For example, you could start Netscape by specifying the string: `netscape.exe`.

This method allows you to obtain the same set of parameters that would be available to a function in a DLL. This is accomplished through the definition of substitution variables. Whenever a `&0`, `&1`, `&2`, `&3`, `&4` or `&5` is found in the string you specify, it is replaced with the following:

- **&0** ppluginPath_
&1 pdllPath_
&2 builderId_
&3 menuContextId_
&4 partsIds_
&5 path to the GUI Designer’s root directory.

Using the Netscape example above, suppose the vendor provided an HTML file with its plugin, and this particular menu item is intended to display that HTML file. Assume also that the plugin file is located in d:\vendor\plugins, and the HTML file is d:\vendor\plugins\htmlsrc\plugpage.html. To have the plugin display this web page, the definition of the command line might be as follows:

gnetscape &8htmlsrc\plugpage.html

Which would expand to, and be run as:

gnetscape d:\vendor\plugins\htmlsrc\plugpage.html

**Menu_Name**

This is either a string or a string resource id that indicates what the menu item should be. These strings have the format:

submenu1/submenu2/.../submenuN/menuitem

where submenu1 through submenuN are optional submenus.

For example:

Menu_Name: "Plug-Me-In Inc./Who am I?"

where "Plug-Me-In Inc." is the submenu and "Who am I?" is the menu item.

**Menu_Info_Strings**

This is a list of strings or string ids that are associated with the corresponding submenus/menuitem as specified in **Menu_Name**. The association works backwards.

For example, if you specify one submenu and one menu item in **Menu_Name**, but only specify one string in **Menu_Info_Strings**, then the string you specify in **Menu_Info_Strings** will be associated with the menu item, and the submenu will be ignored. (It is possible that a previous menu item addition defined an info-area string for the given menu item.)

**Supported_Menus**

As mentioned in **Function_Name**, the menuContextId_ indicates a type of menu. **Supported_Menus** indicates the menus to which this particular entry should be added.

**Help_Id**

If a help file has been specified, and there is help associated with this command, provide the help id here in the ulong_panel parameter of **Help_Id**. If the optional_force_window_parameter is provided and is a non-zero value, then the help will be displayed in a full help window rather than in the default context popup. This field takes the form of:

Help_Id: ulong_panel optional_force_window_parameter
and a sample of how this is actually coded is:

```
Help_Id: 1000 1
```

**Accelerator**

This optional field specifies the accelerator to be associated with the item. It consists of one of F1 through F12, followed by one or more modifiers (SHIFT, ALT, CONTROL)

**Note:** <F1/10>, <Alt-F5/7/8/9/10>, and <Shift-F9/10> are already reserved by the designer, and if specified will be ignored.

To use this function, provide the following information:

```
Accelerator: [F1 | F2 | F3 | ... | F12] [SHIFT] [CONTROL] [ALT]
```

This field, when used will look like the following sample:

```
Accelerator: F8 Shift
```

**End_of_Definition**

This indicates to the parser that a function definition has ended and that a new one may begin.

**Template for .plg file and sample**

When using the previously described fields to create a .plg file for the GUI designer you need to follow this format:

**Note:** There must be at least one Function_Name or Command_Line definition. There is no limit on the maximum number allowed.
Note:

- All filenames are given relative to the location of the .plg file.
- The unloaders are optional, but if you choose to have one, you can only have one or the other.
- Either the Function Name or Command Line may be specified.

The following is a specific example of a simple .plg file. There are some plugin samples provided with VisualAge RPG. The files can be found in the X:\...\WDSC\samples\vndplugs\ directory on the workstation where VisualAge RPG is installed (where X corresponds to the drive letter).
Creating the .EXE file

To create an .EXE file for the GUI designer you need to be aware of the following:

When using VisualAge RPG to create plugins, you use the *component part to interact with the designer. By setting values for the PlugDLL, PlugId, PlugCmd, PlugRC, and PlugResult attributes, all of the necessary information can be communicated between the designer and the plugin.

To create a working plugin you must establish proper communication by providing the designer with the following information:

**builderId**
- This is the same id that was provided by the designer when the plugin was invoked.

**target**
- This is a string representing the aspect of the designer you wish to interact with.

**command**
- This is the specific action you wish the designer to take.

**parameters**
- Any arguments required by the command.

**Note:** In a VARPG program, builderId_ corresponds to *component’s PlugId attribute. In order to make a call to the Plugin interface, you must first have set PlugId and also PlugDLL. PlugDLL indicates to the VARPG runtime where the dll containing the builder’s plugin interface is located. When issuing a command, first concatenate the values of target_, command_ and parameters_ using blanks for delimeters and then use the result to set *component’s PlugCmd attribute.

You get a result and an error code in return. In the case of a function call, one of the parameters is set to contain the result string, and the returned unsigned long value contains the error code. The following are some basic return codes common to all commands. Any additional error codes will be defined in the respective table of Targets and Commands.
Return Code

Meaning

0  All went well, and the command was handled.
1  The target was not recognized.
2  The target did not recognize the command.
4  The builder can not be found.
5  Some unknown error occurred and the results of the command can not be trusted.

Targets/Commands and the associated Return Values

A list of the valid targets and commands follows, along with the semantics of their parameters and returned values.

Table 13. Target: Project

<table>
<thead>
<tr>
<th>Command</th>
<th>Parameter(s)</th>
<th>Meaning/Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build</td>
<td>[win32</td>
<td>java]</td>
</tr>
<tr>
<td></td>
<td>Default: win32</td>
<td></td>
</tr>
<tr>
<td>BuildOptions</td>
<td>[win32</td>
<td>java]</td>
</tr>
<tr>
<td></td>
<td>Default: win32</td>
<td></td>
</tr>
<tr>
<td>CurroredPart</td>
<td>none</td>
<td>Returns a string containing the partId for the currently cursored part. If no design window is open, or if no open design window is the active design window, then this is an empty string.</td>
</tr>
<tr>
<td>ExpandAll</td>
<td>[1]</td>
<td>If this parameter=1, then the entire treeview is expanded; otherwise, it is collapsed.</td>
</tr>
<tr>
<td>ForceOpen</td>
<td>[projectName]</td>
<td>Opens the specified project without checking whether or not the current project needs to be saved. Returns a 1 indicating that the ForceOpen was successful, or a 0 to indicate that the ForceOpen was unsuccessful.</td>
</tr>
<tr>
<td>Get</td>
<td>ProjectDir</td>
<td>Returns the current project’s root directory.</td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------------------------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>ProjectFileName</td>
<td>Returns the current project’s fully qualified .IVG file name.</td>
</tr>
<tr>
<td></td>
<td>ProjectTargetName</td>
<td>Returns the name of the file that will be generated when the project is built (for example, &quot;myproj.exe&quot;).</td>
</tr>
<tr>
<td></td>
<td>ProjectTitle</td>
<td>Returns the title of the current project.</td>
</tr>
<tr>
<td></td>
<td>ProjectFileStub</td>
<td>Returns the filename (minus the extension) of the names of the current project (for example, &quot;myproj&quot;).</td>
</tr>
<tr>
<td>IsSaveRequired</td>
<td>none</td>
<td>Returns 1 to indicate that the project has been modified, 0 to indicate that it hasn’t been touched since being opened.</td>
</tr>
<tr>
<td>IsTemporary</td>
<td>none</td>
<td>Returns a 1 if this is an unnamed project, 0 otherwise.</td>
</tr>
<tr>
<td>MostRecentlyUsed</td>
<td>n</td>
<td>Returns the n’th most recently opened project, where n is equal to or greater than 1. Returns an empty string if the index is out of bounds.</td>
</tr>
<tr>
<td>Open</td>
<td>projectName</td>
<td>Checks to see if the user wants to save the project before opening another project. Returns 1 if a project was successfully opened, and 0 otherwise.</td>
</tr>
</tbody>
</table>
| PartId             | partName [windowName \[0\|1\|2\]] | Returns a partId when given a part name. If you specify a windowName, then this will either return the id of the part, or if there is no such part, it will return an empty string. If you specify a searchType, then the following rules will be used when searching for a part with the given name:  
  0 (default) - Return the first part with the given name.  
  1 - Return all parts with the given name.  
  2 - If there is only one part with this name, return it; otherwise return nothing. |
| PromptedSave       | none                              | Prompts the user for a project name and then saves the project. Returns a 1 to indicate that a successful save took place; otherwise it returns a 0. |
Table 13. Target: Project (continued)

<table>
<thead>
<tr>
<th>PromptExisting</th>
<th>none</th>
<th>Prompts the user for an existing project. Returns the project filename.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run</td>
<td>none</td>
<td>Runs the current project.</td>
</tr>
<tr>
<td>Save</td>
<td>none</td>
<td>Saves the current project.</td>
</tr>
<tr>
<td>SaveAs</td>
<td>projectName</td>
<td>Saves the current project with the specified project name.</td>
</tr>
<tr>
<td>SelectedParts</td>
<td>none</td>
<td>Returns a string containing the partIds of all of the parts currently selected in the project’s treeview.</td>
</tr>
</tbody>
</table>

Table 14. Target: PartClass

<table>
<thead>
<tr>
<th>Command</th>
<th>Parameter(s)</th>
<th>Meaning/Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllAttributes</td>
<td>ClassName</td>
<td>Returns a list of the attributes supported by the given ClassName.</td>
</tr>
<tr>
<td>AllClasses</td>
<td>none</td>
<td>Returns a list of all available part classes. Each list item is embedded in double quotation marks since some may consist of multiple words (for example, in the case of vendor parts).</td>
</tr>
<tr>
<td>AllEvents</td>
<td>ClassName</td>
<td>Returns all registered events for the given class.</td>
</tr>
<tr>
<td>IBMClasses</td>
<td>none</td>
<td>Returns a list of all IBM supplied, non-vendor part classes.</td>
</tr>
<tr>
<td>IconDll</td>
<td>ClassName</td>
<td>Returns the path of the dll containing the icon that represents the given part class.</td>
</tr>
<tr>
<td>IconId</td>
<td>ClassName</td>
<td>Returns the resource id of the icon (in the dll given by “IconDll”) for the given class.</td>
</tr>
<tr>
<td>IsType</td>
<td>TypeName</td>
<td>Returns 1 to indicate that the ClassName given class is of the given type; otherwise it returns a 0. Possible values for TypeName include: Frame, Canvas, MenuBar, NoteBook, NoteBookPage, PopUpMenu, SubMenu, MenuItem, Subfile and SubfileEntryField.</td>
</tr>
<tr>
<td>VendorClasses</td>
<td>none</td>
<td>Returns a list of all available vendor part part classes.</td>
</tr>
</tbody>
</table>

Table 15. Target: Part

<table>
<thead>
<tr>
<th>Command</th>
<th>Parameter(s)</th>
<th>Meaning/Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>Parameters</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ActionSubroutine</td>
<td>partId, eventName</td>
<td>Locates the linked action subroutine, eventName or creates a link and scans to it if it does not already exist.</td>
</tr>
<tr>
<td>ActionSubroutines</td>
<td>partId</td>
<td>Returns a list of action subroutines defined for this part.</td>
</tr>
<tr>
<td>AllEvents</td>
<td>partId</td>
<td>Returns all registered events for the given part.</td>
</tr>
<tr>
<td>Children</td>
<td>[partId]</td>
<td>Returns a list of blank delimited partIds enumerating all of the specified part’s children. If no partId is provided, a list of all of the project’s windows is returned.</td>
</tr>
<tr>
<td>ClassName</td>
<td>partId</td>
<td>Returns the classname of the indicated part.</td>
</tr>
<tr>
<td>CreateChild</td>
<td>partId, className</td>
<td>Creates a part of the given class name className as a child of the specified part. Returns the partID of the newly created part.</td>
</tr>
<tr>
<td>CreateFrame</td>
<td>ClassName</td>
<td>Creates a part of the specified class. The class must be a frame based part. Returns the partID of the newly created part.</td>
</tr>
<tr>
<td>DataInfo</td>
<td>dataType, dataLength, decimalPlaces</td>
<td>Returns a string of three numbers, each separated by a blank. Applicable parts include entry field, static text, and subfile entry field.</td>
</tr>
<tr>
<td>ExtraColorAreas</td>
<td>partId</td>
<td>Returns a count of the color areas that this part supports if this part supports color areas other than foreground and background.</td>
</tr>
<tr>
<td>FileName</td>
<td>partId</td>
<td>Obtains the file name set for this part. If the part doesn’t support files, the return value is an empty string.</td>
</tr>
<tr>
<td>GetColor</td>
<td>partId, [x]</td>
<td>Get the color for the specified area. Returns a string with 4 blank delimited numbers:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>useDefault - (0 or 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>redMix - (0 - 255)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>greenMix - (0 - 255)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>blueMix - (0 - 255)</td>
</tr>
</tbody>
</table>
Table 15. Target: Part  (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetFont</td>
<td>partId [x]</td>
<td>Gets the part’s font. Returns an empty string if the font is not supported. Otherwise, the first part of the returned string is a 0 or 1 indicating whether or not the default font is being used, the second word of the string is a point size, the third word of the string is a number which (ORs) together applicable font styles from the following: 1 - bold 2 - italic 4 - underscore 8 - strikeout 16 - outline The rest of the string is the font’s facename.</td>
</tr>
<tr>
<td>GetRect</td>
<td>partId</td>
<td>Gets the coordinates (x y width height) of the part relative to its parent.</td>
</tr>
<tr>
<td>HasFile</td>
<td>partId</td>
<td>Returns a 1 to indicate that the part supports a file (for example, canvas, image, media, ...); otherwise it returns a 0.</td>
</tr>
<tr>
<td>IsColorArea</td>
<td>partId [x]</td>
<td>Returns a 1 to indicate that the color area is supported; otherwise it returns a 0.</td>
</tr>
<tr>
<td>IsFontArea</td>
<td>partId [x]</td>
<td>Returns a 1 to indicate that the font area is supported; otherwise it returns a 0.</td>
</tr>
<tr>
<td>Label</td>
<td>partId</td>
<td>Returns the part’s label (if it has one).</td>
</tr>
<tr>
<td>LinkedEvents</td>
<td>partId</td>
<td>Returns a list of events for which this part has action links.</td>
</tr>
<tr>
<td>Name</td>
<td>partId</td>
<td>Returns the name of the part as shown in the treeview and settings notebook.</td>
</tr>
<tr>
<td>OpenDesignWindow</td>
<td>partId [1]</td>
<td>When set to 1, opens and sets focus to the design window to which the indicated part belongs. If set to 0, then the design window is closed instead.</td>
</tr>
<tr>
<td>OpenPart</td>
<td>partId</td>
<td>Either opens the part’s settings notebook, or if the part is a frame, opens the part’s corresponding design window.</td>
</tr>
<tr>
<td>OpenSettings</td>
<td>partId</td>
<td>Opens the part’s settings notebook.</td>
</tr>
<tr>
<td>Method</td>
<td>Parameters</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SetColor</td>
<td>partId, colorArea, useDefault, redMix, greenMix, blueMix</td>
<td>Sets the color for the given area. See GetColor for more information for the allowed values for each of the parameters.</td>
</tr>
<tr>
<td>SetCursered</td>
<td>partId</td>
<td>If the part’s design window is open the part becomes the active part but selection state doesn’t change. If the design window is not open, this has no effect.</td>
</tr>
<tr>
<td>SetDataInfo</td>
<td>partId, dataType, dataLength, decimalPlaces</td>
<td>Sets the data properties of a part. Does not update the Properties Notebook of a part that is already open, or in use. The programmer must ensure new values are compatible with existing ones already defined for the part. Applicable parts include entry field, static text, and subfile entry field.</td>
</tr>
<tr>
<td>SetFileName</td>
<td>partId, newFileName</td>
<td>Sets the file name for this part. Nothing happens if the part doesn’t support files.</td>
</tr>
<tr>
<td>SetFont</td>
<td>partId, fontArea, setToDefault, pointSize, styles, faceName</td>
<td>Sets the part’s font.</td>
</tr>
<tr>
<td>SetLabel</td>
<td>partId, newLabel</td>
<td>Attempts to set the part’s label. If the given label is invalid, an error message is shown. Returns 1 to indicate that the label was set; otherwise it returns 0.</td>
</tr>
<tr>
<td>SetName</td>
<td>partId, newName</td>
<td>Attempts to set the name of the part. If the set fails, an error message is displayed. If the part has action links associated with it, a message is displayed asking the user if they wish to break the links. Returns 1 to indicate success, or 0 to indicate failure.</td>
</tr>
<tr>
<td>SetRect</td>
<td>partId, x, y, width, height</td>
<td>Sets the coordinates (x y width height) of the part relative to its parent.</td>
</tr>
</tbody>
</table>
Table 15. Target: Part (continued)

<table>
<thead>
<tr>
<th>Command</th>
<th>Parameter(s)</th>
<th>Meaning/Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetSelected</td>
<td>partId [0</td>
<td>1] [0</td>
</tr>
<tr>
<td>SetStyles</td>
<td>partId styles extendedStyles [0</td>
<td>1]</td>
</tr>
<tr>
<td>Styles</td>
<td>partId [0</td>
<td>1]</td>
</tr>
<tr>
<td>Zoom</td>
<td>partId [0</td>
<td>1]</td>
</tr>
</tbody>
</table>

**Note:** A part will have a Foreground (1) color area, a Background (0) color area or no color area, or it will have Extra color areas. The window part, for example has no color areas. The Checkbox has foreground and background color areas. The graph has Extra color areas. Therefore, 0 and 1 only necessarily indicate background and foreground color if the part has no Extra colors.

The following require that the source file be open in LPEX.

Table 16. Target: Subroutine

<table>
<thead>
<tr>
<th>Command</th>
<th>Parameter(s)</th>
<th>Meaning/Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeleteActionSub</td>
<td>routineName</td>
<td>Deletes the action subroutine with the given name.</td>
</tr>
<tr>
<td>DeleteUserSub</td>
<td>routineName</td>
<td>Deletes the user subroutine with the given name.</td>
</tr>
</tbody>
</table>
### Table 16. Target: Subroutine (continued)

<table>
<thead>
<tr>
<th>UserSubroutine</th>
<th>routineName</th>
<th>If the subroutine does not exist, creates a user subroutine with the given name and locates it in the source file. If it does exist, it is located in the source file.</th>
</tr>
</thead>
<tbody>
<tr>
<td>UserSubroutines</td>
<td>none</td>
<td>Returns a list of user subroutines.</td>
</tr>
</tbody>
</table>

### Table 17. Target: Grid

<table>
<thead>
<tr>
<th>Command</th>
<th>Parameter(s)</th>
<th>Meaning/Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>IsOn</td>
<td>none</td>
<td>Returns a 1 if the grid is currently on, and a 0 if it is off.</td>
</tr>
<tr>
<td>TurnOn</td>
<td>[0</td>
<td>1]</td>
</tr>
</tbody>
</table>

### Table 18. Target: Lpex

<table>
<thead>
<tr>
<th>Command</th>
<th>Parameter(s)</th>
<th>Meaning/Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DoIt</td>
<td>Any_LPEX_command</td>
<td>Passes your parameters to LPEX's &quot;DoIt&quot;.</td>
</tr>
<tr>
<td>IsSourceFileOpen</td>
<td>none</td>
<td>Returns a 1 to indicate that the source file is open; otherwise it returns a 0.</td>
</tr>
<tr>
<td>OpenSourceFile</td>
<td>none</td>
<td>Opens the source file in LPEX.</td>
</tr>
<tr>
<td>Query</td>
<td>Any_LPEX_query</td>
<td>Passes your parameters to LPEX's &quot;Query&quot;.</td>
</tr>
</tbody>
</table>

### Table 19. Target: Plugin

<table>
<thead>
<tr>
<th>Command</th>
<th>Parameter(s)</th>
<th>Meaning/Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddPlugin</td>
<td>filename</td>
<td>Attempts to add the specified plugin. Returns &quot;0&quot; if successful.</td>
</tr>
<tr>
<td>get</td>
<td>PluginCount</td>
<td>Returns the number of plugins currently installed.</td>
</tr>
<tr>
<td></td>
<td>Plugin oneBasedIndex</td>
<td>Returns the fully qualified path of the oneBasedIndex’th plugin. If n is less than 1 or greater than the number of plugins, a null string is returned.</td>
</tr>
<tr>
<td>Plugins</td>
<td></td>
<td>Returns a list of the fully qualified paths of all plugins.</td>
</tr>
<tr>
<td>InvokePlugin</td>
<td>oneBasedIndex target command parameters</td>
<td>Invokes the plugin using a target/command interface.</td>
</tr>
</tbody>
</table>

### Table 20. Target: Registry

<table>
<thead>
<tr>
<th>Command</th>
<th>Parameter(s)</th>
<th>Meaning/Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeleteKey</td>
<td>key</td>
<td>This command will delete the given key from the registry (including any subkeys).</td>
</tr>
</tbody>
</table>
Table 20. Target: Registry (continued)

<table>
<thead>
<tr>
<th>Command</th>
<th>Key Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get</td>
<td>key [&quot;defaultValue&quot;]</td>
<td>If the key does not exist, the return value is defaultValue, otherwise it is the value of the key in the registry. When entering substitute the 'defaultValue' string with the string of your choice. The double quotation marks are required.</td>
</tr>
<tr>
<td>GetRect</td>
<td>key [&quot;x y width height&quot;]</td>
<td>This command will retrieve a rectangle from the registry, and if the element with the given key is not found, the default values supplied will be returned instead. The double quotation marks are required.</td>
</tr>
<tr>
<td>Set</td>
<td>key value</td>
<td>Use this command to set a string value into the registry. No return value.</td>
</tr>
<tr>
<td>SetRect</td>
<td>key &quot;x y width height&quot;</td>
<td>This command will store the given rectangle in the registry using normalized coordinates. The double quotation marks are required.</td>
</tr>
</tbody>
</table>

A note on using the registry commands.

Plugins are strongly urged to use an initial subkey that is likely to be uniquely theirs, so that they don’t interfere with other plugins’ registry entries.

All registry entries made using these commands will be restricted to a common subsection of VARPG’s registry entry, however, it is possible to overlap across plugins.

To avoid such overlaps, plugins could use a variation on the pathname of the .PLG file as the initial subkey as follows:

If the plugin’s pathname is:
"c:\plugins\My_Plugins\myplug.plg",

and the registry entry is to be used to store a window position, then an appropriate key to use for this value would be:
"c__plugins_my_plugins_myplug.plg\Window Position"

(Note that case was eliminated from the path portion of the key, and that the colon and back-slashes were converted to underscores.) The keys and values specified must be enclosed in quotes, since the keys can contain spaces. Thus, if you were trying to set a string value you would use:
Set( "c__plugins_my_plugins_myplug.plg\Some relevant keyname" "The new value." )

Embedded quotes are prefaced with a backslash:
Set( "c__plugins_my_plugins_myplug.plg\Some relevant keyname" "The new \"quoted\"value." )
There are some other commands which apply to (some of) the GUI Designer’s own constituent windows. (For example, the parts catalog)

Applicable Targets:

**MainWindow**
This is the main Gui Designer window.

**Catalog**
The parts catalog.

**DBRefDlg**
The Define Reference Fields window.

**ImportDlg**
The Import Display File window.

**LPEX**
The editor window.

Note that these only apply when the indicated window is open.

**Table 21. Target: GUI Designer constituent windows**

<table>
<thead>
<tr>
<th>Command</th>
<th>Parameter(s)</th>
<th>Meaning/Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetHandle</td>
<td>none</td>
<td>Returns the Windows HANDLE for the given window.</td>
</tr>
<tr>
<td>GetIWindowPointer</td>
<td>none</td>
<td>Returns the IWindow pointer for the given window.</td>
</tr>
<tr>
<td>MoveSizeTo</td>
<td>X Y Width Height</td>
<td>Sets the window’s size and position.</td>
</tr>
<tr>
<td>MoveTo</td>
<td>X Y</td>
<td>Moves the window to position (X, Y).</td>
</tr>
<tr>
<td>Position</td>
<td>none</td>
<td>Returns the position of the window in the form &quot;X,Y&quot;.</td>
</tr>
<tr>
<td>Rect</td>
<td>none</td>
<td>Returns the window’s rectangle in the form &quot;X,Y,Width,Height&quot;</td>
</tr>
<tr>
<td>SetFocus</td>
<td>none</td>
<td>Sets focus to the indicated window.</td>
</tr>
<tr>
<td>SetSize</td>
<td>Width Height</td>
<td>Sets the size of the window.</td>
</tr>
<tr>
<td>ShowSetFocus</td>
<td>none</td>
<td>Shows the window (if it’s not already visible) and sets focus to it.</td>
</tr>
<tr>
<td>Size</td>
<td>none</td>
<td>Returns the size of the window in the form &quot;X Y&quot;.</td>
</tr>
<tr>
<td>NotifyOnClose</td>
<td>Window handle</td>
<td>Specifies which window is to be notified when the GUI Designer is closed.</td>
</tr>
</tbody>
</table>

**Sample Plugin Source Code**
The following is the source code for the plugin that corresponds to the plg file used the in section above.
Program ID . . : WhoAmi

Description . : Sample program to illustrate the Vendor plugin interface of VARPG.

When invoked from the Vendor menu item on the GUI Designer this program will use the plugin interface to gather information about the current project and display it on a window named MAIN.

The following plugin file, WHOAMI.PLG, was specified when adding this plugin to the GUI designer:

// WhoAmi.plg plug in file
Vendor_Name: "Plug-Me-In Inc."
Plugin_Name: "Who Am I?"
Begin_Details: 
Who Am I?
This plug-in will display information about the current project including its directory name and file name.
End_Details:
Command_Line: "d:\myproj\whoami\rt_win32\whoami.exe &1 &2"
Menu_Name: "Plug-Me-In Inc./Who am I?"
Supported_Menus: 1
Accelerator: F7 Shift
End_of_Definition

D Cmd S 255A

C *Entry Plist
C Parm PlugDLL 64
C Parm PlugID 64
Chapter 26. Creating Plugins
Packaging Your Application

The final step in creating the plugin is the compilation of the .EXE file. Select **Build** from the **File** pull-down menu, and then select the platform that you would like to use the plugin on. Once the file is compiled, it is ready for use. Refer back to the instructions for **Adding a Vendor Plugin** in order to add the plugin. From this point, you may use the plugin, or you can do any further testing that may be required.

Considerations when Creating Plugins using VisualAge for C++

The process for creating plugins using VisualAge for C++ is the same as for using VisualAge RPG. The only difference is that when you create the plugin using VisualAge for C++, you do not have the direct use of the *component* part. Instead, to allow VisualAge for C++ programs to be used as plugins, the IBMExecuteVDECommand() function has been provided. Its use is demonstrated in the sample plugin, “TreeSamp”.

If necessary, you can cut and paste the code from the x:\...\WDSC\samples\vndplugs\treesamp (where x corresponds to the letter of the drive onto which you installed VisualAge RPG) and the x:\...\WDSC\samples\vndplugs\plugutil directories in order to create the correct calls to the IBMExecuteVDECommand().

Considerations when Creating Plugins using REXX

The process for creating plugins using REXX is almost identical to that of the VisualAge RPG example. REXX programmers do not have direct access to the GUI designer using the *component* part. To facilitate the use of REXX scripts as plugins, the RexxExecuteVDECommand() function is included amongst this set of functions. An example of how to use this function in a REXX file can be found in the sample plugin, “RexxSamp”.

If necessary, you can cut and paste the code from the x:\...\WDSC\samples\vndplugs\rexxsamp (where x corresponds to the letter of the drive onto which you installed VisualAge RPG) directory in order to create the correct calls to the REXXExecuteVDECommand().
Part 5. Distributing Your Application

Chapter 27, “Packaging Runtime Code and Applications,” on page 415
Describes using the packaging utility.

Chapter 28, “Installing Windows NT/95/98 Runtime Code and Applications,” on page 423
Describes using the installation utility for Windows applications.
Chapter 27. Packaging Runtime Code and Applications

After you build and test your application, you can package it and distribute it to other workstations that have the VisualAge RPG runtime code installed.

This section describes how to package the VisualAge RPG runtime code and VisualAge RPG applications.

Note: If the application will be using an iSeries server other than the one it accessed during the development cycle, all server objects used by the application should also be packaged and restored to the new server. They are not packaged by the VisualAge RPG application packaging utility.

Before You Begin

Make sure that the files needed by your application are stored in the appropriate runtime directory (RT_WIN32 for Windows or RT_JAVA for Java). Some files are automatically placed in the runtime directory after you build your application (for example, the .MSG, .HLP, .DLL, .BND, .RST, and .EXE files); others you will have to put in yourself (for example, any .BMP, .GIF, .ICO, .JPG, .MID, and .WAV files).

If you put additional files in the runtime directories before you package your application, make sure that the file names are not the same as those of any existing application files.

After you put all the files in the runtime directory, also make sure that no two files have the same name and first two characters in their file extension. For example, if you have two files FILEA.ABC and FILEA.ABB in the RT_WIN32 directory, one will be overwritten during the packaging process.

If you plan to package to diskettes, it is convenient to have pre-formatted diskettes available before you begin the packaging process.

Packaging the VisualAge RPG Runtime Code and Applications

This section describes the process you must follow to package either the VisualAge RPG runtime code, an application, or shared components.

Note: Verify that the remote location name in the RST file is the same server as your users will be using. If not, modify the Remote Location column for the entry on the Servers page in the Define iSeries Information notebook. To access this notebook from the GUI Designer, select Define iSeries information from the Server pull-down menu. To access it at run time, use the Define iSeries Information icon. To access it at packaging time, use the Change server button.

In the RST file, you must also specify the correct protocol used by your users.

For SNA, the remote location name is the name of the router defined in Client Access.

For TCP/IP, the remote location name is the iSeries server host name defined in your TCP/IP server list.
Starting the Packaging Utility

To start the packaging utility, use one of these methods:
- Select Project>Package from the Project Organizer
- Select the Package option from the project icon’s pop-up menu
- Select Application Packaging Utility from the Start > Programs > IBM WebSphere Development Studio Client for iSeries > VisualAge RPG menu.

The Application Package window appears:

![Application Package Window](image)

Specify the target system for your application:
- **Windows**
  Packages the Windows version of your application for the Windows platform.
- **Java applications for Windows**
  Packages the Java version of your application for the Windows platform.
- **Java application/applet for all platforms**
  Packages the Java version of your application for other operating systems.

Packaging Windows Applications for Windows

Select Windows and press OK. The Package Windows dialog appears:

![Package Windows Dialog](image)

Specify the following:
- What you want to package
- The application package information
- The runtime package information
Specifying What You Want to Package
In the Package window, specify the following information:

**Application Name**
The fully qualified application project name. You can type over the default (if any), or use the Find push button to invoke the Find Projects for Packaging window. When you specify an application project name, the title of the application will be displayed for your information.

**What you want to package**
Use the check boxes to indicate whether you want to package the application, the shared components, the runtime code, or all three. If you select shared components, a window containing a list of shared components will open. You can select which shared components in the list should be packaged.

Use the radio buttons to indicate whether you want to package the entire application or only selected components. If you choose to package selected components, a window will open that contains a list of components within the application that you can select. You can also package additional shared components created by other applications. These shared components will be automatically added to the list the next time you choose to package by component.

Specifying the Application Package Information
After making your selections on the Package window, press OK. The package information window appears:

In the Package information window, specify the following information:

**Target directory**
The target directory for packaging. If you package to a diskette, the target directory can only be the root directory of the diskette; there can be no subdirectory. If you package to a directory, that directory must not contain other files.

**Company Name**
The company name that the application will be registered under.

**Note:** When you package an updated version of a previously distributed application, use the same company name for the revised application. Otherwise, the revised application will be treated like a new application.
**Version**
The version of the application.

**Title**
The title of the application.
Here you can use the Change server button to display a list of servers (remote locations) used by your application. You can modify the list so that the package will use the new names.

Select Package to begin the packaging. A progress indicator window appears. Messages are displayed to tell you what labels to put on the various diskettes as you create them. When packaging is completed, a completion message is displayed.

**Specifying the Runtime Package Information**
If you indicated that you want to package the runtime code, you must specify the target directory in the Package Run-Time window:

If you package to a diskette, the target directory can only be the root directory of the diskette; there can be no subdirectory. If you package to a directory, that directory must **not** contain other files.

Select Package to begin the packaging. A progress indicator window appears. A completion message indicates when packaging is completed.

**Packaging Java Applications for Windows**
Packaging the Java version of your application for the Windows platform follows a similar set of dialogs as the Windows version.

Specify what you want to package and whether you want to package the run time. If you want to package the application, specify the application project name and select the application check box. Use the runtime checkbox to package the run time.
Packaging Java Applications for Other Platforms

Select Java application/applet for all platforms on the Application Package window and press OK. The Package Java window appears:

Specify the following:
• What you want to package: the application, run time, or both.
• The format of your package: the Application files (valid only for the Application choice) or the Jar file.

Specifying What You Want to Package
In the Package Java window, specify the following information:

Application Name
The fully qualified application project name. You can type over the default (if any), or use the Find push button to display the Find Projects for Packaging window. When you specify an application project name, the title of the application will be displayed for your information.

What you want to package
Indicate whether you want to package the application or the runtime jar file.

For the Application choice, you can choose one of the following formats:
• Application files
  Include all of the files in the runtime directory and place them in the target directory.
• Jar file
  Include all of the files for a component in its own jar file. (Any GIF image files will just be copied and not included in the jar file.)
• Include JDBC
  Include the JDBC class functions file in the jar.

Packaging the Application Jar File
If you select to have your application packaged as a Jar file, the following window appears:
Specify the target directory. You can specify Jar options, as well. If you select **Include HTML file**, the default HTML page for the application will be copied to the target directory. If you select **Export to iSeries**, the Smart Guide to export files to the server will be displayed.

**Note:** If your application has multiple components, each component will have its own jar file. Also, any GIF image files will just be copied and not included in the jar file.
Packaging the Run Time
If you select to have a runtime Jar file, the following window appears:

Specify the jar file name and the target directory. If you select Export to iSeries, the Smart Guide to export files to the server will be displayed.
Chapter 28. Installing Windows NT/95/98 Runtime Code and Applications

This section describes installing the runtime code and applications for Windows NT/95/98, using InstallShield.

Note: The runtime code must always be installed before you install an application. Only one copy of the runtime code is installed on a workstation, regardless of how many applications are installed.

Installing the Runtime Code

Start the installation utility by running the

setup.exe

command in the package and follow the steps given in the dialog boxes.

The Define Server Logon, the Define AS/400 Information utility and the Define TCP/IP Server List are installed with the runtime code. Use these utilities to maintain and update the names and location of AS/400 resources at run time. See Chapter 8, “iSeries Connectivity,” on page 193 for more information.

A Note About Embedded SQL

If your application has embedded SQL and is referencing a database to which your application was not bound at build time, you have to re-bind your application to a database to which it does have access.

Installing an Application

Install the application by calling the

setup.exe

in the package and follow the steps given in the dialog boxes.

The Define AS/400 Information utility is installed optionally with the application. Use this utility to maintain and update the names and location of AS/400 resources at run time. See Chapter 8, “iSeries Connectivity,” on page 193 for more information.

Maintaining the Runtime Code and Applications

To update the runtime code or application, use the same setup.

To remove the runtime code or VisualAge RPG applications do the following:
1. From the Windows NT/95/98 Start pop-up menu on the Task Bar, select Settings and then Control Panel.
2. Invoke the Add/Remove Programs utility.
Installing From the LAN

This section applies to Windows applications running on Windows NT/95/98.

To run from the LAN:
1. Package the runtime code or application to a LAN server.
2. Install the runtime code or application to the root directory of the same server. The directory name should be VRPGRT_LAN for the runtime code or XXX_LAN for the application. XXX is the name of the application’s executable file.
3. Install the runtime code to the client workstation using the package from step 1. Select the compact option.
4. Install the application using the package from step 1. Select the compact option.

Installing Silently from the LAN

This section describes how to install the runtime or application code silently from a LAN server. The basic steps are:
1. Package the runtime or application code to your LAN server using the packaging utility. (See “Packaging the VisualAge RPG Runtime Code and Applications” on page 415 for instructions.)
2. Install the run time or application on the LAN server using the -r setup option:
   ```
   setup -r
   ```
   The r parameter enables the system to record your keystrokes during the installation process. This information is stored in the setup.iss file created in your Windows directory. Your keystrokes will be used for silent installation.
3. Copy the setup.iss file from your Windows directory to the LAN directory where you packaged the run time or application. For example, if c:\winnt is your Windows directory, the setup.iss file can be found under c:\winnt.

   **Note:** Make sure you copy the runtime setup.iss file before you install the application to its LAN directory with the r option. Otherwise the setup.iss file from the application install will overwrite the setup.iss file created by the runtime install step.
4. Modify the copy of the setup.iss file in the LAN directory where you packaged the run time or application. Change the szDir entry to point to the target drive and directory where the run time or application package will be installed to.
5. From the client workstation, go to the LAN directory where the run time or application was packaged. Run the following command:
   ```
   setup -s
   ```
   After installing the run time, shut down and restart your operating system.
Part 6. Appendixes
Appendix A. Application Files

This section describes all the files that VisualAge RPG produces when you create a GUI, write logic, and build an application. Unless specified, do not edit, rename, or remove these files from the directory in which they were created.

**Note:** For Java applications, it is RTJAVA. For Windows NT/95/98 applications, it is RT_WIN32.

*Table 22. Application files*

<table>
<thead>
<tr>
<th>File name</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename.CLASS</td>
<td>Binary</td>
<td>The runtime directory contains the filename.CLASS file, which is created when a project is compiled for Java.</td>
</tr>
<tr>
<td>filename.DLL</td>
<td>Binary</td>
<td>The runtime directory contains the filename.DLL file, which is created using the .VPG file. A VisualAge RPG DLL is the program object for the application. The compiler can also create a utility DLL and its accompanying .LIB file.</td>
</tr>
<tr>
<td>filename.EVT</td>
<td>ASCII</td>
<td>The source directory for the application contains the filename.EVT file, which contains compiler feedback errors.</td>
</tr>
<tr>
<td>filename.EXE</td>
<td>Binary</td>
<td>The runtime directory contains the filename.EXE file, which contains the runtime mainline. The compiler can also create an EXE that is self-contained.</td>
</tr>
<tr>
<td>filename.HLP</td>
<td>Binary</td>
<td>The runtime directory contains the filename.HLP file, which is the compiled help file that was created using the .IPF, .IPM, .VPG, and .TXM files.</td>
</tr>
<tr>
<td>filename.HTM</td>
<td>ASCII</td>
<td>The source directory for the application contains the filename.HTM file, which includes HTML code for launching the compiled Java program as an applet.</td>
</tr>
<tr>
<td>filename_applet.HTM</td>
<td>ASCII</td>
<td>The source directory for the application contains the filename_applet.HTM file, which includes HTML code that checks the VARPG Java run time to ensure that the user has the correct version of the run time installed as an extension.</td>
</tr>
<tr>
<td>filename.IPF</td>
<td>ASCII</td>
<td>The source directory for the application contains the filename.IPF file, which contains all the control information needed to build online help.</td>
</tr>
<tr>
<td>File name</td>
<td>Format</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>--------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>filename.IPM</td>
<td>ASCII</td>
<td>The source directory for the application contains the filename.IPM file, which contains all the second-level help you write for messages for windows and their parts.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Do not rename or remove this file from the directory it was created in.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Use the GUI Designer (Define Messages window) to edit this file. If you must edit this file outside of the GUI Designer, limit your changes to simple text editing such as correcting grammar and spelling mistakes. Do not remove or modify resource IDs, add messages, or delete messages.</td>
</tr>
<tr>
<td>filename.JAVA</td>
<td>ASCII</td>
<td>The source directory for the application contains the filename.JAVA file, which contains the generated Java source resulting from a Java compile.</td>
</tr>
<tr>
<td>filename.LIB</td>
<td>Binary</td>
<td>The filename.LIB file contains all the exported procedures that are part of a utility filename.DLL.</td>
</tr>
<tr>
<td>filename.LST</td>
<td>ASCII</td>
<td>The source directory for the application contains the filename.LST file, which contains the compile listing.</td>
</tr>
<tr>
<td>filename.ODF</td>
<td>Binary or ASCII</td>
<td>The source directory for the application contains the filename.ODF file, which contains all the information about your application’s windows and their parts.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Do not rename or remove this file from the directory it was created in.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• This file can only be edited using the GUI Designer.</td>
</tr>
<tr>
<td>filename.ODX</td>
<td>ASCII</td>
<td>The source directory contains the filename.ODX file, which is created using filename.ODF and is used at run time.</td>
</tr>
<tr>
<td>filenameResources.properties</td>
<td>ASCII</td>
<td>The runtime directory for the application contains the filenameResources.properties file, which contains all messages you write for the windows and their parts, in Java format.</td>
</tr>
<tr>
<td>filename.RST</td>
<td>ASCII</td>
<td>The source directory contains the master copy of this file. filename.RST contains all the server aliases, file overrides, data area overrides, program overrides, and lock level information you define for your application. You can change its contents at build time using the GUI Designer, or at run time using the Define AS/400 Information utility.</td>
</tr>
<tr>
<td>File name</td>
<td>Format</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>--------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| filename.TXC | ASCII | The source directory for the application contains the filename.TXC file, which contains all the programming notes you keep in the multiline edit fields provided for storing technical descriptions.  
  • Do not rename or remove this file from the directory it was created in.  
  • Use a part’s properties notebook to change its settings. |
| filename.TXM | ASCII | The source directory for the application contains the filename.TXM file, which contains all messages you write for the windows and their parts.  
  • Do not rename or remove this file from the directory it was created in.  
  • Use the GUI Designer (Define Messages window) to edit this file. If you must edit this file outside of the GUI Designer, limit your changes to simple text editing such as correcting grammar and spelling mistakes. Do not remove or modify resource IDs (resids), add messages, or delete messages. |
| filename.VCX | Binary | The runtime and source directories for the application contain the filename.VCX file, which contains persistence information for any ActiveX parts used in your application. |
| filename.VPF | ASCII | The source directory for the application contains the filename.VPF file, which contains all the help text you write for windows and their parts.  
  • Do not rename or remove this file from the directory it was created in.  
  • You can edit this file by using either the GUI Designer’s editor (using a pop-up menu for a part or using a properties notebook). |
| filename.VPG | ASCII | The source directory for the application contains the filename.VPG file, which contains all the VisualAge RPG application source code you write.  
  • Do not rename or remove this file from the directory it was created in.  
  • Use the GUI Designer to edit the source code. |
Appendix B. Writing Thin Client Applications

VisualAge RPG applications that mainly run on and utilise workstation resources are called **thick client** applications. **Thin client** applications mostly rely on the iSeries server to perform their processing and only leave the GUI handling to the client.

Thick client applications follow very much the same programming style found in today’s RPG III or RPG IV applications, but they run mostly on the workstation instead of the iSeries server. File specifications are used to specify which database files to access, and native RPG operations like READ, CHAIN, and so on, are used to access the data on the server. The iSeries server functions as a data server and does minimal computing to support the VARPG application.

The thick client model has several disadvantages over its thin client counterpart. Its capability for module reuse is very limited and there is an increased overhead cost associated with change management. As well, moving processing onto the client workstation under utilises the processing power of the server.

Making the client portion of an application thinner, offers the following advantages:
- The amount of code running on the client can be easily reduced.
- Application reusability will be greatly enhanced.
- Maintainance of complex code will be easier.

This section discusses two possible implementations of the thin client model. Both implementations exploit multiple VARPG capabilities that provide integration with the iSeries server. The two examples include capabilities that use:
- External description of data structures to define externally described data in a data structure easily without the use of direct file access
- Remote call interface to provide a simple way of calling server programs and passing data
- Reference fields in the GUI designer. Fields in the subfile of the user interface are defined as reference fields, no additional definition of database fields is needed.

Implementing a VARPG Thin Application Model

The thin VARPG application model can be implemented in several, different ways. Two ways are described here. One implementation uses remote calls to an iSeries server; the other uses data queues on the iSeries server. The same user interface is used in both examples.

The simple client application reads data from a customer file and fills a subfile with 10 records at a time. The following illustration shows the user interface for this application:
The interface consists of a window with canvas, a subfile, and a push button to load one More page of records into the subfile. The subfile size for this particular example is 10 records. It can be changed by increasing the height of the subfile part.

The following names are used in this example:

**Part Name**

**Window**

WIN1

**Subfile**

SUB1

**Push button**

PSBMORE

---

**Sample Application Using Remote Calls**

In traditional RPG programs, user interface code and database access logic are intermixed in one module. Part of this structure stems from the history of RPG, and part from the usage of the Original Program Model (OPM) that forces the programmer to achieve good performance. One way to implement the thin application model is to split the user interface logic completely from the database access logic and have each piece run on different systems. The user interface logic runs on a Windows client, the database access logic runs on the iSeries server.

This sample application shows how to support reading records of data from the iSeries database and placing this data into a GUI subfile. The program on the iSeries server could just as well support full database access (READ and WRITE). This could be implemented by supplying one program for each different access method, or by passing the desired operations as parameters to a single server program.
The following diagram shows how this sample works:

The client program gets requests from the user interface. It calls a server program that reads records from a database program and passes this data back to the client through parameters. The subfile gets filled with the returned data.

The Client Program

The main part of the client side program is the user interface. It is created in the same way as all VARPG applications and can utilize the external database descriptions of the server by using database reference fields. Any validation checking specified in the database is done automatically on the client by the VARPG run time. The Client program requests data from the server by calling a server data access program, the data itself is passed via parameters. The Client program does not use file specifications; instead, the data definition is done through external described data structures. This way the programmer still gets the benefits of external field descriptions in the VARPG program.

Sample RPG Source for the Client Side

The VARPG program consists of D and C specifications. The D specifications contain the following data definitions:

- The fields used as parameters:
  - cust, a multi occurrence structure
  - custelem, a numeric field, contains the maximum number of records being requested
  - eof, a named indicator, gets passed when the end-of-file indicator is set to ON in the server program
  - nrecords, a numeric field, contains the number of records returned

- Two working fields:
  - fileend, a named indicator, for keeping the file end condition
  - counter, a counter for the DO loop

- getrec, a constant, defines the program being called on the server. It defines the linkage to the server and the actual name of the server program. The program name must be specified in UPPERCASE.

H  D cust   e ds   extname(customer)
D    occurs(10)
D     inz
D    eof   s   n   inz
The C specifications contain one action subroutine that is linked to 3 events:

- **Press** event from push button *PSBMore*
- **Create** event from window *Win1* (Linked to PSBmore/press action subroutine)
- **Pageend** event from subfile *Subf1*

The first statement is a call to the server program to fetch more records. The rest of the logic just processes the data passed via parameter and moves it from the multi occurrence data structure to the subfile. After the subfile is filled with a set of records the highest record number in the subfile is applied to the SETTOP attribute to move this set of record into the visible area of the subfile.

At the end, if the end-of-file is reached, the More push button gets set to be disabled. Note that the Page Down keys still work. It is still possible to cause an event that will trigger this action subroutine even with a disabled push button.

As you can see, the client end of this code is straight forward and minimizes the processing on the workstation.

### The Server Program

Since the VARPG client program excludes the database access logic, this function is now provided by the server program. The server program contains all FILE definitions and operations to handle the database processing. Data is exchanged by moving a data structure as a parameter between the client and the server program. The data structure contains the field definitions of the data file record format. In this example, a multi occurrence data structure is used for accessing a collection of records. The number of occurrences is equal to the numbers of records to be passed; in this example, 10. Any operational information, such as error information for example, could be passed by parameter, as well. The server program gets
invoked by the Call in the VARPG client program and ends after each request. The 
Return operation code is used to end the program and keep the invocation 
environment. This will benefit performance in subsequent calls, since no 
initialization is needed. This also requires the program to be created to run in a 
named activation group, since *NEW would destroy the invocation environment 
and free storage immediately.

Sample RPG Source for the Server Side
The File specification defines the external database file Customer. The data 
definition specifications define the parameters to be passed. These must be defined 
the same as on the client side. Count represents a work variable for the counter in 
the DO loop. Custelem contains the number of elements in the data structure 
CUST, it is used as a limit for the DO loop.

* Program to read a set of records into a data structure
*******************************************************************************
Fcustomer if e
D cust e ds extname(customer)
D                      occurs(20)
D eof s                n
D count s              2 0
D custelem s            2 0

At the top of the calculation specifications, the PLIST operation code defines the 
parameters being passed to this program. The DO loop reads from the database 
file and puts the data into data structure CUST, which will be passed back as a 
parameter to the client program. The other two parameters just indicate the status 
of the database access:
* EOF will be set to ON if indicator 99 gets set by the READ statement.
* Count contains the number of records being passed back to the client in data 
structure CUST.

C *entry       plist
C               cust
C               custelem
C               eof
C               count
C eval         count=1
C count         cust
C read          customer
C dow           count<>custelem and not *in99 
C eval          count=count+1
C count         cust
C read          customer
C enddo
C if            *IN99
C eval          count=count-1
C eval          eof=**on
C endif
C return

When compiling the server program, be sure not to specify *NEW for Activation 
Group. If *NEW is specified (the default), any storage allocated by this program is 
freed when RETURN is executed. One of the benefits of this thin client example is 
the reusability of the server application by different applications. Even traditional 
5250 applications can use the server modules for database access. This approach 
certainly makes it easier to maintain applications since changes in a server module 
are reflected in all applications that use it.
Sample Application Using Data Queues

The iSeries server provides built in support for data queues to allow applications to communicate with each other asynchronously. This sample application exploits data queues, instead of parameter passing, to exchange the data from the database with the VARPG client program. This application is based on 2 data queues on the server that are used by the client and server program. The server program in this example gets launched as an independent program on the server using the NOWAIT keyword in the D specifications of the client program.

The following diagrams illustrate how this example works.

First, two data queues are created and server program DATAQ is started. The server program begins requesting data from data queue 'O' and remains in an indefinite wait.

Server program waits for requests

![Diagram of server program waiting for requests]

The next state is entered when a GUI event requests more data. (See Figure 114 on page 432 for the client interface.) The three events that trigger the action subroutine are:

- Create event from the window
- Press event from the More... push button
- Pageend event from the subfile

The client program then waits on data queue 'I' for data. The server program accesses the database file and gets the data.
In the third state, the server program fills data queue 'I'. The client program becomes active and puts the data into the subfile. After this, the program returns to its initial state and the process starts again.

The Client Application

The user interface is the same as that in the previous application, basically a subfile getting filled with data from the iSeries server database. The filling of the subfile starts with the create event of the window, and continues when the More... push button is pressed or a pageend event occurs using the page down keys. This is essentially the same as in the previous example.

The setup for the data queues is done in the create window action subroutine, which calls a program on the server to create 2 data queues in a library on the iSeries server. To create unique data queues for each client, we use the *component part’s hostname attribute to retrieve the client’s hostname and IP address. The last 5 characters of the IP address portion of the returned string get tagged onto the name of the data queues. The characters ‘I’ or ‘O’ at the end of the data queue name provide the unique names for the Input or Output data queues.

The server job receives commands from the ‘O’ data queue; the client program sends commands to the ‘O’ data queue.
After creating the data queues, the client program calls the server program and passes it the 2 data queue names. The server program waits on data queue 'O' for commands from the client program.

The client program gets activated by GUI events and then sends requests to data queue 'O'. It then waits on data queue 'I' until this data queue is filled by the server job.

When the client program gets a termination request, the *TERMSR subroutine is invoked to signal the server program to end and the 2 data queues will be deleted.

**Client Sample Source**

This program is a bit larger because the data queue environment has to be managed in it, as well.

**Data definitions**

The data definitions for the client program:

D*
D* This program uses the *component part attribute hostname
D* variable to store the host name and IP address of the client.
D entipadd s 100a
D*
D* Command strings to create and delete data queues
D QCMDEXC s 10 inz('QCMDEXC')
D linkage(*server)
D* Variables to hold command information
D cmd s 256 INZ
D cmdlen s 15p 5 inz(%size(cmd))
D cmd1 s 256 INZ('CRTDTAQ DTAQ(QGPL/)')
D cmd1 s 256 INZ('DPTDTAQ DTAQ(QGPL/)')
D cmd2 s 9 inz(' ') MAXLEN('
D*
D* Prefix for data queue name
D qname1 s 4 inz('CUSQ')
D*
D* Variables that contain the 2 data queue names used for one client
D qname1 s 10
D qnameo s 10
D
D* Define RCVDTAQ and SNDDTAQ programs as server programs
D QRCVDTAQ s 10 inz('QRCVDTAQ')
D linkage(*server)
D QSNDTAQ s 10 inz('QSNDTAQ')
D linkage(*server)
D* RPG IV server program definition
D DATAQ s 10 inz('DATAQ')
D linkage(*server) nowait
D*
D* Data structure containing customer database data
D CustDS e ds extname(customer) occurs(10)
D*
D* Data structure containing process information
D rinfo ds
D e0f n
D nrecords 2 0
D filler 20
D* Limit for loop
D custelem s 2 0 inz(%elem(CustDS))
D* Indicator for file end reached
D fileend s n
The create window action subroutine

The data queues are created and the server RPG program DATAQ is started, the program gets invoked with the NOWAIT keyword; the client program will not wait for it to end. Both programs are working completely asynchronously.

C*
C WIN1 BEGACT CREATE WIN1
C* Get client IP address to build unique data queue names.
C* entipadd will contain the client's full host name and IP address.
C*
C eval entipadd = %GetAttr('*component':'
C 'component':'hostname')
C* hostname returns the string 'hostname IPAddress'.
C* We only use the IPAddress portion of the returned string.
C* Substring the second part of the returned string:
C eval entipadd = %subst(entipadd: %scan(':'
C entipadd)+1)
C* Create the names for the 'I' and 'O' data queues.
C* Use the last 5 characters of IPAddress and add 'I' or 'O'.
C eval qnamei= qname1 +
C %subst(entipadd: %len
C (%trim(entipadd))-4:5) + 'I'
C eval qname0= qname1 +
C %subst(entipadd: %len
C (%trim(entipadd))-5:
C 5) + 'O'
C* Set up command parameters to create data queues.
C eval cmd=%trim(%trimr(cmd1) +
C qnamei + cmd2 +
C %editc(%size(
C CustDS:*all):'Z') + '')
C* Create the data queues.
C call QCMDEXC cmd 98
C parm cmd
C parm cmdlen
C*
C eval cmd=*blank
C eval cmd=%trim(%trimr(cmd1) +
C qnameo + cmd2 +
C %editc(%size(
C CustDS:*all):'Z') + '')
C call QCMDEXC cmd 98
C parm cmd
C parm cmdlen
C* Call server program to access database on server
C call DATAQ qnamei 98
C parm qnamei
C parm qnameo
C* Initialization is done; now, do event processing.
The action subroutine to handle requests for more data

A request is sent to data queue 'O'. The client program then waits for a response from the server program DATAQ, on data queue 'I'. After receiving the data, the subfile gets filled in a loop.

C* Action subroutine gets invoked from:
C* - Press event of PSBMORE push button
C* - Pageend event from subfile
C* PSBmore BEGACT PRESS win1
C*
C* Get data from server and display it in the subfile.
C* EXSR callhost
C* ENDACT
C*
C* User subroutine 'callhost' requests server data.
C* callhost begsr
C* As long as there is data, get more data.
C* if not fileend
C* Send a request to data queue 'O' to fetch data.
C* eval nrecords=10
C* call QSNDDTAQ
C* parm qnameo
C* parm 'QGPL ' NAME_OF_LB 10
C* parm 23 MSG_SZ 5 0
C* parm rinfo
C* Wait on data queue 'I' for data.
C* Expecting processing data here in DS rinfo.
C* eval wait_time=-1
C* eval MSG_sz=23
C* call QRCVDTAQ
C* parm qnamei
C* parm 'QGPL ' NAME_OF_LB
C* parm MSG_SZ
C* parm rinfo
C* parm WAIT_TIME
C* Expecting data from database here.
C* eval Msg_sz=%size(custds:*all)
C* call QRCVDTAQ
C* parm QNAMEi
C* parm 'QGPL ' NAME_OF_LB
C* parm MSG_SZ
C* parm CustDS
C* parm WAIT_TIME
C* For as many records as the server has read, fill the subfile.
C* eval count=1
C* dow count<=nrecords and not fileend
C* count occur CustDS
C* write sub1
C* eval count=count+1
C* enddo
C* eval %setatr('win1':
C* If end-of-file was signaled, disable the push button.
C* End of user subroutine
C* Terminate subroutine

A termination request is sent to server program DATAQ and the 2 data queues are deleted.

C* When client app ends, clean up server environment
C* Indicate end of program to server program and send data to dataq 'O'
C* Delete both data queues
C* Application ends

The Server Program

After the server program gets launched, it enters a loop and waits on data queue 'O' until it gets a request from the client program. Two different requests are possible in this example. The program determines which request has been sent: to read more data, or to end.

For a request for more data it will read 10 more records from the database and then send 2 items to data queue 'I'.

The first item contains process information, how many records were actually read, and whether an end of file situation has occurred.

The second item contains the multi occurrence data structure with the data from the database file.

The client program will receive these records from data queue 'I' and fill the subfile accordingly.
When the server program gets signaled that a termination is requested, the LR indicator will be set on and the DO loop will end. This will end the program. Any other cleanup will be managed by the client program.

Server Sample Source

File and data definitions

```
Fcustomer if e disk
D* data structure containing database data to be passed to client
D CustDS e ds extname(customer) occurs(10)
D* data structure to pass control information between client and server
D rinfo ds
d e eof n
d count 2 0
d fill 20
d* number of occurs in DS for loop limit
D custelem s 2 0 inz(%elem(CustDS))
d* library name for dataq and data size to be send to dataq and wait time
D Name_of_LB s 10 inz('QGPL')
D msg_sz S 5 0
d wait time s 5 0
d* name of dataq's passed from client
D qnamei s 10
d qnameo s 10
```

Main line program

Process the DO loop, wait on data queue 'O' until requests arrive, read more records from the database, send the data to data queue 'I', and wait again for more requests.

```
C* Beginning of mainline
C* entry
C  plist
c parm qnamei
C  parm qnameo
C* DO loop runs forever until client program signals that it terminates
C  dow not *inlr
C* Wait for client program to signal that it needs data
C  eval wait_time=-1
C  eval MSG_sz=23
C  call 'QRCVDTAQ'
C  parm qnameo
C  parm NAME_OF_LB
C  parm MSG_SZ
C  parm rinfo
C  parm WAIT_TIME
C*
C* Read 10 records from database file
C* count = 0 means client program is terminating
C  if count >0
C  eval count=1
C  count occur CustDS
C  read customer
C  dow count=custelem and not *in99
C  eval count=count+1
C  count occur CustDS
C  read customer
C  enddo
C* Determine whether there is more data in file
C  if *IN99
C  eval count=count-1
C  eval eof==on
C  endif
```
C* Send information to the data queue.
C* Send one record with information on how many records are read and
C* whether end-of-file was reached
C* 
C call 'QSNDDTAQ' 98
C parm QnameI
C parm NAME_OF_LB
C parm 23 MSG_SZ
C parm rinfo
C* Send the data in DS from database file to dataq
C eval msgSz=%SIZE(custds:*all)
C call 'QSNDDTAQ' 98
C parm qnamei
C parm NAME_OF_LB
C parm MSG_SZ
C parm CUSTds
C* When client program ends, it sends nrecords 0, then ends this
C* program as well
C else
C eval *inlr=*on
C* 
C end
C enddo
C* 
C* End of MAINLINE

Other Possible Implementations

In addition to these specific examples, variations and combinations of both implementations are possible. The goal is to minimize client processing and use the server's power to run these applications. Reuse of modules on the server can be accomplished since 5250 and GUI applications can use the same server programs.

One possible implementation is to use requests in form of an SQL statement that gets passed to a server program, This server program issues the SQL statement and routes the received data to a data queue. The client program, waiting on the data queue, uses the data passed back to satisfy the end user request. In this particular application, a single keyed data queue is used instead of multiple data queues.

Another implementation could pass all input data from the user interface to a server program to do error checking and processing on the server; any error conditions would get passed back to the client. This approach allows a high degree of reusability of business logic between 5250 applications and GUI applications, and provides an even thinner client.

Reusable Server Program Example

The following RPG IV code for a 5250 subfile application uses the same server program, but drives a 5250 interface. This example shows the reuse capabilities of server programs for GUI and 5250 applications. The business logic contained in the 5250 server program can readily be used by thin client GUI programs. The following illustration shows a sample of the 5250 screen that displays the same database information as in the GUI sample in the beginning of this chapter.
This subfile only fits 8 records on the screen because each record occupies 2 rows on the subfile. The program source for this application follows:

H* Program to list customer records
F* Workstn file containing DSPF
Fgetrecs cf e workstn sfile(sub1:recnum)
D= Data structure to pass data from server program to the subfile
Dcust e ds extname(customer) occurs(8)
D inz
Ddef s 1 inz(+off)
Dnrecords s 2 0
Dfileend s 1 inz(+off)
Dcount s 2 0
Dcustelem s 2 0 inz(%elem(cust))
Drecnum s 5 0 inz(1)

* Main program to invoke subfile sub routine and end the program
* In91 indicates that database file has not reached the end
C EVAL *IN91=*ON
C exsr more
C eval *inr=*on
C
C more BEGsr
* LOOP to fetch more data and display in subfile
C dow not *in03
* Call server program to get data first time and when page down
* is used.
C if *in91
C call 'GETREC'
C parm cust
C parm custelem
C parm eof
C parm nrecords
C eval count=1
* recnum1 is sbfrcdnbr in subfile control record, to position top record
* to be shown on screen
C EVAL RECNUM1=RECNUM
* Loop to fill subfile with new records
C dow count<nrecords
C count occur cust
* After the set of records is added to subfile, display it, plus header * and footer record formats

```plaintext
C write subl
C eval count=count+1
C eval recnum=recnum+1
C enddo
```

* Handle none pagedown keys
```
C else
C read sublctl 99
C endif
```

* IN90 enables PAGEDOWN key for additional records to be read. * At file end it gets disabled.
```
C eval *in90*on
C eval recnum=recnum1
C endif
C enddo
```

* Leave the LOOP when Exit requested
C ENDSR
Appendix C. Creating and Compiling Non-GUI Programs from MS-DOS

You can create standalone VARPG applications within the VARPG GUI designer, or by issuing commands in an MS-DOS command prompt. This section provides the commands you can use from an command prompt. To use the GUI Designer, see Chapter 22, “Creating Non-GUI VisualAge RPG Programs,” on page 375.

To start the editor and create source in an MS-DOS command prompt, enter:

codeedit filename.VPG

Be sure to include the .VPG extension in the filename so you can benefit from the editor’s tokenizing and syntax checking of your source.

To run the FVDFNFE compiler, issue the compiler command from an MS-DOS command prompt. The syntax of the command is:

fvdfnfe filename.VPG [/compiler_option1 ... /compiler_optionn]

Note: You must include the .VPG extension in the source file name; it is not assumed.

The compiler options, which you can type in upper- or lowercase, are optional. They are as follows:

Option Description

/BL name
  Link library name. If more than one, enclose in double quotes.

/D
  Generate debug information

/GL 1-99
  Generation severity level

/L
  Generate an output listing

/LI
  Indentation

/LX
  Generate a cross-reference (XREF) listing

/LV
  Generate a Visual cross-reference (XREF) listing

/LD
  Expand DDS in listing

/LC
  Expand /COPY in listing

/LE
  Show external references

/LM2
  Show second-level messages in listing

/LS
  Show excluded lines in listing

/LP 10-99
  Lines per page (listing)

/HCU
  Host cache enabled

/HCR
  Host cache refresh

/RF
  Fix numeric
Accessing an AS/400 System

If the program is to access data on the AS/400 system or call AS/400 programs, you must create a Remote Server Table (RST) file. The RST file is an ASCII file that is read by the compiler and the VARPG run time to determine information about the system to use and the location of files. An RST file is created by the GUI Designer when you use the Define AS/400 Information dialog. During compile time, the RST file must be in the same directory as the program source. During run time, the RST file must be in the same directory as the program executable files.

Here is an example of an RST file:

```
DEFINE_SERVER
  SERVER_ALIAS_NAME(MYAS400)
  REMOTE_LOCATION_NAME(TORAS40Z)
  NETWORK_PROTOCOL(*TCP)
  TEXT(Development - RST)

DEFINE_FILE
  FILE_ALIAS_NAME(CUSTOMER)
  REMOTE_FILE_NAME(PRODUCT/CUSTMAST)
  SERVER_ALIAS_NAME(MYAS400)
  TEXT()
```

The DEFINE_FILE statement indicates where the files defined in the ’F’ specifications can be found on the AS/400 system. In this example, the file named CUSTOMER in the ’F’ specification actually refers to file CUSTMAST in library PRODUCT. If the file is in your library list, the DEFINE_FILE statement can be omitted. In that case, the library list, *LIBL, will be searched for the file.

The TEXT keyword is used for comments and can be blank.
Appendix D. Secure Sockets Layer (SSL) Setup

Secure Sockets Layer (SSL) provides secure connections by encrypting the data exchanged between a client and an iSeries 400 server session and performing server authentication. SSL can be used only with a SSL capable iSeries 400 server running OS/400, Version 4 Release 4 or later. The use of SSL connection significantly decreases performance of the product compared to the use of connection without encryption. It is recommended that SSL be used only when the sensitivity of data transferred merits the decrease in performance.

This section provides instructions on configuring your iSeries 400 server for Secure Sockets Layer support.

Note: If any fixes are applied to the VARPG run time, the following procedures must be repeated. For SSL to work properly on the iSeries 400 server, the QSECOFR user profile password must not be expired.

SSL Considerations

• A good knowledge of SSL is required for setup. You should know how to use the Digital Certificate Manager (DCM) program on the iSeries 400 server to perform SSL-related tasks, such as generating system certificates. For information on the DCM program, invoke its online help or go to the Information Center at URL http://www.ibm.com/eserver/iseries/infocenter.

• Ensure that you adhere to import and export regulations. IBM iSeries Client Encryption products provide SSL version 3.0 encryption support using non-exportable 128-bit for U.S. and Canada use only and exportable 56-bit for international use. In customer configurations where Client Encryption products might be downloaded across national boundaries, the customer is responsible for ensuring that the non exportable products are not made available outside the U.S. and Canada. Both the non-exportable and exportable Client Encryption products can be used in combination to allow the appropriate Client Encryption product to be downloaded on different Web sites.

Prerequisites

The following prerequisites must be met before setting up SSL:

• Software:
  – Cryptographic Access Provider licensed program (5769-AC1, 5769-AC2, or 5769-AC3)
  – IBM iSeries Client Encryption program (5722-CE2 or 5722-CE3) where 5722-CE2 (56-bit) is for use in countries other than the U.S. or Canada, and 5722-CE3 (128-bit) is for use only in the U.S. and Canada.

• Hardware:
  – IBM HTTP Server for iSeries (5722-DG1)
  – Base operating system option 34 Digital Certificate Manager (DCM)

• Authorization:
  Provide proper authorizations for users to access the SSL files. Follow these steps to change the authority:
  1. Enter the command:
     wrklnk '/QIBM/ProdData/HTTP/Public/jt400/*
  2. Choose option 9 in the correct directory (SSL56, or SSL128).
  3. Give users *RX authority to the directory.
Note: Individual users or groups of users can be authorized.

SSL Setup for the iSeries 400 Server

Follow the instructions below to setup your iSeries 400 server for SSL:

Start DCM:
1. Start the HTTP servers by using the command:
   STARTCPVR SERVER(*HTTP) HTTPSVR (*ADMIN)
2. Access the iSeries 400 Administration server by typing in your server’s URL address and port number. For example:
   http://your.server.name:2001/
   The proper security officer authority, plus *secadm and *allobj authorities are needed.
3. Enter your iSeries 400 user ID and password.
4. From the AS/400 Tasks page, access DCM by selecting the DCM link.

Obtain a system certificate for your iSeries 400 server from the Digital Certificate Manager program:

Click on the ? icon for instructions on obtaining a system certificate. You can either obtain the system certificate from a trusted Certificate Authority (CA) or build your own.
• To obtain a system certificate from a trusted CA:

   Note: This is what you should use if your application is an internet application and you need to distribute your run time.

   Select a CA. You can get a certificate from one of the following companies:
   – VeriSign, Inc.
   – Integration Financial Network
   – Thawte Consulting
   – RSA Data Security

   Obtain request data for submission to the trusted CA from DCM. Refer to the DCM help for the exact steps you need to follow. Your CA will process your request form and provide you with the certificate. To install it onto your system, use the **Receive a System Certificate** option.

• To obtain your own system certificate:

   Note: This should only be used for an intranet because the run time created can only be used for your specific iSeries 400 server.
   1. Create your CA on the iSeries 400 server.
   2. Generate the system certificate from your own CA.

   For information on creating a CA, on the DCM page click on **Certificate Authority**.

Apply the system certificate to the following server applications:

QIBM_OS400_QZBS_SVR_CENTRAL
QIBM_OS400_QZBS_SVR_DATABASE
QIBM_OS400_QZBS_SVR_DTAQ
QIBM_OS400_QZBS_SVR_NETPRT
QIBM_OS400_QZBS_SVR_RMTCMD
QIBM_OS400_QZBS_SVR_SIGNON
QIBM_OS400_QZBS_SVR_FILE
QIBM_OS400_QRW_SVR_DDM_DRDA
SSL Setup for the Workstation

To set up SSL for the workstation, follow the steps below:

Download the appropriate version of the SSL Client Encryption software to your workstation from the iSeries 400 server:

<table>
<thead>
<tr>
<th>Product</th>
<th>File to Download</th>
<th>Location for Download</th>
</tr>
</thead>
<tbody>
<tr>
<td>5722-CE2</td>
<td>sslightx.zip</td>
<td>/QIBM/ProdData/HTTP/Public/jt400/SSL56</td>
</tr>
<tr>
<td>5722-CE3</td>
<td>sslightu.zip</td>
<td>/QIBM/ProdData/HTTP/Public/jt400/SSL128</td>
</tr>
</tbody>
</table>

If you built your own certificate, perform the following steps to download the CA certificate:

1. Download SSLTools.jar from the directory in which you downloaded the SSL Client Encryption.
2. Add SSLTools.jar and sslightu.zip to your CLASSPATH statement.
3. Create a temporary directory. For example, c:\tempkey.
4. Create the following subdirectory under c:\tempkey:
   com\ibm\as400\access
5. From the tempkey directory, run the following command in one line:
   
   java com.ibm.sslight.nlstools.keyrng com.ibm.as400.access.KeyRing
   
   connect <systemname>:<port>

   Where <systemname> is the name of your iSeries 400 server and <port> is your port number.

   **Note:** The server port can be any of the host servers to which you have access. For example, you can use 9476, which is the default port for the secure sign-on server on the iSeries 400 server. When you are prompted to enter a password, enter toolbox. This is the only password that works. The SSL tool then connects to the iSeries 400 server and lists the certificates it finds.

6. Enter the CA certificate number. Be sure to use the CA certificate and not the site certificate. A message will be issued stating that the certificate is being added to com.ibm.as400.access.KeyRing.class.
7. Delete the file SSLTools.jar.

A KeyRing.class file has been created in the directory.

Create a customized varpg.jar file by following the instructions below:

1. Create a temporary directory. For example, c:\tempjar.
2. Copy the file sslightu.zip to c:\tempjar.
3. Run the following command:
   
   ```sh
jar xvf sslightu.zip
   ``
   
   You will find the meta-inf and com\ibm\sslight subdirectories created for you in c:\tempjar.

4. If you built your own certificate:
   
   a. Under subdirectory c:\tempjar\com\ibm add subdirectory \as400\access.
   
   b. Copy the KeyRing.class file in c:\tempkey\com\ibm\as400\access subdirectory to the c:\tempjar\com\ibm\as400\access subdirectory.

5. Copy the varpg.jar file into c:\tempjar.

6. From c:\tempjar, run the following command:
   
   ```sh
   jar uvf varpg.jar -C ./ com
   ``
   
   **Note:** C is in uppercase.
   
   Running this command updates the varpg.jar file so that it can be used with your SSL-enabled VARPG applications. This will also work for regular, non-SSL-enabled applications.

Enable your application to use SSL:

1. Rename the varpg.jar in your VARPG install directory \WDSC\JAVA.
2. Compile your application with the /SSL user-defined option in the Build options dialog.
3. Run the application with the customized file varpg.jar.
Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user’s responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Lab Director
IBM Canada Ltd. Laboratory
B3/KB7/8200/MKM
This publication is intended to help you to create and manage VisualAge RPG applications and user interfaces on the workstation, in a client/server environment. This publication documents General-Use Programming Interface and Associated Guidance Information provided by IBM WebSphere Development Studio Client for iSeries.

The following terms are trademarks or registered trademarks of the International Business Machines Corporation in the United States or other countries or both:

<table>
<thead>
<tr>
<th>Application System/400</th>
<th>AS/400</th>
<th>AS/400e</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common User Access</td>
<td>CUA</td>
<td>DATABASE 2</td>
</tr>
<tr>
<td>DB2</td>
<td>DB2 Connect</td>
<td>DB2 Universal Database</td>
</tr>
<tr>
<td>IBM</td>
<td>OS/400</td>
<td>SQL/DS</td>
</tr>
<tr>
<td>VisualAge</td>
<td>WebSphere</td>
<td>400</td>
</tr>
</tbody>
</table>

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

Lotus is a trademark of Lotus Development Corporation in the United States, or other countries, or both.

ActiveX, Microsoft, Windows, and Windows NT are trademarks or registered trademarks of Microsoft Corporation in the United States, or other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.
Glossary

This glossary includes terms and definitions from:

- The American National Dictionary for Information Systems ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 1430 Broadway, New York, New York, 10018. Definitions are defined by the symbol (A) after the definition.
- The Information Technology Vocabulary developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Committee (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol (1) after the definition; definitions taken from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol (T) after the definition indicating that the final agreement has not yet been reached among participating National Bodies of SC1.

A

action. (1) Synonym for action subroutine. (2) An executable program or command file used to manipulate a project’s parts or participate in a build.

action subroutine. Logic that you write to respond to a specific event.

active window. The window with which a user is currently interacting. This is the window that receives keyboard input.

activeX part. A part that adds ActiveX control objects to the project. VARPG applications can then access their attributes and monitor for events.

anchor. Any part that you use as a reference point for aligning, sizing, and spacing other parts.

animation control part. A part that allows the playback of video files, with the AVI extension, in Windows, or the playback of animated GIF sequences in Java applications.

API. Application programming interface.

applet. A program that is written in Java and runs inside of a Java-compatible browser or AppletViewer.

application. A collection of software components used to perform specific user tasks on a computer.

application programming interface (API). A functional interface supplied by the operating system or a separately orderable licensed program that allows an application program written in a high-level language to use specific data or functions of the operating system or the licensed program.

ASCII (American National Standard Code for Information Interchange). The standard code, using a coded character set consisting of 7-bit coded characters (8 bits including parity check), that is used for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters. (A)

B

BMP. The file extension of a bitmap file.
build. The process by which the various pieces of source code that make up components of a VARPG application are compiled and linked to produce an executable version of the application.

button. (1) A mechanism on a pointing device, such as a mouse, used to request or start an action. (2) A graphical mechanism in a window that, when selected, results in an action. An example of a button is an OK push button that, when selected, initiates an action.

C

calendar part. A part that adds a calendar that can be modified by the user to include text, color and other attributes.

canvas part. A part onto which you can point and click various other parts, position them, and organize them to produce a graphical user interface. A canvas part occupies the client area of either a window part or a notebook page part. See also notebook page with canvas part and window with canvas part.

check box part. A square box with associated text that represents a choice. When a user selects a choice, an indicator appears in the check box to indicate that the choice is selected. The user can clear the check box by selecting the choice again. In VisualAge RPG, you point and click on a check box part in the parts palette or parts catalog and click it onto a design window.

click. To press and release a mouse button without moving the pointer off of the choice or object. See also double-click.

client. (1) A system that is dependent on a server to provide it with data. (2) The PWS on which the VARPG applications run. See also DDE client.

client area. The portion of the window that is the user’s workspace, where a user types information and selects choices from selection fields. In primary windows, the area where an application programmer presents the objects that a user works on.

client/server. The model of interaction in distributed data processing in which a program at one site sends a request to a program at another site and awaits a response. The requesting program is called a client; the answering program is called a server. See also client, server, DDE client, DDE server.

clipboard. An area of storage provided by the system to hold data temporarily. Data in the clipboard is available to other applications.

cold-link conversation. In DDE, an explicit request made from a client program to a server program. The server program responds to the request. Contrast with hot-link conversation.

color palette. A set of colors that can be used to change the color of any part in your application’s GUI.

combination box. A control that combines the functions of an entry field and a list box. A combination box contains a list of objects that a user can scroll through and select from to complete the entry field. Alternatively, a user can type text directly into the entry field. In VisualAge RPG, you can point and click on a combination box part in the parts palette or parts catalog and click it onto a design window.

Common User Access architecture (CUA architecture). Guidelines for the dialog between a human and a workstation or terminal.

compile. To translate a source program into an executable program (an object program).

component. A functional grouping of related files within a project. A component is created when the NOMAIN and EXE keywords are not present on the control specifications.

component reference part. A part that enables one component to communicate with another component in a VARPG application.

*component part. A part that is the “part representation” of the component. One *component part is created for each component automatically, and it is invisible.

CONFIG.SYS. The configuration file, located in the root directory of the boot drive, for the DOS, OS/2, or Windows operating systems. It contains information required to install and run hardware and software.
configuration. The manner in which the hardware and software of an information processing system are organized and interconnected (T).

container part. A part that stores related records and displays them in a details, icon, or tree view.

CUA architecture. Common User Access architecture.
cursor. The visible indication of the position where user interaction with the keyboard will appear.

D
database. (1) A collection of data with a given structure for accepting, storing, and providing, on demand, data for multiple users. (T) (2) All the data files stored in the system.
data object. An object that conveys information, such as text, graphics, audio, or video.

DBCS. Double-byte character set.

DDE. Dynamic data exchange.

DDE client. An application that initiates a DDE conversation. Contrast with DDE server. See also DDE client part, DDE conversation.

DDE client part. A part used to exchange data with other applications, such as spreadsheet applications, that support the dynamic data exchange (DDE) protocol.

DDE conversation. The exchange of data between a DDE client and a DDE server. See also cold-link conversation and hot-link conversation.

DDE server. An application that provides data to another DDE-enabled application. Contrast with DDE client. See also DDE conversation.
default. A value that is automatically supplied or assumed by the system or program when no value is specified by the user. The default value can be assigned to a push button or graphic push button.
default action. An action that will be performed when some action is taken, such as pressing the Enter key.
dereferencing. The action of removing the association between a part and an AS/400 database field.
design window. The window in the GUI designer on which parts are placed to create a user interface.
details view. A standard contents view in which a small icon is combined with text to provide descriptive information about an object.
dimmed. Pertaining to the reduced contrast indicating that a part can not be selected or directly manipulated by the user.
direct editing. The use of techniques that allow a user to work with an object by dragging it with a mouse or interacting with its pop-up menu.

DLL. Dynamic link library.

double-byte character set (DBCS). A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require double-byte character sets. Because each character requires 2 bytes, the typing, displaying, and printing of DBCS characters requires hardware and programs that support DBCS. Four double-byte character sets are supported by the system: Japanese, Korean, Simplified Chinese, and Traditional Chinese. Contrast with single-byte character set (SBCS).

double-click. To quickly press a mouse button twice.
drag. To use a mouse to move or to copy an object. For example, a user can drag a window border to make it larger by holding a button while moving the mouse. See also drag and drop.
drag and drop. To directly manipulate an object by moving it and placing it somewhere else using a mouse.
drop-down combination box. A variation of a combination box in which a list box is hidden until a user takes explicit acts to make it visible.

drop-down list. A single selection field in which only the current choice is visible. Other choices are hidden until the user explicitly acts to display the list box that contains the other choices.

dynamic data exchange (DDE). The exchange of data between programs or between a program and a datafile object. Any change made to information in one program or session is applied to the identical data created by the other program. See also DDE conversation, DDE client, DDE server.

Dynamic link library (DLL). A file containing executable code and data bound to a program at load time or run time, rather than during linking. The code and data in a dynamic link library can be shared by several applications simultaneously.

E

EBCDIC. Extended binary-coded decimal interchange code. A coded character set of 256 8-bit characters.

emphasis. Highlighting, color change, or other visible indication of conditions relative to an object or choice that affects a user’s ability to interact with that object or choice. Emphasis can also give a user additional information about the state of a choice or an object.

every field part. An area on a display where a user can enter information, unless the field is read-only. The boundaries of an entry field are usually indicated. In VisualAge RPG, you point and click on an entry field part in the parts palette or parts catalog and click it onto a design window.

error logging. Keeps track of errors in an error log. The editor takes you to the place in the source where the error occurred.

event. A signal generated as a result of a change to the state of a part. For example, pressing a button generates a Press event.

exception. (1) In programming languages, an abnormal situation that may arise during execution, that may cause a deviation from the normal execution sequence, and for which facilities exist in a programming language to define, raise, recognize, ignore, and handle it. (I) (2) In VisualAge RPG, an event or situation that prevents, or could prevent, an action requested by a user from being completed in a manner that the user would expect. Exceptions occur when a product is unable to interpret a user’s input.

EXE. The extension of an executable file.

EXE module. An EXE module consists of a main procedure and subprocedures. It is created when the EXE keyword is present on the control specification. All subroutines (BEGSR) must be local to a procedure. The EXE must contain a procedure whose name matches the name of the source file. This will be the main entry point for the EXE, that is, the main procedure.

export. A function that converts an internal file to some standard file format for use outside of an application. Contrast with import.

F

field. (1) An identifiable area in a window, such as an entry field where a user types text. (2) A group of related bytes, such as a name or amount, that is treated as a unit in a record.

file. A collection of related data that is stored and retrieved by an assigned name. A file can include information that starts a program (program-file object), contains text or graphics (data-file object), or processes a series of commands (batch file).

focus. Synonym for input focus.

font palette. A set of fonts that can be used to change the font of a part in your application’s GUI.
G

graph part. A part that allows the user to add a graph to the GUI. The graph styles available are line, bar, line and bar, or pie chart.

graphical user interface (GUI). A type of user interface that takes advantage of high-resolution graphics. A graphical user interface includes a combination of graphics, the object-action paradigm, the use of pointing devices, menu bars and other menus, overlapping windows, and icons.

graphic push button part. A push button, labeled with a graphic, that represents an action that will be initiated when a user selects it. Contrast with push button part.

group box part. A rectangular frame around a group of controls to indicate that they are related and to provide an optional label for the group. In VisualAge RPG, you point and click on a group box part in the parts palette or parts catalog and click it onto a design window.

group marker. A mark that identifies a part as being the first one in a group. When a user moves the cursor through a group of parts and reaches the last part, the cursor returns to the first part in the group.

GUI designer. A suite of tools used to create interfaces by dragging and dropping parts from the parts palette to the design window.

H

hide button. A button on a title bar that a user clicks on to remove a window from the workplace without closing the window. When the window is hidden, the state of the window, as represented in the window list, changes. Contrast with maximize button and minimize button.

horizontal scroll bar part. A part that adds a horizontal scroll bar to a window. This part allows users to scroll through a pane of information, from left-to-right or right-to-left.

hot-link conversation. In DDE, an automatic update of a client program by a server program when data changes on the server. Contrast with cold-link conversation.

I

ICO. The file extension of an icon file.

icon. A graphical representation of an object, consisting of an image, image background, and a label.

icon view. A standard contents view in which each object contained in a container is displayed as an icon.

image part. A part used to display a picture, from a BMP or ICO file, on a window.

import. A function that converts AS/400 display file objects to the appropriate VARPG part. Contrast with export.

inactive window. A window that cannot receive keyboard input at a given moment.

index. The identifier of an entry in VARPG parts such as list boxes or combination boxes.

information area. A part of a window in which information about the object or choice that the cursor is on is displayed. The information area can also contain a message about the normal completion of a process. See also status bar.

Information Presentation Facility (IPF). A tool used to create online help on a programmable workstation.

Information Presentation Facility (IPF) file. A file in which the application's help source is stored.

INI. The file extension for a file in the OS/2 or Windows operating system containing application-specific information that needs to be preserved from one call of an application to another.

input focus. The area of a window where user interaction is possible from either the keyboard or the mouse.
**input/output (I/O).** Data provided to the computer or data resulting from computer processing.

**IPF.** Information Presentation Facility

**item.** In dynamic data exchange, a unit of data. For example, the top left cell position in a spreadsheet is row 1, column 1. This cell position may be referred to as item R1C1.

**J**

**JAR files (.jar).** In Java, abbreviation for Java ARchive. A file format that is used for aggregating many files into one.

**Java.** An object-oriented programming language for portable interpretive code that supports interaction among remote objects. Java was developed and specified by Sun Microsystems, Incorporated.

**java bean part.** A part that allows VARPG applications to access Sun Microsystem’s JavaBeans.

**JavaBeans.** In Java, a portable, platform-independent reusable component model.

**Java Database Connectivity (JDBC).** An industry standard for database-independent connectivity between Java and a wide range of databases. The JDBC provides a call-level application programming interface (API) for SQL-based database access.

**Java 2 Software Development Kit (J2SDK).** Software that Sun Microsystems distributes for Java developers. This software includes the Java interpreter, Java classes, and Java development tools. The development tools include a compiler, debugger, disassembler, AppletViewer, stub file generator, and documentation generator.

**Java Native Interface (JNI).** A programming interface that allows Java code that runs inside of a Java Virtual Machine (JVM) to interoperate with functions that are written in other programming languages.

**Java Runtime Environment (JRE).** A subset of the Java Developer Kit for end users and developers who want to redistribute the JRE. The JRE consists of the Java Virtual Machine, the Java Core Classes, and supporting files.

**Java Virtual Machine (JVM).** The part of the Java Runtime Environment (JRE) that is responsible for interpreting Java bytecodes.

**L**

**link event.** An event that a target part receives whenever the state of a source part changes.

**list box part.** A control that contains scrollable choices that a user can select. In VisualAge RPG, you can point and click on a list box part in the parts palette or parts catalog and click it onto a design window.

**M**

**main procedure.** A main procedure is a subprocedure that can be specified as the program entry procedure and receives control when it is first called. A main procedure is only produced when creating an EXE. See EXE module

**main source section.** In a VARPG program, the main source section contains all the global definitions for a module. For a component, this section also includes the action and user subroutines.

**main window.** See primary window.

**manipulation button.** See mouse button 2.

**maximize button.** A button on the rightmost part of a title bar that a user clicks on to enlarge the window to its largest possible size. Contrast with minimize button, hide button.

**media panel part.** A part used to give the user control over other parts. For example, a media panel part can be used to control the volume of a media part.

**media part.** A part that gives a program the ability to process sound files and video files.
**menu.** A list of choices that can be applied to an object. A menu can contain choices that are not available for selection in certain contexts. Those choices are dimmed.

**menu bar part.** The area near the top of a window, below the title bar and above the rest of the window, that contains choices that provide access to other menus. In VisualAge RPG, you can point and click on a menu bar part in the parts palette or parts catalog and click it onto a design window.

**menu item part.** A part that is a graphical or textual item on a menu. A user selects a menu item to work with an object in some way.

**message.** (1) Information not requested by a user but displayed by a product in response to an unexpected event or when something undesirable could occur. (2) A communication sent from a person or program to another person or program.

**message file.** A file containing application messages. The file is created from the message source file during the build process. See also build.

**message subfile part.** A part that can display predefined messages or text supplied in program logic.

**migrate.** (1) To move to a changed operating environment, usually to a new release or version of a system. (2) To move data from one hierarchy of storage to another.

**MID.** The file extension of a MIDI file.


**minimize button.** A button, located next to the rightmost button in a title bar, that reduces the window to its smallest possible size. Contrast with maximize button and hide button.

**mnemonic.** A single character, within the text of a choice, identified by an underscore beneath the character. See also mnemonic selection.

**mnemonic selection.** A selection technique whereby a user selects a choice by typing the mnemonic for that choice.

**mouse.** A device with one or more push buttons used to position a pointer on the display without using the keyboard. Used to select a choice or function to be performed or to perform operations on the display, such as dragging or drawing lines from one position to another.

**mouse button.** A mechanism on a mouse used to select choices, initiate actions, or manipulate objects with the pointer. See also mouse button 1 and mouse button 2.

**mouse button 1.** By default, the left button on a mouse used for selection.

**mouse button 2.** By default, the right button on a mouse used for manipulation.

**mouse pointer.** Synonym for cursor.

**multiline edit (MLE) part.** A part representing an entry field that allows the user to enter multiple lines of text.

**N**

**navigation panel.** A group of buttons that can be used to control the visible selection of records in a subfile.

**NOMAIN module.** A module that contains only subprocedures. There are no action or standalone user subroutines in it. A NOMAIN module is created when the NOMAIN keyword is present on the control specification.

**notebook part.** A graphical representation of a notebook. You can add notebook pages to the notebook part and then group the pages into sections separated by tabbed dividers. In Windows, a notebook is sometimes referred to as a Windows tab control. See also notebook page part, notebook page with canvas part.

**notebook page part.** A part used to add pages to a notebook part. See also notebook.

**notebook page with canvas part.** A combination of a notebook page part and a canvas page part. See also notebook, canvas part.
object. (1) A named storage space that consists of a set of characteristics that describe itself and, in some situations, data. An object is anything that exists in and occupies space in storage and on which operations can be performed. Some examples of objects are programs, files, libraries, and folders. (2) A visual component of a user interface that a user can work with to perform a task. An object can appear as text or an icon.

object-action paradigm. A pattern for interaction in which a user selects an object and then selects an action to apply to that object.

object-oriented programming. A method for structuring programs as hierarchically organized classes describing the data and operations of objects that may interact with other objects. (T)

object program. A target program suitable for execution. An object program may or may not require linking. (T)

odbc/jdbc part. A part that allows VAPRG applications to access and process database files that support the Windows ODBC API or Sun Microsystems’s JDBC API.

operating system. A collection of system programs that control the overall operation of a computer system.

outline box part. A part that is a rectangular box positioned around a group of parts to indicate that all the parts are related.

P

package. A function used to collect all the parts of a VAPRG application together for distribution.

parts. Objects that make up the GUI of a VAPRG application.

parts catalog. A storage space for all of the parts used to create graphical user interfaces for VAPRG applications.

parts palette. A collection of parts that are most appropriate for building the current graphical user interface for an application. When you finish one GUI, you can wipe the palette clean and add parts from the parts catalog that you require for the next application.

plugin. A function created by the user or an outside vendor that can be used in VAPRG programs.

point and click. (1) A selection method which is used to copy a part from the parts palette or catalog to the GUI design window, the icon view, or the tree view. (2) To place a part in any of the desired views, point to and click on the part, then move the cursor to the chosen window and point the cursor and click where you want the part to appear. In the icon and tree views, the part will be placed on the parent part, and you will then have to move it where you would like it to appear in the design window.

pop-up menu. A menu that, when requested, appears next to the object with which it is associated. It contains choices appropriate for the object in its current context.

pop-up menu part. A part that, when added to an object on your interface, appears next to the object with which it is associated when requested. You can point and click on a pop-up menu part in the parts palette or parts catalog and click it onto a design window.

pop-up window. A movable window, fixed in size, in which a user provides information required by an application so that it can continue to process a user request. Synonymous with secondary window.

primary window. The window in which the main interaction between the user and the application takes place. Synonymous with main window.

procedure. A procedure is any piece of code that can be called with the CALLP operation code.

procedure interface definition. A procedure interface definition is a repetition of the prototype information within the definition of a procedure. It is used to declare the entry parameters for the procedure and to ensure that the internal definition of the procedure is consistent with the external definition (the prototype)
**programmable workstation (PWS).** A workstation that has some degree of processing capability and that allows a user to change its functions.

**progress bar part.** A part that can be used to indicate graphically the progress of a process, such as copying files, loading a database, and so on.

**progress indicator.** One or more controls used to inform a user about the progress of a process.

**project.** The complete set of data and actions needed to build a single target, such as dynamic link library (DLL) or an executable file (EXE).

**prompt.** (1) A visual or audible message sent by a program to request the user’s response. (T) (2) A displayed symbol or message that requests input from the user or gives operational information. The user must respond to the prompt in order to proceed.

**properties notebook.** A graphical representation that resembles a bound notebook containing pages separated into sections by tabbed divider pages. Select the tabs of a notebook to move from one section to another.

**prototype.** A prototype is a definition of the call interface. It includes information such as: whether the call is bound (procedure) or dynamic (program); the external name; the number and nature of the parameters; which parameters must be passed; the data type of any return value (for a procedure)

**pull-down menu.** A menu that extends from a selected choice on a menu bar or from a system-menu symbol. The choices in a pull-down menu are related to one another in some manner.

**push button part.** A button labeled with text that represents an action that starts when a user selects the push button. You can point and click on a push button part in the parts palette or parts catalog and click it onto a design window. See also graphic push button part.

**PWS.** Programmable workstation.

**R**

**radio button part.** A circle with text beside it. Radio buttons are combined to show a user a fixed set of choices from which only one can be selected. The circle is partially filled when a choice is selected. You can point and click on a radio button part in the parts palette or parts catalog and click it onto a design window.

**reference field.** An AS/400 database field from which an entry field part can inherit its characteristics.

**restore button.** A button that appears in the rightmost corner of the title bar after a window has been maximized. When the restore button is selected, the window returns to the size and position it was in before it was maximized. See also maximize button.

**S**

**SBCS.** Single-byte character set.

**scroll bar.** A part that shows a user that more information is available in a particular direction and can be moved into view by using a mouse or the page keys.

**secondary window.** A window that contains information that is dependent on information in a primary window, and is used to supplement the interaction in the primary window. See also primary window. Synonym for pop-up window.

**secure sockets layer (SSL).** A popular security scheme that was developed by Netscape Communications Corp. and RSA Data Security, Inc. SSL allows the client to authenticate the server and all data and requests to be encrypted. The URL of a secure server that is protected by SSL begins with https rather than http.

**selection border.** The visual border that appears around a VARPG part or a custom-made part, allowing the part to be moved with the mouse or keyboard.

**selection button.** See mouse button 1.
server. A system in a network that handles the requests of another system, called a client.

server alias. A name you define that can be used instead of the server name.

shared component. A component that can be accessed by more than one project.

single-byte character set (SBCS). A character set in which each character is represented by a one-byte code. Contrast with double-byte character set (DBCS).

sizing border. The border or frame around a part (or set of parts) that you select to resize the part (or set of parts) using the mouse or the keyboard.

slider part. A visual component of a user interface that represents a quantity and its relationship to the range of possible values for that quantity. A user can also change the value of the quantity. You can point and click on a slider part in the parts palette or parts catalog and click it onto a design window.

slider arm. The visual indicator in the slider that a user can move to change the numerical value.

source directory. The directory in which all source files for a VARPG application are stored.

source part. A part that can notify target parts whenever the state of the source part changes. A source part can have multiple targets.

spin button part. A type of entry field that shows a ring of related but mutually exclusive choices through which a user can scroll and select one choice. A user can also type a valid choice in the entry field. You can point and click on a spin button part in the parts palette or parts catalog and click it onto a design window.

SSL. Secure sockets layer.

static text part. A part used as a label for other parts, such as a prompt for an entry field part.

status bar. A part of a window that displays information indicating the state of the current view or object. See also information area.

status bar part. A part on a window that can display additional information about a process or action for the window.

subfile field. A field used to define fields in a subfile part. See also subfile part.

subfile part. A part used to display a list of records, each consisting of a number of fields. This part is similar to an AS/400 subfile. See also subfile field.

submenu. A menu that appears from, and contains choices related to, a cascading choice in another menu. Submenus are used to reduce the length of a pull-down menu or a pop-up menu. See also submenu part.

submenu part. A part used to start a submenu from a menu item or existing menu, or to start a pull-down menu from a menu item on a menu bar. See also submenu and menu item part.

subprocedure. A subprocedure is a procedure specified after the main source section. It must have a corresponding prototype in the definition specifications of the main source section.

syntax checking. Verifies that the syntax of each line is correct while you are editing the source. By doing so, it can avoid compile errors. You can set this option on or off. You can view only certain specification types, such as C specs, or a line with a specific string.

T

tab stop. An attribute used to set a tab stop for a part so that users can focus on it when they use the Tab key to move through the interface.

target part. A part that receives a link event from a source part whenever the state of the source part changes.

target directory. The directory in which the compiled VARPG application is stored after a build. Contrast with target folder.
**target folder.** The object in which the icon representing a VARPG application is placed.

**target program.** The object to be built by the project, such as a dynamic link library (DLL).

**thread.** The smallest unit of operation to be performed within a process.

**timer part.** A part used to track the interval of time between two events and trigger the second event when the interval has passed.

**title bar.** The area at the top of each window that contains the system-menu symbol.

**token highlighting.** Enhances the readability of the code. You can configure highlighting of different language constructs with different colors or fonts to identify the program structures. You can turn token highlighting on or off.

**tool bar.** A menu that contains one or more graphical choices representing actions a user can perform using a mouse.

**topic.** In dynamic data exchange (DDE), the set of data that is the subject of a DDE conversation.

**tree view.** A way of displaying the contents of an object in a hierarchical fashion.

**U**

**user-defined part.** A part, consisting of one or more parts you have customized, that you save to the parts palette or parts catalog for reuse. When in the palette or catalog, you can point and click this part onto the design window as you would any other VARPG part.

**utility DLL.** See NOMAIN module

**V**

**vertical scroll bar part.** A part that adds a vertical scroll bar to a window. This part allows users to scroll through a pane of information vertically.

**W**

**WAV.** The file extension of a wave file.

**wave file.** A file used for audio sounds on a waveform device.

**window part.** An area with visible boundaries that represents a view of an object or with which a user conducts a dialog with a computer system. You can point and click on a window part from the parts palette or parts catalog and click it onto the project window.

**window with canvas part.** A combination of the window part and the canvas part. See also window part and canvas part.

**work area.** An area used to organize objects according to a user’s tasks. When a user closes a work area, all windows opened from objects contained in the work area are removed from the workplace.

**workplace.** An area that fills the entire display and holds all of the objects that make up the user interface.

**workstation.** A device that allows a user to do work. See also programmable workstation.
Bibliography

For additional information about topics related to WebSphere Development Studio Client, refer to the following IBM publications:

WebSphere Development Studio Client manuals:
- *Getting Started with WebSphere Development Studio Client for iSeries, SC09-2625-06*, provides information about WebSphere Development Studio Client for iSeries, giving an overview of the various components, how they work together, and the business advantages of using them.

VisualAge RPG manuals:
- *Programming with VisualAge RPG, SC09-2449-05*, contains specific information about creating applications with VisualAge RPG. It describes the steps you have to follow at every stage of the application development cycle, from design to packaging and distribution. Programming examples are included to clarify the concepts and the process of developing VARPG applications.
- *VisualAge RPG Parts Reference, SC09-2450-05*, provides a description of each VARPG part, part attribute, part event, part attribute, and event attribute. It is a reference for anyone who is developing applications using VisualAge RPG.
- *Java for RPG Programmers* introduces you to the Java language (and RPG IV) by comparing it to the RPG language. It is a good first step in your Java journey. It also includes an interactive CD tutorial on Java and VisualAge for Java, by MINDQ.
- *Experience RPG IV Tutorial* is an interactive CD tutorial that teaches you RPG IV and ILE, in a fun and step-by-step approach. The book is a handbook with questions and exercises to help you get hands-on experience with this exciting new version of RPG.
- Another non-IBM book of interest to VisualAge RPG users is *VisualAge for RPG by Example*.

If you have internet access, you can obtain current iSeries and AS/400e information and publications from the following Web site:
http://www.ibm.com/eserver/iseries/infocenter

For the PDF version of iSeries publications, refer to the CD ROM *iSeries Information Center: Supplemental Manuals, SK3T-4092-00*.

Application Development Manager manuals:
- *ADTS/400: Application Development Manager Introduction and Planning Guide, GC09-1807-00*, describes the basic concepts and the planning needed to make effective use of the Application Development Manager feature.
- *ADTS: Application Development Manager User’s Guide, SC09-2133-02*, describes how to create and manage projects defined to the Application Development Manager feature.
• ADTS/400: Application Development Manager API Reference, SC09-2180-00, describes how application programmers can write their own interface to the Application Development Manager feature.

Information Presentation Facility manual:
• Information Presentation Facility Programming Guide G25H-7110, describes the elements that make up the Information Presentation Facility (IPF). IPF is a tool that supports the design and development of online documents and online help facilities.

SQL manuals:
• IBM SQL Reference Version 2 SC26-8416, Volume 2, compares the facilities of
  – DB2
  – SQL/DS™
  – DB2/400™
  – DB2/6000™
  – IBM SQL
  – ISO-ANSI (SQL92E)
  – X/Open™ (XPG4-SQL).
• DB2 Universal Database Administration Guide S10J-8157, provides information necessary to use and administer the DB2 product.
• DB2 Universal Database Embedded SQL Programming Guide S10J-8158, describes how to design and code application programs that access DB2 Client/Server family servers (such as DB2 or DB2/400). It presents detailed information on the use of Structured Query Language (SQL), and API calls in applications.
Index

Special characters
.BMP file using 84, 243

A
accessing picture files at build time 244 action subroutine modifying link events 26 action subroutine, invoking 26 ActiveX part attributes 47 creating 47 events 47, 50, 89 methods 48 properties 47 purpose of 46 AddItemEnd attribute 151 AddLink attribute controlling media part with 102 for media panel parts 103 AddMsgld attribute 108 AddMstTxt attribute 108 AddOffset attribute 113 AddRcd attribute 68 AllowLink attribute enabling media panel control by setting 102 for media panel parts 103 animation control part attributes 53 events 53 purpose of 53 applets calls 297 create 293 Java build options 294 runtime requirements 295 application installing 423 packaging 415 re-installing 423 re-packaging 423 removing 423 updating 423 Application files description 427 filename.DLL 228, 427 filename.EVT 427 filename.EXE 427 filename.HLP 427 filename.IP 427 filename.IP 427 filename.LIB 427 filename.LST 427 filename.ODF 427 filename.ODX 427 filename.RST 193, 427 filename.TX 427 filename.TXM 427 filename.VPF 427 filename.VPG 427 applications, thin client 431 Arrange attribute 70 array changing during debug session 233 displaying during debug session 233 AS/400 accessing files on the iSeries 400 server 34 creating data files for the sample programs 34 filename.RST 427 messages for translation 262 reusing applications from 213 reusing UIM help 222 attributes AddItemEnd 151 AddLink 102, 103 AddMsgId 108 AddMsgTxt 108 AddOffset 113 AddRcd 68 AllowLink 102, 103 attributes (continued) Arrange 70 AudioMode 101 BackColor 37 BackMix 37, 146 Bottom 37 CharOffset 113 Checked 58, 106 checking event and system attributes 27 ColNumber 69, 160 Count 69, 94, 159 DDEMode 267 DeSelect 62, 93, 94 DragEnable 41 DropEnable 41 Enabled 37, 77, 107, 113 FileName 81, 85, 101, 243 FirstSel 62 Focus 38 FontName 155 FontSize 155 for ActiveX parts 47 for animation control parts 53 for canvas parts 57 for check box parts 58 for combination box parts 60, 61 for component reference parts 65 for container parts 67 for DDE client parts 74 for entry field parts 75 for graphic push button parts 81 for group box parts 82 for horizontal scroll bar parts 83 for image parts 85 for Java Bean parts 89 for list box parts 78, 92 for media panel parts 103 for menu item parts 106 for message subtitle parts 108 for multiline edit parts 112 for notebook canvas parts 119 for notebook page parts 118 for notebook parts 117 for ODBC/JDBC interface parts 120 for outline box parts 137 for pop-up menu parts 138 for positioning parts 37 for progress bar parts 139 for push button parts 140 for radio button parts 142 for slider parts 145 for static text parts 155 for status bar parts 157 for subtitle parts 158 for submenu parts 171 for timer parts 172 for vertical scroll bar parts 179 for window with canvas parts 181 for window without canvas parts 180

© Copyright IBM Corp. 1994, 2005 471
attributes (continued)
  ForeColor 37
  ForeMix 37
  GetItem 94
  GetNewID 68
  GetRcdText 67
  getting and setting 25
  Height 37, 137
  Index 61, 62, 94, 160
  InfoLabel 39
  InsertItem 61, 93
  InsertLine 112
  InsertMode 76
  Interval 172
  Label 39, 82, 106, 140, 155
  Left 37
  LineNumber 112
  Masked 77
  Maximum 151
  Minimum 151
  MsgBoxText 109
  Multiplier 172
  OpenEdit 160
  OpenImmediately 181, 182
  Panel 85
  PanelItem 104
  ParentName 35
  PartName 35
  PartType 35
  Position 102, 104
  ReadOnly 62, 77, 113, 152
  RecordID 69
  RemoveItem 62, 93
  RemoveMsg 109
  RemoveRcd 70
  Selected 62, 93, 94
  SelectItem 64
  Sequence 61, 92
  SetItem 61, 93
  SetRcdIcon 70
  SetRcdText 67
  SetTop 62, 93
  TabLabel 39
  Terminate on close 186
  Text 62, 76, 112, 151
  TextEnd 113
  TextSelect 113
  TextStart 113
  TimerMode 173
  UserData 39
  Validate 76
  Value 145, 151, 173
  View 73
  Visible 38, 172, 181
  Volume 102, 104
  Width 37, 137
  AudioMode attribute 101

B
  BackColor attribute, common uses of 37
  backmatter
    common uses of 37
    for slider parts 146
  BEGACT operation code, responding to events with 26
  bibliography 469

bitmaps, using 243
  Bottom attribute, common uses of 37
  breakpoint
    setting 229, 231
  calendar 54
  purpose of 54
  CALL operation code, example of 273
  CALLB operation code
    calling local functions 267, 268
    calling local programs
      calling local programs 267
      functions using named constants 268
      functions using procedure pointer 269
      functions without required parameters 269
      local functions 268
      local programs 267
      remote programs 272
  canvas part
    attributes 57
    events 57
    purpose of 56
  CHAIN (random retrieval from file)
  operation code 159
  Change event
    and multiline edit parts 113
    for media panel parts 104
    for slider parts 145
  changing a pointer value while debugging 236
  debugger views 236
  the contents of a field while debugging 234
  the representation while debugging 234
  variables, arrays, and structures while debugging 233
  changing position of parts 182
  CharOffset attribute 113
  check box part
    attributes 58
    events 58, 59
    getting and setting states 58
    purpose of 58
  Checked attribute
    for check box parts 58
    for menu item parts 106
  CLEAR operation code
    for subfile parts 159
    purpose of 29
  Close event 186
  ColNumber attribute 69, 160
  color of converted parts after import 221
  combination box part
    adding and changing items 61
    and data transfer 41
    attributes 60
    events 61
    order of items 61
    purpose of 60
    removing items 62
    retrieving a user-selected item 62
  combination box part (continued)
    selecting and deselecting items 62
    common attributes, description of 35
  compiling programs
    filename.EVT 427
    filename.LST 427
  Complete event 102
  component reference part
    attributes 65
    communication between components 265
    events 65
    example 65
    purpose of 65
  components
    communication between 265
    starting 272
    stopping 272
  container part
    attributes 67
    changing views of 70
    events 67
    example adding records to 69
    example removing records from 70
    example updating data in 69
    purpose of 67
  container views, changing 70
  context-sensitive help 246
  control language (CL) program
    ALCOBJ 199
    CVTRPGSRC, ILE RPG conversion tool 224
    QCMD DDM 194
    QCMD EX 194
    STRP CMD 274
  controlling server connections 201
  conversion
    RPG source code using CVTRPGSRC 224
  Count attribute
    for container parts 69
    for list box parts 94
    for subfile parts 159
  Create event, example of 38
  CVTRPGSRC, ILE RPG/400 conversion tool 224

D
  data area overrides 195
  data transfer
    example 42
    parts that support 41
    using 41
  DDE client
    attributes 74
    determining if programs support 74
    events 74
    purpose of 74
  DDEAddLink attribute
    using 267
  DDEMode attribute 267
  debugger
    breakpoints 228
    changing debugger views 236
    changing the contents of a field 234
    changing the representation 234
Double Byte Character Set
application development considerations

DBCS Either data type 381, 382
DBCS Mixed data type 381, 383
DBCS Only data type 381, 382
GETATR operation code 382, 383

Graphic data type 383
Pure DBCS 383

Double Byte Character Set (continued)

SETATR operation code 382, 383
DragEnable attribute 41
DropEnable attribute 41
DSPLY 262

E
edit codes
formatting data into predefined formats 239
purpose of 239
user-defined 240
edit words
body of 241
correcting improperly formatted output 241
expansion positions of 242
parts of 241
purpose of 239, 240
status of 242
editing
data in entry fields and static text parts 239, 240
help files 222
messages 261
RPG source 224
Enabled attribute
common uses of 37
for entry field parts 77
for menu item parts 107
for multiline edit parts 113
ENDACT operation code, responding to events with 26
Enter event
for combination box parts 64
for list box parts 94, 170
entry field part
and data transfer 41
attributes 75
clearing 187
events 76
overriding defined values 29
purpose of 75
starting components 272
storing read values 29
error messages
error referencing parts 182
improperly formatted output 241
Event attributes
defining event and system attributes 27
purpose of 26
event attributes, using 26
events
Change 104, 113, 145
checking for event attribute errors 27
Close 186
coding BEGACT and ENDACT 26
Complete 102
description of attributes 26
Enter 94, 170
for ActiveX parts 47
for animation control parts 53
for check box parts 59
for combination box parts 64
for DDE client parts 74

events (continued)
for entry field parts 76
for graphic push button parts 81
for group box parts 82
for horizontal scroll bar parts 83
for image parts 85
for JavaBean parts 89
for list box parts 92
for media panel parts 103
for media parts 101
for menu bar parts 105
for menu item parts 106
for message subfile parts 108
for multiline edit parts 112
for notebook page parts 118
for notebook parts 117
for ODBC/JDBC Interface parts 121
for outline box parts 137
for progress bar parts 139
for push button parts 140
for radio button parts 142
for slider parts 145
for spin button parts 151
for static text parts 155
for status bar parts 157
for subfile parts 158
for submenu parts 171
for timer parts 172
for vertical scroll bar parts 179
for window with canvas parts 181
for window without canvas parts 180
GofFocus 38, 113
invoking action subroutines 26
listing events for a part 26
LostFocus 38
MenuSelect 107
Notify 66
Press 81, 141
responding to events in your program 26
Select 59, 94, 144, 170
Tick 172
examples
adding records to a container part 69
getting and setting values for spin button parts 152
grouping radio buttons 142
data transfer 42
of parts sharing a program field 30
reading and modifying subfile records 160
removing records from container parts 70
resizing a window 183
updating container parts 69
using component reference part 65
using Create event for a window 38
using subfile part to display database records 161
using subfiles to display server data 161
using the image part 86
using the list box part 94
using the message subfile part 110
using the multiline edit part 113
using the slider part 146

Index 473
examples (continued)
   using the timer part 173
   Video Store Catalog application 3
   window part 187
exchanging information with other PWS applications 265
execute subroutine
invoking action subroutines with 26

F
field parts
   unique names 30
file aliases (overrides) 196
FileName attribute
   for graphic push button parts 81
   for image parts 85
   for media parts 101, 243
finding a message
   a message 261
FirstSel attribute 62
Focus attribute, common uses of 38
FontName attribute 155
FontSize attribute 155
ForeColor attribute, common uses of 37
ForeMix attribute, common uses of 37

G
GETATR
   using 25
GetItem attribute 94
GetNewID attribute 68
GetRcdText attribute 67
getting the record count
   count of records in a subfile 159
   part attributes 25
   state of check box parts 58
   state of radio button parts 144
   text attribute for multiline parts 112
   value for slider parts 145
   values for spin button parts 151
glossary 457
GotFocus event
   and multiline edit parts 113
   common uses of 38
graph 78
   purpose of 78
graphic data type 383
graphic push button part
   attributes 81
   events 81
   purpose of 80
group box attributes 82
   events 82
   purpose of 82
   grouping radio buttons, example 142

H
Height attribute
   common uses of 37
   for outline box parts 137
help
   adding graphics to 245
help (continued)
   creating a help push button 246
   creating for Windows 249
   creating hypertext links 246
   editing 222
   filename.IPIM 427
   filename.VPF 427
   for Java applications 253
   planning your application 19
   reusing UIM 222
   translating 245
   types of 246
   help push button, creating 246
   hidden subfile fields 161
   horizontal scroll bar
      attributes 83
      events 83
      purpose of 83
   hypertext links, creating 246
   Java bean part
      associated JARs 90
      attributes 89
      classpath setup 90
      creating 89
      properties and methods 91
      purpose of 89
   java methods, calling 279
   Java methods, prototyping 280
   java restrictions 287
   java runtime differences 290
   java source changes 288
   java, compiling 287
   JavaHelp, creating 253
   Label attribute
      common uses of 39
      for group box parts 82
      for menu item parts 106
      for push button parts 140
      for static text parts 155
      purpose of 30
   labels
      description 39
      substitution 259
   Left attribute, common uses of 37
   level checking 199
   library lists
      Define iSeriesInformation notebook
      considerations 196
      job description 194
      QCMDDDM 194
      QCMDExC 194
      setting up a server 194
   LineNumber attribute 112
      for multiline edit parts 112
   linking parts 265
   list box part
      and data transfer 41
      attributes 78, 92
      events 92
      purpose of 92
      locking database files 199
   LostFocus event, common uses of 38

M
Make Message File utility 262
Masked attribute 77
Maximize button 183
Maximum attribute 151
media panel part
   attributes 103
   controlling media part with 102
   events 103
   purpose of 103
media part
   attributes 101
   controlling with media panel part 102
   events 101
   purpose of 101
   signaling events 102
menu bar
  attributes 105
  events 105
  purpose of 105

menu item part
  attributes 106
  events 105, 106
  purpose of 106

menubar
  purpose of 105
MenuSelect event
  for menu item parts 107
message subfile part
  and data transfer 41
  attributes 108
  events 108
  example of 110
  purpose of 108
messages
  choosing type of 260
  compiling for translation 262
  creating 259, 260
  deleting 261
  designing 21
  editing 261
  editing for translation 263
  filename.TXM 427
  finding 261
  types of 259
  using as labels 263
  using with logic 262
Minimize button 183
Minimum attribute 151

mnemonics
  for check box parts 59
  for menu items 106
  for notebook pages 118
  for push buttons 140
  for radio buttons 142
  translating 22

modifying
  link events to action subroutines 26
  resource IDs 427
MsgSubText attribute 109

multiline edit part
  and data transfer 41
  attributes 112
  events 112
  example of 113
  purpose of 112
multiple procedures
  prototyped call 275
Mutliplier attribute 172
MultiSelect attribute
  for list box parts 78, 92
  for subfile parts 158

N
non-GUI programs 375
non-GUI programs from DOS 447

notebook page part
  attributes 118
  events 118
  purpose of 118

notebook page with canvas part
  (continued)
  events 119
  purpose of 119
notebook part
  attributes 117
  events 117
  purpose of 117

Notify event 66

O
ODBC/JDBC interface part
  access table data 122
  attributes 120, 139
  connect to a database 121
  create a record set 121
  data types 122
  events 121, 139
  purpose of 120
  retrieve table rows 123

Open Immediately attribute 181
OpenEdit attribute 160

operation codes
  CALLB 268
  CHAIN 159
  CLEAR 159, 187
  DELETE 159
  READ 76
  READC 159
  READS 159
  SETATR 85
  SHOWWIN 182
  START 65, 272
  STOP 272
  UPDATE 159
  WRITE 76, 159

Outline Box part
  attributes 137
  events 137
  purpose of 137

overrides
  accessing data areas 195
  accessing database files 196
  calling iSeries server programs 273

P
packaging
  application 415
  prerequisites 415
  runtime code 415
Packaging utility 415
Panel attribute 85
Panelltem attribute 104
ParentName attribute, common uses of 35

part colors
  common uses of 37
  slider part example 146
part type
  description 275
PartName attribute, common uses of 35
  "component 188
  ActiveX 46

parts (continued)
  animation control 53
  canvas 56
  changing colors of 37
  check box 58
  combination box 60
  Combination box 60
  component reference 65
  Component Reference 65
  container 67
  Container 67
  DDE client 74
  enabling parts 37
  entry field 75
  graph 78
  graphic push button 80
  group box 82
  horizontal scroll bar 83
  image 84
  Java Bean 89
  linking 265
  list box 86, 92
  listing events for a part 26
  media 101
  media panel 103
  menu bar 105
  menu item 106
  message subfile 108
  multiline edit 112
  notebook 117
  notebook page 118
  notebook page with canvas 119
  ODBC/JDBC interface 120
  outline box 137
  placement on various monitor resolutions 36
  pop-up menu 138
  positioning 37
  progress bar 139
  push button 140
  radio button 142
  referencing 25
  slider 145
  spin button 151
  static text 155
  status bar 157
  subfile 158
  submenu 171
  support data transfer 41
  timer 172
  vertical scroll bar 179
  window 180, 181
  window frame 180

PartType attribute, common uses of 35

Picture file
  for image parts 84
  using 243

pictures, adding 243
planning your application 19

pointer
  changing the value while debugging 236
  displaying while debugging 234
pop-up menu part
  attributes 138
  events 138
  purpose of 138

Index 475
Position attribute
  for media panel parts 104
  setting 102
position of parts, changing 182
positional entries, and conversion during import 218
Press event
  for graphic push button parts 81
  for push button parts 141
programs, non-GUI 375
progress bar part
  purpose of 139
prototyped call
  prototyping, Java methods 280
publications, list of 469
push button part
  attributes 140
  events 140
  purpose of 140

Q
QCMDDDM
  changing the library list 194
QCMDDDM, changing the library list 194
QCMDEXC
  changing the library list 194
QCMDEXC, changing the library list 194

S
sample programs
  building 34
  installing 33
  running 34
  special instructions for samples requiring iSeries server data 34
secure sockets layer setup 449
Select event
  for combination box parts 64
  for list box parts 94
  for radio button parts 144
  for subfile parts 170
  signaling 59
Selected attribute
  for combination box parts 62
  for list box parts 93, 94
  selecting items in combination boxes 62
SelectItem attribute 64
Sequence attribute
  for combination box parts 61
  for list box parts 92
server connections, runtime control 201
servers
  accessing data areas 195
  accessing database files 196
  calling iSeries 400 programs with workstation files 274
  calling server programs 273
  database considerations 200
  defining iSeries Information 193
  issuing CL commands 194
  level checking 199
  library list considerations 194
  locking database files 199
  notebook considerations 193
  overriding database files 199
  setting up for developing/running applications 194
servers (continued)
  using your application as a DDE Server 266
SETATR
  using 25
SETATR (set attribute) operation code for image parts 85
  reflecting stored values on the screen 29
SetItem attribute
  for combination box parts 61
  for list box parts 93
SetRcdIcon attribute 70
SetRcdText attribute 67, 70
setting
  debug breakpoints 229
  debug fonts 237
SetTop attribute
  for combination box parts 62
  for list box parts 93
sharing program fields, example of parts 30
SHOWWIN operation code
  loading window into memory 182
Signon API, sample program 205
Signon API, using 201
slider part
  attributes 145
  events 145
  purpose of 145
sound files, using 243
sound, adding 243
source code
  editing 224
  filename.VPG 427
spin button part
  events 151
  example of 152
  purpose of 151
SSL setup 449
standalone programs 375
START (start a component) operation code
  and component reference parts 65
  calling local programs using 270
  description 272
  restrictions when calling local programs with 271
starting
  debug window 227
  the debugger 227
starting a component 272
starting components
  starting components 272
static text
  and data transfer 41
  attributes 155
  events 155
  overriding defined values 29
  purpose of 155
  storing read values 29
  unique names 30
status bar part
  attributes 157
  events 157
  purpose of 157

R
radio button part
  attributes 142
  events 142
  example showing how to group 142
  purpose of 142
re-packaging 423
READ (read a record) operation code
  database files 198
  purpose of 29
READC (read next modified record) operation code 159
ReadOnly attribute
  for combination box parts 62
  for entry field parts 77
  for multiline edit parts 113
  for spin button parts 152
READS (read selected record from subfile) operation code 159
RecordID attribute 69
recursion
  recursive calls 277
referencing
  parts on different windows 25
  parts on the same windows 25
RemoveItem attribute 62, 93
RemoveMsg attribute 109
RemoveRcd attribute 70
removing
  an application 423
  the runtime code 423
RESET
  purpose of 29
resizing windows 183
reusing
  applications from iSeries 400 213
  display files 217, 218
  RPG source 224
  UIM help files 222
RGB color value 37
RPC source 224
running with breakpoints
  debug breakpoints 231
  programs while debugging 232
runtime
  deleting 423
  filename.DLL 427
  filename.EXE 427
  filename.HLP 427
  filename.ODX 427
  filename.RST 427
  re-installing 423
  re-packaging 423
  removing 423
  updating 423
runtime code
  installing 423
  packaging 415
servers (continued)
  using your application as a DDE Server 266
SETATR
  using 25
SETATR (set attribute) operation code for image parts 85
  reflecting stored values on the screen 29
SetItem attribute
  for combination box parts 61
  for list box parts 93
SetRcdIcon attribute 70
SetRcdText attribute 67, 70
setting
  debug breakpoints 229
  debug fonts 237
SetTop attribute
  for combination box parts 62
  for list box parts 93
sharing program fields, example of parts 30
SHOWWIN operation code
  loading window into memory 182
Signon API, sample program 205
Signon API, using 201
slider part
  attributes 145
  events 145
  purpose of 145
sound files, using 243
sound, adding 243
source code
  editing 224
  filename.VPG 427
spin button part
  events 151
  example of 152
  purpose of 151
SSL setup 449
standalone programs 375
START (start a component) operation code
  and component reference parts 65
  calling local programs using 270
  description 272
  restrictions when calling local programs with 271
starting
  debug window 227
  the debugger 227
starting a component 272
starting components
  starting components 272
static text
  and data transfer 41
  attributes 155
  events 155
  overriding defined values 29
  purpose of 155
  storing read values 29
  unique names 30
status bar part
  attributes 157
  events 157
  purpose of 157

476 Programming with VisualAge RPG
stepping over
while debugging 232
STOP (stop a component) operation code
description 272
subfile part
attributes 158
events 158
example displaying server data 161
example of reading and updating
records 160
hidden fields 161
purpose of 158
submenu part
attributes 171
events 171
purpose of 171
substitution labels
defining text for 259
description 39
system attributes
%DspHeight 27, 38
%DspWidth 27, 38
System attributes
checking for system attribute
text 228
errors 27

**V**

Validate attribute 76
Value attribute
for slider parts 145
for spin button parts 151
for timer parts 173
Vendor Plugins
adding 389
creating 391
invoking 389
managing 389
vertical scroll bar
attributes 179
events 179
purpose of 179
Video Catalog application
adding messages 15
adding online help 15
creating the Comedy window 7
creating the Preview window 11
description of 3
designing 5
installing 3
running 4
View attribute 73
views, changing 70
Visible attribute
common uses of 38
for timer parts 172
for window parts 181
visual RPG
breakpoints list 230
changing a pointer value 236
changing variables, arrays, and
structures 233
debug startup information 229
debug tool bar 232
debug window 227
displaying a pointer value 234
displaying the assembly source
code 228
displaying the load occurrence
breakpoint 228
displaying variables, arrays, and
structures 233
running breakpoints 231
setting breakpoints 229
setting debug fonts 237
Volume attribute
for media panel parts 104
for media parts 102
windows
attributes 180, 181
attributes for operation codes 30
creating at startup 182
default settings 181
designing content of 20
displaying 181
displaying pictures on 243
events 180, 181
giving input focus 38
loading into memory 182
method for moving 182
OpenImmediately attribute 181
operation codes for 29
positioning without use of title
bar 182
purpose of 180, 181
referencing 182
resizing 183
setting focus 185
setting window list contents 185
specifying when to display 38
style considerations 21
system menu settings 186
terminating on Close 186
unique names for entry field and
static text parts 30
using sound 243
Visible attribute 181
when you can set attributes 182
Windows help, creating 249
WRITE (create new records) operation
code
database files 198
for subfile parts 159
purpose of 29
reflecting stored values on the
screen 29

**T**

TabLabel attribute 39
Terminate on close 186
terminating a program 186
text
for combination box parts 62
for entry field parts 76
for spin button parts 151
purpose of 30
Text attribute 112
TextEnd attribute 113
TextSelect attribute 113
TextStart attribute 113
thin clients 431
Tick event 172
TickLabel attribute
for slider parts 145
timer part
events 172
purpose of 172
TimerMode attribute 173
translating
compiling messages for 262
editing messages for 263
messages 22
mnemonics 22
tips for 22, 39

**U**

update subfile record
for subfile parts 159
User Interface Manager, reusing
files 222
UserData attribute, common uses of 39
utilities
Define iSeries Information 194, 415
Make Message File 262
Packaging 415

**W**

Width attribute
common uses of 37
for outline box parts 137
window part
attributes 180
events 180
purpose of 180
window with canvas part
attributes 181
events 181
purpose of 181

Index 477