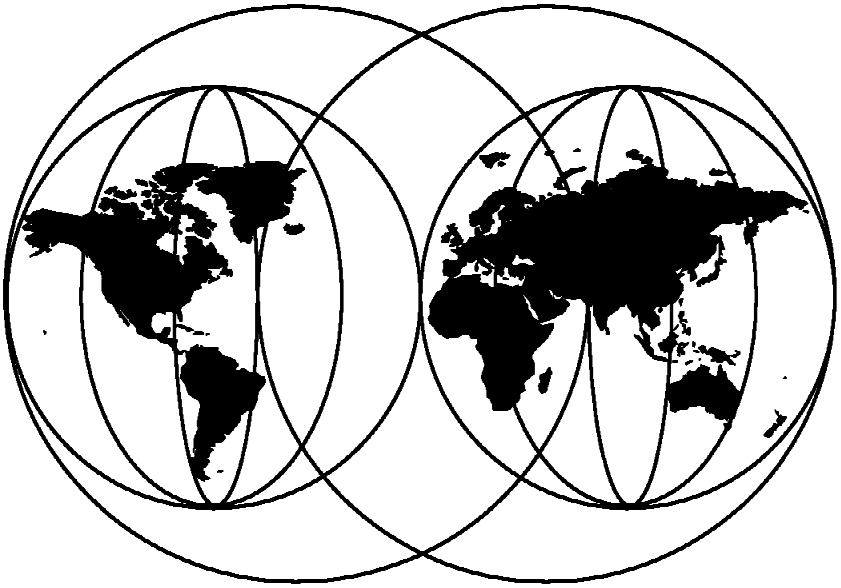
AS/400 Programming with VisualAge for RPG

Andi Bitterer
Reinhard Leising



International Technical Support Organization

http://www.redbooks.ibm.com

SG24-2222-00

SG24-2222-00

International Technical Support Organization

AS/400 Programming with VisualAge for RPG

May 1998

Take Note!

Before using this information and the product it supports, be sure to read the general information in Appendix D, “Special Notices.”

First Edition (May 1998)

This edition applies to Version 3 Release 1 of VisualAge for RPG for Windows 95/NT, for use with OS/400 V3R1 or later.

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. QXXE Building 80-E2
650 Harry Road
San Jose, California 95120-6099

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

Contents

Figures

Tables

Preface

This redbook demonstrates how to use VisualAge for RPG in a Windows 95 or Windows NT client/server environment. The reader learns how to visually develop a graphical user interface for workstation programs that access the AS/400. This book gives a broad understanding of various issues related to client/server programming with VisualAge for RPG, such as data access on the AS/400, calling programs and procedures, using messages and help, managing projects, or national language support.

The book contains two parts. Part 1 covers VisualAge for RPG topics in detail and the reader is presented with solutions to everyday problems, such as exception handling or sharing of components. Many RPG source examples are developed along the way to aid the reader.

Part 2 consists of education material that can be used in class. This part takes a step-by-step approach to explain the basic features of VisualAge for RPG, for example, actions, components, menus, notebooks, or containers. By following the samples, the reader develops a small application. The book demonstrates how to use and create VisualAge parts, use events to trigger RPG logic, define subfiles and more.

AS/400 Programming with VisualAge for RPG was written for AS/400 programmers that are using ILE RPG and now want to move into a client/server environment. Some knowledge of AS/400 operations, DB2/400 and RPG is assumed.

ITSO on the Internet

Internet users may find additional material about new redbooks on the ITSO World Wide Web home page. Point your Web browser to the following URL:

http://www.redbooks.ibm.com

IBM internal users may also download redbooks or scan through redbook abstracts. Point your Web browser to the internal IBM Redbooks home page:

http://w3.itso.ibm.com

If you do not have World Wide Web access, you can obtain the list of all current redbooks through the Internet by anonymous FTP to:

ftp.almaden.ibm.com

```
cd /redbooks
get itsopub.txt
```

The FTP server, *ftp.almaden.ibm.com*, also stores the samples described in the book. To retrieve the sample files, issue the following commands from the */redbooks* directory:

```
cd SG242222
binary
get VARPG.ZIP
get UPSAVF.ZIP
get CONTEXMP.ZIP
ascii
get VARPG.TXT
```

All users of ITSO publications are encouraged to provide feedback to improve quality over time. Send questions about and feedback on redbooks to:

```
redbook@vnet.ibm.com        or
REDBOOK at WTSCPOK          or
USIB5FWN at IBMMAIL
```

If you find an error in the software that is described in this book, please send a bug report with a description of the problem to

```
bugs@vnet.ibm.com
```

To receive regular updates on new redbooks and general IBM announcements, you can subscribe to the IBM Announcement Listserver. It automatically supplies an Internet e-mail user with timely new announcement information (titles and optionally the letter or abstract) from selected categories. To get started, send an e-mail to

```
announce@webster.ibmlink.ibm.com
```

The keyword SUBSCRIBE must be the only word in the body of the e-mail; leave the subject line blank. You will receive a category form and listserver details. To immediately start your subscription to, for example, AS/400 announcements, put the words SELECT HW120 in the body of the note. On the afternoon of an announcement day, you will receive e-mail with the announcement, along with a list of newly available redbook titles. To obtain a full abstract of a particular redbook, use GET SG242222 in the note.

VisualAge for RPG on the Internet

VisualAge for RPG users are encouraged to visit the VisualAge for RPG Web site regularly for latest information, solutions, fixes, PTFs, tips and techniques or to ask questions about VisualAge for RPG and its features. From your favorite Web browser, enter

http://www.software.ibm.com/ad/varpg

You will find sections on the following and other VisualAge for RPG and CODE/400 topics:

- News
- Support
- Downloads
- Solutions
- Education

About the Authors

Andi Bitterer works as a consultant for the International Technical Support Organization at the Almaden Research Center in San Jose, California. He graduated with a degree in computer science from the Technical University in Darmstadt, Germany. Andi joined IBM in 1987 and worked as an application development specialist in large customer projects for IBM Integration Systems Services and at the German AS/400 Field Support Center. Since joining the ITSO in 1994, he has taught workshops worldwide on object-oriented application development, Digital Library, e-business, and the Internet. Andi is the author of three VisualAge for Smalltalk books published by Prentice Hall. You can reach him by e-mail at bit@acm.org.

Reinhard Leising is working as a software specialist for the AS/400 Software Support in Muenster, Germany. He graduated with a degree in computer science from the Technical College in Dortmund, Germany. Since joining IBM and the AS/400 Software Support in 1990, he focussed on code defects and customer problems in the programming languages area as well as application development tools like CODE/400 and VisualAge for RPG. On his first ITSO residency in 1993, he worked on the first redbook about ILE and the ILE C/400 compiler. This was followed by two residencies in 1994 and 1995, that were held to produce presentation and education material about Visual RPG Client/2 Release 1 and 2. You can reach Reinhard by e-mail at rleising@vnet.ibm.com.

Acknowledgments

Many thanks go to the following people for their invaluable contributions to this project: Claus Weiss, Larry Schweyer, Jenny Wong, May Tang, and Joe Sobura, IBM Toronto Laboratory, for their help during the beta phase of the product and for providing us with the latest drivers and more, Michele Chilanti, IBM Rochester ITSO, for his help with AS/400 database issues, Bob Slaney, IBM Education & Training, Atlanta, and Erminia Nicoletti, IBM Italy, for developing the original education material.

Thanks also go to the staff of the ITSO San Jose Center, in particular Elsa Barron, Mary Comianos, Alan Tippett, and Dave Wray, for the great administrative support, to our managers, Laymond Pon and Hans-Werner Henschel, for their support to run this project, to Shirley Hentzell, for an excellent editing job, and to Stephanie Manning and Liz Rice, for the editorial assistance.

Comments Welcome

Your comments are important to us!

We want our redbooks to be as helpful as possible. Please send us your comments about this or other redbooks in one of the following ways:

- Fax the evaluation form found in “ITSO Redbook Evaluation” to the fax number shown on the form.
- Use the electronic evaluation form found on the Redbooks Web sites:

  For Internet users `http://www.redbooks.ibm.com/`
  For IBM Intranet users `http://w3.itso.ibm.com/`

- Send us a note at the following address:

  `redbook@us.ibm.com`

Part 1. Programming with VisualAge for RPG

Chapter 1. Introduction to VisualAge for RPG

This chapter describes VisualAge for RPG, introduces the key components, and defines key terms.

What Is VisualAge for RPG

VisualAge for RPG is a feature of the IBM Application Development ToolSet Client/Server for the AS/400 product. You can use it to create applications that have graphical user interfaces (GUIs) that are processed by an enhanced version of the RPG language. Applications created on your Windows NT/95 workstation run on Windows NT, Windows 95, or Windows 3.1. Applications created on your OS/2 workstation run on OS/2 or Windows 3.1. The applications you create can access data stored on an AS/400 server.

What Is Client/Server

In its simplest form, the phrase client/server refers to a relationship between two or more computers. The relationship involves requesting and sending data. The client can request data from the server.

The programmable workstation (PWS) with its graphical user interface is the client. Using the PWS, the user extracts, presents, and manipulates information.

The natural strengths of AS/400 make it ideal for the role of the server. It can store huge amounts of data, it has object-oriented design, and it can incorporate new technologies without adversely affecting existing applications.

Building Client/Server GUI Applications

You start by designing your application and then build it from the top down. You first focus on the look and feel of the interface, and then you tie all the parts together with workstation RPG logic.

VisualAge for RPG automatically generates the code to display the GUI you create. The only code you have to write is the code that drives the GUI. That is, you have to write subroutines to respond to GUI events. For example, you have to write a subroutine to handle the selection of a menu item by the user. Your program can handle any functions that could be done in a regular RPG subroutine.

All the application development—such as creating the GUI, editing, compiling, and debugging program logic, and packaging for distribution to end users—can be performed within the GUI Designer. Writing and testing your code is simplified by the language-parsing editor and the debugger that are integrated with the IBM Application Development ToolSet Client Server for the AS/400 development environment.

Once you have created the GUI and written the RPG code, you build the application using the VisualAge for RPG compiler.

Even your existing OS/400 applications can be converted to run on a workstation. An import function lets you reuse existing display files and convert them to GUI parts. With some customizing, you can create a new look for an application that used to display only on a 5250.

Now you can build real GUI applications without having to invest time and money retraining in a whole new language. You can use the strengths of both AS/400 and the workstation in a client/server environment and build on your existing RPG skills.

The Application Development Environment

The VisualAge for RPG application development environment is based on a GUI project organizer. The Project Organizer simplifies the process of organizing and building software projects and provides an application development environment that allows you to organize your files and customize the way you work with them.

In this interface, your application is organized into projects. A project represents the complete set of objects (source code files, GUI objects, and so on) that you need to build a single target program, such as a dynamic link library (DLL) or executable file (EXE).

Associated with each project is a set of actions. In this environment, an action is described as the use of a tool or a function of a tool to manipulate a project. Edit, compile, and debug are examples of actions.

You can organize complex applications into project hierarchies to give you a visual perspective of how your code is organized. You can perform a build from any point in a project hierarchy, giving you added control over the way you build your applications.

Tools such as an editor and a debugger are integrated into this environment so that you have quick and easy access to them as you program.

Project Organizer Help

You can get help on how to use any menu choice, window, or control. You can access this context-sensitive help by pressing F1 when a field, menu item, or control is *in focus*. You can also press the Help push button, where available.

The GUI Designer

Use the GUI Designer to create your VisualAge for RPG applications. It consists of the project window, the parts palette, and the parts catalog (Figure 1).

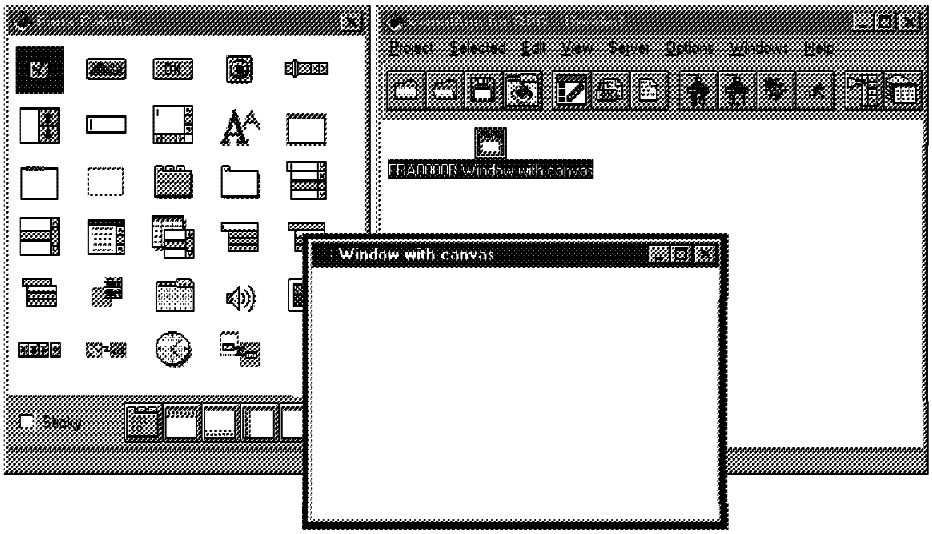


Figure 1. GUI Designer Project Window and Parts Palette

The Project Window

The project window is your easel, the place where you actually create your GUI. The project window consists of the following:

Project view

A working area in the center of the window that lists the contents of a. project.

Menu bar

Lists the actions you can invoke using menu items.

Tool bar

Contains graphic push buttons you can press to invoke frequently performed actions.

Status bar

Contains descriptive text when you place the cursor over certain areas of the GUI Designer.

The Project View

The project view displays a representation of the current working project in VisualAge for RPG. The representation is displayed as either a tree view or an icon view.

The tree view (Figure 2) is a hierarchical structure showing an icon, name, and (where applicable) the label for each window and part in the project.

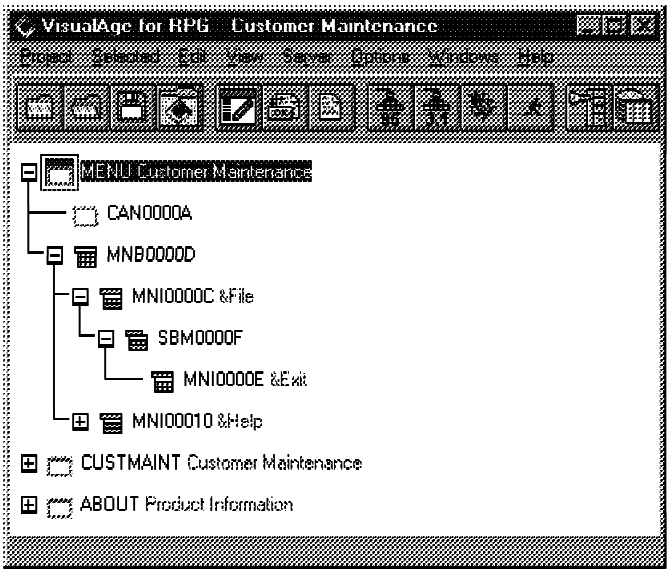


Figure 2. Tree View

The icon view (Figure 3) shows a window part icon for each of the windows in the project. Parts contained within windows are not displayed with this view.

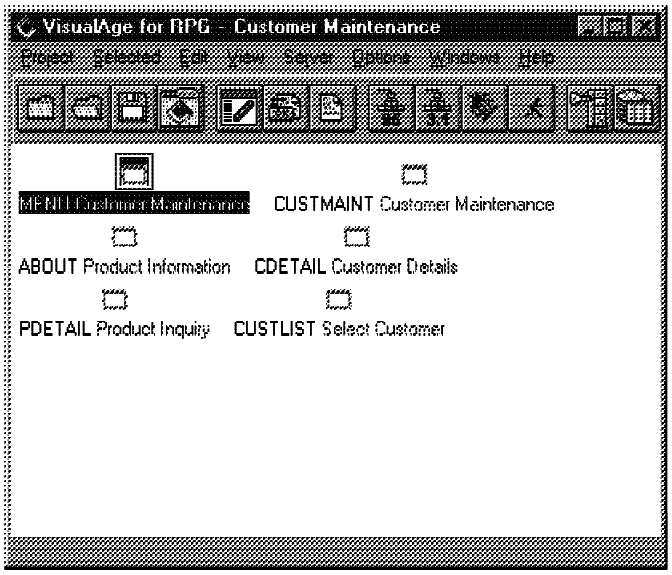


Figure 3. Icon View

The Menu Bar

The VisualAge for RPG product makes extensive use of menus to assist you in designing an application. There are two types of menus:

- Pull-down menus (from the menu bar)
- Pop-up menus (from objects within the project view)

On the project window, the menu bar appears below the title bar. Actions on the pull-down menus from the menu bar let you perform a variety of tasks when you design an application, such as creating and running your application, customizing the GUI Designer, and using help.

Pop-up menus provide you with a list of choices specific to the object that the mouse pointer is over, such as a part, a group of parts, or a window. These menus are accessed directly from an object by pressing mouse button 2 when the mouse pointer is over the object in a design window or the project view. The same choices can also be accessed from the selected pull-down menu when an object or group of objects is selected.

The choices on menus are called *menu items*. Some menu items contain a right arrow (➔) at the right side. This marking indicates there is additional information you need to be aware of in a secondary list of choices. This secondary list is called a *submenu.* Some menu items contain an ellipsis (...)

at the right side. This marking indicates that additional information will appear in a window when this item is selected.

The Tool Bar

The tool bar is a menu of graphic push buttons. These push buttons represent certain pull-down menu items, and are intended to provide quick access to common actions you perform, such as saving a project. Use the tool bar as an alternative to using some menu items. When you place the cursor over one of these push buttons, hover-help appears next to the button, giving a short description of the action. A description of the action is also displayed in the status bar.

The initial location of the tool bar is at the top of the project window, but you can change its position, as well as its contents.

The Status Bar

The status bar (also referred to as the *information area*) provides you with information for tool bar action icons and menu items. Position the mouse over any of these items and the status bar displays a short description of it.

Some other windows within the GUI Designer also have a status bar that displays a short description of an object the mouse is positioned over.

By default, the status bar is located at the bottom of the project window. You can change its position or you can hide the status bar if it is not required.

The Parts Palette

The Parts Palette (Figure 4) contains a customizable set of parts that you use to create the screen layout for your VisualAge for RPG application. Use the palette as a painter uses a paint palette. It's the place where you keep the parts that are most appropriate for the design you are currently working on. As you finish a design, you can wipe the palette clean and start over with other parts from the parts catalog. Use the palette if you wish to access your own personalized set of parts when designing a GUI.

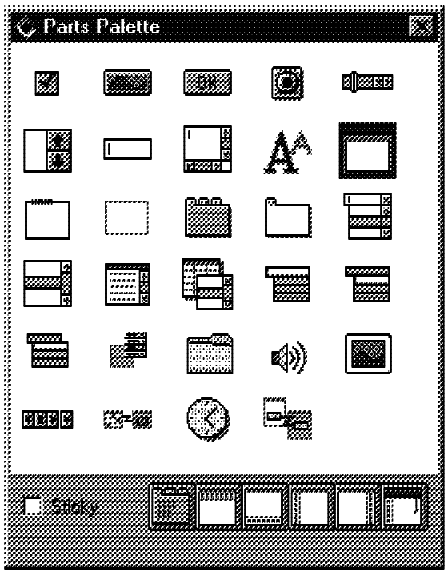


Figure 4. Parts Palette

By default, the Parts Palette is not part of the project window but appears in a window of its own. You can configure the Parts Palette to be part of the project window, if you like.

The Parts Catalog

The Parts Catalog (Figure 5) contains all of the parts that you use to create the screen layout for your VisualAge for RPG application. You can select parts from the catalog and copy them to the Parts Palette if you intend to use them frequently. Use the catalog as an artist uses a box of oil paints. It's the place where you keep all of the various parts you accumulate over time. Use the catalog if you wish to access all of the parts when designing a GUI.

Note!

All the parts supplied by IBM are contained here. You cannot delete these IBM-supplied parts from the catalog.

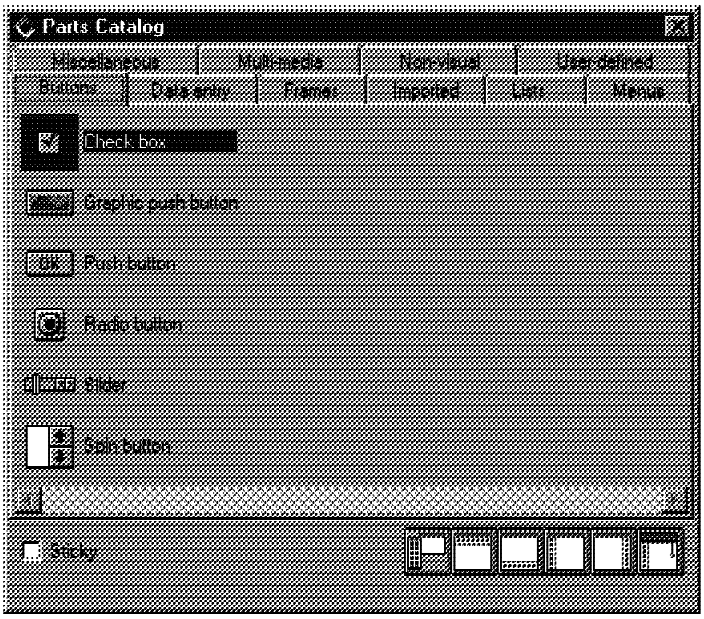


Figure 5. Parts Catalog

You can create and define your own customized parts, called *user-defined parts*. You can delete customized parts from the catalog if you no longer want them.

The Editor

The editor that is integrated with the IBM Application Development ToolSet Client Server for AS/400 is called Live Parsing Editor (LPEX). You can use this editor to edit source files (for example, program source files and help source files) and to customize the VisualAge for RPG source templates. You can also use this editor when you define output specifications, define user subroutines, and enter array data.

This programmable editor gives you powerful editing capabilities such as:

Prompting

Makes writing column-oriented source code easier. It is provided for source code and comments, single line, or continuous inserts (insert prompting).

Token highlighting

Enhances the readability of the code. You can configure highlighting to show different language constructs with different

colors or fonts to identify the program structures. You can turn token highlighting on or off.

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.

Error logging
Keeps track of errors in an error log. The editor takes you to the place in the source where the error occurred.

Language-sensitive help
Provides help for RPG-related topics.

With this editor you can also:

- Use commands in the prefix or sequence number area.
- Edit more than one file at a time.
- Open multiple views of one file.
- Cut and paste between files or between different views of a single file.
- Undo changes you made to a file.
- Save files on a workstation or a host until you are ready to compile them.

The Compiler

The compiler for VisualAge for RPG is based on the RPG IV language. The VisualAge for RPG compiler contains the enhancements you need to create RPG client/server programs with graphical user interfaces. Operations and built-in functions have been added to reap the benefits of the GUI environment on the workstation. You create your application (consisting of source code and the GUI) and compile it on a Windows NT/95 or OS/2 workstation.

The VisualAge for RPG compiler enables you to access objects on the AS/400 server (for example, DB2/400 database files) from your workstation. You can use RPG file operations to access data on an AS/400 server. VisualAge for RPG insulates the developers from many of the details of the data access.

The Debugger

The debugger helps you detect and diagnose errors in your application. You can use different views to display the image of the program you are debugging. You can run your application, set breakpoints, and monitor variables, registers, the call stack, and storage.

Graphical debugging features include:

Source-level debug
: Allows you to follow the execution of a program while you view the source. You can quickly and effectively identify errors in the code.

Step mode debug
: Allows you to bypass previously debugged code and focus on a problem area to save time.

Fix-and-resume
: Allows you to continue running your program after you fix a run-time error.

Chapter 2. Debugging

The AS/400 Cooperative Debugger[1] is a client/server program that helps you detect and diagnose errors in programs written in the following languages: ILE RPG, ILE COBOL, ILE CL ILE C, ILE C + + , OPM RPG, OPM COBOL, and OPM CL. You can use the debugger to run your program, set breakpoints, step through program instructions, display the call stack, and watch, display or change variables.

This chapter lists the considerations you need to be aware of, and preparatory items you should complete, before you run the debugger on your program.

Select from these topics:

- Preparing for debugging
- Compiling a program with debug data
- Starting a debugging session
- Ending the debugging session
- Locating source code
- Frequently used features of the debugger

Preparing for Debugging

Before you can run the debugger on an AS/400 application, you must:

- Compile a program with debug data.
- Set up a debugger port (optional).
- Start the debug server if it is not already started.
- Specify an AS/400 host name.
- Set environment variables for the debugger (optional)

To perform most of these tasks, your user profile must have the appropriate authorities.

[1] This chapter is derived from the online document *Using the AS/400 Cooperative Debugger*, which comes with the ADTS Client/Server for AS/400 product. For the most part the information found there also applies to VisualAge for RPG.

Authorities Required for Using the Debugger

The user profile that you use to sign on to an AS/400 system from the debugger must have the following authorities:

- *USE authority for the Start Debug command (STRDBG)
- *USE authority for the End Debug command (ENDDBG)
- *USE authority for the Start Service Job command (STRSRVJOB)
- *USE authority for the End Service Job command (ENDSRVJOB)
- Either *CHANGE authority for the program being debugged, or *USE authority for the program being debugged and *SERVICE special authority.

If the job that you are debugging is running under a user profile different from the user profile you use to sign on to AS/400 from the debugger, your user profile must have the following authorities:

- *USE authority for the user profile that the job you are debugging is running under.
- *JOBCTL special authority, if you do not explicitly use fully qualified program names (library/program). In other words, if you use *CURLIB or *LIBL or you do not specify a library name, you need to have *JOBCTL special authority.

The group profile QPGMR will give a user the correct authority for the STRDBG, ENDDBG, STRSRVJOB, and ENDSRVJOB commands, and for the *JOBCTL special authority.

Setting Debugger Ports

The debugger client and server communicate with each other using TCP sockets, which communicate through ports. When shipped, the debug server on the host is set up to listen for connection requests on TCP port 3001. If port 3001 is being used by another application, you can change it by using the Work with Service Table Entry command on the AS/400.

The service table is used to manage the mapping of network services to ports and to record the protocols the services use.

To change the port used by the debug server, modify the service called QDBGSVR by using the WRKSRVTBLE SERVICE(QDBGSVR) command. Add a QDBGSVR service entry for the new port and remove the QDBGSVR service entry for port 3001. Then set the ICCASDBGPORT environment variable on all debugger clients to match the port specified. For information

on environment variables, see “Setting Environment Variables for Debugging.”

Note: Before changing the port, end the debug server by using the End Debug Server command (ENDDBGSVR). Then change the port and start the debug server again using the Start Debug Server command (STRDBGSVR).

You must have special system configuration authority (*IOSYSCFG) to add service entries to and remove them from the service table.

Starting the Debug Server

Before you can use the debugger, the debug server must be started on the AS/400 with the Start Debug Server command (STRDBGSVR).

The Start Debug Server command starts the debug server router function. Only one debug server router can be active at a time. Once started, the debug server router remains active until it is ended using the End Debug Server command (ENDDBGSVR).

Ending the Debug Server

Normally, you do not need to end the debug server at the end of your debug session, because the debug server is active for the entire AS/400 system. If necessary, the debug server router function can be ended using the End Debug Server command (ENDDBGSVR). This prevents any new debug sessions from starting, but active sessions continue until completion or until they are cancelled.

Subsequent connection requests fail until the debug server router function is started again using the STRDBGSVR command.

Specifying an AS/400 Host Name

Because the debugger is a client/server program, a method of communication is required to allow the client and server to communicate. The debugger uses the sockets programming services for communication. A Windows debugger client must know the name of the host where the debugger server and your application will run.

You can specify the name of the debug host through the ICCASDEBUGHOST environment variable, or through the */e hostname* command-line parameter, or through the AS/400 logon window of the debugger. If you do not specify the */e hostname* command-line parameter, the host name specified in the ICCASDEBUGHOST environment variable is used. If you do not specify the /e parameter and the ICCASDEBUGHOST environment variable is not set, you must specify the host name in the AS/400 logon window. See “Setting Environment Variables for Debugging” for information on the

ICCASDEBUGHOST environment variable. See “Starting a Debugging Session” for information on the */e hostname* command-line parameter and the AS/400 logon window.

Setting Environment Variables for Debugging

You can use several environment variables with the debugger. The following environment variables may be set for use with the debugger:

- ICCASDEBUGHOST
- ICCASTAB
- ICCASUPRD
- ICCASDEBUGPATH
- ICCASDBGPORT

ICCASDEBUGHOST

Specifies the name of the AS/400 host you want to connect to. ICCASDEBUGHOST is an optional environment variable.

To set the name of the host through ICCASDEBUHOST, type the following at the Windows command prompt:

```
SET ICCASDEBUGHOST=debughostname
```

where debughostame is one of the following:

- An IP address, such as 9.1.150.100

  To determine the IP address of your AS/400, type `GO CFGTCP` on an AS/400 command line, and choose menu option 1.

- A host name defined in your local host table or on a name server to which you link, such as name.toronto.ibm.com.

  To determine the host name of your AS/400, type `GO CFGTCP` on an AS/400 command line, and choose menu option 12. (Be careful not to change the local domain name and host name when you use option 12.)

Note: ICCASDEBUGHOST is an optional environment variable. ICCASDEBUGHOST is not used if you specify the host name through the */e hostname* command-line parameter. If you do not specify the /e parameter, you must specify the host name in the AS/400 logon window.

ICCASTAB

Specifies the number of spaces per tab. ICCASTAB is an optional environment variable.

To set tab stops at particular intervals, type the following at the Windows command prompt:

```
SET ICCASTAB=number
```

where the number is from 1 through 64. For example, SET ICCASTAB=5 will cause text after a tab to begin in columns 6,11,16, 21, 26, and so on. If ICCASTAB is not set, the default is 8.

ICCASUPRD
Specifies whether to allow the update of production files. ICCASUPRD is an optional environment variable. To allow the update of production files, type the following at the Windows command prompt:

```
SET ICCASUPRD=Y
```

where Y allows production files to be updated, and N does not allow production files to be updated. By default, ICCASUPRD is set to N. If ICCASUPRD is not set, the debugger will use the AS/400 default set for the UPDPROD parameter of the STRDBG command.

ICCASDEBUGPATH
Specifies the search paths for source files residing on your workstation. ICCASDEBUGPATH is an optional environment variable.

To set the search path for the source code file used by the debugger, type the following at the Windows command prompt:

```
SET ICCASDEBUGPATH=pathl;path2;
```

where path* is the path, including the drive and directory names, to the location where the source code files are stored. Multiple paths must be separated with semicolons.

ICCASDBGPORT
Specifies the port used to connect to the AS/400. ICCASDBGPORT is an optional environment variable. To connect to the AS/400 using a port other than the default port (3001), type the following at the Windows command prompt:

```
SET ICCASDBGPORT=port number
```

where the port number is a value between 1 and 64,767 that matches the port number specified for the QDBGSVR entry on the AS/400.

For information on debugger ports and the QDBGSVR entry, see “Setting Debugger Ports.”

Compiling a Program with Debug Data

If you want to debug your application, you must first compile your modules or programs to include debug data. The debugger lets you debug applications that are written in one or more of the following AS/400

languages: ILE C + + , ILE C, ILE COBOL, ILE RPG, ILE CL, OPM COBOL, OPM RPG, and OPM CL.

Compiling ILE C + + Programs

To create debug data in your program, do the following:

1. Compile your code with one of the following options to request one or more debug views:

   /Ti+ Compiles your program to produce a module that includes a source view, a listing view, and a statement view.

   /Til Compiles your program to produce a module that includes a listing view and a statement view.

   /Tis Compiles your program to produce a module that includes a source view and a statement view.

   /Tin Compiles your program to produce a module that includes only a statement view.

2. Bind the program.

Compiling ILE C, ILE RPG, ILE COBOL, or ILE CL Programs

If you want to debug programs written in ILE C, ILE RPG, ILE COBOL, ILE CL, you need to:

1. Compile your code with one of the following debug view options:

   ***ALL** Compiles your program or module to include a source view, a listing view, and a statement view.

   ***LISTING** Compiles your program or module to include a listing view and a statement view.

   ***SOURCE** Compiles your program or module to include a source view and a statement view.

   ***STATEMENT** Compiles your program or module to include only a statement view.

2. Bind the program if it is made up of modules compiled with the CRTxxxMOD command.

Note: In the ILE program model, a module is the object that results from compiling source code with the CRTxxxMOD command. A module cannot be run. To be run, the module (or modules) must be bound into a program using the CRTPGM command or into a service program using the CRTSRVPGM command.

Compiling OPM RPG, OPM COBOL, or OPM CL Programs

If you want to debug programs written in OPM RPG, OPM COBOL, or OPM CL, you need to compile your code with one of the following debug options:

***SRCDBG** Compiles your program to include a source view and a statement view.

***LSTDBG** Compiles your program to include a listing view and a statement view.

> **Note!**
>
> The *LSTDBG option is not supported for CL.

Debugging Optimized Code

Generally, the higher the optimization level, the more efficiently the procedures in the module run. However, higher optimization levels adversely affect breakpoints that are set with the debugger.

While debugging your code, set the optimization level to the minimum level (*NONE). This allows you to accurately display and change variables. After you have completed your debugging session, set the optimization level to the maximum level. This provides the highest levels of performance for the procedures in the module.

> **Watch this!**
>
> Even at optimization level *NONE, some optimization may be done in certain cases that could affect the debugger's ability to accurately display the program's stopped location.

Starting a Debugging Session

To start the debugger from the Windows command prompt, perform the following steps:

1. Ensure that the debug server is running on the AS/400.
2. Start the debugger on your Windows workstation by entering the debugger invocation command IDEBUGAS. To use parameters with this command, type

   ```
   idebugas /x program-name
   ```

   where /x represents any number of debugger parameters. Choose from the following parameters:

/a Run the program until it completes or hits a breakpoint.

/e *hostname* Specify the name of the host system that you want to connect to for your debugging session. If you do not specify the /e command-line parameter, the host name specified in the ICCASDEBUGHOST environment variable is used. If you do not specify the /e parameter and the ICCASDEBUGHOST environment variable is not set, you must specify the host name in the AS/400 logon window.

/i Step into the program. This parameter has the same function as the *Step into program* check box on the Startup Information window.

/j*job* Specify the job to debug, in the following format: job-name/user-name/job-number. No spaces should intervene between the /j parameter and job

If you specify the /j parameter and the program name, you will bypass the Startup Information window.

/p+ Use program profile information, if program profile information is available. This is the default.

/p- Do not use any program profile information.

The AS/400 logon window opens if one of the following is true:

- You have not specified the /e parameter and the ICCASDEBUGHOST environment variable is not set.
- You are connecting to this particular AS/400 host for the first time after rebooting your workstation.

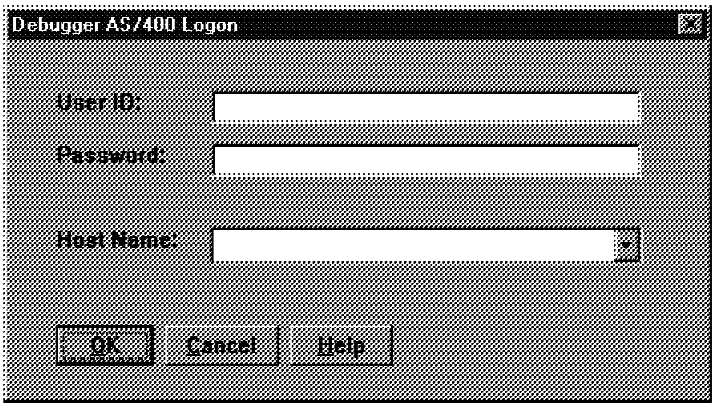


Figure 6. AS/400 Logon Window

3. Enter your user ID, password, and the name of your AS/400 host in the AS/400 logon window (Figure 6) and click on OK. Next, the Debug Session Control window and the Startup Information window (Figure 7) open.

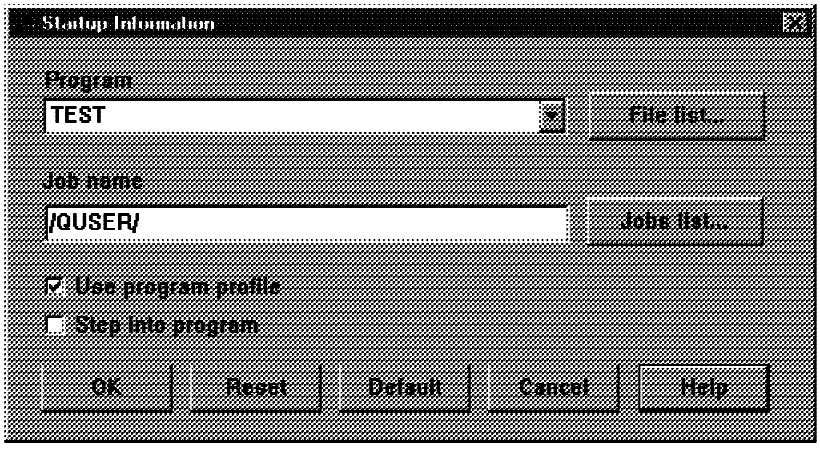


Figure 7. Startup Information Window

4. In the Startup Information window, enter the following:
   - In the *Program* entry field, either type the name of the program you want to debug, or click on the down arrow on the *Program* entry field and select a program.
   - In the *Job name* entry field, type the name of the AS/400 job you will use when the program to be debugged is run. The job is specified in the following format: job-name/user-name/job-number.

     Often, this is the job associated with a 5250 terminal session, but it can be any AS/400 job, a batch job, a prestarted job, or an invoked job. It you select *Jobs list,* the Jobs List window is shown. From this window, select the job you want and select OK.

     The Startup Information window is updated and the job name you selected now displays in the *Job name* entry field.

     Note: A subset of the jobs list may be requested by specifying part of the fully qualified job name:

     – ABC*// will list all jobs with job names starting with ABC.

     – /QUSER/ will list all jobs associated with the user name QUSEFi.
   - Enable the *Use program profile* check box to restore the debugger windows and breakpoints when debugging a program more than

once. The program profile is stored separately for each program debugged.

- Enable the *Step into program* check box to stop your program at the next runable statement encountered. Use this option to debug C++ constructors for objects that are located in static storage or to debug a program that is already running. You should also enable this check box for ILE and OPM languages other than C and C++.
- Select *OK* to accept the information you have entered. The Startup Information window closes, and the debugger is started. A debugger message appears that asks you to start the program to be debugged on the AS/400.

5. Start the program on the AS/400.
6. Press the *OK* button in the debugger message window to continue. The source of your program is displayed.

Ending the Debugging Session

End the debugging session in one of the following ways:

- Press F3 in any of the debugger windows
- Select *Close debugger* from the *File* menu-bar choice in a debugger window.

You may want to end the debugger session in one of the following situations:

- In the Startup Information window, you specify the name of a job that is an active AS/400 job, but it is not the job that you wanted to use.
- You specify the name of a program on the Startup Information window, but the program fails when you call it (for example, a signature violation occurs when you call the program).
- The job ends on the AS/400 before the startup program is called.

Locating Source Code

How the debugger finds source code depends on whether this source code is located on the Windows workstation, or on the AS/400.

C++ Source Code

When you compile a C++ program and specify the source debug view by using the /Ti+ option, the compiler stores the name of the source file and its directory path in the module object. When you try to display the source file, the debugger tries to find the file using the directory path and file name

that was captured when the program was compiled. The debugger searches for the file in this order:

1. In the directory path that was stored when the module was compiled.
2. In the directory where the last file, if any, was found.
3. In the directories defined in the ICCASDEBUGPATH environment variable.

If the file is not found in any of these directories, you will be prompted for the name of the file. If the source file is not available, press the *Cancel* push button on the file prompt and a different view will be used.

Source Code Written in Other ILE or OPM Languages

When you compile other ILE (C, RPG, COBOL, and CL) modules, or OPM (CL, COBOL, RPG) programs, the name of the AS/400 source member is stored in the module or program object. When you try to display the source for a module or program, the debugger attempts to read the information from the AS/400 source member used to create the module or program.

If the source member is not found, the debugger searches your workstation for a source file using the following steps:

1. The debugger builds a name using the module name or program name and a file extension based on the language. The file extensions used are as follows:

   RPG For RPG language source

   C For C language source

   CBL For COBOL language source

   CL For CL language source.

2. The debugger searches for the file in this order:

   a. In the directory where the last file, if any, was found

   b. In the directories listed in the ICCASDEBUGPATH environment variable.

   If the file is not found, you are prompted to enter the name of the file.

   If you do not have the source available, press the *Cancel* push button on the file prompt and a different view will be used.

Frequently Used Features of the Debugger

This section introduces

- The tool buttons
- Ways to run your program
- How to set breakpoints
- How to step through your program
- How to display and change variables
- The use of multiple statements in program code
- Debugger performance considerations
- Debug limits

Using the Tool Buttons

Tool buttons provide easy access to frequently used features. Check the tool bar on each window to determine which buttons are active for the window. Table 1 shows the most commonly used tool buttons.

Table 1 (Page 1 of 2). Tool Buttons

Button	Description
	*Step over* runs the current, highlighted line in the called procedure, but does not stop in it. If the current line is a call, the program is halted when the call is completed.
	*Step debug* runs the current, highlighted line in the program. The debugger steps over any procedure for which debugging information is not available (for example, library and system routines), and steps into any procedure for which debugging information is available.
	*Run* runs the program. Control returns to the debugger when: • The program stops at an enabled breakpoint or watch. • The program encounters an exception • The program ends. When the debugger is running, the run button is grayed out.
	*View* changes the current program view window to one of the other program view windows, depending on the views available for the module or program. For example, you can change from the *LISTING view to the *STATEMENT view.
	*Call stack* displays the Call Stack window, which allows you to view all of the call stack entries. The procedures are displayed in the order that they were called.

Table 1 (Page 2 of 2). Tool Buttons	
Button	**Description**
	*Monitor*, from a program view window, displays the Monitor Expression window. This button also appears on the left side of the tool bar on the Program Monitor window and the Local Variables window.
	*Breakpoints* displays the Breakpoints List window, which allows you to view all the breakpoints that have been set.
	*Debug Session Control* transfers control to the Debug Session Control window.
	*Change Representation* displays the next representation of the selected variable.
	*Delete* deletes the highlighted expression, variable, or breakpoint.
	*Delete All* deletes all expressions, variables, or breakpoints.
	*Toggle Direction* toggles the order of the entries on the call stack.

Running a Program

You can run your program by using Step commands, the Run command, or the Run to location command:

- Step commands control how the program runs. The execution of a line of code is reflected in all open views.

  The step commands are located in the tool bar and under the *Run* menu-bar choice of the program view windows.

- The Run command runs the program until a breakpoint or watch is encountered, the program is halted, or the program ends.

  The Run command is located in the tool bars and under the *Run* menu-bar choice of the program view windows and the Debug Session Control window.

- The Run to location command runs the program to the current position in the current program view window. This command performs the following steps:

- Add a breakpoint at the target line
- Run the program
- Remove the breakpoint.

To select a position, press mouse button 1 once on the prefix area of a runable statement. The prefix area is the area on the left of the program view window where the line numbers are displayed.

The Run to location command is located under the *Run* menu-bar choice of the program view windows. It is also available in the pop-up menu for the line number, if mouse button 2 is configured for pop-up.

Setting Breakpoints

A breakpoint is a place in a program where the system stops the processing of the program and gives control to the debugger:

- Line Breakpoints

  A line breakpoint stops your program at a specific location. Set line breakpoints in one of the following ways:

  - Select the *Breakpoints* menu from the Debug Session Control window or from any of the program views windows.
  - Double-click in the prefix area of a runable statement in any of the program view windows. The prefix area is the area on the left of the program view window where line numbers are displayed. The prefix area turns red, indicating that the breakpoint has been set.

- Watch Breakpoints

  A watch allows you to request a breakpoint when a specified variable or expression is changed from its current value. The variable or expression is used to determine the address of the storage location to watch and must resolve to a location that can be assigned to. The scope of the variable or expression in a watch is defined by the execution stop point when the watch is set.

  After the watch is successfully set, a change to the contents of the watched storage location by a program in your session will cause your application to be stopped. If the program has debug data and a debug view is available, the statement after the change is highlighted. A message will indicate which watch condition was satisfied.

  If the program that caused the change has not been added to the debugger, it is automatically added if:

  - The program contains debug data.
  - You have authorization for the program.

If the program cannot be debugged, or you do not have sufficient authority to debug the program, you receive a message specifying the program and location where the watch condition was encountered. Program execution resumes when you click on the message's OK button.

Set watches in one of the following ways:

- Highlight a variable in the source window. Click on *Breakpoints* and select the *Set watch* action for the selected variable. The Watch window opens. It contains the name of the highlighted variable.
- Select the *Set watch* action on the *Breakpoints* pull-down menu which is available from several debugger windows.

  If the watch dialog is invoked from a window other than the source window (in which case default values are provided by the debugger), you must fill in the variable or expression to be watched. Otherwise, the watch will fail. If you do not specify values for the other entry fields in this dialog, default values are used. These defaults are:

  Bytes to Watch The number of bytes defined by the variable length is the number watched, up to 128 bytes. The default value is 0 which means that the declared type length of the variable is watched.

  Thread Every

  Frequency

  From 1

  To Infinity

  Every 1

Note!

If multiple watch conditions are satisfied by a single program statement, only the first one detected will be shown.

Once a watch condition is established, it remains in effect until it is deleted or modified from the Breakpoints List window, or until the debugger is ended.

It is important to understand that when a watch condition is established, the watched storage location address does not change until the watch is deleted or modified. This can lead to unpredictable results if a temporary storage location is watched and that storage location is reused while the application is running. An example of this is the automatic storage of an ILE C procedure, which can be reused if the

procedure ends. You should delete watches for automatic storage before the program or procedure ends or you may see unexpected watch breakpoints.

The maximum number of watches that can be active across the entire system is 256. This includes watches set by the Dedicated Service Tool (DST). A user session may have as many watches as are available. If the system maximum is exceeded the watch will not be set. If a watch condition crosses a page (4 KB) boundary, the system will internally use two of the available 256 watches; therefore, depending on the watch locations, the system has the capability to set at least 128 watches and up to 256. The Display Debug Watch command (DSPDBGWCE) can be issued from an AS/400 job to display active watches and which jobs set them.

Stepping Through a Program

You can step through your program using the tool bar step buttons in the source view window:

Step Over executes the current line but does not enter any called procedure or program.

Step Debug executes the current line and enters any called procedure or program that has debug information.

Note: You can also step through your program by selecting the *Step Over* or *Step Debug* choice from the *Run* menu pull-down in the source window, or by using mouse button 2 if it is set for stepping.

Displaying and Changing Variables

To display the value of a variable, double-click on the name of the variable in a source view window. If the variable contains special characters, the debugger may not correctly select the whole variable name. You must select the variable with the mouse before double-clicking. To change the value of a variable:

- Double-click on the value of the variable in the monitor window.
- Type the new value in the edit area that is displayed.
- Press Enter to set the new value.

Writing Code That the Debugger Supports

There are a few points to remember when writing code:

- Multiple statements on the same line are difficult to debug.

Because individual statements cannot be accessed separately when you set breakpoints or when you use step commands, you may want to avoid the use of multiple statements on the same line.

- Although it is possible to create programs that contain more than one module with the same name (if the modules are in different libraries), the debugger does not support them.

  Attempting to debug programs that contain two or more modules with the same name will result in the display of an error message.

Debugger Performance Considerations

In general, you should not have to worry about debugger performance, but if you find that the debugger is stepping slowly, you may want to consider the points discussed in the following sections.

Expression Evaluations

- Complex expressions take longer to evaluate than simple expressions.

  Performance is an issue only when you are monitoring an expression, since the expression must be evaluated each time the debugger stops.

- The settings of the Default Data Representation window affect the performance of expression evaluation:
  - Representing character pointers, arrays, and character arrays as hexadecimal pointers gives the best performance.
  - Representing structures using system defaults performs better than that using user defaults.
  - Representing a character array as a string is faster than representing it as an array.
- Evaluating all of the elements of a large array takes longer than evaluating single elements. Use the Monitor Expression window to evaluate a single element.

Step Performance

Step performance is affected by the number of enabled monitor windows, the numbers of expressions in the windows, and the complexity of the expression. Step performance can be improved by:

- Disabling or deleting expressions that no longer need to be monitored.
- Displaying only single elements of an array.
- After following a chain of pointers to a variable, disabling the pointers used and leaving only the variable active in the monitor.
- Not stepping when the Local Variables window is enabled.

- Not stepping when the Call Stack window is enabled. (Minimizing the Call Stack window will disable it.)

Using PC Files Instead of AS/400 Source Members

For non-C++ programs, performance can be improved by copying the files to the PC and using the *Change text file* choice from the *View* menu to specify the path name of the PC file. This technique is especially useful when debugging from remote sites.

Searching for Strings in Text View

String searches can be speeded up by the following:

- Keeping the source file on the workstation.
- Using the *Find Procedure* choice to search for procedures
- Searching the *LISTING view instead of a source view that is on the AS/400.

Using the From/To/Every Entry Fields on Line Breakpoints

Large values specified for these options will significantly slow your program, because the debugger must stop for the breakpoint and evaluate the From/To/Every clause at each stop. Even though you do not see the program stop, it is in fact stopping so that the debugger can evaluate the stop conditions.

If possible, an alternative is to set a conditional breakpoint by specifying an expression.

Setting a Large Number of Watches

When a watch is set, the system checks after each instruction whether the value of the monitored variable or expression has changed. Setting many watches may lead to slower performance.

Debug Limits of the Cooperative Debugger

The following applies when you debug applications with the ADTS Client Server for AS/400 cooperative debugger.

Limits

The limits are these:

- The largest string that can be displayed is 4080 characters.
- The largest number of bytes that can be displayed in a hexadecimal dump is 1024.
- The largest number of elements displayed in a COBOL array or an RPG array is 500.

- The largest number of fields displayed in a COBOL record or an RPG structure is 500.
- Only 256 elements per dimension are returned for C and C + + arrays.
- Only 140 characters per line are displayed for the text views retrieved from the AS/400. Files residing on the workstation will have the entire line displayed regardless of the line length.
- A maximum of 128 bytes can be watched for any variable.
- A maximum of 256 watches can be set system-wide. This number may be lower, because watching a variable can sometimes require more than one system watch.

Source Files with a Record Size Greater than 240

Source physical files that have a record size greater than 240 characters cause a message to be written to the job log for each record read. (The job log that the messages are written to is the job log of the debug server job that is serving your debug session.) This behavior slows down processing and may cause your debug session to end if the job log grows too large.

Chapter 3. Data Access

In this chapter, we show you different ways of accessing data from a VisualAge for RPG application. This includes not only the record I/O support for AS/400 files and local PC files, but also the integration of SQL to access DB2 databases. Additionally, we explain how AS/400 data areas can be used in your VisualAge for RPG applications.

AS/400 Files and Record I/O

Accessing AS/400 database files using record I/O in VisualAge for RPG is relatively straightforward and you will have your first file maintenance example running quite soon. However, when you look deeper, there are some additional considerations such as open data paths, overrides, locking, record blocking, and commitment control, you may want to know a bit more about.

An Easy Approach

Before an AS/400 file can be accessed through the different file operation codes, it has to be defined in the file description specifications. First, there are the fixed-format, column-oriented specifications, such as:

- File name
- File type
- File designation (full procedural, table, or output)
- Whether or not file addition is allowed at the end of the file
- Whether the file's format is internally or externally described
- Record length
- Record address type
- Whether it is a DISK, PRINTER, or SPECIAL file

This is nothing really new, if you are already familiar with ILE RPG/400. However, some restrictions apply, if you want to define an AS/400 database file as a DISK file within your VisualAge for RPG application:

- Its format must be externally described.
- The file designation can be full-procedural or output.

Refer to the *VisualAge for RPG Language Reference* manual for a list of all available options in the different columns and the various option combinations that can be used.

As in ILE RPG/400, any additional information is provided through keywords specified in columns 44-80. The most important keyword for an AS/400 database file in VisualAge for RPG is the REMOTE keyword, which is the indicator for a file that resides on an AS/400 server, rather than a local file (Figure 8).

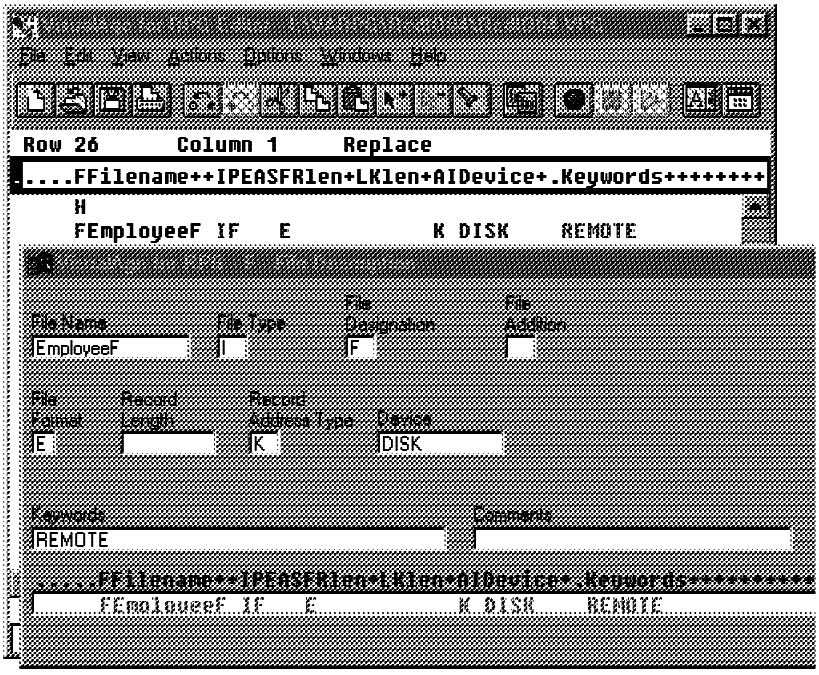


Figure 8. File Description Specification Window

If nothing else has been specified, the file is searched on the default AS/400 server as defined on the *Servers* page of the *Define AS/400 Information* window. The library list of the user ID specified during logon to the AS/400 is used for this search.

If the database file has a different name on the AS/400 than in your program, or if you would like to specify a certain library rather than having the libraries of the library list searched, you can specify this information on the *Files* notebook page of the *Define AS/400 Information* window (see Figure 9). The information provided is used by both the compiler and the VisualAge for RPG runtime to find the file on the AS/400 server.

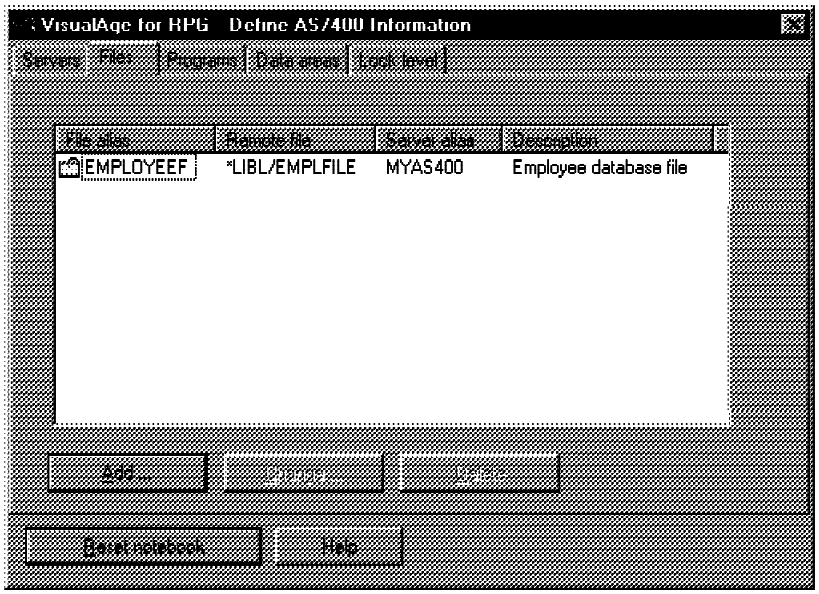


Figure 9. Define AS/400 Information Window, Files Page

An entry for your file may also be necessary here, if the files of an application reside on more than just one AS/400 server. Through this notebook, you can specify different servers for each of the accessed files. VisualAge for RPG resolves the information and takes care that every file is accessed on the correct server.

Additionally, it is possible to override the default setting (*FIRST) for the file member to be accessed, if the file contains two or more members (see Figure 10). In this case, use the format *Library/File(Member)* to qualify the particular member you need to access.

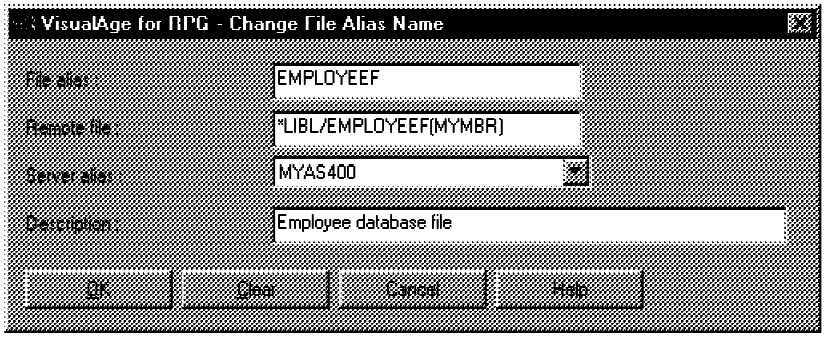


Figure 10. Overriding the Default Member

Keep in mind that this kind of member override is resolved on the workstation during program initialization. Therefore, it is not really an override as the AS/400 understands it, and you won't see it on the AS/400 using option 15 of the DSPJOB command.

All this information is stored in the .RST file in the project's directory. This ASCII file is packaged and shipped with your application and can be maintained using a standard editor or by calling program FVEDRST.EXE, which is the command interface of the *Define AS/400 Information* notebook. You can pass the .RST file's name as a parameter to the command. See Figure 11.

RST File

```
DEFINE_SERVER  SERVER_ALIAS_NAME(MYAS400)
               REMOTE_LOCATION_NAME(IVL01234)
               TEXT(My default server)

DEFINE_FILE  FILE_ALIAS_NAME(EMPLOYEEF)
             REMOTE_FILE_NAME(*LIBL/EMPLOYEEF(MYMBR))
             SERVER_ALIAS_NAME(MYAS400)
             TEXT(Employee database file)
```

Figure 11. Code for the RST File

Let's go back into our VisualAge for RPG source. We could use additional keywords to define the file in further detail: some of the keywords allow you to change the information that is included from the external description of the file, such as the RENAME, PREFIX, INCLUDE, and IGNORE keywords.

For example, the RENAME keyword allows to change the name of a record format. Through this keyword, it is possible to include two record formats, which have the same name on the AS/400. Additionally, if there are identical field names coming from different files, consider the use of the PREFIX keyword.

See Figure 12, which shows the compiler listing of a sample VisualAge for RPG program. The listing includes an AS/400 database file twice, renaming the record format as well as the fields for one of them.

Listing

```
FEmployeeF IF   E           K DISK    REMOTE
 *----------------------------------------------------------------
 *                              RPG name         External name
 * File name. . . . . . . . . :  EMPLOYEEF        REINHARD/EMPLFILE
 * Record format(s) . . . . . :  EMPRCD           EMPRCD
 *----------------------------------------------------------------
FEmplFile  IF   E           K DISK    REMOTE
F                                     RENAME(EMPRCD:EMPLREC)
F                                     PREFIX(B)
 *----------------------------------------------------------------
 *                              RPG name         External name
 * File name. . . . . . . . . :  EMPLFILE         REINHARD/EMPLFILE
 * Record format(s) . . . . . :  EMPLREC          EMPRCD
 *----------------------------------------------------------------
IEMPRCD
 *----------------------------------------------------------------
 * RPG record format  . . . . :  EMPRCD
 * External format  . . . . . :  EMPRCD : REINHARD/EMPLFILE
 *----------------------------------------------------------------
I                             S    1    7 0NUMBER
I                             A    8   27  FIRSTNAME
I                             A   28   47  LASTNAME
IEMPLREC
 *----------------------------------------------------------------
 * RPG record format  . . . . :  EMPLREC
 * Prefix . . . . . . . . . . :  B : 0
 * External format  . . . . . :  EMPRCD : REINHARD/EMPLFILE
 *----------------------------------------------------------------
I                             S    1    7 0BNUMBER
I                             A    8   27  BFIRSTNAME
I                             A   28   47  BLASTNAME
```

Figure 12. Compiler Listing of a Sample Program

Other keywords allow you to enable or disable commitment control (COMMIT) or help in file exception handling (INFSR and INFDS). For a complete list and a detailed description for each of them, please refer to the *VisualAge for RPG Language Reference* manual.

To access the data from an AS/400 database file defined in your VisualAge for RPG program, use one of the following operation codes:

CHAIN Random retrieval from a file

CLOSE Close files

COMMIT Commit all changes

DELETE Delete record

EXCEPT Calculation time output

FEOD Force end of data

OPEN Open file for processing

POST Post information into the information data structure

READ Read a record

READE Read equal key

READP Read prior record

READPE Read prior equal

ROLBK Roll back all changes

SETGT Set greater than

SETLL Set lower limit

UNLOCK Unlock a data area or release a record

UPDATE Modify existing record

WRITE Create a new record

Which of these file operation codes are available for your file depends on its definition and current status. For example, an UPDATE operation code (opcode) will not work on a file that is opened for input only or is not opened yet. Again, refer to the *VisualAge for RPG Language Reference* manual for all the options, relationships, and dependencies of the different operation codes available.

The sample VisualAge for RPG source (Figure 13) shows you how easy it is to access an AS/400 database file. In the action subroutine of the *READ* push button part, the opcodes SETLL and READ are used to read the content of a keyed AS/400 database file sequentially, until the end-of-file indicator (*IN80) is set on.

RPG

```
FEmployeeF IF   E           K DISK    REMOTE
 *
 *
C     READ          BEGACT    PRESS         MAIN
 *
C     *START        setll     EmloyeeF
C                   read      EMPRCD                                 80
C                   dow       *IN80 <> *ON
C                   write     EMPLLIST
C                   read      EMPRCD                                 80
C                   enddo
 *
C                   ENDACT
```

Figure 13. Reading an AS/400 Database File

Note that the WRITE opcode is not accessing an AS/400 database file here. Instead, it is used to place the content of the previously read record into a subfile part of the graphical user interface of the sample. Keep this in mind, when you examine a VisualAge for RPG source you don't know.

The special words *START and *STOP are unique for VisualAge for RPG. Their purpose is to replace the *LOVAL and *HIVAL figurative constants that are not converted from ASCII to EBCDIC, when they are used as key values to access the AS/400 database. Using *START and *STOP instead makes sure that the program is reaching the correct record in the file, also considering a possibly different coded character set identifier (CCSID) on the AS/400.

If you build your VisualAge for RPG application, a service job is evoked on the AS/400 in subsystem QCMN. You can find it by looking for a job that is named as the LU of your PC workstation and a user name as the user ID specified for the logon during the build.

AS/400 screen

```
Subsystem/Job  User        Type  CPU %  Function        Status
QCMN           QSYS        SBS     0,2                  DEQW
  SC02013M     LEISING     EVK     1,1                  RUN
```

In the joblog, you will find several messages of type *Request* talking about 'Client Request - Executing the program ...', which can help you identify the correct job.

To reduce compile time, select the *Use cache* check box on the *Compile* page of the *Build options* notebook (Figure 14). During the next build, the information for the accessed AS/400 files is stored on the PC and will be used during future builds instead of resolving the information from the AS/400.

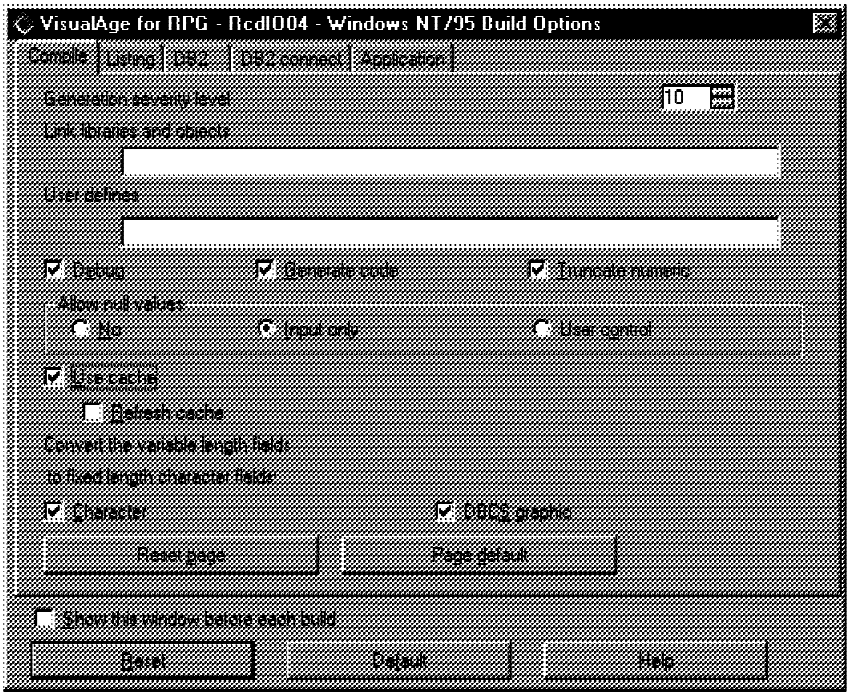


Figure 14. Build Options Window, Compile Page

Select the *Refresh cache* check box to have the file information on the PC refreshed on the next compile. This is necessary if the structure of those files, for which this information was stored earlier on the PC, has changed.

During runtime, a distributed data management (DDM) job is started in subsystem QCMN as soon as the first OPEN (implicit or explicit) operation is performed in your VisualAge for RPG application. The DDM job handles all database requests coming from the same VisualAge for RPG application to the AS/400 server. Even if your application consists of different components and several (or all of them) access AS/400 database files, the serving job on the AS/400 will always be the same DDM job.

The job will be ended automatically, as soon as the last database file of the VisualAge for RPG application is closed.

If a user is reporting abnormal terminations or other strange behavior of an application, more often than not, the problem's circumstances and symptoms are not described properly. In such cases take a look at the job log of the DDM job. It contains messages for errors that occurred while accessing the AS/400 database, such as the sample in Figure 15, where the user tried to add a new record to a database file with a UNIQUE key that already existed.

AS/400 screen

```
                      Display All Messages
                                                          System:   MCEAS4
Job . . :  SC02013M     User . . :  LOOSER      Number . . . :  145022

Job 145022/LEISING/SC02013M started on 23.04.97 at 20:11:55 in
  subsystem QCMN in QSYS. Job entered system on 23.04.97 at 20:11:55.
Target DDM job started by source system.
Duplicate record key in member EMPLOYEEF.
Duplicate key not allowed for member EMPLOYEEF.
Duplicate key not allowed for member EMPLOYEEF.
```

Figure 15. Sample Job Log Showing Error Messages

Open Data Paths and Overrides

While we are on the AS/400, let's take a closer look at open data paths (ODP) and overrides and see how those fit into the VisualAge for RPG environment.

An open data path is basically a temporary object through which all input/output operations for a file are performed. In a sense, it connects the program to a file.

As soon as an OPEN operation is performed for a file, an ODP is created. Its scope depends on whether the program performing the OPEN is running in the default activation group or not, and on the value specified for the OPNSCOPE parameter on the Override with Database File command (OVRDBF) or Open Database File command (OPNDBF).

For programs that run in the default activation group, such as the programs doing the OPEN in the DDM job serving VisualAge for RPG's database accesses, the ODP is scoped to the call level. It stays there until the file is closed, which happens explicitly through a CLOSE operation, by executing the Reclaim Resources command (RCLRSC) or implicitly when the program that initiated the ODP's creation is terminated.

Sharing Open Data Paths

By default, every OPEN for an AS/400 file results in the creation of an ODP, enabling the system to have multiple programs in the same job perform I/O operations to the same file without interfering with each other. Record locking as well as the current position in the file (as set by the last SETLL, SETGT, CHAIN or READ operation) will be changed for that ODP through which the last file operation was done (see “File Locking”).

This default behavior can be changed by using the SHARE parameter available on the following commands:

- Create physical file (CRTPF)
- Create logical file (CRTLF)
- Change physical file (CHGPF)
- Change logical file (CHGLF)
- Override with database file (OVRDBF)

Specifying SHARE(*YES) on one of these commands will have the ODP opened in shared mode. This means that only the first OPEN in a job creates an ODP object. Any following OPEN in the same job to the same file specifying the same open options (such as the CCSID of the file) will reuse the ODP. Record locking and file positioning will then be performed through the same ODP from different parts of the application.

Note that if the options are not the same for all OPENs on the shared ODP, the first (shared) OPEN needs to include the options of all following OPENs. Otherwise, the options of subsequent OPEN operations may be ignored (message CPF4123) or the OPEN may even fail. For example, if the first OPEN is of read-only type, then a subsequent OPEN for update will be ignored, causing problems when trying to perform an UPDATE operation.

A shared open data path can be identified through the *Display Open Files* screen (Option 14, F11) of the *Display Job* (DSPJOB) menu. Figure 16 shows two ODPs, all created for the same database file, *EMPLOYEEF*. While the first one was done for a nonshared open, the second ODP serves two different open requests.

AS/400 screen

```
                              Display Open Files

Job . . :  SC02013M     User . . :  LEISING     Number . . . :  086370
Number of open data paths . . . . . . . . . . :      2


                   Member/    Record   File  I/O  ----Open--- Relative
File      Library  Device     Format   Type Count Opt Shr-Nbr  Record
EMPLOYEEF LEISING  EMPLOYEEF           PHY      0 I   NO
EMPLOYEEF LEISING  EMPLOYEEF           PHY      0 I   YES  2
```

Figure 16. Two Open Data Paths

So far, so good. But how does VisualAge for RPG behave in this area? We have already learned in the previous section (“An Easy Approach”), that two different VisualAge for RPG applications result in two different DDM jobs started on the AS/400. As an ODP cannot be shared between programs running in different jobs on the AS/400, separate ODPs will be created for every separate VisualAge for RPG application, even if they are accessing the same file.

So what happens, if the same AS/400 file is accessed from two or more components of the same VisualAge for RPG application, or even within the same component using different file definitions (file aliases)?

If you keep the default *NO for the SHARE parameter of the AS/400 database file that is accessed by VisualAge for RPG, you always get a separate instance of the file here as well. A new ODP is created and scoped to the call level (*ACTGRPDFN in the default activation group). It is kept open until an explicit CLOSE is performed through the same ODP or the component is terminated.

If the SHARE parameter is changed to *YES for the accessed file, only the first open creates an ODP. Any following open to the same file will try to reuse this ODP, as outlined earlier.

Basically, the AS/400 file behaves the same in this environment as in any other AS/400 job. Whatever value (*YES or *NO) is specified when you open a file from VisualAge for RPG, it will be treated like a file from any other native AS/400 application.

If you find VisualAge for RPG ignoring the SHARE(*YES) parameter and open duplicate ODPs, you are probably missing one of the PTFs that enable this

support for VisualAge for RPG. Refer to Table 2 to find the correct PTF for your OS/400 release.

Table 2. PTFs for Shared ODP Support

Release	PTF
V3R1M0	SF32580 and SF34222
V3R2M0	SF34214
V3R6M0	SF38099
V3R7M0	SF38113

Given the correct PTFs installed, the question is, how and where do you specify this value for the SHARE parameter using VisualAge for RPG? If you do it through the CRTPF, CRTLF, CHGPF, or CHGLF commands, it is permanently set at the file level and the behavior will be the same for all applications (VisualAge for RPG as well as native AS/400) that access the same file.

You get into trouble with this permanent setting approach as soon as you need different settings for the SHARE parameter for different applications. As the difference in behavior is quite significant, you cannot just do a CHGPF SHARE(*YES), if your program needs this setting. Doing so will change the value system-wide, and you could cause other applications in different jobs to fail, if they rely on SHARE(*NO).

Overriding Database Files

The *Define AS/400 Information* notebook does not allow you to change the SHARE parameter, so you cannot have VisualAge for RPG change the value for you. The only solution is to override during run time using the Override with Database File command (OVRDBF) as you would do in a native AS/400 application.

The interface to perform an OVRDBF command from a VisualAge for RPG application is through the QCMDDDM API. Its functionality and interface are similar to those of the QCMDEXC API known from the AS/400. It allows you to execute commands on the AS/400. The difference between these two APIs is the AS/400 server job for which they execute the specified commands.

While the QCMDEXC API runs its commands in a separate server job, which is also used to run all AS/400 programs called by VisualAge for RPG through the CALL (LINKAGE(*SERVER)) opcode, the QCMDDDM API executes commands in the DDM job responsible for the database access of the application. Both will run on the AS/400 server specified on the *Program* page of the *Define AS/400 Information* notebook.

Figure 17 shows a sample source illustrating how the OVRDBF SHARE(*YES) operation can be done using the QCMDDDM interface:

RPG

```
D QCmdDDM         C                   'QCMDDDM'
D                                     LINKAGE(*SERVER)
 *
D OvrDBF          C                   'OVRDBF EMPLOYEEF +
D                                             SHARE(*YES) +
D                                             OVRSCOPE(*JOB)'
 *
C                   call      QCmdDDM
C                   parm      OvrDBF        Cmd             255
C                   parm      255           CmdLen           15 5
```

Figure 17. Using the QCMDDDM Interface

It is important to specify *JOB for the OVRSCOPE parameter of the OVRDBF command. By default, the override scope is the call level in the default activation group, which would result in the override being removed as soon as QCMDDDM returns control to your VisualAge for RPG program. With OVRSCOPE(*JOB) the override remains active throughout the lifetime of the job, or until it is deleted using the DLTOVR command.

If you do an override this way, however, you must keep the following in mind:

- The file for which the override is done must not be open already. If the override is performed in the same component that also accesses the file, you need to specify the USROPN keyword on the file description specification and use the OPEN opcode to open the file after the override is done. Otherwise, the override is ignored.
- If there is an entry for the file in the *Define AS/400 information* notebook, the file name specified on the OVRDBF must be the same as the *remote file name* entry.
- As the override scope is the job level, the override is also active for any other component of the same VisualAge for RPG application that accesses the same AS/400 database file.
- This kind of override is possible only for the AS/400 server that was specified on the *Program* page of the *Define AS/400 Information* notebook. If you need to run the QCMDDDM API for DDM jobs on different AS/400 servers, you have to add multiple alias entries here, one for each server.

Open Query File

So far, so good. Let us now have a look at another topic, which always comes up when talking about shared ODPs: the Open Query File command (OPNQRYF).

Now, where we have managed to perform an override on an AS/400 database file from our VisualAge for RPG application, we are able to have a shared ODP opened. So, it shouldn't be too difficult to use the OPNQRYF command in our DDM job as well, right?

Well, it's not that easy. Even if you can use the QCMDDDM API to have the OPNQRYF command performed in the DDM job and the OPNSCOPE parameter allows us to have the open done on job level, the shared ODP that gets created won't be reused by any VisualAge for RPG application.

The problem is that the ODP created by the OPNQRYF command gets a CCSID assigned according to the CCSID of the DDM job. On a US-English system, this will be 037. On the other hand, VisualAge for RPG tries to open an ODP with a PC code page that uses 437 for US-English. As a result, the OPEN operation of VisualAge for RPG will fail, issuing message CPF417C (error code 1). Unfortunately, changing the CCSID of the job to an ASCII code page is not possible.

In the older releases V3R1M0, V3R2M0, and V3R6M0, the only solution is to either:

- Change the design of the AS/400 database structure to incorporate some of OPNQRYF's functions. For example, sorting or selection specifications could be implemented through a key or through select/omit criteria in a logical file.
- Or, use SQL support of VisualAge for RPG (refer to section “SQL Support”).

With V3R7M0, the CCSID parameter has been introduced for the OPNQRYF command, allowing us to specify the CCSID in which data from character, DBCS-open, DBCS-either, and graphic fields will be returned. Take a look at the example in Figure 18, where we use the OVRDBF and OPNQRYF commands to read into a subfile only those records from file *Employeef* that have an employee number ranging between two variable values.

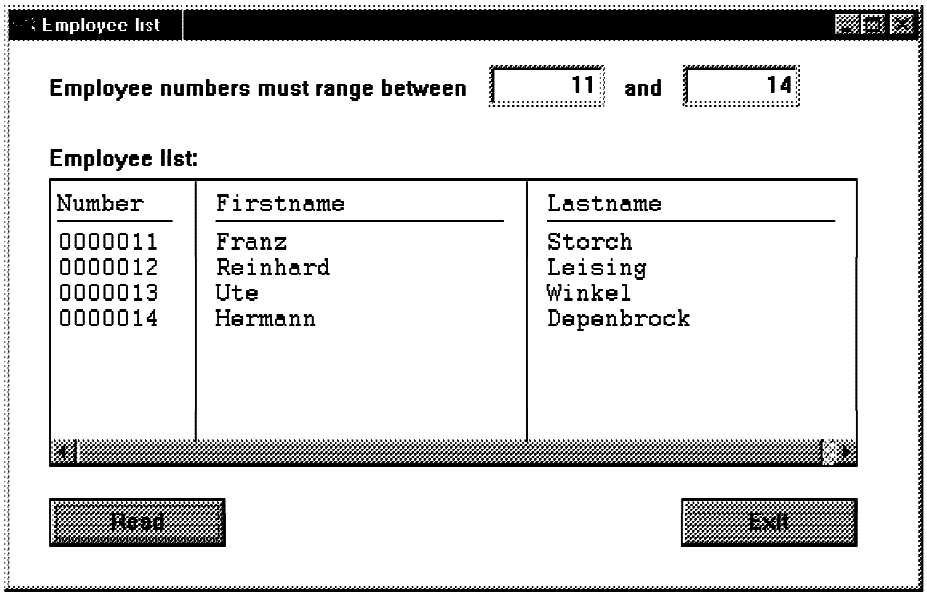


Figure 18. The OPNQRYF Sample of Records to Be Read

The action subroutine of the PRESS event changes the SHARE parameter for the DDM job to *YES, creates a shared open data path using OPNQRYF, and opens the file with the same attributes, so that it shares the ODP before it fills the subfile. The code is shown in Figure 19.

RPG

```
FEmployeeF IF   E             DISK    REMOTE USROPN
 *
D OvrDBF          C                   'OVRDBF EMPLOYEEF +
D                                             SHARE(*YES) +
D                                             OVRSCOPE(*JOB)'
 *
D OpnQryF1        C                   'OPNQRYF EMPLOYEEF +
D                                              OPNSCOPE(*JOB) +
D                                              CCSID(437) +
D                                              QRYSLT(''NUMBER *GE '
 *
D OpnQryF2        C                   ' *AND NUMBER *LE '
D OpnQryF3        C                   ''')'
```

Figure 19 (Part 1 of 2). Sharing an Open Data Path Using OPNQRYF

RPG (continued)

```
D CloF            C                   'CLOF EMPLOYEEF'
D QCmdDDM         S             10A   INZ('QCMDDDM')
D                                     LINKAGE(*SERVER)
 *
D Cmd             S            255A   INZ(*BLANKS)
D CmdLen          S             15P 5 INZ(%size(Cmd))
 *
C     READ          BEGACT    PRESS         MAIN
 *
C                   eval      Low  = %getatr('MAIN':'LOW':'Text')
C                   eval      High = %getatr('MAIN':'HIGH':'Text')
C                   if        Low > High
C     InvRange      dsply                   RC               9 0
C                   else
 *
C                   call      QCmdDDM
C                   parm      OvrDBF        Cmd
C                   parm                    CmdLen
 *
C                   eval      Cmd = OpnQryF1 + %trim(Low) + OpnQryF2
C                               + %trim(High) + OpnQryF3
C                   call      QCmdDDM
C                   parm                    Cmd
C                   parm                    CmdLen
 *
C                   open      EmployeeF
 *
C                   read      EMPRCD                                 80
C                   dow       *IN80 <> *ON
C                   write     EMPLLIST
C                   read      EMPRCD                                 80
C                   enddo
 *
C                   close     EmployeeF
C                   call      QCmdDDM
C                   parm      CloF          Cmd
C                   parm                    CmdLen
 *
C                   endif
 *
C                   ENDACT
```

Figure 19 (Part 2 of 2). Sharing an Open Data Path Using OPNQRYF

File Locking

File and record locks for AS/400 database files accessed by a VisualAge for RPG application are maintained by OS/400 in the same way as for any other database file. When file locking is performed by OS/400, one of the following lock states is placed on a file:

- Shared for read (*SHRRD)
- Shared for update (*SHRUPD)
- Shared, no update (*SHRNUP)
- Exclusive, allow read (*EXCLRD)
- Exclusive, no read (*EXCL)

When VisualAge for RPG opens a file, only one of the first two lock states is used. Which state is used depends on the type of OPEN performed:

INPUT Shared for read

UPDATE Shared for update

ADD Shared for update

OUTPUT Shared for update

Locking Conflicts

A file lock is always held by a job, so that it applies only when a program in another job tries to use the file concurrently. Programs within the same job are not affected by file lock states. If an application tries to open a file and a lock state is already set on the file that doesn't allow a second lock, then an error message is issued.

As all components of a single VisualAge for RPG application use the same DDM job for database access, file locks do not apply between these components. However, other VisualAge for RPG applications may access the same files and try to set locks themselves.

Your VisualAge for RPG application should be able to handle this contention, either by specifying error indicators for the different file operation codes or by supplying an information subroutine (INFSR). (For more information regarding exception handling, please refer to section “Exception Handling.”)

Additionally, you may consider the WAITFILE parameter of the CRTPF, CRTLF, CHGPF, CHGLF and OVRDBF commands to specify how long the application is to wait for another job's lock to be released. (Keep in mind that the user of your application will have to wait as well.) It may be a better idea to handle the exception and issue a meaningful message.

Record Locking

Record locks are handled in a different way than file locks. There are usually only two lock states (locked for update and released), and they are maintained through an open data path, not a job. So, there are far more occurrences when a record lock could be and will be set through another ODP, preventing access.

A record lock applies only when the file to be accessed has been opened for update. Performing this kind of open indicates your intention to change a record that you have read before. OS/400 then needs to make sure that no other program is trying to read the same record for update at the same time.

To handle this situation, OS/400 applies a lock to the record as soon as it is read for update by your VisualAge for RPG program. Any attempt to perform another read for update through another ODP will fail.

Dead-Lock Situations

As any open of a file (first shared or not shared) in a VisualAge for RPG application results in its own ODP, a record lock could also prevent another component of the same VisualAge for RPG application from reading an already locked record for update.

This is important to understand, as it will also happen if you are accessing the same file within a single component through two different ODPs. If both ODPs have been opened for update and a read for update is performed on an already locked record, you end up in a dead-lock, where your program is waiting for itself to release the record.

It is not really a dead-lock, however, as OS/400 is usually not waiting for a locked record until the end of all days. Instead, it looks for the setting of the WAITRCD parameter of the database file. By default, it is set to 60 seconds, which makes a program (or VisualAge for RPG component) performing a read for update wait for 60 seconds for the record to become released. If that doesn't happen in the specified time, an exception (RNQ1218) is signaled to the component.

Can't Wait!

It is usually a good idea to set the record wait time to a value lower than 60 seconds, if you expect record waits in your application. A wait time of 3 or 5 seconds is more appropriate, because users tend to get nervous when their computer program seems to have stalled.

Read for Update

What exactly is a read for update in VisualAge for RPG language? It is one of the following operation codes performed on a file opened for update:

- CHAIN
- READ
- READE
- READP
- READPE

Once, a record has been locked, the lock remains until one of the following happens through the ODP that placed the lock:

- The record is updated (UPDATE).
- The record is deleted (DELETE).
- Another record is read from the file (CHAIN, READ, READE, READP, READPE).
- A SETLL or SETGT operation is performed against the file.
- An UNLOCK operation is performed against the file.
- An output operation defined by an output specification with no field names included is performed against the file (EXCEPT).
- The component that opened the file ends.

Duration of Record Lock

As you design your application, you should focus on reducing the amount and duration of record locks being held. For example, the following action subroutine is associated with a push button part called *Next* of a database maintenance program (Figure 20). When it is pressed, the next record is read from the file and its content is displayed to the user. So, pressing it multiple times will allow the user to scroll through the records of the file.

```
RPG
FEmployeeF UF A E           K DISK    REMOTE USROPN
 *
 *
C     NEXT          BEGACT    PRESS         MAIN
 *
C                   read      EMPRCD                               80
C                   if        *IN80 = *ON
C     EndOfFile     dsply                   RC
C                   else
C                   write     'MAIN'
C                   endif
 *
C                   ENDACT
```

Figure 20. Action Subroutine for Next Button

As the user will also be able to change the displayed values, the file is opened for update here, so that an update can be made to a record, if necessary. However, updating will also cause a record lock being applied to the record when it is read.

This lock remains until the user presses the *Next* push button again or the *Change* push button, indicating that changes were made and the currently locked record needs to be updated. In our example, we just need to read the content from the entry fields of the window and, as these have the same names as the fields of the database record, perform an UPDATE opcode:

```
RPG
C     CHANGE        BEGACT    PRESS         MAIN
 *
C                   read      'MAIN'
C                   update    EMPRCD
 *
C                   ENDACT
```

The problem is, that you don't know when the user will press one of these push buttons. It might take just one second, while scrolling through the file, but it could also last until the end of the day, as the user is doing something else while displaying the record. You would have to call the user to have the lock released.

Read Without Locking

To prevent this kind of lock difficulty, you could use the (N) operation code extender on the READ opcode. This operation code extender, which is available for every VisualAge for RPG operation code that usually places a record lock, prevents OS/400 from applying the record lock, leaving the record free for any other update request (Figure 20).

RPG

```
C     NEXT          BEGACT    PRESS         MAIN
 *
C                   read(N)   EMPRCD                              80
C                   if        *IN80 = *ON
C     EndOfFile     dsply                   RC
C                   else
C                   write     'MAIN'
C                   endif
 *
C                   ENDACT
```

However, if the user now decides to change a record, you will have to read it again from the file without the operation code extender to place a lock on it. Otherwise, the UPDATE operation would fail as it always expects a locked record.

But looking at the modified action subroutine for the *Change* push button, you can see it is not that simple. The record was accessible, since it was read first by this user. However, it may have been updated or deleted by another user by the time the first user reads the record again for update. Figure 20 shows a modified action subroutine for the *Change* push button.

RPG

```
FEmployeeF UF A E           K DISK    REMOTE USROPN
F                                     PREFIX(F)
 *
D WdwFields       DS
D   Number                       7S 0
D   Firstname                   20A
D   Lastname                    20A
 *
D FileFields      DS
D   FNumber                      7S 0
D   FFirstname                  20A
D   FLastname                   20A
 *
 *
C     CHANGE        BEGACT    PRESS         MAIN
 *
C     FNumber       chain     EMPRCD                            80
C                   if        *IN80 = *ON
C     NotFound      dsply                   RC
C                   else
 * record was found
C                   if        FileFields <> WdwFields
C     OtherUpd      dsply                   RC
C                   else
 * no update since last read
C                   read      'MAIN'
C                   move      WdwFields     FileFields
C                   update    EMPRCD
C                   endif
 *
C                   endif
 *
C                   ENDACT
```

Figure 21. Modified Action Subroutine for the Change Push Button

Therefore, we first need to check whether the CHAIN opcode finds the record or not. If it's not there anymore, a message is issued giving this information to the user. (At this point, you might consider asking the user whether or not to add it again.)

Rereading Record for Update

If the record is found, it now is locked by the CHAIN opcode. The next thing to verify is, if another job updated the record since the first read (with no lock). To be able to read the record again through the CHAIN and compare

the content of the record with the content of the first (no lock) read, we changed the names of the record fields using the PREFIX keyword.

Doing so, we have now three different versions of our record fields for the employee number, the first name and the last name:

- The entry field parts of the GUI show the values as they were changed by the user.
- Since we didn't read the content from these entry field parts, the corresponding VisualAge for RPG variables (*Number*, *Firstname* , and *Lastname*) still contain the values as read from the file through the READ(N) opcode.
- The input buffer of the file (*FNumber*, *FFirstname*, and *FLastname*) itself contains the contents of the record fields as it is available in the database file from the second read (for update).

Comparing the content of the VisualAge for RPG variables with the input buffer fields allows us to determine whether there has been an update or not. If there was no update, we receive the new content from the entry field parts of the GUI into the corresponding VisualAge for RPG variables, copy it to the file buffer variables and update the record in the database file.

Keep in mind, that this approach works only if the key of the database file is unique, so that the CHAIN operation will always fetch the correct record. If the key is not unique, you may be able to identify the correct record using the RECNO keyword.

Commitment Control

The commitment control feature of OS/400 allows you to process several database changes as one unit. Every single database change is made temporarily at first. Only if all of the necessary operations have been completed successfully are the changes permanently applied to the database. If something goes wrong, while some but not all of the necessary changes have been made, you can have OS/400 roll back the changes.

Before we can use the commitment control feature of OS/400 within VisualAge for RPG, we need to set up the environment for it first.

Journaling

On the AS/400, we need to enable journaling for the database file that will be accessed under commitment control. So, we create a journal receiver as well as a journal object and start the journaling for our database to get all changes logged:

AS/400 command

```
CRTJRNRCV JRNRCV(REINHARD/EMPLOYEEF)
          TEXT('Journal receiver object for EMPLOYEEF')

CRTJRN    JRN(REINHARD/EMPLOYEEF)
          JRNRCV(REINHARD/EMPLOYEEF)
          TEXT('Journal object for EMPLOYEEF')

STRJRNPF  FILE(REINHARD/EMPLOYEEF)
          JRN(REINHARD/EMPLOYEEF)
```

Commitment control is started for a job through the Start Commitment Control command (STRCMTCTL). To have this command executed in the DDM job serving the database requests of our VisualAge for RPG application, we must specify a lock level other than <NONE> on the *Lock level* page of the *Define AS/400 Information* notebook (see Figure 22). As you can see, commitment control is restricted to only one AS/400. However, you can still access all files residing on other AS/400 servers without commitment control.

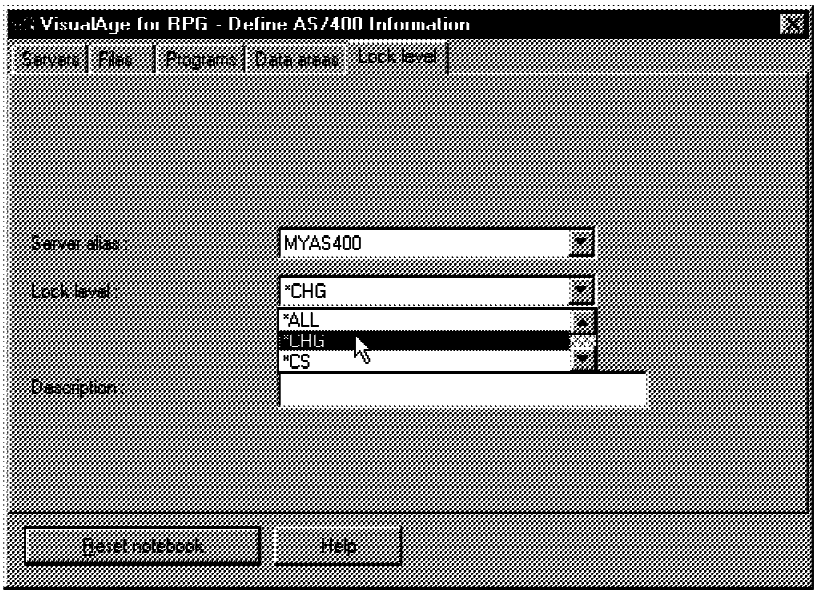


Figure 22. Lock Level for Commitment Control

Lock Level

Which of the available lock levels you choose depends on the type of commitment control you want to have. Commitment control is exercised by keeping record locks for all changed records of the database file until the changes made are permanently applied or removed. With the lock level parameter, you specify which record locks are kept:

***CHG** Every record read for update for a file opened under commitment control is locked. If a record is changed, added, or deleted, it remains locked until the changes made are permanently applied or removed. Records that are accessed for update operations but are released without being changed are unlocked.

***CS** Every record accessed for files opened under commitment control is locked. A record that is read but not changed or deleted is unlocked when a different record is read. Records that are changed, added, or deleted are locked until the changes are permanently applied or removed.

***ALL** Every record accessed for files opened under commitment control is locked until the changes are permanently applied or removed.

As soon as the STRCMTCTL command is executed successfully in the DDM job of our VisualAge for RPG application, the record locks for the file running under commitment control are kept according to this setting. In the file description specification for this file, we need to specify the COMMIT keyword, so that VisualAge for RPG knows which file has to run under commitment control.

Commit and Rollback

All changes to this database file will be temporary, lasting only until the program executes a COMMIT operation code. At this point, all record locks are released and the changes are permanently applied.

If a file operation is unsuccessful, the program can execute a ROLBK operation code. This will remove all changes done, since the last COMMIT opcode was executed (or the file was opened). Record locks are also released. An implicit rollback is performed if the VisualAge for RPG application is terminated (normally or abnormally) before a COMMIT is executed. So, don't forget to supply a COMMIT before leaving your application.

Example

The sample source in Figure 23 gives you an overview of the structure of a VisualAge for RPG application using commitment control

```
RPG
FEmployeeF UF A E           K DISK    REMOTE
F                                     COMMIT
FSalaryF   UF A E           K DISK    REMOTE
F                                     COMMIT

  :

C     UPDATE        BEGACT    PRESS         MAIN
 *
C                   commit
  :
C     Number        chain     EMPRCD                           80
C     EmpNbr        chain     SALRCD                           81
  :
C                   update    EMPRCD                             80
  :
C                   update    SALRCD                             81
C                   if        *IN81 = *ON
C                   rolbk
C                   else
C                   commit
C                   endif
 *
C                   ENDACT
```

Figure 23. Using Commitment Control

The file and the commitment control operation codes don't need to be enclosed in a single action or user subroutine. However, if commitment control spans over multiple action subroutines, you need to make sure that the sequence of the operations is always consistent.

The COMMIT keyword also has an optional parameter that allows you to switch commitment control on and off for a certain file during runtime. The field specified is implicitly defined as a one-byte character field and initialized with '0' (*OFF). As the file needs to be closed when changing this value, you will have to specify the USROPN keyword as well. Figure 24 shows an example of the use of the switching parameter for COMMIT.

```
RPG
FEmployeeF UF A E           K DISK    REMOTE
F                                     USROPN
F                                     COMMIT(CmtMark)

  :

C     *INZSR        BEGSR
C                   eval      CmtMark = *ON
C                   open      EmployeeF
  :
C                   ENDSR

  :

C     CMTOFF        BEGACT    PRESS         MAIN
 *
C                   commit
C                   close     EmployeeF
C                   eval      CmtMark = *OFF
C                   open      EmployeeF
 *
C                   ENDACT
```

Figure 24. Switching COMMIT On and Off

Record Blocking

To improve the performance of an application that sequentially reads data from an AS/400, VisualAge for RPG implements record blocking where applicable. This means that the records of a file are not read one by one, but as a block of multiple sequential records. Blocking reduces the number of remote accesses to the AS/400 and, therefore, can improve performance significantly.

VisualAge for RPG offers default record blocking if any of the following is true:

- The file is opened for output only and contains only one record format.
- The file is a combined file.
- The file is opened for input only and contains only one record format. The file addition entry (column 20) is blank and only the following operation codes are used to access the file:
  - OPEN

  - READ
  - FEOD
  - CLOSE
- The RECNO keyword (allowing access by relative record number) has not been specified on the file description specification.

In addition to this default record blocking, you can enable additional record blocking explicitly through the BLOCK keyword. Specifying BLOCK(*YES) on the F-spec of the accessed files tells VisualAge for RPG to perform record blocking during runtime even if the following operation codes are used on an input-only file:

- CHAIN
- SETLL
- SETGT

RPG

```
FMyFile    IF   E           K DISK    REMOTE
F                                     BLOCK(*YES)
 *
C     MyKey         setll     MyRcd
C                   read      MyRcd                                  80
```

If you don't want to have record blocking at all, you can specify BLOCK(*NO) on a file description specification. This disables even the default record blocking of VisualAge for RPG. It can be useful if you already know that there will be just a few sequential read operations on your file, so that reading blocks of records would affect the performance rather unfavorably.

Exception Handling

Whenever an exception occurs during the access to an AS/400 database file and no error indicator has been specified on the operation codes that raised the exception, the VisualAge for RPG default exception handler receives control (Figure 25).

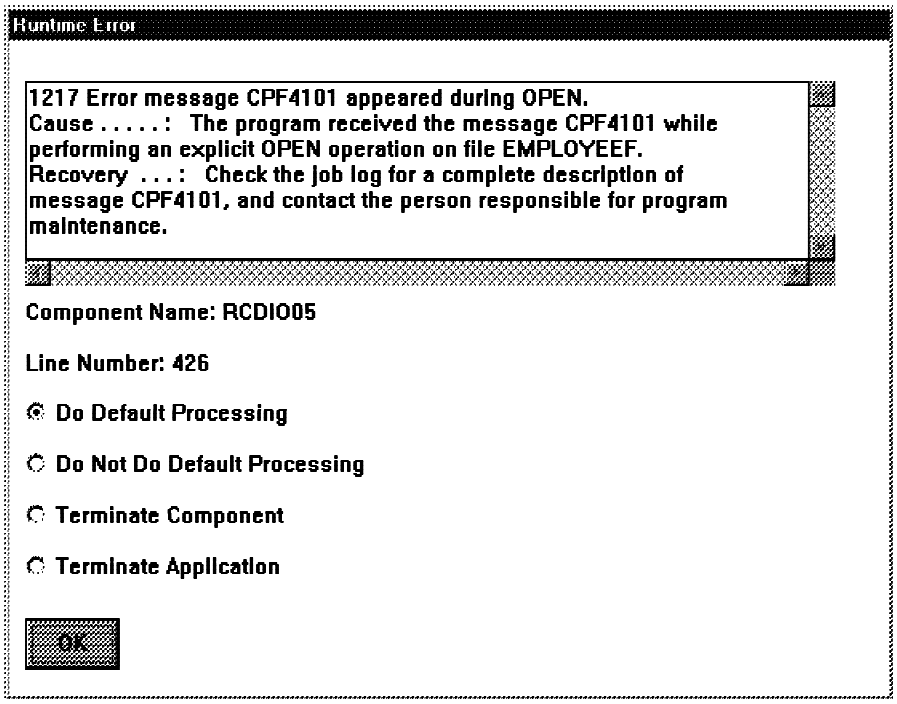


Figure 25. Default Exception Handler

It allows the user to have the runtime perform one of the following actions:

***DEFAULT** Return control from the current action subroutine and perform the default processing associated with the current event.

***NODEFAULT** Return control from the current action subroutine but do not perform any default processing. If the Last Record (LR) indicator is on when processing reaches this point, the component is terminated.

***ENDCOMP** Terminate the component abnormally.

***ENDAPPL** Terminate all currently active components, ending the application.

If you want to have your own exception handler invoked instead of the default, you can register a user subroutine through the INFSR keyword in the F-specs of the database file. The specified subroutine receives control as soon as an exception occurs while accessing the file it is registered for. Within this subroutine, you now can handle file exceptions differently as the default exception handler would do. Additional information regarding the error can be received through the informational data structure (INFDS):

```
RPG
FEmployeeF UF A E           K DISK    REMOTE USROPN
F                                     INFDS(InfDs)
F                                     INFSR(InfSr)
 *
D InfDs           DS
D   Status          *STATUS
```

We are using here only the status field out of the INFDS, which allows us to distinguish between the different exceptions that are possible. For further information about the INFDS and its content, refer to the *VisualAge for RPG Language Reference* manual.

In the example in Figure 26, we have created our own exception handler subroutine. It handles only the exception 1021, which occurs when a WRITE operation is trying to add a record to a unique keyed file that already existed (duplicate key). If this exception is raised during a WRITE operation, we are reading the already existing record and updating its content.

```
RPG
D DupKey          C                   1021
 *
C     InfSr         BEGSR
 *
C                   select
 *
C                   when      Status = DupKey
C                   update    EMPRCD
C                   movel     '*DEFAULT'    ReturnPoint      12
 *
C                   other
C                   movel     *BLANKS       ReturnPoint
 *
C                   endsl
 *
C                   ENDSR     ReturnPoint
```

Figure 26. Example of an Exception Handler Subroutine

Exception Resume

Through the value in factor 2 of the ENDSR opcode, you can determine how the application resumes after leaving your exception handler subroutine. In the case of the duplicate key exception, we specify *DEFAULT, which means

that control returns to the statement immediately following the operation code that raised the exception.

As we want to have the default exception handler take care about all other exceptions, we specify *BLANKS for factor 2. The other available values for the factor 2 of the ENDSR are *NODEFAULT, *ENDCOMP and *ENDAPPL. They correspond to the options available through the default exception handler as shown earlier.

The sample exception handler is called recursively if the CHAIN or UPDATE operations are unsuccessful and, as a result, raise another exception. This results in an infinite loop. You can avoid looping by using an indicator as shown in Figure 27.

RPG

```
D DupKey          C                   1021
 *
C     InfSr         BEGSR
 *
C                   if        *IN01 = *ON
C                   movel     *ON           *INLR
C                   return
C                   else
 *
C                   movel     *ON           *IN01
C                   select
 *
C                   when      Status = DupKey
C                   update    EMPRCD
C                   movel     '*DEFAULT'    ReturnPoint      12
 *
C                   other
C                   movel     *BLANKS       ReturnPoint
 *
C                   endsl
 *
C                   movel     *OFF          *IN01
C                   ENDSR     ReturnPoint
```

Figure 27. Avoiding Recursion in an Exception Handler

Local Files

The interface used in VisualAge for RPG to access a local ASCII file on your PC is very similar to that used for AS/400 database file access. You must define the characteristics of the file through file description specifications. You can then apply the different operation codes used for sequential or relative-record-number processing. Record locking is not available here.

File Definition

Let's first look at what needs to be considered when defining a local file. As a local file is always program-described ('F' in column 22 of the F-spec), we also need to specify the layout of its records. This is done using input and output specifications such at these:

RPG

```
FRstFile   UF   F  100        DISK
F                                      RCDLEN(RcdLen)
F                                      RECNO(RRN)
 *
D RcdLen          S              5P 0 INZ(*ZERO)
D RRN             S              5P 0 INZ(*ZERO)
 *
IRstFile   NS
I                              A    1  100  RstRecord
```

In our example, we defined a program-described file called *RstFile* for update with a maximum record length of 100 bytes. Through its input specification, we specified that we want the entire content of a record read from the file placed in a single field *RstRecord*. So, this field has to be defined with the same length as the record (100 bytes).

Record Length

The RCDLEN keyword was added because we want to control the real length of the records. Local files usually don't contain records of the same fixed length. So, VisualAge for RPG is determining the end of a record by searching for the hexadecimal representation of a carriage-return/line-feed combination (CRLF), when reading from the file. The variable specified as a parameter of the RCDLEN parameter contains the actual length of the record just read.

Additionally, when writing a record to the file (either through an UPDATE, WRITE, or EXCEPT operation code), VisualAge for RPG writes a record of the specified maximum length, placing the CRLF combination at the end. Thus, all updated or added records always have the same length. If you want to

keep variable length records, you can specify a smaller number of bytes to be written to the file using the variable of the RCDLEN keyword.

The RECNO keyword can be used to obtain the relative record number of the record the file is currently positioned to. In a later example, you will see how this can help us to access the correct record.

File Name

During runtime, VisualAge for RPG looks for a file in the application directory with the same name as specified in the F-spec (*RstFile* in our example). If you want to specify a different name, you can use the EXTFILE keyword to specify a variable that contains the external name of your local file:

RPG

```
FRstFile   UF   F  100        DISK    USROPN
F                                     EXTFILE(FileName)
F                                     RCDLEN(RcdLen)
F                                     RECNO(RRN)
 *
D FileName        S             15A   INZ('.\\QCMDEXC.RST')
D RcdLen          S              5P 0 INZ(*ZERO)
D RRN             S              5P 0 INZ(*ZERO)
 *
IRstFile   NS
I                             A    1  100  RstRecord
```

As it is possible to change the value of the variable to a different file name during runtime, the EXTFILE keyword requires the USROPN keyword. You must open and close the file yourself.

Another option would be to specify a different file name through the *Define AS/400 information* notebook. On its *Files* page you can specify the file with a server entry of *CLIENT (Figure 28).

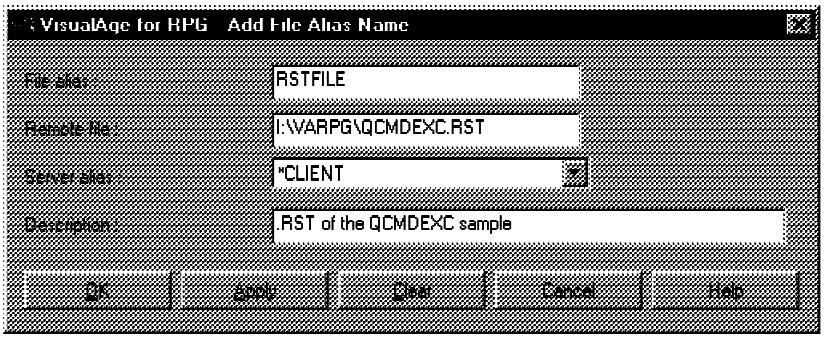


Figure 28. Defining a Local File in the Add File Alias Name Window

Using this interface for your local files allows you to maintain them together with all other external information. As mentioned earlier, a local file can be processed only sequentially or by relative-record number. Thus, only a subset of VisualAge for RPG's operation codes are allowed:

CHAIN Random retrieval from a file through RRN

CLOSE Close files

DELETE Delete record

EXCEPT Calculation time output

OPEN Open file for processing

POST Post information into the information data structure

READ Read a record

READP Read prior record

UPDATE Modify existing record

WRITE Create a new record

We will now use some of the operation codes to access the .RST file of a VisualAge for RPG application that allows to submit commands to an AS/400 using the QCMDEXC API. As we mentioned in “An Easy Approach,” the .RST file contains the information defined through the *Define AS/400 Information* notebook in ASCII format.

The .RST file of our *QCMDEXC* sample contains the definitions of two servers *MCEAS4* and *MCEAS7* as well as the definition for the QCMDEXC API:

RST File

```
DEFINE_SERVER  SERVER_ALIAS_NAME(MCEAS4)
               REMOTE_LOCATION_NAME(IVL$3006)
               TEXT()

DEFINE_SERVER  SERVER_ALIAS_NAME(MCEAS7)
               REMOTE_LOCATION_NAME(IVL$100F)
               TEXT()

DEFINE_PROGRAM  PROGRAM_ALIAS_NAME(QCMDEXC)
                REMOTE_PROGRAM_NAME(*LIBL/QCMDEXC)
                SERVER_ALIAS_NAME(MCEAS4)
                TEXT(QCMDEXC API entry)
```

We want to use this information to fill a combination box in the GUI of our application with the defined server alias names, and change the server alias for the QCMDEXC API, if the user selects another entry from that combination box.

After adding a combination box part called *Server* to the GUI, we use its CREATE event to read the server definitions from the file and fill the list of *Server* with the server alias names (Figure 29).

RPG

```
C     SERVER       BEGACT    CREATE        FRA0000B
 *
C                  open      RstFile
 *
C                  read      RstFile                              80
C                  dow       *IN80 = *OFF
 *
C                  select
C                  when      *IN10 = *OFF and
C                            %scan('DEFINE_SERVER':RstRecord) <> 0
C                  eval      Start = %scan('(':RstRecord) + 1
C                  eval      Length = %scan(')':RstRecord) - Start
C                  eval      %setatr('FRA0000B':'SERVER':'AddItemEnd')
C                             = %subst(RstRecord:Start:Length)
 *
C                  when      *IN10 = *OFF and
C                            %scan('DEFINE_PROGRAM':RstRecord) <> 0
C                  movel     *ON           *IN10
 *
C                  when      *IN10 = *ON and
C                            %scan('SERVER_ALIAS_NAME':RstRecord) <> 0
C                  eval      Start = %scan('(':RstRecord) + 1
C                  eval      Length = %scan(')':RstRecord) - Start
C                  eval      %setatr('FRA0000B':'SERVER':'Text')
C                             = %subst(RstRecord:Start:Length)
C                  movel     *ON           *IN80
C                  endsl
 *
C                  read      RstFile                              80
C                  enddo
 *
C                  ENDACT
```

Figure 29. Using a Create Event to Read Server Definitions

Within a do-while (DOW) loop we are reading through the .RST file sequentially, until the end-of-file condition is raised (*IN80 is set to on). For every found record, we are checking whether it contains information we need for the combination box.

If the string 'DEFINE_SERVER' is detected, we know we have found the alias name for one of the defined servers. So, we scan for the left and right parentheses that enclose this alias name and add it to the list portion of the combination box.

If instead the string 'DEFINE_PROGRAM' is found, *IN10 is set to *ON indicating, that the last server definition has been passed.

The last WHEN opcode checks for the current server alias name of the QCMDEXC API. Therefore, it searches for the next record containing the string 'SERVER_ALIAS_NAME'. The name found between the parentheses is then used to set the Text attribute of the combination box (Figure 30). The relative record number of this record is saved into variable *AliasRRN* as we will need this information later.

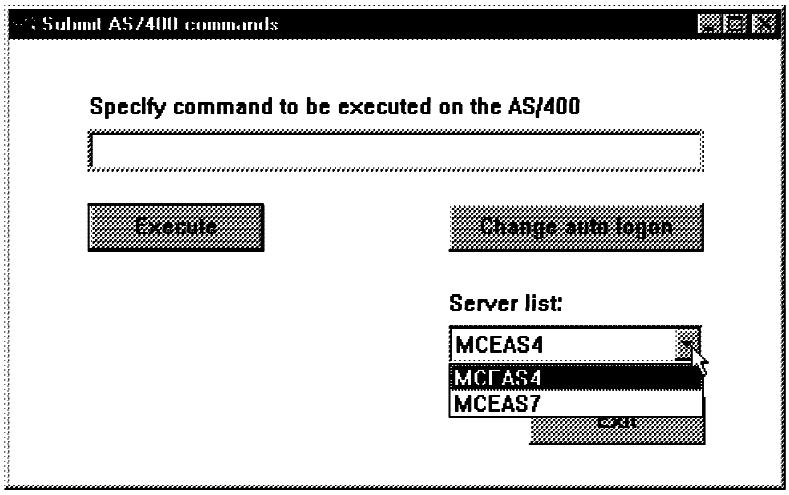


Figure 30. QCMDEXC Sample with Server List

If the user now selects another entry from the list of available servers, we need to change the SERVER_ALIAS_NAME for the QCMDEXC program definition. The action subroutine for the SELECT event of the combination box part is the place to make the necessary changes to our .RST file (Figure 31).

```
RPG
D Server          S             10A   INZ(*BLANKS)
 *
D ServerAlias     DS
D   RstRecord                  100A
 *
IRstFile   NS
I                             A    1  100  RstRecord

  :

C     SERVER        BEGACT    SELECT        FRA0000B
 *
C                   eval      Server
C                              = %getatr('FRA0000B':'SERVER':'Text')
 *
C     AliasRRN      chain     RstFile                           80
C                   eval      %subst(RstRecord:Start:11)
C                              = %trim(Server) + ')'
C                   update    RstFile       ServerAlias
 *
C                   ENDACT
```

Figure 31. Action Subroutine for the Select Event of the Combination Box

Here, we are using our saved relative record number to CHAIN into the file again and read the desired record directly. The buffer of the file gets changed to contain the new server alias name and the record is updated. The link of the QCMDEXC API has been changed now to another server.

This example should give you an idea how local file access can be used within your application. The *VisualAge for RPG Language Reference* manual contains further information about the other operation codes and more complex input and output specifications.

SQL Support

VisualAge for RPG supports the access of DB2 databases through SQL. These databases may be local on your workstation, on other workstation nodes, or on an AS/400 system. We show you how SQL statements can be included into your source, what happens during the build process, and what to consider when the application is installed into another database environment.

VisualAge for RPG supports the level of function available in DB2 Version 1.2. However, your application will be able to access DB2 on other platforms or PCs using more recent releases of DB2 as long as only the Version 1.2 functions and statements are used.

Embedding SQL

In your VisualAge for RPG source, every SQL statement must be preceded by an /EXEC SQL phrase coded in columns 7-15. The statement itself can be coded in the same line starting in column 17 and can span more than one line. Each of these lines must start with a plus (+) sign in column 7 followed by a blank in column 8. The end of a SQL statement must be indicated by coding an /END-EXEC phrase in columns 7-15 of a new line.

RPG

```
C/EXEC SQL
C+  SELECT Number, FirstName, LastName
C+     INTO :Number, :FirstName, :LastName
C+     FROM EmpTable
C+     WHERE Number = 12
C/END-EXEC
```

With the exception of the SQL statements INCLUDE, BEGIN DECLARE, and END DECLARE, that can be defined anywhere in your source before the compile-time data definition (** in columns 1 and 2), all SQL statements must be coded in the calculation specifications. VisualAge for RPG comments (* in column 7) are allowed within an embedded SQL statement. Alternatively, SQL comment indicators (--) can be used to mark the rest of the current line as comment.

Host Variables

The identifiers in the INTO clause of the SQL statement (Figure 32) are host variables, which allow communicating information between SQL and the hosting VisualAge for RPG language. These host variables are identified by a preceding colon (:) and can be of one of the following VisualAge for RPG data types:

- Character
- Graphic
- Packed decimal
- Zoned decimal
- Binary numeric
- Date

- Time
- Time stamp

RPG

```
D/EXEC SQL
D+  INCLUDE SQLCA
D/END-EXEC
 :
C/EXEC SQL
 * This SELECT returns a single row
C+  SELECT Number, FirstName, LastName
C+     INTO :Number, :FirstName, :LastName
C+     FROM EmpTable                          -- employee table
C+     WHERE Number = 12
C/END-EXEC
```

Figure 32. Sample SQL Statement

Although you do not need to specify the same names as those of the corresponding SQL column name, we recommend it for better documentation and program understanding.

Data Type Mapping

How the different SQL data types are mapped to the VisualAge for RPG data types is shown in Table 3.

Table 3. Data Type Mapping between SQL and VisualAge for RPG

SQL Data Type	VARPG Data Type	Format (Col 43)	Length (Cols 44-51)	Decimal Positions (Cols 52)
CHARCTER(m)	character	A	m	n/a
GRAPHIC	graphic	G	m*2	n/a
DECIMAL(m,n)	packed decimal	P	m/2+1	n
SMALLINT	4-digit binary	B	2	0
INTEGER	9-digit binary	B	4	0
DATE	date	D	10	n/a
TIME	time	T	8	n/a
TIMESTAMP	timestamp	Z	26	n/a

Data mapping works in both directions, SQL to VARPG and VARPG to SQL. If a zoned decimal variable is used as host variable, VisualAge for RPG converts the content to packed decimal before and after the DB2 operation.

When the host variable type doesn't match the column definition, appropriate conversions take place. This is also true for the SQL data types REAL, DOUBLE and VARCHAR, which are not supported by VisualAge for RPG. You can still access then using a different host variable data type. The conversion will be made and you will receive a message indicating that truncation occurred.

When you are defining the host variables for the SQL statements in your VisualAge for RPG application, please keep in mind that the limits for the various data types may differ between SQL and VisualAge for RPG. For example, a host variable of the character type may have a maximum length of 254 bytes, while VisualAge for RPG allows specifying up to 32767 bytes. Refer to the documentation of your DB2 database to find these data type limits.

All variables within your source are considered to be used as host variables with the exception of multiple occurrence data structures, indicators (*INxx), tables, indexed arrays, or the reserved words UDATE, UDAY, UMONTH, and UYEAR. This differs from the SQL standard, which expects all host variables to be defined within the BEGIN DECLARE and END DECLARE SQL statements. You can still use these SQL statement to document your host variables. However, the VisualAge for RPG compiler will not indicate any host variable that has not been defined.

Single occurrence data structures with no subfields are treated like variables of normal data character type. Data structures with subfields are considered host structures. They can be used as a convenient way to specify a series of host variables. So, our first sample SELECT statement can be modified to use such a host structure as shown in Figure 33.

RPG

```
D EmplRow         DS
D   Number                       7S 0
D   FirstName                   20A
D   LastName                    20A
 *
C/EXEC SQL
C+  SELECT Number, FirstName, LastName
C+     INTO :EmplRow
C+     FROM EmpTable
C+     WHERE Number = 12
C/END-EXEC
```

Figure 33. Using Host Structure in SELECT Statement

Connecting to the Database

Before an SQL table or view can be accessed within your VisualAge for RPG application, you first need to connect to the database, where the table or view resides. This connection can be made explicitly through the SQL statement CONNECT TO or through an implicit connect.

Using the implicit method means that a connection to a database is established by the system as soon as your VisualAge for RPG application tries to execute its first SQL statement. This connection remains active until the application is terminated.

The name of the database, the user identification, and the password needed to establish the connection must be supplied through the environment variables DB2DBDFT, DB2USERID, and DB2PASSWORD. These can be set in your AUTOEXEC.BAT file or from a command prompt window:

Batch file

```
SET DB2DBDFT=VARPG
SET DB2USERID=STADE7
SET DB2PASSWORD=STADE7
```

Using the explicit connect method allows you to specify the database name, user ID, and password during runtime in the same way as host variables can be specified for this information on the CONNECT TO statement:

```
RPG
D Database                       10A   INZ('VARPG')
D Userid                         10A   INZ('STADE7')
D Password                       10A
 *
C                   movel     'STADE7'      Password
 *
C/EXEC SQL
C+ CONNECT TO :Database USER :Userid USING :Password
C/END-EXEC
```

Regardless of which method you use, the connection is always limited to the VisualAge for RPG application. So, another explicit CONNECT TO is performed in one of the application's components, it will either reset the current connection or cause an error message to be issued (if there are pending row locks and no COMMIT or ROLLBACK was performed before the second connect).

Sample SQL Application

Let's create a sample VisualAge for RPG application that accesses a database file using different SQL statements. We will modify the employee list example, that was already used in “AS/400 Files and Record I/O.”

For this example, a subfile part has to be filled with information for employees, where the lowest and highest employee number of the rows to be displayed can be specified (see Figure 34). To help the user choose an employee number range that makes sense, we add two read-only entry fields showing the lowest and highest employee number available in the table.

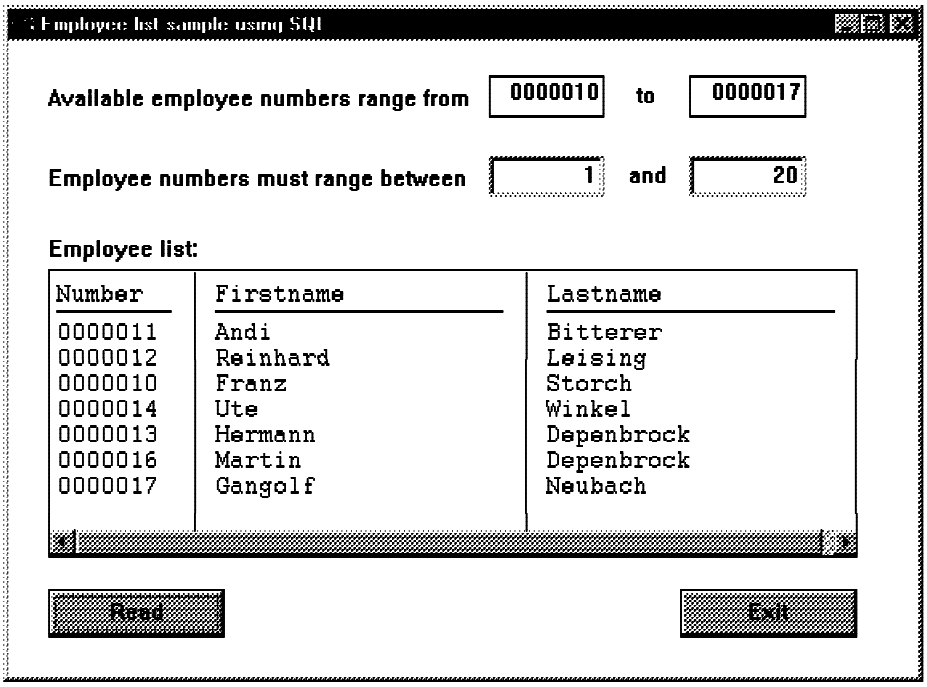


Figure 34. Employee List Example Using SQL

To keep the example simple, the accessed table *EmplTable* consists of only three columns. The following SQL statement was used to create it:

```
SQL
CREATE TABLE EmplTable (Number     DECIMAL(7,0),
                        Firstname  CHARACTER(20),
                        Lastname   CHARACTER(20))
```

Database Connection

The first thing to do is to establish a connection to our database. An explicit CONNECT TO statement could be placed into the initialization subroutine (*INZSR) of our VisualAge for RPG application. However, we want to get the necessary information from the user through a separate window (see Figure 35), so we start an already available component called *DBLOGON*, which does this for us.

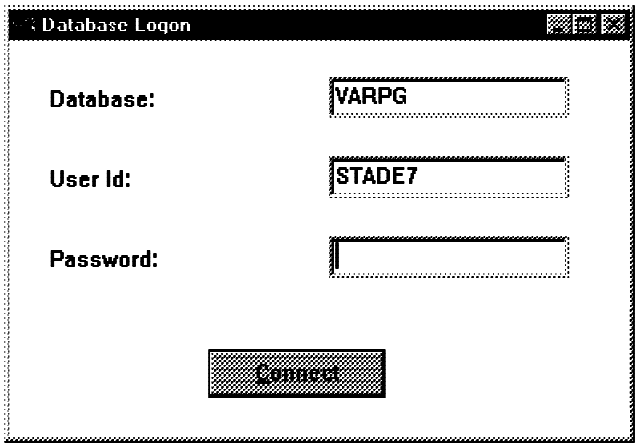


Figure 35. Database Logon Component

We don't look at this component in more detail, beyond noting that it ensures that all necessary information is specified by the user, updates the parameters with the entered information, and signals successful completion in an indicator variable (entry field part) called *DONE*. This is the code:

RPG

```
C     *INZSR        BEGSR
 *
C                   start     'DBLOGON'
C                   parm      'VARPG'       Database        10
C                   parm      'STADE7'      Userid          10
C                   parm                    Password        10
 *
C                   ENDSR
```

The indicator's CHANGE event can be monitored by our application using a component reference part (see Exercise 5, “Using the Component Reference Part” for further details about component reference parts).

When the NOTIFY event for the component reference part is raised, we know that the information necessary to connect to the database is available. Thus, we can now perform the CONNECT TO operation.

To verify that the CONNECT statement is completed successfully, we are checking the content of SQLCOD (SQL codes below zero indicate an error). Although we did not include the SQL communications area (SQLCA), its subfields are available within our application. This is because VisualAge for RPG always includes SQLCA (Figure 36).

RPG

```
DSqlExcpt         M                   MSGNBR(*MSG0001)
D                                     MSGDATA(SQLCOD)
  :
C     DBLOGON       BEGACT    NOTIFY        MAIN
 *
C/EXEC SQL
C+ CONNECT TO :Database USER :Userid USING :Password
C/END-EXEC
C                   if        SQLCOD < 0
C     SqlExcpt      dsply                   RC
C                   movel     *ON           *INLR
C                   else
 *
C/EXEC SQL
C+  SELECT MIN(Number), MAX(Number)
C+     INTO :Min, :Max
C+     FROM EmplTable
C/END-EXEC
C                   write     'MAIN'
C                   eval      %setatr('MAIN':'MAIN':'Visible') = 1
C                   endif
 *
C                   ENDACT
```

Figure 36. Checking the SQL Return Code

Before we make the *MAIN* window of our sample application visible, a SELECT INTO statement is executed on the employee table. The SQL functions MIN() and MAX() determine the lowest and highest employee number in the table and pass this information into the host variables *Min* and *Max*. As these are also the names of the corresponding entry field parts, this information can be updated through a WRITE to the *MAIN* window.

Database Cursor

We cannot use a simple SELECT INTO statement to fill the subfile. This is because more than just one row is likely to be returned (even though the specified range of numbers would result in only one row). As a result, we have to declare an SQL cursor in this case, which is basically nothing more than a temporary space, which holds the result of a SELECT statement and allows accessing the rows one at a time.

Let's next look at the action subroutine for the PRESS event of the *Read* push button to see how SQL cursor processing can be embedded into VisualAge for RPG (Figure 37).

RPG

```
D EmplRow         DS
D   Number                       7S 0
D   FirstName                   20A
D   LastName                    20A
  :
C     READ          BEGACT    PRESS         MAIN
 *
C                   clear                   EMPLLIST
C                   read      'MAIN'
 *
C/EXEC SQL
C+  WHENEVER SQLERROR GOTO SqlErrRead
C/END-EXEC
 *
C/EXEC SQL
C+  DECLARE c1 CURSOR FOR
C+             SELECT Number, FirstName, LastName
C+               FROM EmplTable
C+              WHERE Number BETWEEN :Low AND :High
C/END-EXEC
C/EXEC SQL
C+  OPEN c1
C/END-EXEC
C/EXEC SQL
C+  FETCH c1 INTO :EmplRow
C/END-EXEC
C                   dow       SQLCOD <> 100
C                   write     EMPLLIST
C/EXEC SQL
C+  FETCH c1 INTO :EmplRow
C/END-EXEC
C                   enddo
C/EXEC SQL
C+  CLOSE c1
C/END-EXEC
C                   goto      EndRead
 *
C     SqlErrRead    tag
C     SqlExcpt      dsply                   RC               9 0
C     EndRead       tag
 *
C                   ENDACT
```

Figure 37. Action Subroutine for the Press Event of the Read Push Button

Before we can use an SQL cursor, we need to declare it. This is done through the SQL statement DECLARE CURSOR, where a unique cursor name (*c1*) is assigned to a SELECT statement. The SELECT statement in our example provides all employee rows from the table called *EmplTable* with an employee number in the specified range. *Low* and *High* are the names of the corresponding entry field parts, where the user can specify the range.

When the SQL cursor is opened by executing the SQL OPEN statement, the SELECT statement is actually executed and the results are placed into the temporary space represented by the cursor.

The rows of the cursor can now be accessed through the SQL statement FETCH. In our example, we fetch the rows from the cursor sequentially and receive the content into a host structure called *EmplRow*. Instead, we could have also specified separate host variables for each column of the table (separated by commas).

After every FETCH statement, we are checking the content of SQLCOD. If the FETCH has been successful, the record is added to the subfile part called *EMPLLIST*. A SQL code of 100 signals that there was no row remaining in the cursor and nothing could be fetched. In this case, the cursor is closed through the SQL statement CLOSE.

SQL Exceptions

In the above example, we are using a particular method to handle SQL exceptions. The SQL statement WHENEVER SQLERROR defines a branch tag, which will get control as soon as one of the subsequent SQL statements results in a SQL code below zero. Once a WHENEVER SQLERROR has been coded in your program, the branch tag definition remains active until another WHENEVER SQLERROR statement with a different tag or with the clause CONTINUE is specified. See Figure 38.

RPG

```
C/EXEC SQL
C+  WHENEVER SQLERROR GOTO ...
C/END-EXEC
 *
C/EXEC SQL
C+  FETCH c1 INTO :EmplRow
C/END-EXEC
 *
C/EXEC SQL
C+  WHENEVER SQLERROR CONTINUE
C/END-EXEC
```

Figure 38. Example of a Branch Tag Definition

It is important to keep this in mind, as the branch tag must reside in the same (action or user) subroutine as the SQL statement that might raise an SQL exception. The piece of code shown in Figure 39 fails to compile and returns an error message RNF8530E.

RPG

```
C     READ          BEGACT    PRESS         MAIN
 *
C/EXEC SQL
C+  FETCH c1 INTO :EmplRow
C/END-EXEC
 *
C                   ENDACT
  :
C     *INZSR        BEGSR
 *
C/EXEC SQL
C+  WHENEVER SQLERROR GOTO SqlErr
C/END-EXEC
 *
C                   goto      EndInzSr
C     SqlErr        tag
C     SqlExcpt      dsply                   RC               9 0
C     EndInzSr      tag
 *
C                   ENDSR
```

Figure 39. Example of Branch Tag

Currently, the database connection will be active until the program ends. So, the user is able to press the *Read* push button multiple times, specifying different ranges of employee number. If you would like to end the connection explicitly (for example, because you want to connect to a different database), you can use the SQL statement DISCONNECT. We are using it in the action subroutine for the PRESS event of the *Exit* push button in our example to clean up things:

RPG

```
C     EXIT          BEGACT    PRESS         MAIN
 *
C/EXEC SQL
C+  DISCONNECT :Database
C/END-EXEC
 *
C                   movel     *ON           *INLR
 *
C                   ENDACT
```

Sorting Records

There is one problem left: The list of employees is not sorted. When we used record I/O processing to fill the subfile, we were reading from a keyed physical file, so that the list was always sorted by employee numbers. To achieve the same thing within SQL, we can add an ORDER BY clause to our SELECT statement:

RPG

```
C/EXEC SQL
C+  DECLARE c1 CURSOR FOR
C+             SELECT Number, FirstName, LastName
C+               FROM EmplTable
C+              WHERE Number BETWEEN :Low AND :High
C+              ORDER BY Number
C/END-EXEC
```

This would work perfectly. But wouldn't it be nice to have the user decide whether to sort the list by number, first name, or last name? All we would need to do is add another column name specified for the ORDER BY. But as this is a column name, not just a value, we can't use it as a host variable.

Dynamic SQL

One way of providing user choice in sorting would be to code three different DECLARE CURSOR statements for each sorting choice. However, if we use dynamic SQL instead, then we can assemble the SQL statement during runtime. Figure 40 shows the modified action subroutine for the PRESS event of the *Read* push button:

RPG

```
D DclCsr          C                   'SELECT Number, +
D                                            Firstname, Lastname +
D                                       FROM EmplTable +
D                                      WHERE Number BETWEEN ? AND ? +
D                                      ORDER BY '
D DclCsrC1        S            254A
  :
C     READ          BEGACT    PRESS         MAIN
 *
C                   clear                   EMPLLIST
C                   read      'MAIN'
C                   select
C                   when      %getatr('MAIN':'ORDNBR':'Checked') = 1
C                   movel     'Number    '  OrderBy          10
C                   when      %getatr('MAIN':'ORDFIRST':'Checked') = 1
C                   movel     'Firstname '  OrderBy
C                   when      %getatr('MAIN':'ORDLAST':'Checked') = 1
C                   movel     'Lastname  '  OrderBy
C                   endsl
 *
C/EXEC SQL
C+  WHENEVER SQLERROR GOTO SqlErrRead
C/END-EXEC
 *
C                   movel     *BLANKS       DclCsrC1
C                   eval      DclCsrC1 = DclCsr + OrderBy
 *
C/EXEC SQL
C+  PREPARE DeclareCursor FROM :DclCsrC1
C/END-EXEC
 *
C/EXEC SQL
C+  DECLARE c1 CURSOR FOR DeclareCursor
C/END-EXEC
```

Figure 40 (Part 1 of 2). Modified Action Subroutine for the Press Event

RPG (continued)

```
C/EXEC SQL
C+  OPEN c1 USING :Low, :High
C/END-EXEC
C/EXEC SQL
C+  FETCH c1 INTO :EmplRow
C/END-EXEC
C                   dow       SQLCOD <> 100
C                   write     EMPLLIST
C/EXEC SQL
C+  FETCH c1 INTO :EmplRow
C/END-EXEC
C                   enddo
C/EXEC SQL
C+  CLOSE c1
C/END-EXEC
C                   goto      EndRead
 *
C     SqlErrRead    tag
C     SqlExcpt      dsply                   RC               9 0
 *
C     EndRead       tag
 *
C                   ENDACT
```

Figure 40 (Part 2 of 2). Modified Action Subroutine for the Press Event

The DECLARE CURSOR statement no longer contains the SELECT statement. Instead, an SQL statement named *DeclareCursor* must be specified. This SQL statement name represents the SQL statement prepared before, using the SQL statement PREPARE. This PREPARE routine expects a character host variable (*DclCsrC1*) that contains the SELECT statement to be prepared.

In our example, we have the user choose between the different sorting options by selecting one of three radio buttons *ORDNBR*, *ORDFIRST* and *ORDLAST*. Depending on which one is selected, we assemble the host variable for the SELECT. Figure 41 shows the list sorted by employee last name.

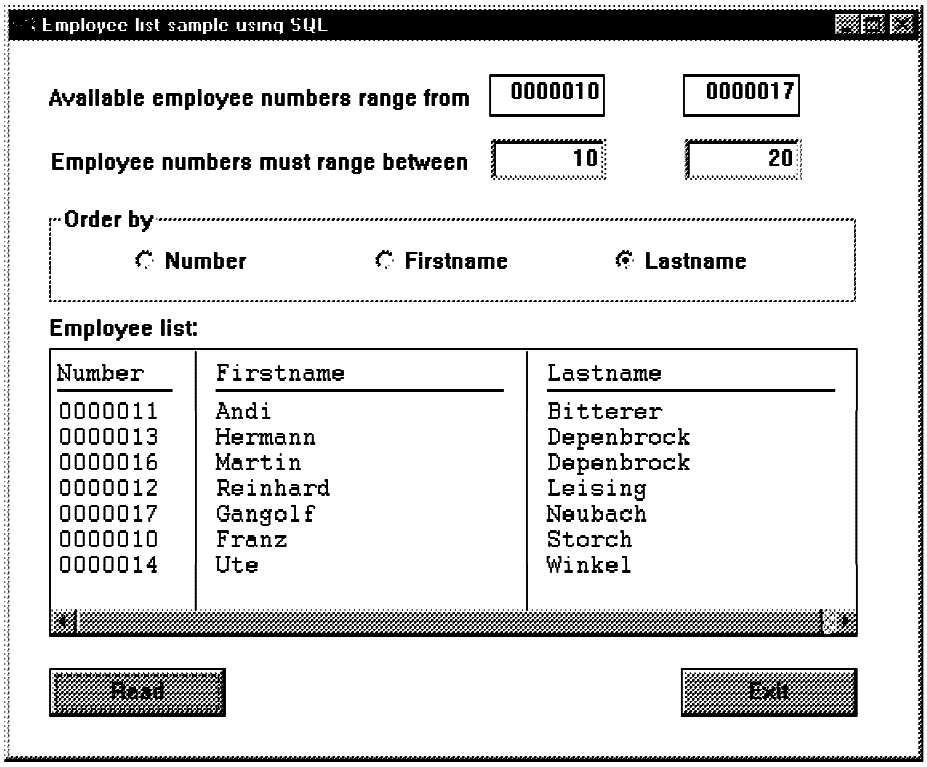


Figure 41. Employee List Sorted by Last Name

Have you noticed that the host variables *Low* and *High* are no longer part of the character variable that is used for the PREPARE portion of the SELECT statement? This is because host variables are not allowed in a SQL statement to be prepared in dynamic SQL. You can, however, use parameter markers that are represented by question marks (?). These parameter markers are replaced during runtime by the content of those host variables specified on the USING clause of the OPEN statement. The sequence in which you specify these host variables must correspond to the (relative) position of the parameter markers.

Build Process and Options

Before VisualAge for RPG is able to compile a component that contains embedded SQL statements, you need to set up the build options properly. There are two pages on the *Build options* notebook, that allow you to do this.

Database Name

The most important information is the *DB2 Database Name* entry on the *DB2* notebook page (see Figure 42), as this will be the DB2 database from which the VisualAge for RPG compiler will retrieve the necessary information about the tables and views accessed.

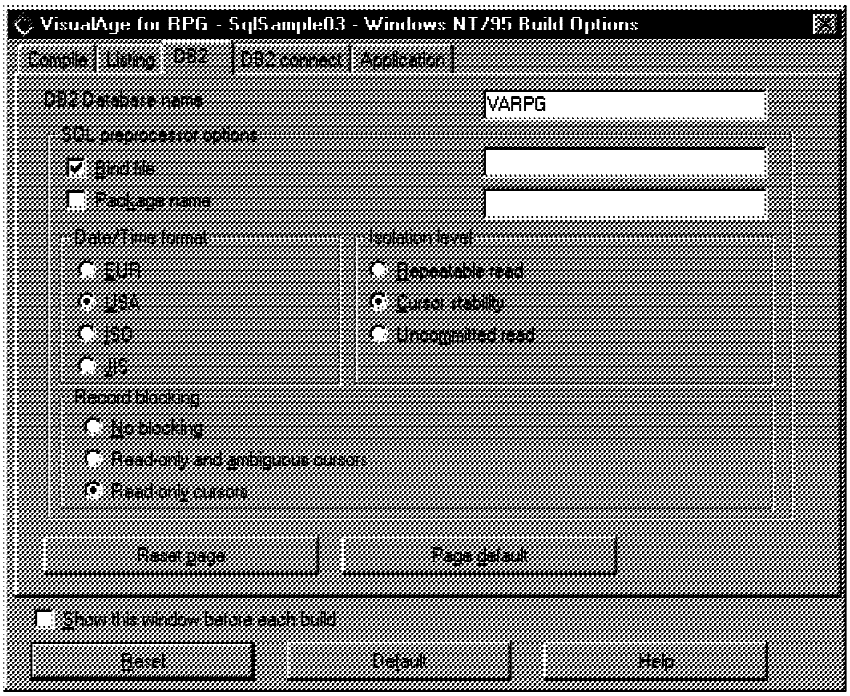


Figure 42. Build Options Window, DB2 Page

During the build, VisualAge for RPG automatically starts the database manager for this DB2 database and establishes an SQL connection to it, using the user ID and password supplied on the *DB2 connect* notebook page (see Figure 43).

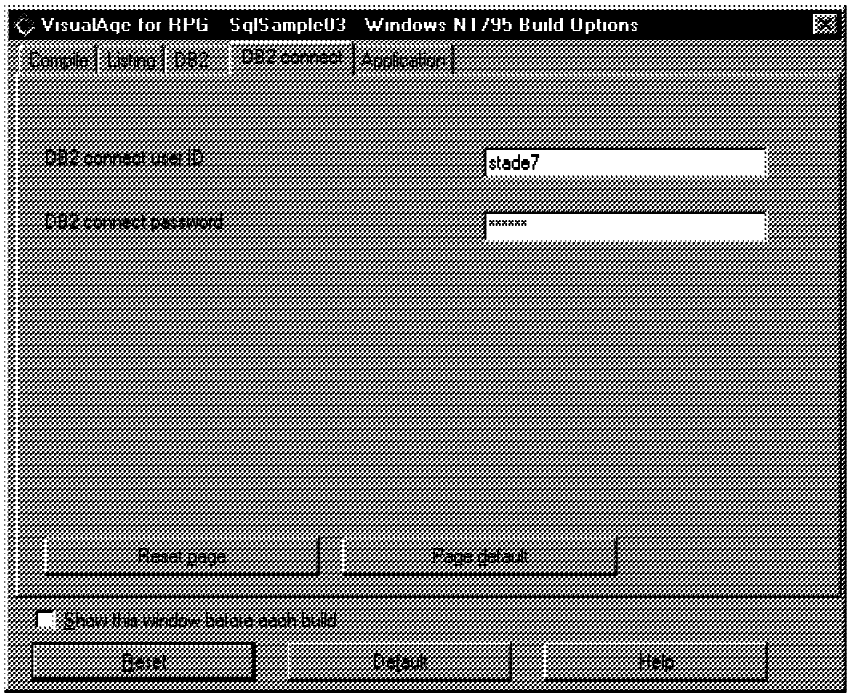


Figure 43. Build Options Window, DB2 Connect Page

Binding to the Database

Providing the information on the Build Options notebook is enough to have VisualAge for RPG compile a component containing embedded SQL statements. During this build process, a .BND file is created containing all the information regarding the different SQL statements as well as the tables and views used in the application.

By default, the .BND file has the same name as all the other files of your application and is placed together with them into the application's runtime directory (RT_WIN32 or RT_WIN). To change its name, type the new name into the entry field beside the *Bind file* check box. This check box is checked by default; if you deselect it, the .BND file is no longer created.

This .BND file is needed to bind the VisualAge for RPG application to the DB2 database that will be used during runtime. An SQL package is created for this application in the database, allowing it to connect to the database and execute its SQL statements.

To have this bind performed automatically as part of the build process, you must select the *Package name* check box on the *DB2 Build Options*

notebook page. You can also invoke the bind process manually by calling the BIND command through the DB2 command line processor. This is very useful if your application needs to be set up for a database that you don't have access to.

SQL CLP

```
I:\>DB2CMD

DB2CLP I:\>DB2 CONNECT TO varpg

   Database Connection Information

 Database product       = DB2/Windows 95 2.1.2
 SQL authorization ID   = STADE7
 Local database alias   = VARPG


DB2CLP I:\>DB2 BIND SqlSmpl3.bnd COLLECTION varpg

LINE    MESSAGES FOR SqlSmpl3.bnd
------  ------------------------------------------------------------
        SQL0061W  The binder is in progress.
        SQL0091N  Binding was ended with "0" errors and "0" warnings.

DB2CLP I:\>DB2 LIST PACKAGES

NAME     CREATOR  BOUNDBY  TOTALSECT VALID FORMAT ISOLATION BLOCK
-------- -------- -------- --------- ----- ------ --------- -----
SQLSAMPL STADE7   STADE7           1 Y     1      CS        U
SQLSMPL2 STADE7   STADE7           2 Y     1      RR        N
SQLSMPL3 STADE7   STADE7           2 Y     1      RR        N
SQLSMPL4 STADE7   STADE7           3 Y     1      RR        U

  4 record(s) selected.

DB2CLP I:\>
```

Figure 44. Binding to Database

The LIST PACKAGES command can be used to verify that the package has been added to the database successfully.

Also included in the package created during bind are the other options available on the *DB2* page of the *Build Options* notebook: *Date/Time format*, *Isolation Level*, and *Record blocking*.

The *Date/Time format* option is used if you are receiving the content of date or time columns into character host variables. Table 4 illustrates the different formats.

Table 4. Date and Time Formats

Date/Time Format Option	Description	Date Format	Time Format
EUR	European Standard	dd.mm.yyyy	hh.mm.ss
USA	U.S. Standard	mm/dd/yyyy	hh.mm AM (or PM)
ISO	International Standard Organization	yyyy-mm-dd	hh.mm.ss
JIS	Japanese Industrial Standard	yyyy-mm-dd	hh.mm.ss

Record Blocking

Record blocking becomes important, when a SELECT statement returns multiple rows. The application must declare a cursor and use the FETCH statement to retrieve the rows one at a time. With a remote database, this means that each request and each reply must travel across the network. With a large number of rows, this leads to a significant increase in network traffic.

When you specify record blocking and use a read-only cursor, the Database Manager at the database server returns a block of rows to the database client in one network transmission. These rows are retrieved one at a time from the database client when Database Manager processes a FETCH request. When all rows in the block have been fetched, Database Manager at the database client sends another request to the remote database, until all output rows have been retrieved.

Record blocking can lead to results that are not entirely consistent with the database when used in combination with the cursor stability or uncommitted-read isolation levels (see below).

The following list shows you the available record blocking options and their effects:

No blocking

No record blocking occurs for any cursor.

Read-only and ambiguous cursors

Record blocking is performed for read-only and ambiguous cursors. An ambiguous cursor is a dynamic cursor that does not include either the FOR READ ONLY clause or the FOR UPDATE clause. A cursor is considered read-only, if one of the following is true:

- The cursor is based on a read-only SELECT statement such as join or union SELECTs.
- The cursor is declared with a DISTINCT, ORDER BY, GROUP BY, or FOR FETCH ONLY clause.

Read-only cursors

For all read-only cursors, record blocking is performed.

Through the *Isolation level* setting you can control which rows of a cursor will be locked until the next SQL COMMIT or ROLLBACK statement is performed:

Repeatable read

All rows accessed by the application will remain locked until the next commit or rollback. Using this isolation level, you can make sure that a record read by your application is not updated by another application.

Cursor stability

Only the last fetched row is locked. As soon as the next row is fetched, the lock is released. However, if an update was done on the row, it remains locked until the next commit or rollback.

Uncommitted read

No lock is placed on a row that is read by a FETCH or SELECT INTO statement. This should be your preferred choice if you only read the rows of a cursor.

The locking of rows affects only different VisualAge for RPG applications, not different components of the same application.

Data Area

In addition to file access through record I/O and SQL, VisualAge for RPG is also able to access AS/400 data areas. These are often used on the AS/400 to exchange information between different programs running in the same or different AS/400 jobs.

Defining a Data Area

To use data areas within a VisualAge for RPG application, you need to define them within the definition specifications or using the DEFINE operation code. In the D-specifications the keyword DTAARA can be specified for a stand-alone field, a data structure, or a data structure subfield. To define a data area through the DEFINE operation code, the reserved word *DTAARA must be specified in its factor 1, while the data area's name is coded in the result field:

```
RPG
D MyDtaAra        S             20A   DTAARA
 *
D MyDs            DS                  DTAARA
D   Part01                      10A
D   Part02                      10A
 *
D MyDta           S             20A
 *
 *
C     *DTAARA       define                MyDta
```

By default, VisualAge for RPG runtime looks for a data area in the library list on the AS/400 with the name specified in columns 7-21 on the D-spec or in the result field of the DEFINE opcode. To override this name, you can specify an external name through the optional parameter of the DTAARA keyword or a factor 2 of the DEFINE opcode.

```
RPG
D MyDtaAra        S             20A   DTAARA(MyDtaAra4)
 *
D MyDta           S             20A
 *
 *
C     *DTAARA       define    MyDtaAs4      MyDta
```

Another method to override the name specified within the VisualAge for RPG source is through an entry on the *Data areas* page of the *Define AS/400 Information* notebook (Figure 45). Please note, if an external name has been specified in your program, the name specified as *Data area alias* must be equal to this name.

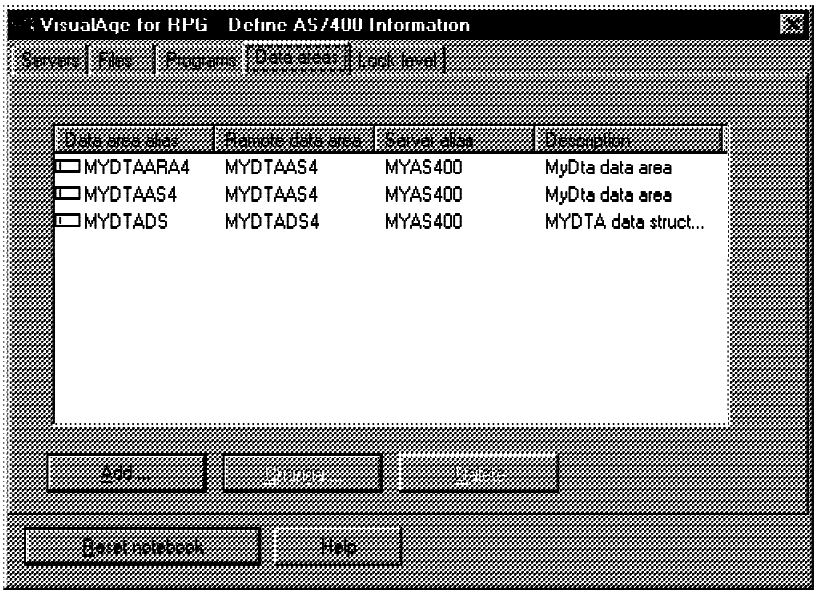


Figure 45. Define AS/400 Information Window, Data Areas Page

Reading and Writing Data Areas

To read the content of a data area, you can use the IN operation code. It retrieves the content of the data area into the defined field or subfields of the defined data structure. If the reserved word *LOCK is specified, an exclusive-read (*EXCLRD) lock is placed on the data area. Other programs are thus allowed to read the content of the data area without being able to place their own lock.

An IN operation with *LOCK in factor 1 is the prerequisite for executing an OUT operation, which writes the content of the defined field or the subfields of the defined data structure back to the data area. Usually, an OUT operation also releases the lock on the data area. If you would like to keep the lock, *LOCK must be specified in factor 1:

RPG

```
C     *LOCK         in        MyDta
 *
C                   in        MyDtaDs
 *
C                   out       MyDta
```

On both operations, the internal name of the data area must be specified in factor 2. Instead, you can use the special word *DTAARA, causing the operation to be performed on all available data areas. But you then need to make sure that all data areas are in the appropriate state. For example, the OUT operation in the following sample will fail as data area *MyDtaDs* has not been locked when read.

```
RPG
C     *LOCK         in        MyDta
 *
C                   in        MyDtaDs
 *
C                   out       *DTAARA
```

Instead of executing an OUT operation with a blank factor 1, you can also use the UNLOCK operation code to release the lock for a data area.

Data Area Data Structure

If you specify a U in column 23 of a data structure definition for a data area, it becomes a data area data structure. This is a special type of data area definition, for which an implicit IN (*LOCK) is executed during program initialization (before *INZSR is executed). When the program terminates, an implicit OUT operation is performed. This happens even if the lock on this data area was released by an UNLOCK:

```
RPG
D MyDtaDs        UDS                 DTAARA(DtaAraAs4)
D   DtaDsFld                   20A
```

If you have defined such a data area data structure, and VisualAge for RPG runtime cannot find the corresponding data area on the AS/400 during startup, it creates it for you in library QTEMP. For any ordinary data area, an exception is raised in this case, which can be handled in the *PSSR subroutine. The status field in the program status data structure will contain one of the error codes listed in Table 5.

Table 5 (Page 1 of 2). Error Codes for Data Area Access

Error Code	Condition
00401	Data area specified on IN/OUT not found
00411	Data area type or length does not match
00412	Data area not locked for output

Table 5 (Page 2 of 2). Error Codes for Data Area Access	
Error Code	**Condition**
00413	Error on IN/OUT operation
00414	User not authorized to use data area
00415	User not authorized to change data area
00421	Error on UNLOCK operation
00431	Data area previously locked by another program
00432	Data area locked by program in the same process

Chapter 4. Programs, Procedures, and Functions

In this chapter, we show the abilities of VisualAge for RPG to invoke and communicate with local and remote programs, VisualAge for RPG procedures, and functions from a foreign dynamic link library. We explain what the different operation codes, such as CALL, CALLP, CALLB, and START do, how parameters are being passed, and how return codes can be retrieved.

AS/400 Programs and Commands

While working in a client/server environment, it may sometimes be necessary to execute programs or commands on your AS/400 servers. For example, you may need to change the library list, do overrides to your database files, use AS/400 objects that can't be accessed by any RPG operation code directly, or use other functionality of the AS/400.

For example, it could be necessary to place certain information into a spooled file on the AS/400. Unfortunately, we can't access an AS/400 printer file from our VisualAge for RPG application. Thus, we need to invoke an AS/400 program that receives the information to be printed and does the spooling for us.

The CALL Opcode

The VisualAge for RPG interface used to invoke an AS/400 program is the CALL operation code. It always calls the program represented by the name in factor 2:

```
RPG
D PrtToAS400      S             20A   INZ('PRTCUST')
D                                     LINKAGE(*SERVER)
 *
C                   call      PrtToAS400                    60
```

This name must be defined in the D-specs either as a constant or as a field initialized with the program name. Optionally, you can specify the library name as well.

```
RPG
D PrtToAS400      C                   CONST('MYLIB/PRTCUST')
D                                     LINKAGE(*SERVER)
 *
C                   call      PrtToAS400                    60
```

Through the keyword LINKAGE(*SERVER), the VisualAge for RPG runtime knows that the program to be called resides on an AS/400 system. If nothing else is specified in your application, the runtime will look for an AS/400 program object with the same name as the CONST (or INZ) value of the D-spec in the specified library. If only a program name is supplied, the library list of the user that signed on to the applications default server is searched.

If an error message tells you that a default server could not be found for your application, you most probably didn't define one through the *Define AS/400 Information* dialog of the GUI Designer. This dialog is also the place where you should specify the location of your AS/400 program, instead of hard-coding the program name and library information into the program itself. On its *Program* notebook page (Figure 46), you can maintain all AS/400 programs called by your application.

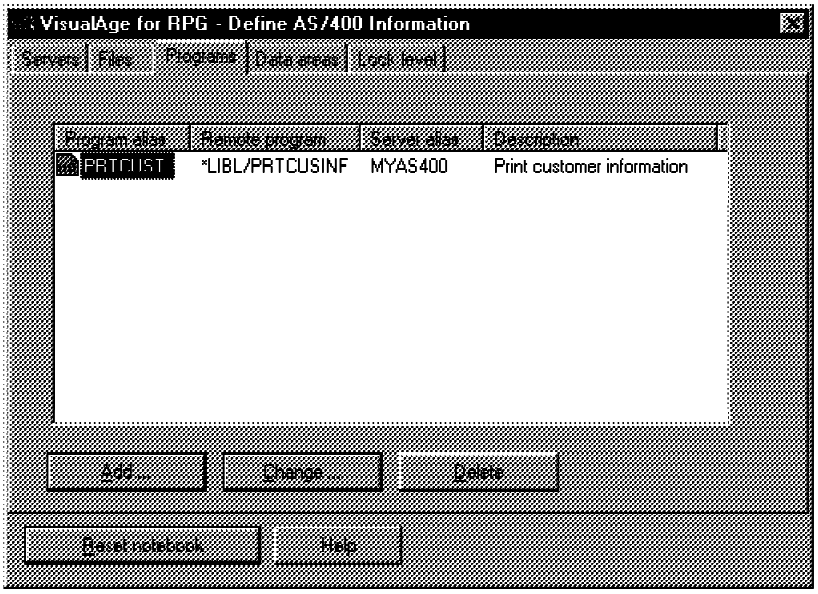


Figure 46. Define AS/400 Information Window, Program Page

The link defined here is stored, together with all other AS/400 related information, in the .RST file of your project. You can easily change it later, when you ship the application and run it in another environment. The definition is:

RST File

```
   :
DEFINE_PROGRAM  PROGRAM_ALIAS_NAME(PRTCUST)
                REMOTE_PROGRAM_NAME(*LIBL/PRTCUSINF)
                SERVER_ALIAS_NAME(MYAS400)
                TEXT(Print customer information)
   :
```

This information is now used by the VisualAge for RPG runtime to locate the program object, if the variable or constant for the program name has been initialized without library information. The specified value is then treated as an alias name that resolves through the appropriate entry in the .RST file.

While the call to PrtToAS4_1 invokes program PRTCUSINF from the library list, the second call to PrtToAS4_2 searches for program PRTCUST in library MYLIB:

RPG

```
D PrtToAS4_1      C                   CONST('PRTCUST')
D                                     LINKAGE(*SERVER)
 *
D PrtToAS4_2      C                   CONST('MYLIB/PRTCUST')
D                                     LINKAGE(*SERVER)
 *
C                   call      PrtToAS4_1                          60
 *
C                   call      PrtToAS4_2                          60
```

Parameters

If you need to pass parameters, you can do so by coding a PLIST name into the result field of the CALL opcode or through PARM opcodes immediately following the CALL:

```
RPG
D PrtToAS400      S             20A   INZ('PRTCUST')
D                                     LINKAGE(*SERVER)
 *
D CustNbr         S              7A
D CustName        S             40A
D CustCity        S             30A
 *
C                   call      PrtToAS400                           60
C                   parm                    CustNbr
C                   parm                    CustName
C                   parm                    CustCity
```

Parameters are always passed by reference; that is, you have direct access to their changed values as soon as control returns back to your VisualAge for RPG application.

As no prototyping is possible, you need to make sure that the attributes of the arguments correspond to the parameter definition of the called AS/400 program. Otherwise, you may experience unpredictable results, depending on what is overlaid by the content of an incorrectly passed parameter.

Exceptions

If an error occurs during the call, a resulting indicator in columns 73 and 74 is set on and the program status data structure is updated. The *STATUS field contains a return code of 00202 or 00211, while the exception type in positions 40-42 is set to *RT.

To get more detailed information about errors that can occur during execution of the AS/400 program, we suggest you implement an additional parameter for your calls. This could be a data structure containing, for example, the exception ID and exception data information from the program status data structure of the called program.

```
RPG
DErrorCode        DS
D ExcptID                        7A   INZ(*BLANKS)
D*                                            Exception Id
D ExcptData                    100A   INZ(*BLANKS)
D*                                            Exception data
```

The AS/400 program is then responsible for monitoring the exceptions and filling the data structure properly. In ILE RPG/400, this could look like the example in Figure 47.

RPG

```
H
FPRTCUSINFFO    E             PRINTER INFSR(*PSSR)
F                                     USROPN
 *
D PSDS           SDS
D  EXCP_TYPE              40     42
D  EXCP_NUM               43     46
D  EXCP_DATA              91    170
 *
D/COPY REINHARD/QRPGLESRC,ErrorCode
 *
C     *entry        plist
C                   parm                    CustNbr
C                   parm                    CustName
C                   parm                    CustCity
C                   parm                    ErrorCode
 *
C                   open      PRTCUSINFF
C                   write     PRTCUSTR
C                   close     PRTCUSINFF
 *
C                   movel     *ON           *INLR
 *
C     *PSSR         begsr
C                   eval      ExcptID   = EXCP_TYPE + EXCP_NUM
C                   eval      ExcptData = EXCP_DATA
C                   movel     *ON           *INLR
C                   endsr     '*DETL'
```

Figure 47. Monitoring Exceptions

Back in your VisualAge for RPG application, you can use this additional information, for example, to react to an error in much more detail. Or, you can display a message containing the exception ID and data for unexpected errors, so that you get more information if an error occurs (Figure 48).

```
RPG
D/COPY *REMOTE REINHARD/QRPGLESRC,ErrorCode
 *
D PrtError        M                   MSGNBR(*MSG0002)
D                                     MSGDATA(ExcptID:ExcptData)
 *
C                   call      PrtToAS400                           60
C                   parm                    CustNbr
C                   parm                    CustName
C                   parm                    CustCity
C                   parm                    ErrorCode
 *
C                   if        Excpt_ID <> *BLANKS
C     PrtError      dsply                   RC
C                   endif
```

Figure 48. Displaying Message with Exception ID and Data

We are using the /COPY *REMOTE compiler directive to include the *ErrorCode* data structure from the AS/400 source member used in the ILE RPG/400 program. Thus, we don't have to maintain two versions of the same include file and we also make sure that this parameter has the same attributes in both programs.

Asynchronous Call

Usually, a call to an AS/400 program is done synchronously, so the VisualAge for RPG application on the PC is waiting for the AS/400 program to complete before it executes the statements following the CALL. However, it is also possible to call an AS/400 program and have the VisualAge for RPG application resume immediately. The only thing to do is to add the NOWAIT keyword to the definition of the program name variable:

```
RPG
D PrtToAS400      S             20A   INZ('PRTCUST')
D                                     LINKAGE(*SERVER)
D                                     NOWAIT
 *
C                   call      PrtToAS400                           60
C                   parm                    CustNbr
C                   parm                    CustName
C                   parm                    CustCity
C                   parm                    ErrorCode
```

Now, the program will be submitted without waiting for its completion. The error indicator in columns 73 and 74 is set to *ON only if the submit itself went wrong. An error during execution of the AS/400 program will not be indicated to the calling VisualAge for RPG application.

Parameters

When using asynchronous calls, parameters are passed by value. As a result, you do not get any feedback about the results of the AS/400 program's execution. You keep the old parameter values, even if they were changed by the called program.

But suppose we do not want to wait for the completion of the AS/400 program, but still need the feedback from it. In our Spooling example, it might take some time to have a complex spooled output finished. It does not make sense to have the user on the PC wait for the completion. The answer is to use a data queue.

Data Queue

We can use the same data queues that we would use in a similar asynchronous environment on the AS/400. As we are able to invoke AS/400 programs, we can call the data queue APIs. The data queue is created with a length of 114 bytes:

- The first 7 bytes are for the customer number, so we know to which record each entry belongs.
- The next 7 bytes contain the message ID, if an error occurred.
- The remaining 100 bytes are for the exception data (the message text).

Instead of setting the *ErrorCode* parameter, the called AS/400 program adds an entry to the data queue object for every entry that is spooled (see Figure 49).

RPG

```
H
FPRTCUSINFFO    E             PRINTER INFSR(*PSSR)
F                                     USROPN
 *
D PSDS           SDS
D  EXCP_TYPE              40     42
D  EXCP_NUM               43     46
D  EXCP_DATA              91    170
 *
C     *entry        plist
C                   parm                    CustNbr
C                   parm                    CustName
C                   parm                    CustCity
 *
C                   open      PRTCUSINFF
C                   write     PRTCUSTR
C                   close     PRTCUSINFF
 *
C                   movel     CustNbr       DtaQEntry
C                   exsr      SndDtaQ
C                   movel     *ON           *INLR
 *
C     *PSSR         begsr
C                   eval      DtaQEntry = CustNbr
C                                       + EXCP_TYPE + EXCP_NUM
C                                       + EXCP_DATA
C                   exsr      SndDtaQ
C                   movel     *ON           *INLR
C                   endsr     '*DETL'
 *
C     SndDtaQ       begsr
C                   call      'QSNDDTAQ'
C                   parm      'PRTCUSINFQ'  DtaQName         10
C                   parm      '*LIBL'       DtaQLib          10
C                   parm      114           DtaQELen          5 0
C                   parm                    DtaQEntry       114
C                   endsr
```

Figure 49. Using Data Queues

When we call our VisualAge for RPG application, the invocation of the AS/400 program changes so that we no longer need to pass the *ErrorCode* parameter. Because it would be passed by value, it would not return any error information.

To get feedback from the data queue, we add a timer part, which sends a TICK event every 10 seconds (Interval attribute is set to 10000, Multiplier defaults to 1). On every TICK event, we have the application check for an entry from the data queue object on the AS/400, as shown in Figure 50.

RPG

```
D PrtError        M                  MSGNBR(*MSG0002)
D                                   MSGDATA(RcdNbr
D                                          :ErrID
D                                          :ErrMsg)
D PrtOK           M                  MSGNBR(*MSG0003)
D                                   MSGDATA(RcdNbr)
 *
 *
 *  TICK event of the new Timer part
 *
C     CHKDTAQ       BEGACT    TICK          PRTMON
 *
C                   call      RcvDtaQ
C                   parm                    DtaQName
C                   parm                    DtaQLib
C                   parm                    DtaQELen
C                   parm                    DtaQEntry
C                   parm                    Wait
 *
C                   select
C                   when      DtaQELen > 0
C                              and %subst(DtaQEntry:15:100) <> *BLANKS
C                   eval      RcdNbr = %subst(DtaQEntry:1:7)
C                   eval      ErrID  = %subst(DtaQEntry:8:7)
C                   eval      ErrMsg = %subst(DtaQEntry:15:100)
C     PrtError      dsply                   RC
C                   when      DtaQELen > 0
C                              and %subst(DtaQEntry:15:100) = *BLANKS
C     PrtOK         dsply                   RC
C                   endsl
 *
C                   ENDACT
```

Figure 50. Using the Tick Event for Monitoring a Data Queue

If there is an entry, we check whether or not an exception occurred. If there was an exception, a message is issued showing the employee number of the record processed when the exception was raised, as well as the exception ID and the message text. If everything went fine, an appropriate completion message is issued for every processed record.

AS/400 Commands

Using the QCMDEXC or QCMDDDM APIs makes it possible to execute commands on an AS/400. While QCMDEXC is run in the AS/400 job that performs all operations on programs and data areas, QCMDDDM can be used to execute commands in the DDM job responsible for the file I/O. This is very helpful, if you need to do things like database overrides or library list changes. But you can, for example, also provide an AS/400 command line within your application through this interface (see Figure 51).

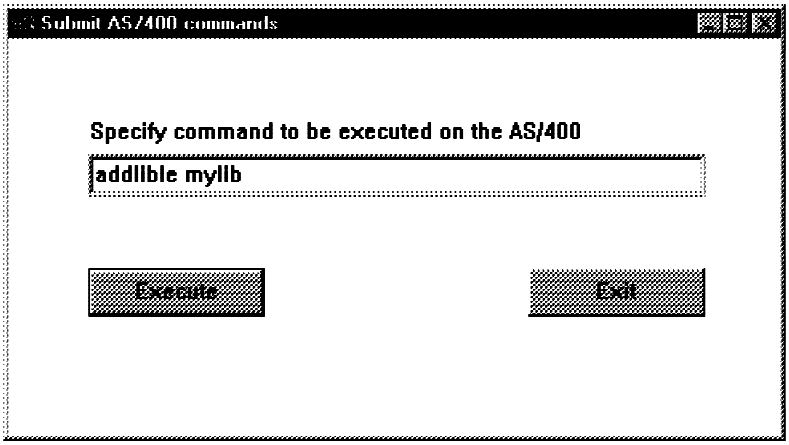


Figure 51. Submit AS/400 Commands Window

The PRESS event for the *Execute* push button reads the command entered into the entry field part and calls QCMDEXC to execute it on the AS/400, as shown in Figure 52.

RPG

```
D QCmdExc         S             10A   INZ('QCMDEXC')
D                                     LINKAGE(*SERVER)
D CmdLen          S             15P 5 INZ(255)
 *
C     EXECUTE       BEGACT    PRESS         FRA0000B
 *
C                   read      'FRA0000B'
 *
C                   call      QCmdExc                              60
C                   parm                    Cmd
C                   parm                    CmdLen
 *
C                   if        *IN60 = *OFF
C     *MSG0001      dsply                   RC                9 0
C                   else
C     CmdError      dsply                   RC
C                   endif
 *
C                   ENDACT
```

Figure 52. Executing a Remote Command

Handling AS/400 Exceptions

There is a problem if an exception occurs, however. The messages from the AS/400 are not accessible from VisualAge for RPG. The program status data structure indicates only a status code 00202, "Called program or procedure failed." which doesn't tell the user what really went wrong.

It may be better to call a small CL program on the AS/400 instead of QCMDEXC directly. This CL program will, in turn, call QCMDEXC, but it will also monitor for any exception that occurs. In that case, the program receives the message ID and text of the last received diagnostic or escape message and provides this information back to your application as two additional parameters, as shown in Figure 53

CL

```
PGM        PARM(&CMD &CMDLEN &MSGID &MSGTEXT)
DCL        VAR(&CMD)     TYPE(*CHAR) LEN(255)
DCL        VAR(&CMDLEN)  TYPE(*DEC)  LEN(15 5)
DCL        VAR(&MSGID)   TYPE(*CHAR) LEN(7)
DCL        VAR(&MSGTEXT) TYPE(*CHAR) LEN(132)
DCL        VAR(&MSG)     TYPE(*CHAR) LEN(1000)

CHGVAR     VAR(&MSGID)   VALUE(' ')
CHGVAR     VAR(&MSGTEXT) VALUE(' ')

CALL       PGM(QCMDEXC) PARM(&CMD &CMDLEN)

MONMSG     MSGID(CPF0000 CPD0000 MCH0000) EXEC(DO)
   RCVMSG     MSGQ(*PGMQ) MSGTYPE(*DIAG) RMV(*NO) +
              MSG(&MSG) MSGID(&MSGID)
   IF         COND(&MSGID *EQ ' ') THEN( +
      RCVMSG     MSGQ(*PGMQ) MSGTYPE(*EXCP) +
                 RMV(*NO) MSG(&MSG) MSGID(&MSGID))
   CHGVAR     VAR(&MSGTEXT) VALUE(%SST(&MSG 1 132))
ENDDO

ENDPGM
```

Figure 53. Monitoring Exceptions in a CL Program

In your VisualAge for RPG application, only the program name in the D-spec needs to be changed and the two new parameters must be added to the CALL. The error indicator on the CALL statement is no longer necessary, as the called CL program handles all exceptions itself. To determine whether an error occurred, we are now looking for the content of the MsgID field, as shown in Figure 54.

You can then issue messages on the workstation with detailed information about the error that occurred on the AS/400 (see Figure 55).

RPG

```
D QCmdExc         S             10A   INZ('CMDEXEC')
D                                     LINKAGE(*SERVER)
D CmdLen          S             15P 5 INZ(255)
 *
C     EXECUTE       BEGACT    PRESS         FRA0000B
 *
C                   read      'FRA0000B'
 *
C                   call      QCmdExc
C                   parm                    Cmd
C                   parm                    CmdLen
C                   parm                    MsgID             7
C                   parm                    MsgText         132
 *
C                   if        MsgID = *BLANKS
C     *MSG0001      dsply                   RC                9 0
C                   else
C     CmdError      dsply                   RC
C                   endif
 *
C                   ENDACT
```

Figure 54. Executing a Remote Command Through CL Program

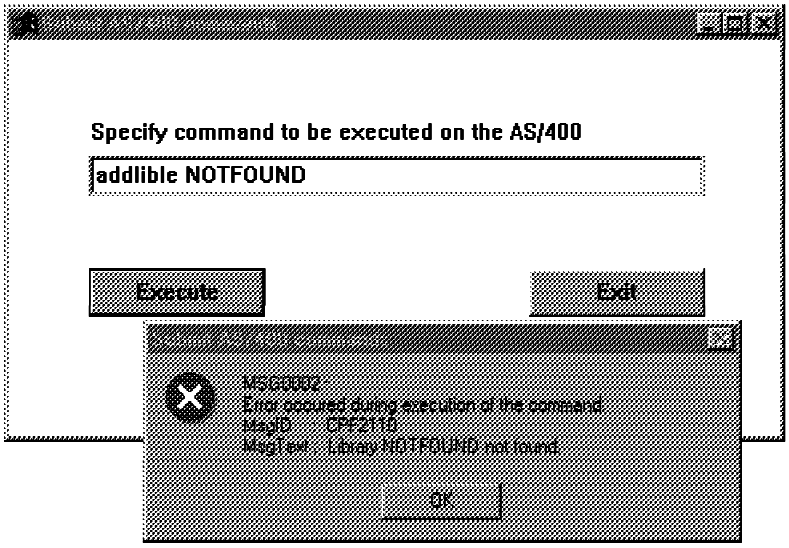


Figure 55. Error Details

Local Programs

From VisualAge for RPG, it is also possible to call programs, that reside on the local PC like .EXE, .BAT, and .COM files. You may, for example, want to call another application or tool from your VisualAge for RPG application, or run system functions of your hosting operating system like copying files from one directory to another.

VisualAge for RPG offers two different operation codes to invoke a local program.

CALLP Operation

The CALLP operation executes the local program synchronously. Thus, your VisualAge for RPG program waits for the called program to complete before it continues execution.

CALLP uses the free-form style, where the extended factor 2 holds the name of the prototype of the called program, as well as all parameters to be passed. The prototype must be defined in the D-specs providing the external name of the called program through the CLTPGM keyword:

```
RPG
D ChgHlpFile      PR                  CLTPGM('CHGHLPF.BAT')
 *
C                   callp     ChgHlpFile
```

The specified value can be a variable, which enables you to specify the program name dynamically during runtime. The program as provided in CLTPGM is searched using the PATH environment variable. Make sure that the extension is included. Otherwise, the VisualAge for RPG runtime is looking for .EXE files only.

If you precede the name by .\\, the VisualAge for RPG runtime will also look into the current directory (the directory, the application is called from):

```
RPG
D ChgHlpFile      PR                  CLTPGM(PgmName)
 *
D PgmName         S             15A   INZ(*BLANKS)
 *
C                   eval      PgmName = '.\\CHGHLPF.BAT'
C                   callp     ChgHlpFile
```

All parameters that need to be passed to the program must have an entry immediately following the prototype line for the program. Please note, that this is just the prototype; the variable definitions need to be coded separately. Additionally, all parameters can be passed by value only. So, the prototypes need to include the keyword VALUE:

RPG

```
D ChgHlpFile      PR                  CLTPGM('.\\CHGHLPF.BAT')
D   FromFile                    12A   VALUE
D   ToFile                      12A   VALUE
 *
D FromFile        S               12A   INZ('CONTAINE')
D ToFile          S               12A   INZ('CONTAENU')
 *
C                   callp     ChgHlpFile(FromFile:ToFile)
```

And here is the batch file, so you see what is called:

Batch file

```
@echo off
@del %2.HLP >nul
@copy %1.HLP %2.HLP >nul
@echo on
```

START Operation

The START operation can be used to invoke a local program asynchronously and immediately continue execution of your calling VisualAge for RPG program.

As this opcode is also used to invoke a VisualAge for RPG component, the VisualAge for RPG runtime needs to distinguish between these two types of objects. It does this through the keyword LINKAGE(*CLIENT). If the variable or constant specified in factor 2 has this keyword in its definition specification, the VisualAge for RPG runtime assumes the value to be a local program:

RPG

```
D ChgHlpFile      S               15A   INZ('.\\CHGHLPF.BAT')
D                                       LINKAGE(*CLIENT)
 *
C                   start     ChgHlpFile
```

As for the CALLP operation, the VisualAge for RPG runtime is searching the directories in the PATH environment variable for the program to be called. Specifying \\ in front of the name will result in the runtime search in the current directory as well.

Parameters do not need to be prototyped and are always passed by value. They can be specified as parameter list (PLIST) in the result field of the START operation or as consecutive PARM opcodes. It is your responsibility to invoke the program, providing arguments with the correct attributes.

Examples

To give you an idea of what can be done with the CALLP and START opcodes, here are some samples.

Changing Password

Assume you would like to have the users of your application change their passwords to the serving AS/400 regularly (always a good idea). What about the automatic logon feature of VisualAge for RPG?

You could have the *Define Server Logon* dialog of VisualAge for RPG invoked by your application automatically through this interface, enabling the user to change the automatic logon password as necessary.

This feature may come in handy when used with the sample of section “AS/400 Commands,” which submits commands to the AS/400. After adding another push button to its graphical user interface, the PRESS event will simply call program FVDEPW.EXE, which is shipped with VisualAge for RPG.

RPG

```
D ChgLogonInf     PR                  CLTPGM('FVDEPW.EXE')

   :

C     CHGPWD        BEGACT    PRESS          FRA0000B
 *
C                   callp     ChgLogonInf
 *
C                   ENDACT
```

The *Define Server Logon* dialog shows up in front of the application window disabling any input to it (Figure 56). A START operation instead of the CALLP used would keep the application window enabled.

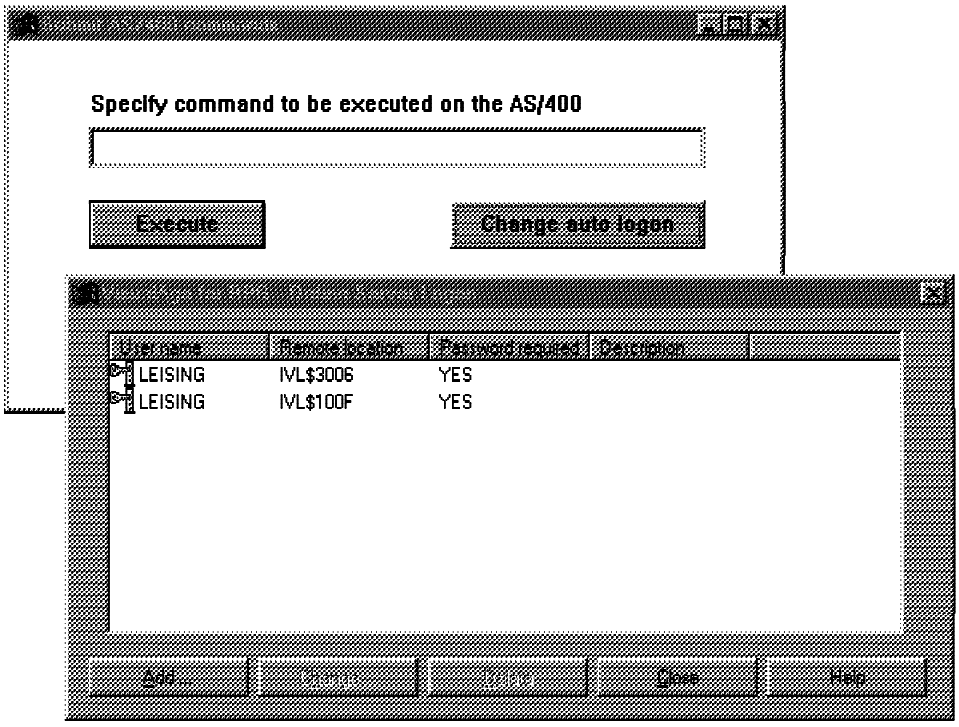


Figure 56. Define Server Logon Window

Changing Server

The same sample application could also be enhanced to be able to change the server on the fly, before submitting the command. The *Define AS/400 Information* dialog can be invoked by calling FVDERST.EXE providing the .RST file as parameter. The action subroutine of the PRESS event for a new push button could do this:

RPG

```
D ChangeServer    PR                  CLTPGM('FVDERST.EXE')
D   PgmName                    255A   VALUE

   :

C     CHGSRV        BEGACT    PRESS          FRA0000B
 *
C                   callp     ChangeServer('QCMDEXC.RST')
 *
C                   ENDACT
```

Please note that *PgmName* is a prototype only and is not used on the CALLP opcode. Instead, the .RST file is passed as literal.

You could also use these two features to write your own setup and maintenance tool. If so, you may want to add another window allowing you to specify the .RST file to be edited. Besides improving the GUI look of your tool, using the FVDERST.EXE for this purpose ensures that the .RST file does not contain invalid data. Invalid data could happen, if you allowed the user to use an ASCII editor instead.

Calling REXX

Another good example would be to call REXX.EXE (shipped with VisualAge for RPG) to enable your application to execute REXX code (Figure 57). It expects a .CMD file with REXX instructions as parameter.

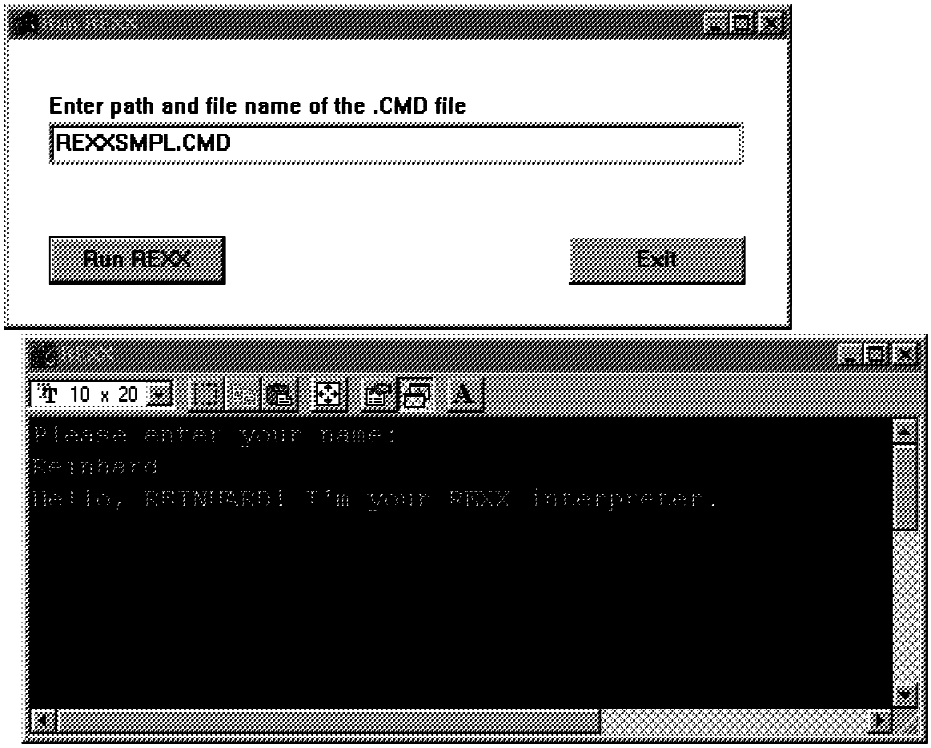


Figure 57. Executing REXX Code

Procedures

VisualAge for RPG supports procedures that enable you to write your code in a more modular way. Local variables and prototyping reduce the risk of unwanted side effects, and your code becomes more reliable. Additionally, VisualAge for RPG allows you to separate certain functions from the rest of your code, making your code more readable and maintainable. Functionality can be reused in other applications.

Prototyping and Invocation

To call a VisualAge for RPG procedure, you first need to define its interface by specifying the prototype in the D-specs. This prototype is examined by the VisualAge for RPG compiler, to make sure that all invocations of the procedure are correct. This includes the number and nature of the parameters as well as the data type of the return code.

The following sample source shows the prototype and invocation of a simple VisualAge for RPG procedure. Its purpose is to place the window of a VisualAge for RPG application into the middle of the screen, before showing it to the user:

RPG

```
D CenterWindow    PR

   :

C     MAIN          BEGACT    CREATE        MAIN
 *
C                   callp     CenterWindow
C                   eval      %setatr('MAIN':'MAIN':'Visible') = 1
 *
C                   ENDACT
```

The CALLP opcode, as used here, is one of the options to invoke a procedure. It calls the procedure synchronously waiting for its completion before continuing execution.

Implementation

Looking at the implementation, you find the executable statements of the procedure enclosed by two procedure specification lines. These P-specifications mark the beginning (B in column 24) and end (E in column 24) of the procedure definition (Figure 58).

RPG

```
PCenterWindow     B
DCenterWindow     PI
 * System attributes
D %DspWidth       S              4P 0
D %DspHeight      S              4P 0
 * Local variables
D winWidth        S              4P 0
D winHeight       S              4P 0
 *
C     'MAIN'        getatr    'Width'       winWidth
C                   eval      %setatr('MAIN':'MAIN':'Left')
C                              = (%DspWidth - winWidth) / 2
 *
C     'MAIN'        getatr    'Height'      winHeight
C                   eval      %setatr('MAIN':'MAIN':'Bottom')
C                              = (%DspHeight - winHeight) / 2
C                                + winHeight
 *
PCenterWindow     E
```

Figure 58. Procedure Specification: CenterWindow

Directly below the beginning P-specification follow the D-specifications that define the procedure's interface: the parameters and the type of the return code. These must match the corresponding prototype. As we don't use parameters or a return code in our first example, the procedure interface definition consists of just one line with a PI entry in columns 24 and 25. Together with the two P-specs, these D-specification lines make up the procedure's header.

The body of the procedure may consist of additional definition or calculation specification lines only. This means that, for example, files must be defined outside of any procedure. However, they are still accessible within a procedure.

The same restriction applies to data areas as well as preruntime and compile-time arrays. They can't be defined within the scope of a procedure, but accessing their data from a procedure is possible.

Variables

All variables defined within a procedure have local scope only and are accessible within the procedure only. If a variable of global scope is defined with an identical name, it is not accessible within the procedure and, therefore, remains unchanged.

By default, local variables are stored into automatic storage. Therefore, if a procedure is called multiple times, they are initialized every time. However, the keyword STATIC can be specified to store a variable in static storage, keeping its value across multiple invocations of the same procedure. This could be used, for example, to implement a counter to keep track of how often the procedure was called. Please note, that the scope of the variable remains unchanged (it's still local).

Parameters

Besides the global variables, that are always accessible in any procedure (with the one exception outlined earlier), you can also use parameters to pass information to your procedure. These are passed by value, by reference, or by constant reference.

Which of these methods is used to pass your parameters has to be specified in the procedure prototype as well as interface. Coding the keyword VALUE for a parameter results in a pass by value. Any change to the parameter within the procedure will not be returned to the calling procedure. This, for example, allows you to specify a literal or even an expression as the value for the parameter.

A parameter is passed by reference, if the VALUE keyword is not specified. Any change to it within the procedure is returned to the calling procedure at the end of the called procedure.

Coding

To show you the coding of parameters, we will enhance our first example and replace the *CenterWindow* procedure by a new *PositionWindow* procedure. This procedure will be able to position the window on the screen depending on two passed parameters.

The first parameter, *VPosition* specifies the vertical position of the window and may have the values *Left*, *Center*, or *Right.* The second parameter, *HPosition*, is used for the horizontal positioning and can contain *Top*, *Center*, or *Bottom.*

Through a third parameter, *ErrCode*, which is passed by reference, we return to the calling procedure one of the following:

- 0, to signal successful completion of the procedure
- -1, to signal an invalid vertical positioning parameter
- -2, to signal an invalid horizontal positioning parameter

The prototype of the new procedure now looks like this:

```
RPG
DPositionWindow   PR
D  VPosition                    6A   VALUE
D  HPosition                    6A   VALUE
D  ErrCode                      1P 0
```

The implementation changes as shown below in Figure 59.

```
RPG
PPositionWindow   B
D                 PI
D  VPosition                    6A   VALUE
D  HPosition                    6A   VALUE
D  ErrCode                      1P 0
 *
 * System attributes
D %DspWidth       S             4P 0
D %DspHeight      S             4P 0
 *
 * Local variables
D winWidth        S             4P 0
D winHeight       S             4P 0
 *
C                   z-add     *ZERO         ErrCode
 *
C     'MAIN'        getatr    'Width'       winWidth
C                   select
C                   when      VPosition = 'Left'
C                   eval      %setatr('MAIN':'MAIN':'Left') = 0
C                   when      VPosition = 'Center'
C                   eval      %setatr('MAIN':'MAIN':'Left')
C                              = (%DspWidth - winWidth) / 2
C                   when      VPosition = 'Right'
C                   eval      %setatr('MAIN':'MAIN':'Left')
C                              = %DspWidth - winWidth
C                   other
C                   eval      ErrCode = -1
C                   endsl
 *
```

Figure 59 (Part 1 of 2). Procedure Specification: PositionWindow

RPG (continued)

```
C     'MAIN'        getatr    'Height'      winHeight
C                   select
 *
C                   when      HPosition = 'Top'
C                   eval      %setatr('MAIN':'MAIN':'Bottom')
C                             = winHeight
 *
C                   when      HPosition = 'Center'
C                   eval      %setatr('MAIN':'MAIN':'Bottom')
C                             = (%DspHeight - winHeight) / 2
C                               + winHeight
 *
C                   when      HPosition = 'Bottom'
C                   eval      %setatr('MAIN':'MAIN':'Bottom')
C                             = %DspHeight
C                   other
C                   eval      ErrCode = -2
C                   endsl
 *
PPositionWindow   E
```

Figure 59 (Part 2 of 2). Procedure Specification: PositionWindow

For the invocation, Factor 2 of the CALLP opcode now contains the name of the procedure prototype followed by the parameters to be passed. As the first two parameters are passed by value, literals are allowed during the invocation. The third parameter value can be examined after returning from the procedure, to see if an invalid value was specified for one of the first two parameters:

RPG

```
C                   z-add     *ZERO         ErrCode           1 0
C                   callp     PositionWindow('Right':'Bottom':ErrCode)
C                   if        ErrCode <> 0
C     DspErr        dsply                   RC                9 0
C                   endif
```

Specifying the keyword CONST for a parameter in the procedure prototype and interface will have it passed as a constant reference. The parameter is passed by reference, but any attempt to change its content within the procedure will result in compile time errors such as RNF5346 ("Result of EVAL operation must not be a field that cannot be modified").

For example, the EVAL opcode in the piece of code shown in Figure 60 would result in this error.

```
RPG
D AddNumbers      PR
D NumA                          6P 0 CONST
D NumB                          5P 0 VALUE
 *
D NumA            S             6P 0
D NumB            S             5P 0
 *
C                   callp     AddNumbers(NumA:NumB)
 *
 *
P AddNumbers      B
D                 PI
D NumA                          6P 0 CONST
D NumB                          5P 0 VALUE
 *
C                   eval      NumA = NumA + NumB
 *
P AddNumbers      E
```

Figure 60. Procedure Prototype with Constant Reference Parameter

If you want to allow a value for one of the parameters to be omitted by using the *OMIT figurative constant, you must enable your procedure to handle this situation by specifying the OPTIONS(*OMIT) keyword for these parameters.

A parameter must be passed by reference or as a CONST reference to allow the *OMIT feature, as shown in Figure 61.

RPG

```
D AddNumbers      PR
D NumA                          6P 0
D NumB                          5P 0 CONST OPTIONS(*OMIT)
 *
D NumA            S             6P 0
D NumB            S             5P 0
 *
C                   callp     AddNumbers(NumA:*OMIT)
 *
 *
P AddNumbers      B
D                 PI
D NumA                          6P 0
D NumB                          5P 0 CONST OPTIONS(*OMIT)
 *
C                   if        NumB = *NULL
C                   eval      NumA = NumA + 1
C                   else
C                   eval      NumA = NumA + NumB
C                   endif
 *
P AddNumbers      E
```

Figure 61. Using a CONST Reference to Allow the OMIT Option

Return Values

Using procedures gives you a lot of advantages over user subroutines. Your code becomes more modular and parameter checking is done. However, with the definition of a return value, procedures become even more powerful.

A return value is defined in the D-spec of the procedure interface line (PI in columns 24 and 25). As its data type and length is checked by the compiler, it needs to be prototyped in the procedures prototype line (PR in columns 24 and 25) as well.

Looking at our example, the *PositionWindow* procedure could be enhanced to pass the information of the third parameter (the error code) through the return value back to its caller. Here is the prototype of the procedure:

```
RPG
DPositionWindow   PR             1P 0
D  VPosition                     6A   VALUE
D  HPosition                     6A   VALUE
```

In the implementation, the procedure header is changed accordingly and the code that manipulated the *ErrCode* parameter is replaced by a RETURN operation. The appropriate error code values that are to be passed back to the caller have to be placed into the RETURN statement's factor 2, as shown in Figure 62.

```
RPG
PPositionWindow   B
D                 PI             1P 0
D  VPosition                     6A   VALUE
D  HPosition                     6A   VALUE
 *
 * System attributes
D %DspWidth       S              4P 0
D %DspHeight      S              4P 0
 * Local variables
D winWidth        S              4P 0
D winHeight       S              4P 0
C     'MAIN'        getatr    'Width'       winWidth
C                   select
 *
C                   when      VPosition = 'Left'
C                   eval      %setatr('MAIN':'MAIN':'Left') = 0
 *
C                   when      VPosition = 'Center'
C                   eval      %setatr('MAIN':'MAIN':'Left')
C                              = (%DspWidth - winWidth) / 2
 *
C                   when      VPosition = 'Right'
C                   eval      %setatr('MAIN':'MAIN':'Left')
C                              = %DspWidth - winWidth
 *
C                   other
C                   return    -1
C                   endsl
 *
```

Figure 62 (Part 1 of 2). PositionWindow Procedure with Return Value

RPG (continued)

```
C     'MAIN'        getatr    'Height'      winHeight
C                   select
 *
C                   when      HPosition = 'Top'
C                   eval      %setatr('MAIN':'MAIN':'Bottom')
C                             = winHeight
 *
C                   when      HPosition = 'Center'
C                   eval      %setatr('MAIN':'MAIN':'Bottom')
C                             = (%DspHeight - winHeight) / 2
C                               + winHeight
 *
C                   when      HPosition = 'Bottom'
C                   eval      %setatr('MAIN':'MAIN':'Bottom')
C                             = %DspHeight
 *
C                   other
C                   return    -2
C                   endsl
 *
C                   return    0
 *
PPositionWindow   E
```

Figure 62 (Part 2 of 2). PositionWindow Procedure with Return Value

To receive the return value, we need to replace the CALLP by an EVAL opcode. The *PositionWindow* procedure now is invoked as part of an expression. When control returns, the return value is assigned to the error code variable:

RPG

```
C                   z-add     *ZERO         ErrCode          1 0
C                   eval      ErrCode
C                             = PositionWindow('Right':'Bottom')
C                   if        ErrCode <> 0
C     DspErr        dsply                   RC               9 0
C                   endif
```

It is important to remember that a procedure invoked within an expression always needs to return a value. When designing a procedure, you need to make sure that a return without a return value is not possible. However, as

a procedure can be invoked as part of an expression, we can put it directly into the expression of the IF opcode this way:

```
RPG
C                   if        PositionWindow('Right':'Bottom') <> 0
C     DspErr        dsply                   RC               9 0
C                   endif
```

External Procedures

Up to now, we have examined the basics about procedures: how to define and prototype them, and where they get invoked. The modularity we can achieve with this knowledge is a big advantage. But we still cannot separate procedures from the rest of our code and make them reusable from another VisualAge for RPG application.

Utility DLL

To address the separation difficulty, VisualAge for RPG supports the creation of a utility dynamic link library (DLL), that can be linked to any VisualAge for RPG application that intends to use the contained procedures.

Before we implement such a utility DLL, let's first have a look at its general structure in Figure 63.

RPG

```
H NOMAIN
 *
 * Global definitions
FMyFile ...
IMyFile ...
D GlobalVar       S            100A
 *
 * Prototype
DExtProc          PR             4P 0
D P1                             5P 0 VALUE
D P2                            10A   VALUE
DIntProc          PR             5P 0
D P1                            10A   VALUE
 *
 * Procedure definition (will be exported)
PExtProc          B                   EXPORT
D                 PI             4P 0
D P1                             5P 0 VALUE
D P2                            10A   VALUE
 *
C      here's the code of ExtProc
C                   eval      P1 = IntProc('Literal')
C      some more code
 *
PExtProc          E
 *
 * Procedure definition (internal only)
PIntProc          B
D                 PI             5P 0
D P1                            10A   VALUE
 *
C      here's the code of IntProc
 *
PIntProc          E
```

Figure 63. General Structure of Utility DLL

The keyword NOMAIN has to be added to the control specification of your source. This is the indicator that instructs the VisualAge for RPG compiler to generate a .DLL and .LIB file instead of the .EXE and .DLL file for a usual VisualAge for RPG application.

The NOMAIN keyword can be followed by file and input specifications or definition specifications for global variables. Everything defined here is accessible across all procedures of the utility DLL.

You must also include prototypes for any procedure defined in the DLL that must be accessible from outside. If a specific procedure is to be accessible from outside, indicate that with the keyword EXPORT in the beginning P-spec of the procedure's definition header. Without this keyword, the procedure will be callable from any procedure in the DLL, but completely hidden to the outer world.

When dealing with utility DLLs, consider the following restrictions:

- A utility DLL may contain calculation specifications within procedures only. Therefore, BEGSR and ENDSR operations must be completely enclosed within a procedure.
- No GUI operation codes or built-in functions are allowed. This includes START, STOP, SETATR, GETATR, SHOWWIN, CLSWIN, and READS as well as %GETATR and %SETATR. The DSPLY opcode is allowed, but will be ignored when the procedure is called by a VisualAge for RPG application. To include any of these opcodes in your procedure, place them into separate source files and include them through the /COPY compiler directive.
- *INZSR and *TERMSR are not permitted.
- *ENTRY PLIST is not allowed.
- The default exception handler is not invoked when an exception occurs in the utility DLL.

We now extend our window positioning example to address the fact that the *PositionWindow* procedure doesn't calculate the new vertical and horizontal itself. Instead, it invokes two new procedures *VPosWindow* and *HPosWindow*, passing either procedure the width (or height, respectively) of the display and the window as well as the positioning information *Left*, *Center*, or *Right* (or *Top*, *Center*, or *Bottom,* respectively).

The return value of these two procedures will contain the newly calculated value for the LEFT (or BOTTOM, respectively) attribute of the window part to be positioned. If you pass to the procedures a window size that is larger than the corresponding size of the display, -1 will be returned. If the positioning value is invalid, the return value will be -2.

We now look at the *PositionWindow* procedure, from which the two new procedures will be invoked. Please note that the procedure can't be exported, as it contains GUI operation codes. It remains a part of our VisualAge for RPG application. Figure 64 shows the code for the procedure.

RPG

```
PPositionWindow   B
D                 PI             1P 0
D  VPosition                     6A   VALUE
D  HPosition                     6A   VALUE
 * System attributes
D %DspWidth       S              4P 0
D %DspHeight      S              4P 0
 * Local variables
D winWidth        S              4P 0
D winHeight       S              4P 0
D newPos          S              4P 0
 * Prototype of local procedures
DVPosWindow       PR             4P 0
D dspHeight                      4P 0 VALUE
D winHeight                      4P 0 VALUE
D position                       6A   VALUE
DHPosWindow       PR             4P 0
D dspWidth                       4P 0 VALUE
D winWidth                       4P 0 VALUE
D position                       6A   VALUE
 *
C     'MAIN'        getatr    'Width'       winWidth
C                   eval      newPos
C                              = VPosWindow(%DspWidth:winWidth
C                                           :VPosition)
C                   if        newPos >= 0
C                   eval      %setatr('MAIN':'MAIN':'Left') = newPos
C                   else
C                   return    -1
C                   endif
 *
C     'MAIN'        getatr    'Height'      winHeight
C                   eval      newPos
C                              = HPosWindow(%DspHeight:winHeight
C                                           :HPosition)
C                   if        newPos >= 0
C                   eval      %setatr('MAIN':'MAIN':'Bottom') = newPos
C                   else
C                   return    -2
C                   endif
 *
C                   return    0
 *
PPositionWindow   E
```

Figure 64. PositionWindow Procedure with Local Procedure Prototype

The two new procedures are prototyped within the scope of the calling *PositionWindow* procedure. They can be called from this procedure only. If

they need to be called from anywhere else in the program, the prototype must be repeated there as well.

The SELECT constructs have been replaced by the invocation of the procedures. As the return value may contain error codes (-1 or -2), we need to make sure that the return value is not used as the new value for the LEFT and BOTTOM attributes of the window. Instead, we will return from *PositionWindow* with an appropriate return value.

You can avoid the negative return codes by checking the parameters before calling one of the procedure. In this case, you can code the invocation as follows:

RPG

```
   :
 *
C     'MAIN'        getatr    'Width'       winWidth
C                   if          VPosition <> 'Left'
C                             or VPosition <> 'Center'
C                             or VPosition <> 'Right'
C                   return    -1
C                   else
C                   eval      %setatr('MAIN':'MAIN':'Left')
C                              = VPosWindow(%DspWidth
C                                           :winWidth
C                                           :VPosition)
C                   endif
 *
   :
```

VPosWindow and *HPosWindow* reside in a utility DLL called *SERVICE.DLL*.

As the prototype information needs to be placed into the utility DLL as well as into all VisualAge for RPG programs that may call the procedures, we placed it into a separate /COPY file (Figure 65). This way we make sure that the interface of the procedures is defined correctly for all occurrences.

RPG

```
H NOMAIN
 *
 * Prototype of the DLL's interface
D/COPY I:\VARPG\SERVICE\SERVICE.CPY
 *
PVPosWindow       B                   EXPORT
D                 PI             4P 0
D dspWidth                       4P 0 VALUE
D winWidth                       4P 0 VALUE
D position                       6A   VALUE
 *
C                   if        dspWidth < winWidth
C                   return    -1
C                   else
C                   select
C                   when      position = 'Left'
C                   return    0
C                   when      position = 'Center'
C                   return    (dspWidth - winWidth) / 2
C                   when      position = 'Right'
C                   return    dspWidth - winWidth
C                   other
C                   return    -2
C                   endsl
C                   endif
 *
PVPosWindow       E
 *
PHPosWindow       B                   EXPORT
D                 PI             4P 0
D dspHeight                      4P 0 VALUE
D winHeight                      4P 0 VALUE
D position                       6A   VALUE
 *
C                   if        dspHeight < winHeight
C                   return    -1
C                   else
C                   select
C                   when      position = 'Top'
C                   return    winHeight
C                   when      position = 'Center'
C                   return    (dspHeight - winHeight) / 2 + winHeight
C                   when      position = 'Bottom'
C                   return    dspHeight
C                   other
C                   return    -2
C                   endsl
C                   endif
 *
PHPosWindow       E
```

Figure 65. Using a COPY File for Procedure Prototypes

The build process generates a SERVICE.DLL and a SERVICE.LIB file. While the DLL is stored in the RT_WIN32 subdirectory, the .LIB file is placed in the utility component's project directory.

Binding

To use the procedures of this utility DLL in our window positioning example, we need to link the DLL to it during the program's build process. This can be done by specifying the .LIB file through the *Link libraries and objects* entry field of the build options (Figure 66).

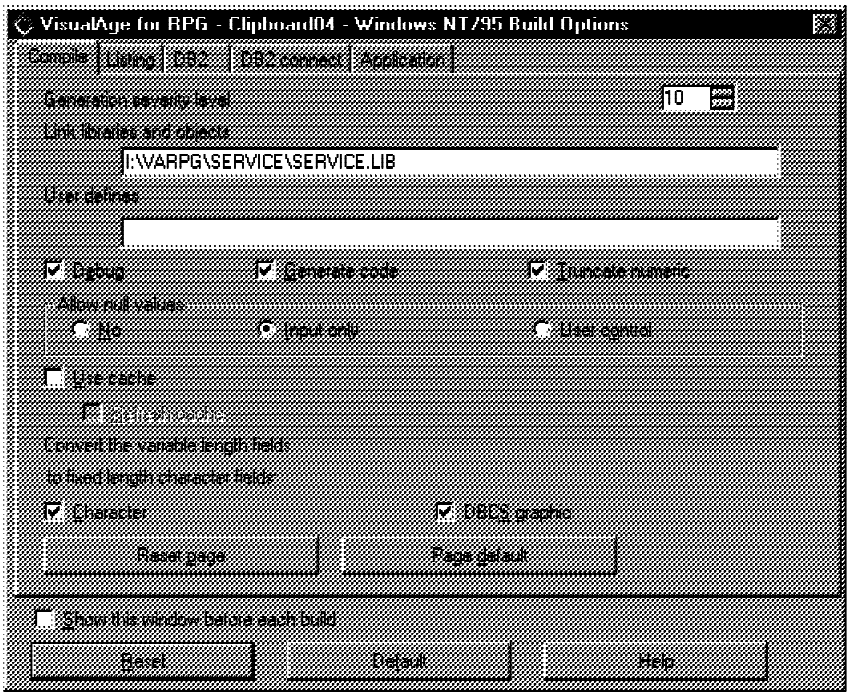


Figure 66. Build Options Window, Compile Page

The procedures get bound to the application statically. The compiler listing shows the external information in its *External References* section:

```
Listing

                    E x t e r n a l   R e f e r e n c e s

   Statically bound procedures:
      Procedure                              References
      VPOSWINDOW                                  0
      HPOSWINDOW                                  0

* * * * *  E N D   O F   E X T E R N A L   R E F E R E N C E S  *
```

It is also possible to have an EXE file created from a source that contains only procedures. However, the EXE keyword must be specified in the control specifications.

In addition to the restrictions of a utility DLL, the source file must have a procedure definition with the same name as the source file itself. This is the main procedure that is invoked when the EXE is called. Any other included procedures cannot be exported to the outside world. They can be called from other procedures in the same EXE file only.

The only allowed method of passing parameters is by value. A return value must be of type binary or integer, but is not accessible if the calling program is a VisualAge for RPG application.

The general structure of such an EXE file is as shown in Figure 67.

RPG

```
H EXE
 *
 * Global definitions
FMyFile ...
IMyFile ...
D GlobalVar       S            100A
 *
 * Prototype
DMainProc         PR
D P1                             5P 0 VALUE
D P2                            10A   VALUE
DIntProc          PR             5P 0
D P1                            10A   VALUE
 *
 * Main procedure definition
PMainProc         B
D                 PI
D P1                             5P 0 VALUE
D P2                            10A   VALUE
 *
C      here's the code of MainProc
C                   eval      P1 = IntProc('Literal')
C      some more code
 *
PMainProc         E
 *
 * Procedure definition (internal only)
PIntProc          B
D                 PI             5P 0
D P1                            10A   VALUE
 *
C      here's the code of IntProc
 *
PIntProc          E
```

Figure 67. General Structure of an EXE File Example

You can call this EXE file from whatever application you want. If it is another VisualAge for RPG program, the CALLP together with the CLTPGM keyword is to be used (see “Local Programs”).

Functions from Foreign DLLs

It may sometimes be necessary to invoke from a DLL a function that was written in a different language than VisualAge for RPG. For example you may need to perform mathematical calculations that are not supported by VisualAge for RPG, or to access the APIs of Client Access/400.

Prototyping

VisualAge for RPG allows the invocation of functions from foreign DLLs, as long as they follow the __cdecl linkage convention. The two operation codes available are CALLP and CALLB. Additionally, a prototyped function can be invoked within an expression using IF or EVAL opcodes.

CALLP should be your first choice, as CALLB doesn't support any prototyping and offers no additional features over CALLP. It is provided for compatibility only, as VisualAge for RPG applications that use this opcode may have been migrated from OS/2.

As for the CALLP syntax, it really makes no difference whether you are calling a procedure of a VisualAge for RPG utility DLL or a function from a DLL written in a different programming language. Look at “Procedures” for basic syntax information.

Considerations

There are some special considerations when working with foreign DLLs, for example, function names. By default, the VisualAge for RPG compiler is looking for a function with the name of the prototype in any of the specified DLLs. However, this search is not case sensitive and, therefore, wouldn't be successful if the DLL exports a function in lower or mixed case.

We need to establish a link between the prototype name and the name of the function exported from the DLL. This is done through the EXTPROC keyword, which can be specified on the prototype D-spec:

RPG

```
D Sin             PR             8F   EXTPROC('getSin')
D   Radian                       8F   VALUE
 *
D Cosin           PR             8F   EXTPROC('getCosin')
D   Radian                       8F   VALUE
 *
 *
C                   eval      SINE = Sin(RADIAN)
C                   eval      COSINE = Cosin(RADIAN)
```

Instead of the literal, you can code a procedure pointer. This allows you to dynamically call one of several possible functions. However, you must make sure that the return values and the attributes of all passed parameters are the same (Figure 68).

RPG

```
D Sin             PR             8F   EXTPROC('getSin')
D   Radian                       8F   VALUE
D Cosin           PR             8F   EXTPROC('getCosin')
D   Radian                       8F   VALUE
 *
D TrigonFunc      PR             8F   EXTPROC(TrigonFuncP)
D   Radian                       8F   VALUE
D TrigonFuncP     S               *   PROCPTR
 *
D Function        S              5A   INZ(*BLANKS)
D Result          S              8F   INZ(*BLANKS)
 *
 *
C                   select
C                   when      Function = 'sin'
C                   eval      TrigonFuncP = %paddr('getSin')
C                   when      Function = 'cosin'
C                   eval      TrigonFuncP = %paddr('getCosin')
C                   other
C     InvalidFct    dsply                   RC               9 0
C                   endsl
 *
C                   eval      Result = TrigonFunc(RADIAN)
```

Figure 68. Procedure Pointer

Note, that the prototypes of the callable functions need to be included in the code. Otherwise, the procedure address cannot be resolved.

During runtime, VisualAge for RPG is looking for the .DLL file in the directory of your application. However, placing it into a directory that is part of the PATH environment variable should work as well. This would have the added advantage that only one copy of a commonly used DLL needs to exist on the target PC.

If implemented in C, the functions *getSin* and *getCosin* look like this:

```c
#include <math.h>

double getSin(double radian)
{
   return( sin(radian) );
}

double getCosin(double radian)
{
   return( cos(radian) );
}
```

Null-Terminated Strings

If the functions to be called are written in C or C++, alphanumeric parameters often need to be passed as null-terminated strings.

One way to do this is to define a data structure that consists of the alphanumeric variable to be passed and a 1-byte field initialized with x'00', like this:

```rpg
D MyFunc          PR                  EXTPROC('myFunc')
D   String                     32767A   OPTIONS(*VARSIZE)
 *
D NullString      DS
D   MyParm                       10A
D   Nullterm                      1A   INZ(X'00')
 *
C                   callp     MyFunc(NullString)
```

Please note that the keyword OPTIONS(*VARSIZE) can be used, if the function is able to handle strings with variable lengths (such as (char *) in C). This allows you to pass character strings of any size (up to 32767 bytes).

Using the OPTIONS(*STRING) keyword, passing null-terminated strings is much easier. Have a look at the following example, which calls a C function to determine the real length (without the trailing blanks) of an alphanumeric field:

RPG

```
D Length          PR             4B 0 EXTPROC('Length')
D   String                       *   CONST OPTIONS(*STRING)
 *
D String          S              255A
D StrLen          S                4B 0
 *
C                   eval      StrLen = Length(%trimr(String))
```

The C function *Length* simply returns the output of the standard *strlen()* function:

C

```
#include <string.h>

long int Length(char *string)
{
   return( strlen(string) );
}
```

Through the OPTIONS(*STRING) keyword, an alphanumeric parameter is first copied into a temporary variable. (That's why the parameter must be passed by value or by constant reference.) It then gets an X'00' added at its end, which is needed by most C functions. Using the %TRIMR built-in function makes sure that the X'00' is added directly after the last nonblank character.

The %LEN built-in function, together with %TRIMR, would give the same result as the C function:

RPG

```
D String          S              255A
D StrLen          S                4B 0
 *
C                   eval      StrLen = %len(%trimr(String))
```

With the null-terminated string support of VisualAge for RPG you will be able to create your own VisualAge for RPG procedure to calculate the length of a given string. Even literals and expressions are allowed parameter values (see Figure 69).

RPG

```
D Length          PR             4B 0
D   String                        *   CONST OPTIONS(*STRING)
 *
D String          S            255A
D StrLen          S              4B 0
 *
C                   eval      StrLen = Length(String)
 *
 *
P Length          B
D                 PI             4B 0
D   String                        *   CONST OPTIONS(*STRING)
 *
C                   return    %len(%trimr(%str(String)))
 *
P Length          E
```

Figure 69. Using Null-Terminated String

The %STR built-in function is used here to provide the string, which has a pointer variable pointing to it, and which is terminated by X'00'.

Chapter 5. Messages

Messages are always important for an application, as they build one of the interfaces to the user. Good messages can help greatly while the user is interacting with the application. In this chapter, we show you the different ways to define and display various types of messages. We show how they can be used during validity checking of user input, whether to enforce a decision from the user, or to confirm a requested action.

Messages in Your Source

One way to define a message is to code appropriate definition specifications (M in column 24) using various VisualAge for RPG keywords. This definition is always split into two parts.

Message Style

First, you need to specify the type and look of the message. The STYLE keyword is used to define whether a message is displayed as information (*INFO), warning (*WARN), or exception (*HALT) message:

```
RPG
D Warning         M                   STYLE(*WARN)
D                                     BUTTON(*OK)
```

Depending on this keyword, an appropriate icon is displayed in the message window (see a warning message sample in Figure 70).

Buttons

With the BUTTON keyword, the number and text of the available push buttons can be defined. The available values are *OK, *CANCEL, *RETRY, *ABORT, *IGNORE, *ENTER, *NOBUTTON, and *YESBUTTON. You can specify up to three values separated by a colon (:) character.

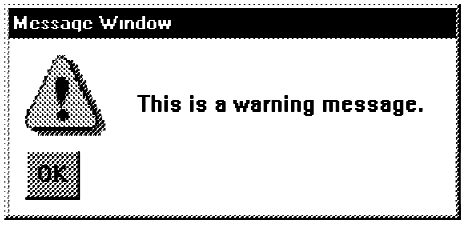


Figure 70. Example of a Warning Message

The second part of the message definition is done through another message D-spec. Here, the message text to be displayed must be specified through the MSGTEXT keyword:

```
RPG
D Message         M                   MSGTEXT('This is a warning -
D                                     message.')
```

To display a message, the DSPLY operation code is used. It expects the message text as factor 1 and the message type definition as factor 2:

```
RPG
C     Message       dsply     Warning       RC                9 0
```

The result field must be a numeric field with a length of 9 bytes and 0 decimal positions. If control returns to the VisualAge for RPG application, the variable specified here will contain a numeric representation of the push button pressed by the user.

Return Codes

The return value from the message box was not useful in our first example, where the only choice was the *OK* push button. However, suppose a user is in the middle of changing the fields of a certain record. By accident, the user presses the *Exit* push button, which ends the application. Wouldn't it be good to display a message to the user asking for a confirmation (Figure 71)?

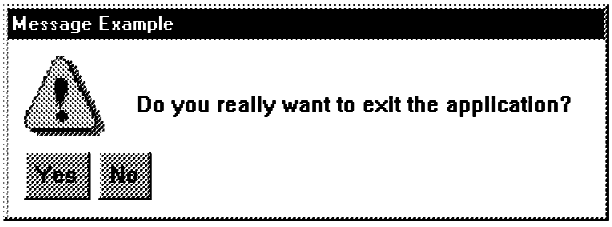


Figure 71. Example of a Message Seeking Confirmation

This can easily be done, as the VisualAge for RPG sample in Figure 72 shows:

```
RPG
D ExitType        M                   STYLE(*WARN)
D                                     BUTTON(*YESBUTTON
D                                            :*NOBUTTON)
 *
D ExitPgm         C                   'Do you really want to -
D                                     exit the application?'
 *
D RC              S              9P 0
 *
C     EXIT          BEGACT    PRESS         MAIN
 *
C     ExitPgm       dsply     ExitType      RC
C                   if        RC = *YESBUTTON
C                   movel     *ON           *INLR
C                   endif
 *
C                   ENDACT
```

Figure 72. Sample Code to Insert a Confirmation Message

You don't need to code a message D-spec for factor 1 of the DSPLY opcode; any literal, named constant, or variable is allowed as well.

You can check the return code variable in the result field against the numeric representation of the available push buttons. However, the easiest and best documenting method is to use the same figurative constants in the compare expression as you used as values for the BUTTON keyword.

Define Messages Dialog

The other method of defining the messages for your application is through the *Define messages* dialog of the GUI designer (Figure 73). You should prefer this method, as you can omit most of the message definitions from your VisualAge for RPG source. Additionally, this prepares your application for national language support (NLS) enhancements (refer to chapter Chapter 7, “National Language Support”).

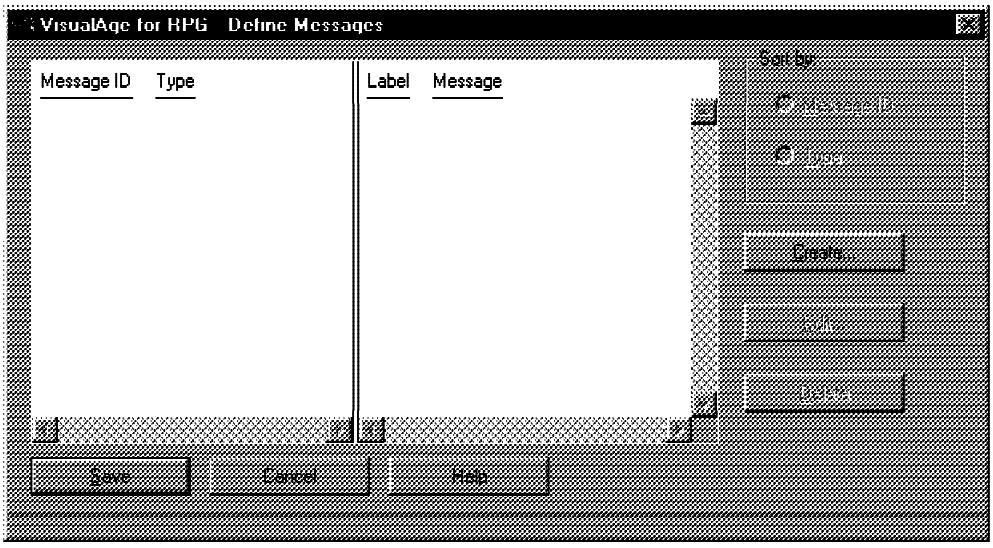


Figure 73. Define Messages Dialog Window

Message Definition

If you select the *Create* push button, another window appears allowing you to define the parts of a message (Figure 74). Everything that is available through the various VisualAge for RPG keywords is also available here. The *Type* combination box allows you to specify the style of the message, while the *Message* entry field shows the text to be displayed. The BUTTON keyword is represented by the *Button* combination box.

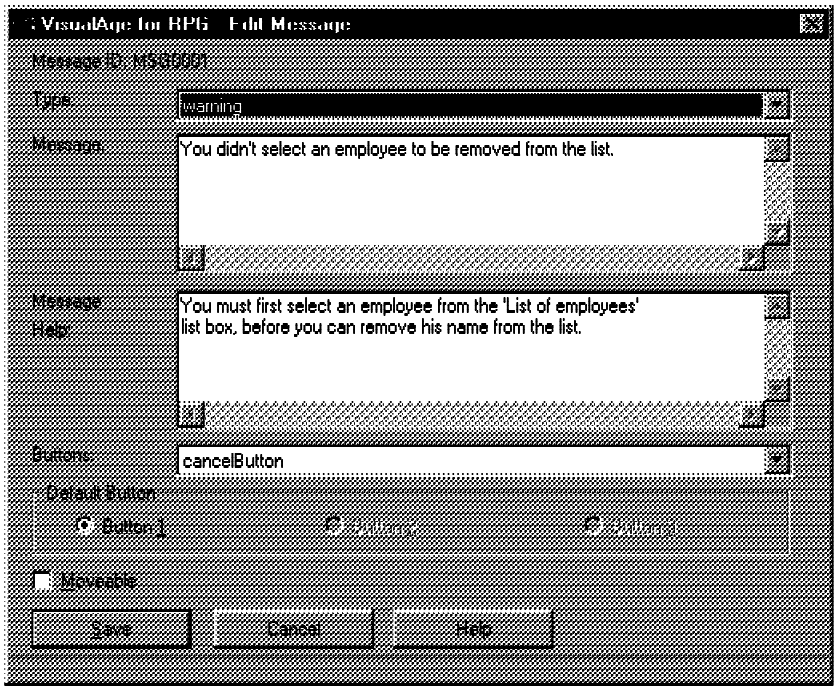


Figure 74. Edit Message Window

In addition, you can also define additional attributes for the message window, that are not available through VisualAge for RPG keywords. For example, if more than one push button is defined for the message window, you can select a default push button that receives the PRESS event, when the Enter key is pressed.

Selecting the *Movable* check box allows the user to move the message window into the background and go on with the application. By default, this check box is not selected, so that the message window becomes application modal, forcing the user to reply to the message before being able to continue the application.

The *Message Help* multiline edit allows you to specify help information for the message window. For further information regarding second-level help text and information presentation facility (IPF) tags, refer to Chapter 6, “Defining Help.”

Displaying Messages

How can these messages be displayed to the user? Once again, the DSPLY opcode does the job.

Instead of specifying the message text in factor 1, you code the message ID assigned to the message on the *Define Messages* dialog (see Figure 74), preceded by a asterisk (*) character:

```
RPG
C                   if        %getatr('MAIN':'EMPLIST':'NbrOfSel') = 0
C     *MSG0001      dsply                   RC
C                   else
 *
 * Code to be performed, if there was an entry selected
 *
C                   endif
```

The message D-spec in factor 2 representing the style becomes obsolete, as this information is available through the message ID. However, if you specify a message D-spec in factor 2, it overlies the definition of the *Define Message* dialog.

Instead of the message ID, you can also specify a message definition name, which is linked to the message ID through the MSGNBR keyword:

```
RPG
D NotSelected     M                   MSGNBR(*MSG0001)
 *
C     NotSelected   dsply                   RC
```

This makes your code more readable, as someone looking at the source gets an idea of what kind of message is displayed without having to invoke the *Define Messages* dialog. If you want to keep the message ID variable, you can also code like this:

```
RPG
D Number          S              8A
D NotSelected     M                   MSGNBR(Number)
 *
C                   movel     '*MSG0001'    Number
C     NotSelected   dsply                   RC
```

Replacement Variables

There is more you can do with this kind of predefined message. For example, assume you are designing an application to maintain a list of employees. Most probably, there will be a *Remove* push button to allow the

user of the application to remove the name of an employee who has left the company.

To avoid an accidental removal, you may want to display a message, when the *Remove* push button is pressed, asking the user to confirm the action. Wouldn't it help to have the information of the selected entry in the message text?

Text Substitution

To achieve this, VisualAge for RPG offers the text substitution feature. In the *Define Messages* dialog, you can specify replacement variables within the message text. Replacement variables are entered as a percent sign (%) followed by a number ranging from 1 to 9 (see Figure 75).

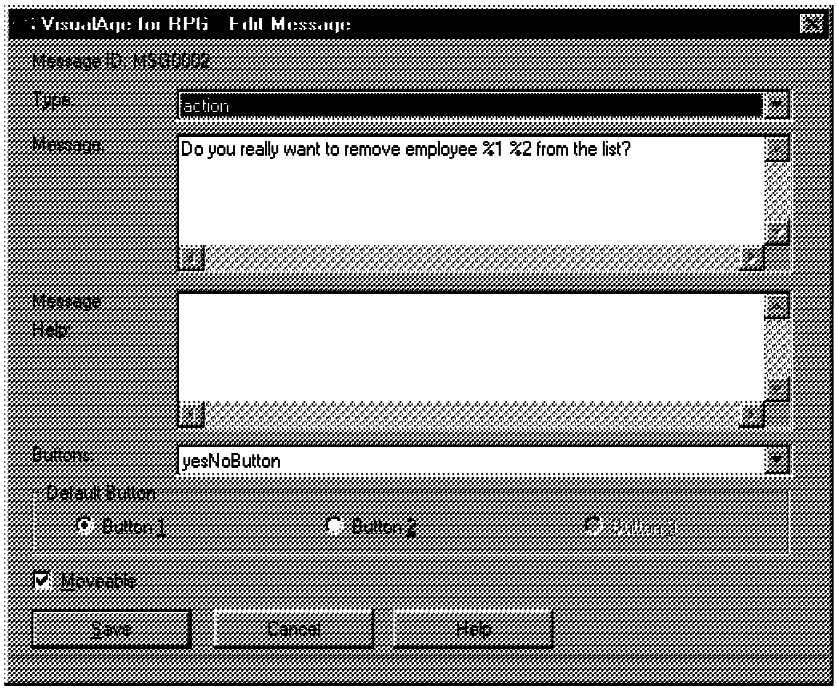


Figure 75. Specifying Replacement Variables in the Edit Messages Window

These replacement variables are filled with the content of those variables specified through the MSGDATA keyword on the message definition specification. See Figure 76.

RPG

```
D FirstName       S             10A
D SurName         S             10A
 *
D RemoveEmp       M                   MSGNBR(*MSG0002)
D                                     MSGDATA(FirstName:SurName)
 *
C                   if        %getatr('MAIN':'EMPLIST':'NbrOfSel') = 0
C     *MSG0001      dsply                 RC
C                   else
C     RemoveEmp     dsply                 RC
C                   if        RC = *YESBUTTON
C                   eval      %setatr('MAIN':'EMPLIST':'RemoveItem')
C                              = %getatr('MAIN':'EMPLIST':'FirstSel')
C                   else
C                   eval      %setatr('MAIN':'EMPLIST':'DeSelect') = 0
C                   endif
C                   endif
```

Figure 76. Coding for Replacement Variables

In our example, this is the first and last name of the employee. The message will be shown as in Figure 77.

Figure 77. Message Window with Substitution Text

Did you notice the blanks following the first and the last name? This is because VisualAge for RPG is filling up each entire field, defined as characters 10 bytes in length. If you want to get rid of these trailing blanks, define the message with just one replacement variable. The variable then needs to contain the rest of the message, starting from its first variable part:

```
RPG
D RemoveEmp       M                   MSGNBR(*MSG0002)
D                                     MSGDATA(RemoveEmpE)
 *
D RemoveEmpE      S             35A   INZ(*BLANKS)
 *
C                   eval      RemoveEmpE = %trim(FirstName) + ' '
C                                        + %trim(SurName)
C                                        + ' from the list box?'
C     RemoveEmp     dsply                   RC
```

The message will now be displayed as shown in Figure 78.

Figure 78. Substitution Text without Blanks

Message Subfile

Another way of displaying messages in VisualAge for RPG is through a message subfile part. Like a message window, it is able to display messages defined in the *Define Messages* dialog as well as text provided by your program logic.

Whether a subfile message is better or worse than a message window depends on the situation.

Multiple Messages

While the message window always displays just one message, a message subfile can show multiple messages at the same time. As the user can scroll through the list of messages, this might be the better choice if the sequence of messages is important or if more than one error occurs at the same time.

On the other hand, if you have an inquiry message that needs a response from the user, a message subfile doesn't make much sense. It is better to use a message window that allows the user to select one of the displayed push buttons.

If you add a message subfile to the GUI of your application, it is always placed at the bottom of the window frame (Figure 79). It is not possible to place it elsewhere or resize its width. This is done automatically when the window itself is resized. The height of the message subfile can be changed to fit your needs. If more messages are added to the message subfile than can be displayed at one time, the user will be shown a scroll bar, allowing him or her to scroll through the messages.

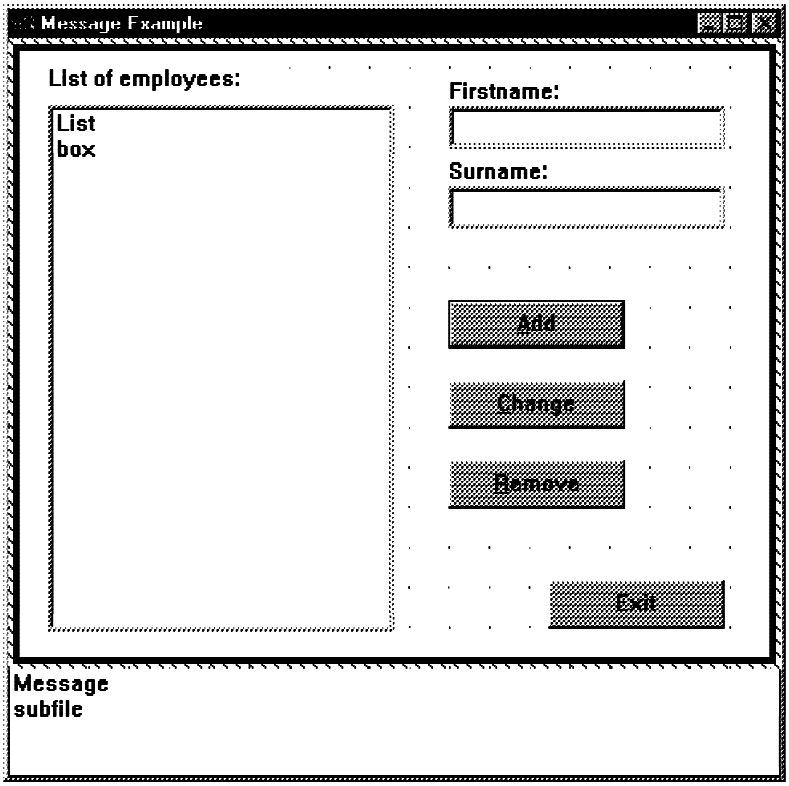


Figure 79. Example of an Employee List Window with Message Subfile

Adding Messages

Within your VisualAge for RPG source code, you can add messages to the message subfile using its AddMsgID or AddMsgText attributes. While the AddMsgText expects a character string, the AddMsgID needs a numeric variable or literal representing the ID of the message as defined in the *Define Messages* dialog.

In our example, we are issuing messages if no first or no last name was specified before pressing the *Add* push button. If both are missing, we display both messages in the subfile (Figure 80).

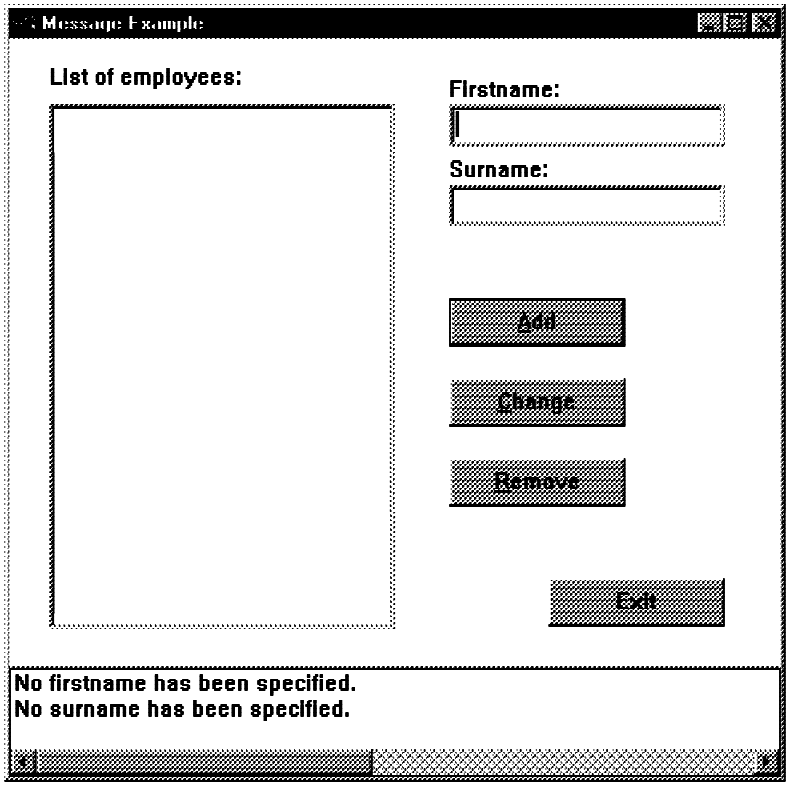


Figure 80. Message Example Window—Subfile with Two Error Messages

Using a message window here would mean that the user first gets a message for the missing first name and, after this error is fixed, another message for the missing last name. This is hardly optimal application design.

Coding a Message Subfile

In the action subroutine of the PRESS event of the *Add* push button, after clearing the message subfile through its RemoveMsg attribute, checks are performed on the content of the *FirstName* and *SurName* entry fields. If they contain blanks, an appropriate message is added to the message subfile for each of them, and the focus is set to the first blank entry field (Figure 81).

RPG

```
C                   read      'MAIN'
 *
C                   eval      %setatr('MAIN':'MSGSFL':'RemoveMsg') = 0
 *
C                   if        FirstName = *BLANKS
C                   eval      %setatr('MAIN':'MSGSFL':'AddMsgID') = 3
C                   eval      %setatr('MAIN':'FIRSTNAME':'Focus') = 1
C                   endif
C                   if        SurName = *BLANKS
C                   eval      %setatr('MAIN':'MSGSFL':'AddMsgID') = 4
C                   if        %getatr('MAIN':'MSGSFL':'Count') = 0
C                   eval      %setatr('MAIN':'SURNAME':'Focus') = 1
C                   endif
C                   endif
 *
C                   if        %getatr('MAIN':'MSGSFL':'Count') = 0
 *
 * Add the code that adds the item to the list here.
 *
C                   endif
```

Figure 81. Adding Messages to the Message Subfile

The Count attribute makes it possible to determine the number of messages currently in the message subfile. If there is no existing message, and all necessary information has been specified by the user, a new item can be added to the subfile.

The code that adds the item to the list will also be enhanced to issue a completion message into the message subfile. This message, defined through the *Define Message* dialog, contains two replacement variables representing the first and last name of the employee who was added to the list.

To fill the variables with proper values, we are using the MsgSubText attribute of the message subfile. As we are passing two substitution texts, we need to place a blank between them, so that VisualAge for RPG knows where the first ends and the second begins.

RPG

```
C                   eval      %setatr('MAIN':'EMPLIST':'Sequence') = 1
C                   eval      %setatr('MAIN':'EMPLIST':'InsertItem')
C                              = Name
C                   eval      %setatr('MAIN':'MSGSFL':'MsgSubText')
C                              = %trim(Firstname)
C                                + ' ' + %trim(SurName)
C                   eval      %setatr('MAIN':'MSGSFL':'AddMsgID') = 5
C                   movel     *BLANKS       Name
```

The piece of code in Figure 82 shows the action subroutine of the SELECT event of our message subfile. This event occurs if a message was selected or deselected by the user. We are using the NbrOfSel, Count, Index, Selected, and GetItem attributes of a message subfile to find the selected entry. We then focus on the appropriate entry part for which this message was issued.

RPG

```
C     MSGSFL        BEGACT    SELECT        MAIN
 *
C                   if        %getatr('MAIN':'MSGSFL':'NbrOfSel') > 0
C     'MSGSFL'      getatr    'Count'       Count             5 0
C     1             do        Count         Idx               5 0
C                   eval      %setatr('MAIN':'MSGSFL':'Index') = Idx
C                   if        %getatr('MAIN':'MSGSFL':'Selected') = 1
C     'MSGSFL'      getatr    'GetItem'     MyMessage       255
C                   select
C                   when      %scan('firstname':MyMessage) <> 0
C                   eval      %setatr('MAIN':'FIRSTNAME':'Focus') = 1
C                   when      %scan('nurname':MyMessage) <> 0
C                   eval      %setatr('MAIN':'SURNAME':'Focus') = 1
C                   endsl
C                   endif
C                   enddo
C                   endif
 *
C                   ENDACT
```

Figure 82. Handling the SELECT Event of the Message Subfile

We achieved here a functionality similar to the one available during the build process of the GUI designer. If errors are found during the build process, double-clicking on the error message in the *Error Messages*

window positions you on the source statement that caused the error message.

Messages as Labels

Messages that are defined in the *Define Messages* dialog can also be used to set the Label attribute of a part (if the part supports the Label attribute). The following example source shows how this is done:

RPG

```
C                   eval      %setatr('MAIN':'MYTEXT':'Label')
C                              = '*MSG0001'
```

The message identifier must be enclosed by single quotes. apostrophes. Therefore, it is not possible to specify the name of a message definition here.

Using messages as labels is very useful, if your intention is to write an NLS-enabled application. If you want to know more about this topic, refer to Chapter 7, “National Language Support” for further details.

Here, we use this feature to implement an information area as it is provided in the GUI designer of VisualAge for RPG in a sample clipboard editor. After defining the informational messages in the *Define Messages* dialog, we first add a static text part, which has a Label attribute, at the bottom of our GUI for the application as shown in Figure 83.

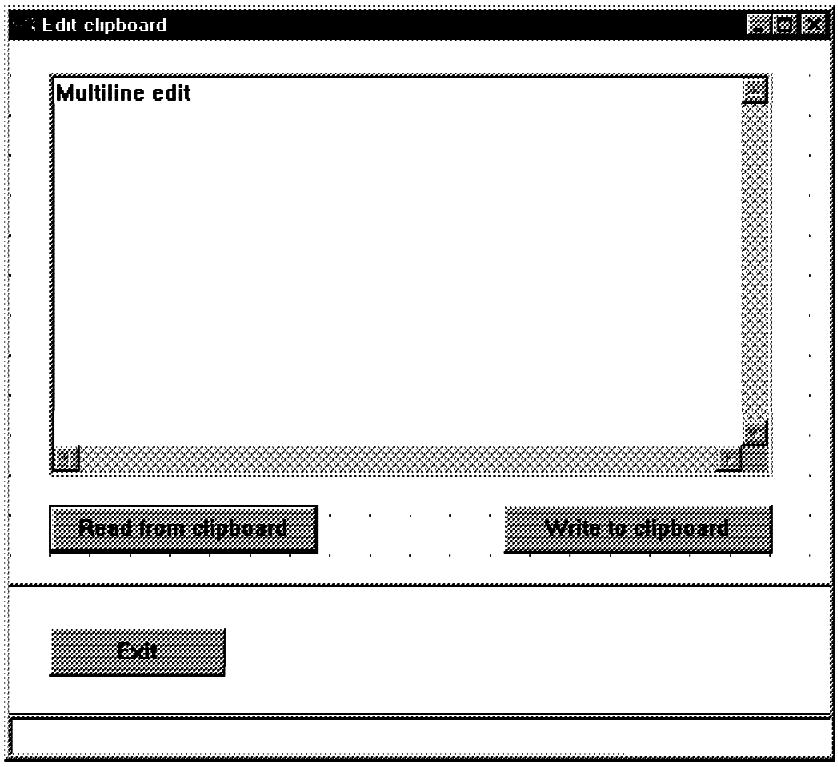


Figure 83. Edit Clipboard Window—Example with Static Text Part

In the VisualAge for RPG source code, we need to add action subroutines for the MOUSEMOVE event of those parts, for which help information should appear in the information line. The following is the action subroutine for the MOUSEMOVE event of the WRITE push button:

```
RPG
C     WRITE         BEGACT    MOUSEMOVE     MAIN
 *
C                   eval      %setatr('MAIN':'INFOLINE':'Label')
C                              = '*MSG0003'
 *
C                   ENDACT
```

If the user now moves the mouse pointer over different parts of the application's GUI, the static text part *INFOLINE* shows the appropriate information (Figure 84).

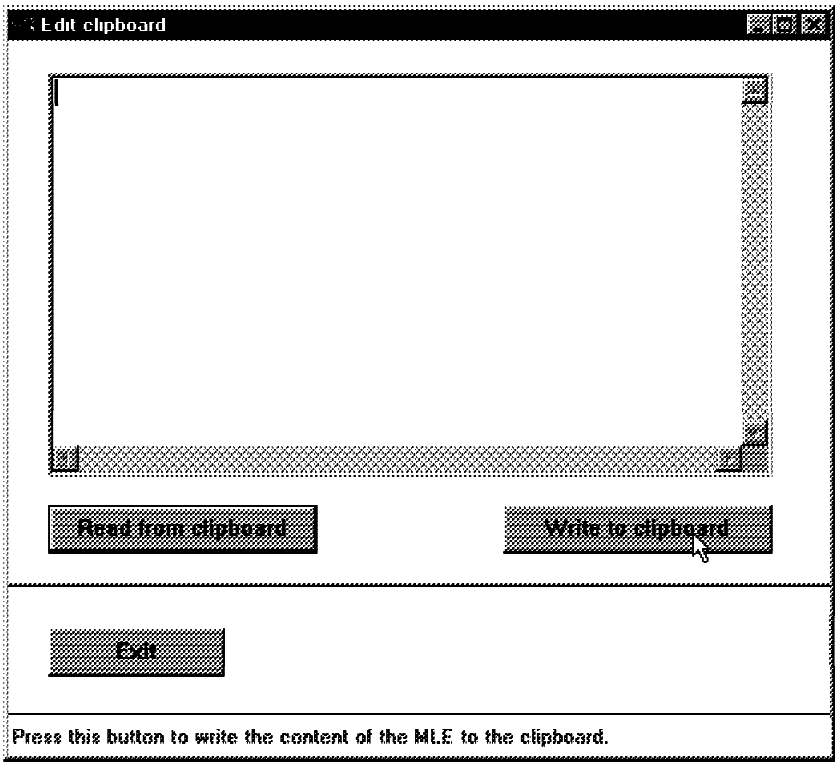


Figure 84. Clipboard Example with Information Line

Chapter 6. Defining Help

In this chapter, you explore the help facilities of VisualAge for RPG and learn how to define help text for context-sensitive help and second-level help for messages. We also explain the basics of the Information Presentation Facility (IPF), which is used to create the .HLP file for applications, and show you how its tags can be used to create your own application help.

Context-Sensitive Help

The first type of help we examine is context-sensitive help. This is the help for a part of your graphical user interface. The help window will be displayed, if the user presses the F1 function key while the part has got the focus. Figure 85 shows the help text defined for the *Read from the clipboard* push button of our sample clipboard editor.

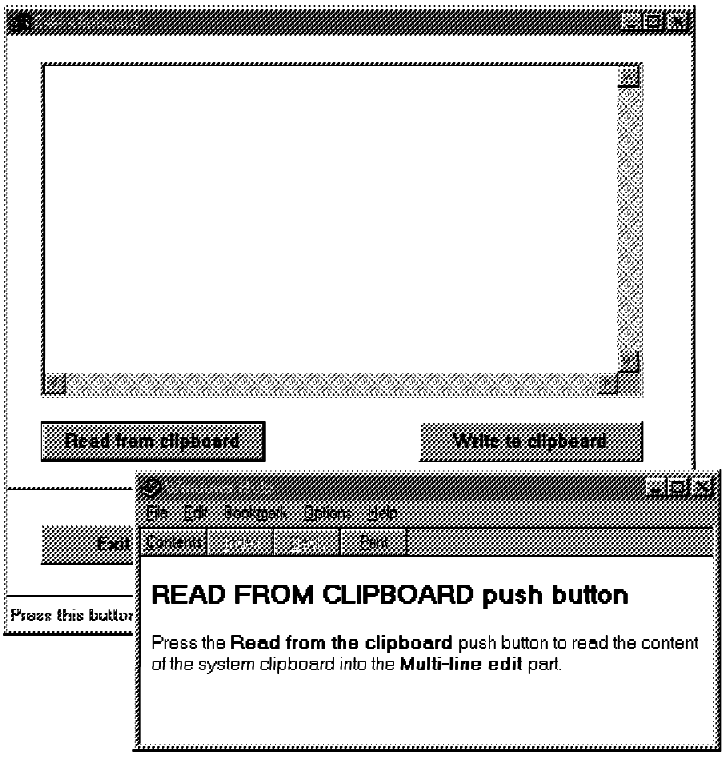


Figure 85. Context-Sensitive Help for a Push Button

LPEX Editor

To add context-sensitive help for a GUI part you must invoke the LPEX editor from the GUI designer of VisualAge for RPG via the context menu of that part (Figure 86).

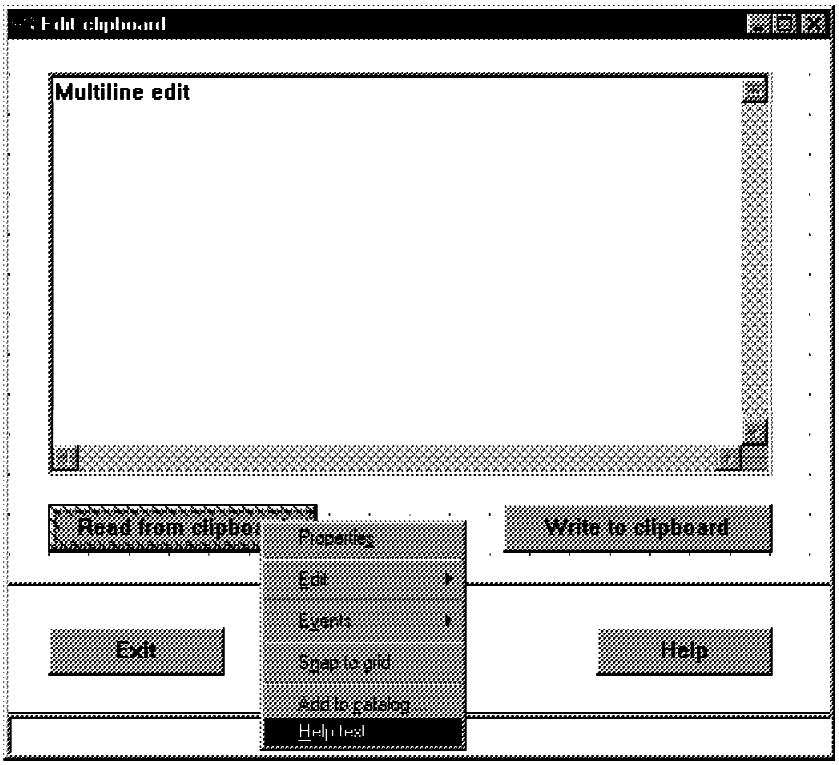


Figure 86. The Context Menu of a Push Button Part

Selecting the *Help text* menu choice results in making LPEX load (or create) a .VPF file with the name of your application (see Figure 87). In this .VPF file, VisualAge for RPG will save the context-sensitive help for all parts of the application.

Default Header

If you invoke the *Help text* menu option for the first time, a default header is added at the bottom of the .VPF file. The first three IPF tags are already there.

- The .* identifies the line as comment line.
- The :h1. marks a header of level 1. Levels between 1 and 6 are allowed. You can use those headers in your help as well.

The res= (resource) parameter is used by VisualAge for RPG to assign this piece of help information as context-sensitive help for the part used to invoke the editor. Therefore, it is important that you don't change the supplied number. Otherwise, VisualAge for RPG may no longer find the defined help. The rest of the line represents the header text to be displayed at the top of the help window. By default, this contains the VisualAge for RPG name of the part.

- The :p. marks the beginning of a new paragraph in your text. Use this tag every time, if you don't want IPF to attach the subsequent text to the end of the previous paragraph. (Any blank line is removed during the build of a .HLP file.)

  In Figure 87, the word *Help* represents the help text you want to key in as help for the selected part.

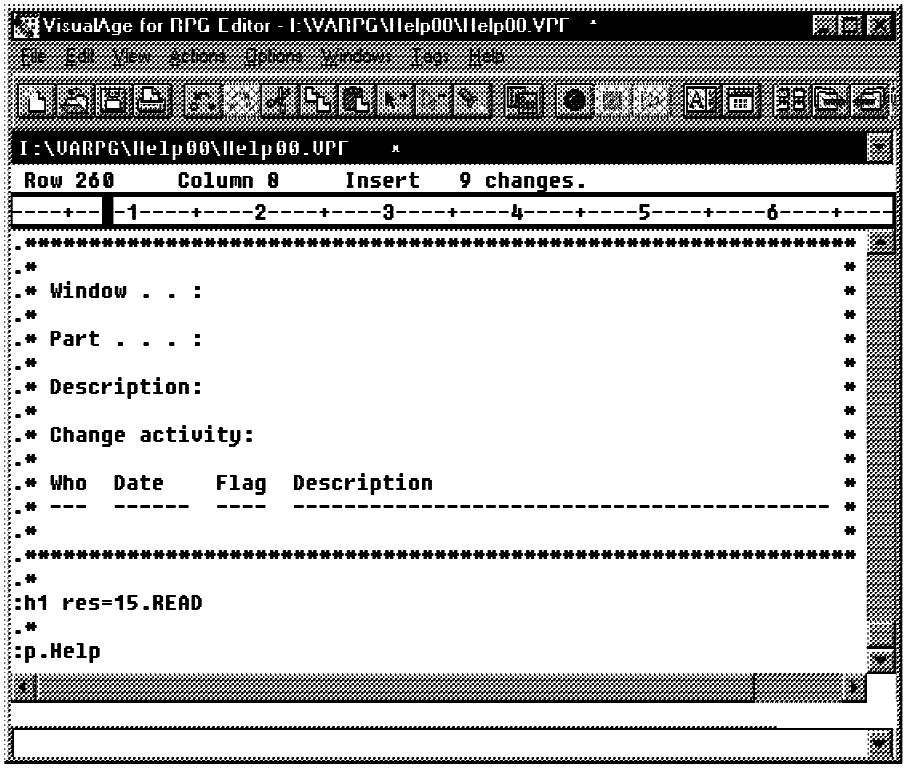


Figure 87. Default .VPF File

The following IPF source was used as the source for the context-sensitive help for the *Read from the clipboard* push button as already shown in Figure 85:

```
IPF
:h1 res=15.READ FROM CLIPBOARD push button
.*
:p.Press the :hp2.Read from the clipboard:ehp2. push
button to read the content of the system clipboard into the
:hp2.Multi-line edit:ehp2. part.
```

Highlighting

Did you notice the :hp2. and :ehp2. tags? Those are highlighting tags, that show the enclosed text as **bold**. More highlighting tags are available (ranging from 1 to 9), so you can display a text in various styles. For example, the :hp1./:ehp1. tags cause the enclosed text to be displayed as *italics*.

After saving the application, the VisualAge for RPG build process also generates the .HLP file using this source. If there is an error in your .VPF file, the *Error List* window comes up with an error message telling you what's wrong. The *Error List* window shown in Figure 88 resulted from a missing :ehp2. tag.

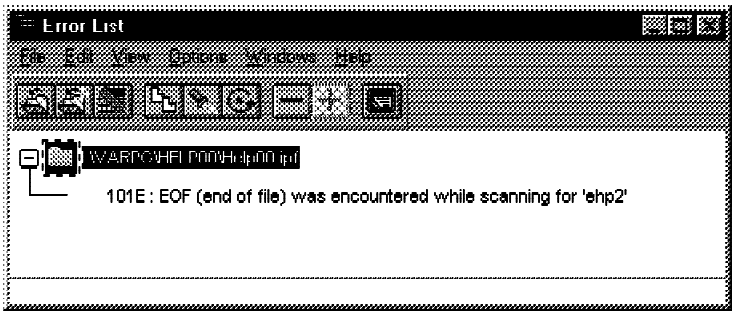


Figure 88. Error List Window Showing a Sample Message

As we want to show you some more basic tags, let's look at the context-sensitive help for the multiline edit part contained in the sample clipboard editor (Figure 89). In addition to some further highlighting, it includes an ordered list as well as a note at the bottom.

Looking at the source, you will see how easily this can be defined. The items of the ordered list are enclosed by the tags :ol. and :eol.. Every item is preceded by a :li. tag. To point the user to an important part of the text, enclose it into the note tags :nt. and :ent., as done in Figure 90.

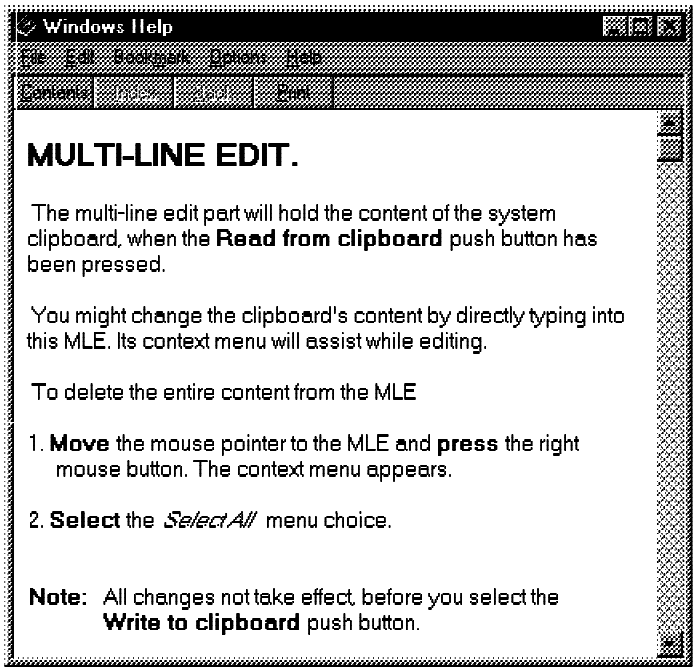


Figure 89. Windows Help Window—Help for the Multiline Edit

IPF

```
:h1 res=12.MULTI-LINE EDIT.
.*
:p.
The multi-line edit part will hold the content of the system clipboard,
when the :hp2.Read from clipboard:ehp2. push button has been pressed.
:p.
You might change the clipboard's content by directly typing into
this MLE. Its context menu will assist while editing.
:p.
To delete the entire content from the MLE
:ol.
:li.:hp2.Move:ehp2. the mouse pointer to the MLE
and :hp2.press:ehp2. the right mouse button. The context menu appears.
:li.:hp2.Select:ehp2. the :hp1.Select All:ehp1. menu choice.
:eol.
:nt.
All changes not take effect, before you select the :hp2.
Write to clipboard:ehp2. push button.
:ent.
```

Figure 90. Source for Multiline Edit Help Window

Resource Identifier

Note that the multiline part got a different resource ID assigned as the *Read from clipboard* push button. All parts used in your project will have such a unique resource identifier.

You may wonder whether it is possible to invoke the editor with the .VPF file just once and code the context sensitive help for all your parts directly, even though you don't know the resource IDs for all the parts. Yes, it's possible.

You can find the resource ID for a certain part by invoking its properties notebook (Figure 91). In the window's title line, the resource ID is displayed in brackets.

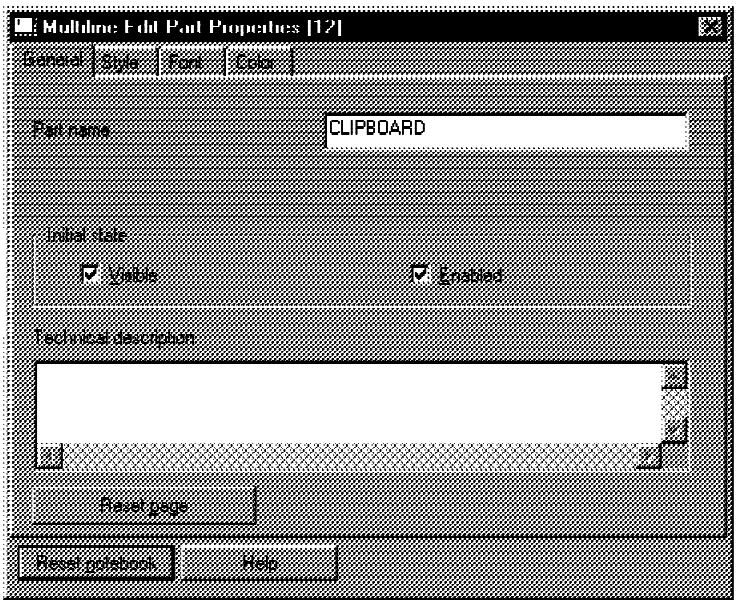


Figure 91. Properties Notebook with Bracketed Resource ID

You have to be very careful if you are editing the .VPF file this way, however. Not all parts support the context-sensitive help, and you could include a typo that might cause unpredictable results.

If you invoke the editor for every part first, then default headers get added to the .VPF file. You can still edit the .VPF file afterwards. Remember, do not touch the `res=` parameter if you are not sure.

The Help Push Button

Providing context-sensitive help is always a good thing to do. But it may also be necessary to implement a *Help* push button, which, if it is pressed, displays more general help about the application or the currently displayed window.

Adding such a *Help* push button to our graphical user interface is easily done. But how to get the help displayed when the push button is gets pressed is another matter. Usually, the PRESS event occurs and the associated action subroutine is executed.

Redirecting Events

There is a function in VisualAge for RPG that allows us to redirect the PRESS event of a push button part. On the *Action* page of its properties notebook (see Figure 92) we can select the *Display help* radio button, which will cause help information to be displayed when the PRESS event occurs.

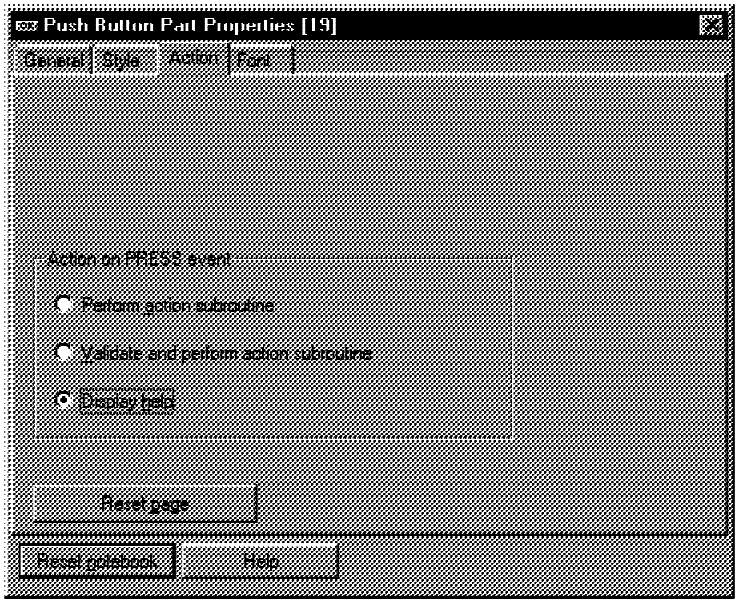


Figure 92. Push Button Part Properties Notebook, Action Page

The help displayed here, however, is the context-sensitive help of the push button part itself. We must then add the help text to the .VPF file. If the *Help* push button is pressed now, the help window displaying general help about the application pops up (see Figure 93).

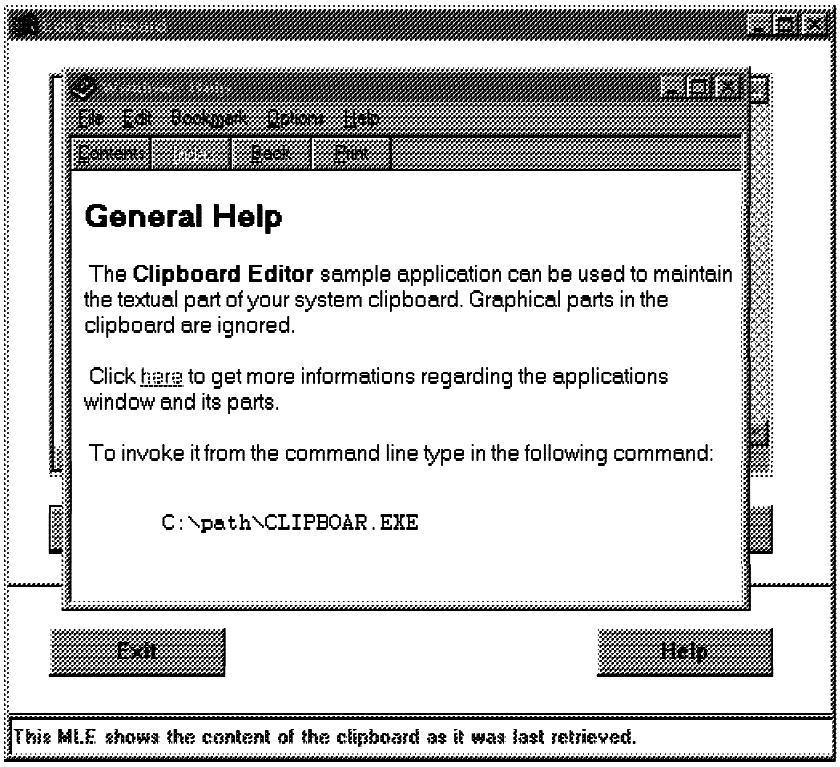


Figure 93. General Help Window for the Clipboard Editor

Symbols

If we look at the source, we can find out how the command-line invocation of the sample application is included:

```
IPF
:h1 res=19.General Help
.*
:p.
The :hp2.Clipboard Editor:ehp2. sample application can be
used to maintain the textual part of your system clipboard. Graphical
parts in the clipboard are ignored.
:p.
Click :link reftype=hd res=11.here:elink. to get more
information regarding the applications window and its parts.
:p.
To invoke it from the command line type in the following command:
:xmp.
     C:&bsl.path&bsl.CLIPBOAR.EXE
:exmp.
```

That may look strange. It is fairly clear that the :xmp. and :exmp. are the tags used to enclose an example. They change the font to monospaced and switch automatic reflowing off, so that the lines stay as they were specified. But what about those &bsl. tags? These are predefined symbols, used to display a character that may not be on your keyboard. The &bsl. stands for *backslash* and inserts the \ character. For a complete list of available predefined symbols, please refer to the IPF reference manual.

Hypertext Links

There is more to discover in the general help text. In Figure 93, the word *here* is not just underlined, it represents a hypertext link to a context-sensitive help text of another part. The resource ID as specified for the `res=` parameter of the :link. tag is 11. Looking through the properties notebooks of the available parts shows that this ID is associated with the window part itself.

If we move the mouse pointer to this word, it changes to a pointing hand. Pressing the left mouse button will show the help defined for the window part (see Figure 94).

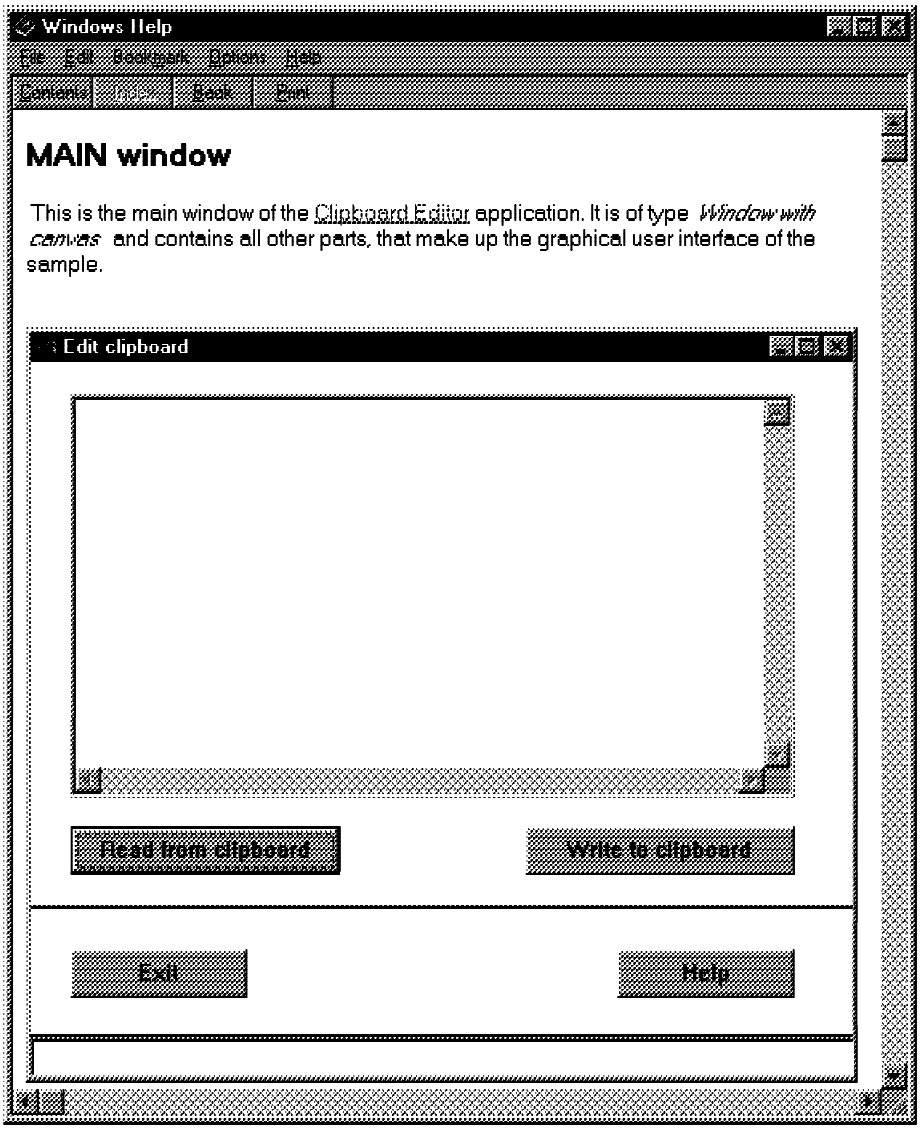


Figure 94. Help for the Window Part

On the window in Figure 94, the words *Clipboard Editor* are again hypertext links. The source in Figure 95 shows that the resource ID is 19, which is associated with the help defined for the *Help* push button.

Embedded Images

Directly following the first paragraph in Figure 95, we included a bitmap of the application's main window, so that the user can look at the parts of the window, which are described further down in the help text. The :artwork. keyword can be used for this purpose.

```
IPF
:h1 res=11.MAIN window
.*
:p.
This is the main window of the :link reftype=hd res=19.Clipboard
Editor:elink. application. It is of type :hp1.Window with
canvas:ehp1. and contains all other parts, that make up the
graphical user interface of the sample.
:p.
:artwork name='CLIPEDIT.BMP' align=center.
:p.
The other parts used to build the graphical user interface are:
:ul.
:li.a :link reftype=hd res=12.Multi-line edit:elink.
:li.a :link reftype=hd res=15.Read from
clipboard:elink. push button
:li.a :link reftype=hd res=13.Write to
clipboard:elink. push button
:li.an :link reftype=hd res=16.Exit:elink. push
button
:li.a :link reftype=hd refid=HELPPB.Help:elink.
push button
:li.and an :link reftype=hd res=18.Information
area:elink.
:eul.
```

Figure 95. Source Code for the Main Window Help

Further down in the help text for the window part, we list all parts of the application. All have been implemented as hypertext links, allowing the user to jump from one help window to another.

Please note that we are using the `refid=` parameter for the *Help push button* help. Here, we want to explain the use of the push button itself, not the display of the general help for the application. So, we use this referencing parameter instead of the `res=` parameter. The header information of the corresponding help text definition has to replace the `res=` parameter with an `id=` parameter:

```
IPF
:h1 id=HELPPB.HELP push button
```

The list itself is an unordered list, instead of the already used ordered list (:ol./:eol.). The begin and end tags for an unordered list are :ul. and :eul..

Using the IPF tags described (and other IPF tags) enables you to create helpful and comprehensive information for the user of your application. The extent to which you need to use these features will depend on user characteristics and skill level.

Displaying Table of Contents

When the user invokes help via F1 or the *Help* push button, the standard Windows 95 or Windows NT help facility is used. This also allows the user to display the table of contents of the entire help file by pressing the *Contents* push button. For our VisualAge for RPG sample application, the table of contents consist of all header tag (:h1-6.) entries defined in the IPF source as hypertext links in the order they were added to the .VPF file (see Figure 96).

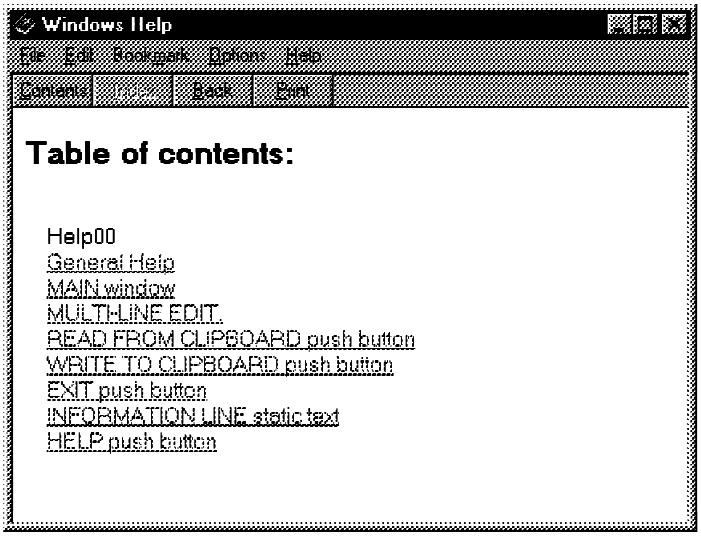


Figure 96. Sample Table of Contents of a Help File

All are at the same level, and VisualAge for RPG adds an entry to the file, representing the application's name (*HELP00* in this case).

To change the heading, you can replace some of the :h1. entries with :h2. or another heading tag. Rearranging the help text can also be used to change the sequence of the table entries.

The additional entry comes from another file residing in the source directory of your application. The .IPF file shown below controls the .VPF and the .IPM files used to create the .HLP file: (The .IPM file contains the second-level text for predefined messages. This is discussed in further detail in "Second-Level Help for Messages.")

IPF

```
:userdoc.
:h1. Help00
:p. Help00.
.im Help00.ipm
.im Help00.vpf
:euserdoc.
```

Commenting the :h1. and :p. lines out removes this additional item from the table of contents (Figure 97).

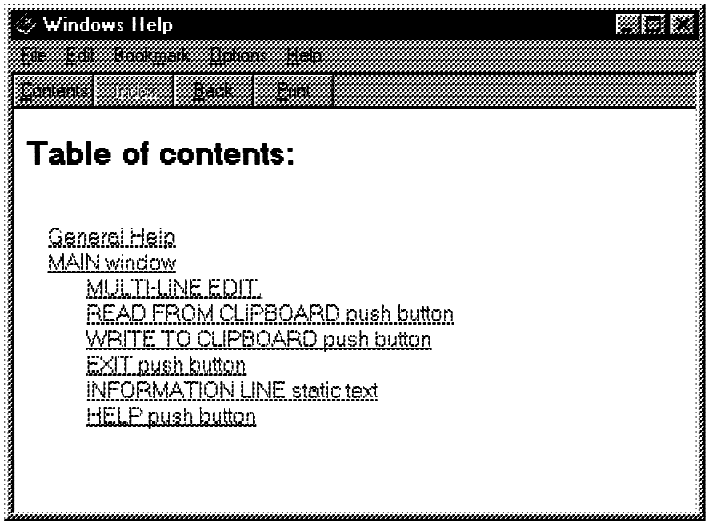


Figure 97. Altered Sample Table of Contents

Second-Level Help for Messages

The other type of help text available within VisualAge for RPG is second-level help for messages, which is also defined using IPF.

To add help information for a predefined message, invoke the *Define messages* dialog. If you select the *Edit* push button the *Edit Message* window comes up. In Figure 98, this window shows the confirmation message for the *Exit* push button of our sample clipboard editor.

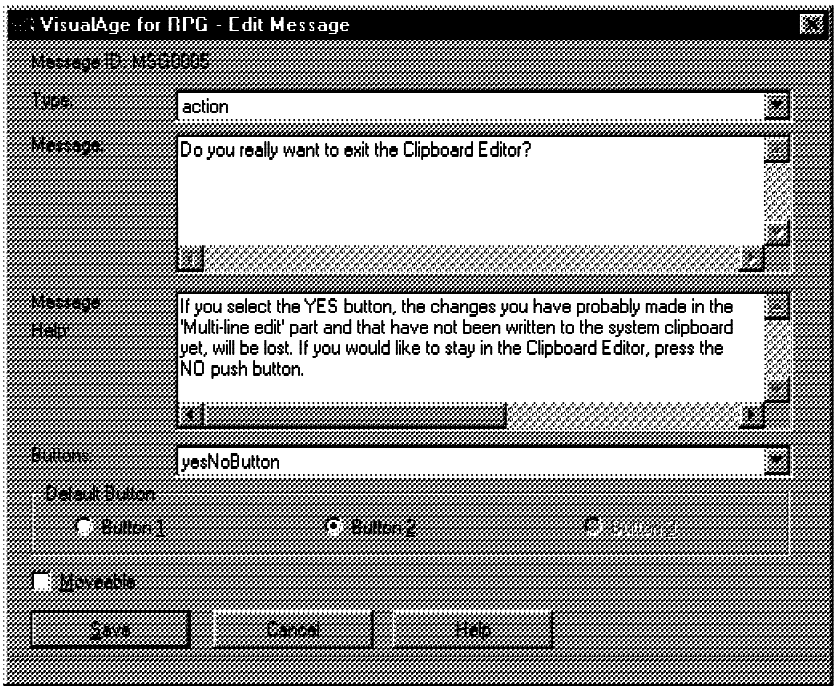


Figure 98. Adding Message Help in the Edit Message Window

In the *Message Help* multiline edit field, you can add the help information for the message. Doing this will result in adding a *Help* push button to the message window (Figure 99).

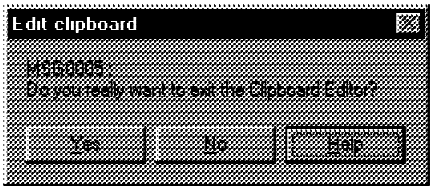


Figure 99. Message Window with Additional Help Push Button

Pressing the *Help* push button (or the F1 function key) provides additional information to the user as to why the message occurred, and what consequences selecting *Yes* will have (Figure 100).

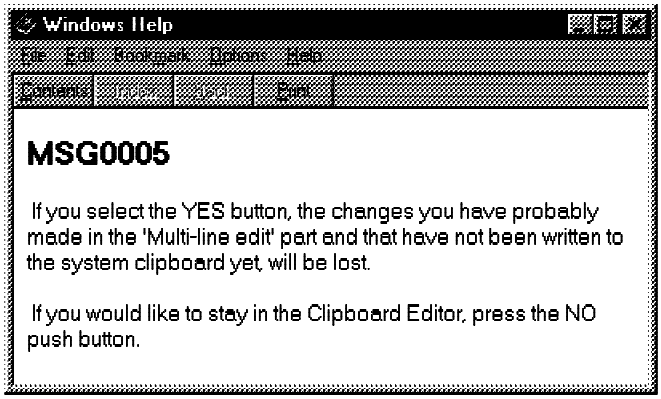


Figure 100. Help for the Exit Push Button

As mentioned, the information specified in the *Message Help* multiline edit field is stored as an IPF source file with the extension .IPM in your project's directory. Therefore, instead of using quotation marks or uppercase to highlight text portions, you can use IPF tags to format the information. If, for example, you want the *Yes* and *No* push buttons displayed as **bold**, add the :hp2. and :ehp2. IPF tags (see Figure 101).

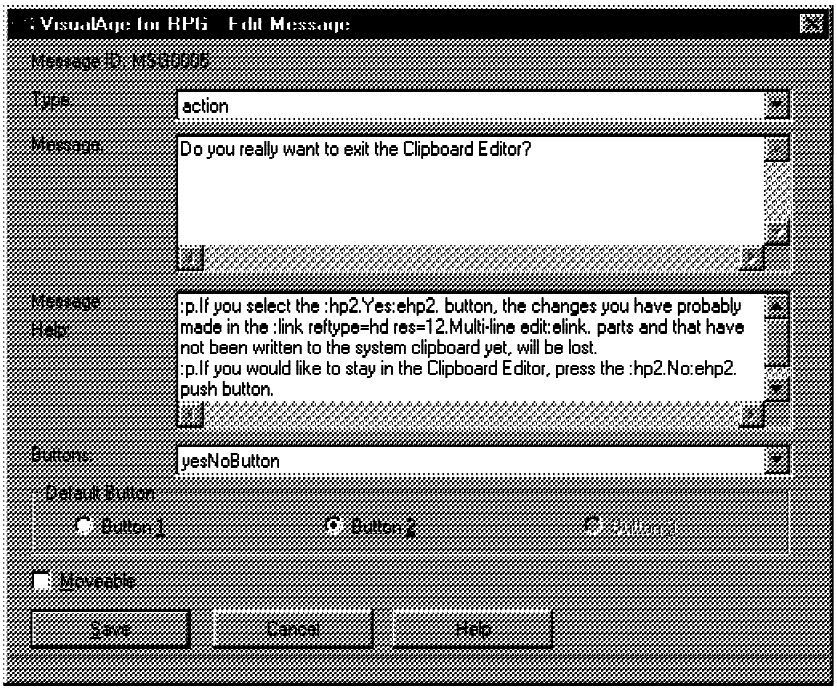


Figure 101. Use IPF Tags in the Message Help Multiline Edit Field

Please note that we also made the words *Multiline edit* a hypertext link to the context-sensitive help information for that part of the GUI. This allows us to reuse this already available piece of help information in the second-level help text of a message. If you do, the help text now shows up as shown in Figure 102.

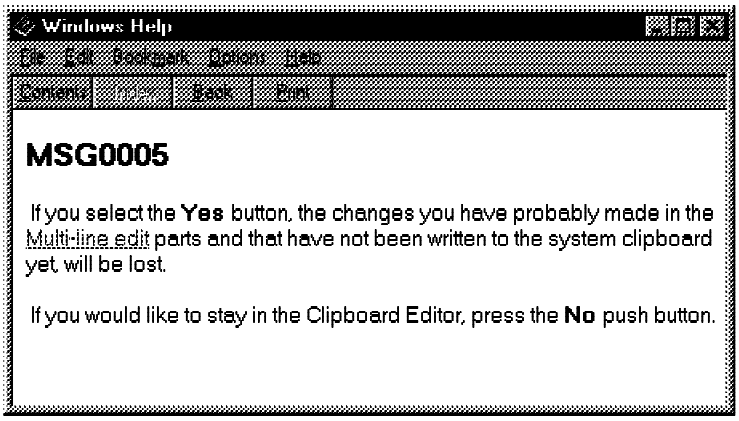


Figure 102. Message Help Window MSGOOO5 with IPF Tags

If we look at the table of contents (Figure 103), we can see that the second-level help text has been added to it.

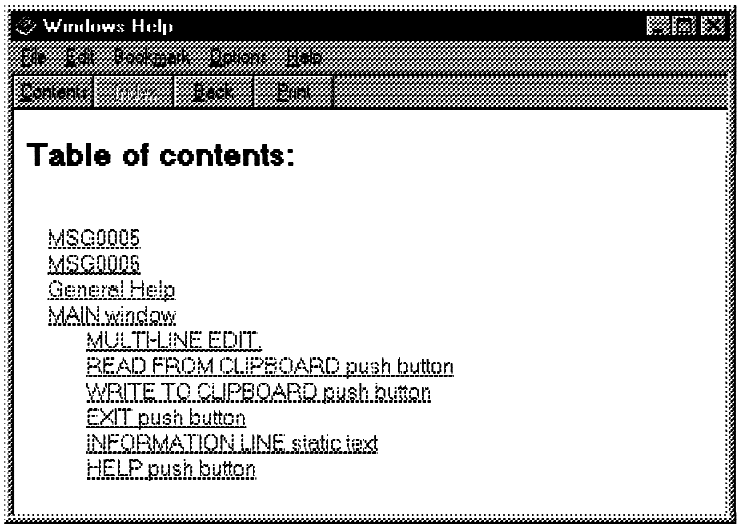


Figure 103. Table of Contents with Second-Level Topics

Every second-level help text is shown as an entry at the top of the list. To change this so that the messages appear at the end of the list, look at the .IPF file of your project, which controls the other files (.VPF and .IPM) during the build:

IPF

```
:userdoc.
.* :h1. Help02
.* :p. Help02.
.im Help02.ipm
.im Help02.vpf
:euserdoc.
```

Among other things, the .IPF file controls the sequence of the .IPM and .VPF file. Through the .im tag, the .IPM file gets imbedded into the help file first, followed by the .VPF file. Replacing these two .im lines makes the message help appear below the context-sensitive help topics (Figure 104).

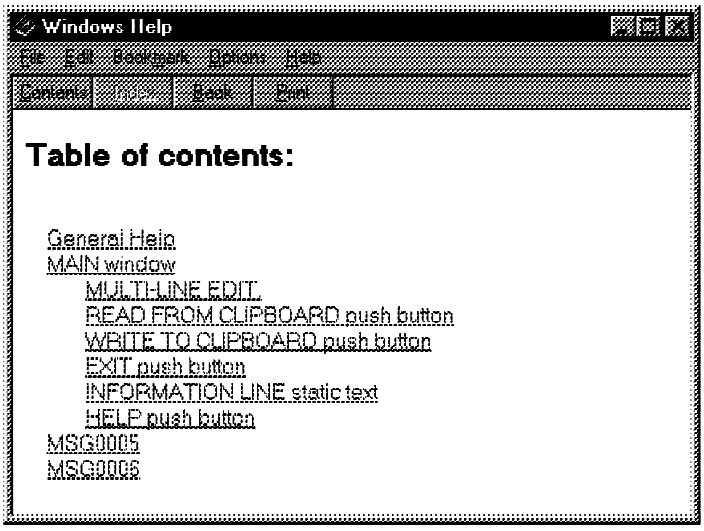


Figure 104. Table of Contents with Second-Level Help at the Bottom

Chapter 7. National Language Support

In this chapter, we discuss the national language support features of VisualAge for RPG. We point out the areas where problems can occur when developing applications for different languages, and we also show you coding techniques that make your code more language independent.

Labels in GUI Parts

First, we look at the design phase of a graphical user interface. Many of the parts you are using to compose the windows of your application have TEXT and LABEL attributes. They may be specified directly through the properties notebook of these parts. For example, look at Figure 105. It shows the GUI of the Container example shipped with VisualAge for RPG. Any text displayed here, such as the window and container titles, the push button labels, or the static text represents attributes that were specified while putting the parts for the GUI parts together.

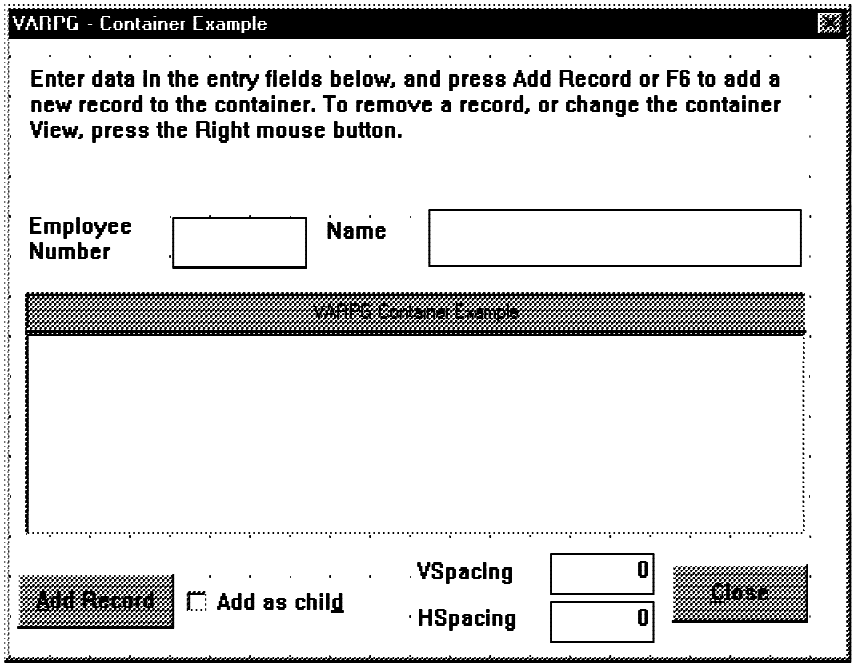


Figure 105. GUI for Container Example

The advantage of this approach is that you need not alter these settings within your RPG code. If these attributes need changes during run time, as

for implementing an information area such as the GUI Designer, a SETATR opcode has to be coded.

Label Substitution

If you are focusing on writing an application that is independent of national language, hardcoding TEXT and LABEL attributes into the GUI itself is not a good approach. You must then change these attributes for all affected parts through your RPG code when starting the application—for example, for the CREATE event of the window. Therefore, you would have to maintain different versions of your RPG code for every language you intend to support.

The better alternative is to use label substitution instead. This can be specified for any TEXT and LABEL attribute by coding a caret sign (^) followed by a free label name. In Figure 106, the originally specified title of the window is replaced by the substitute label ^Title, which is defined through the properties notebook.

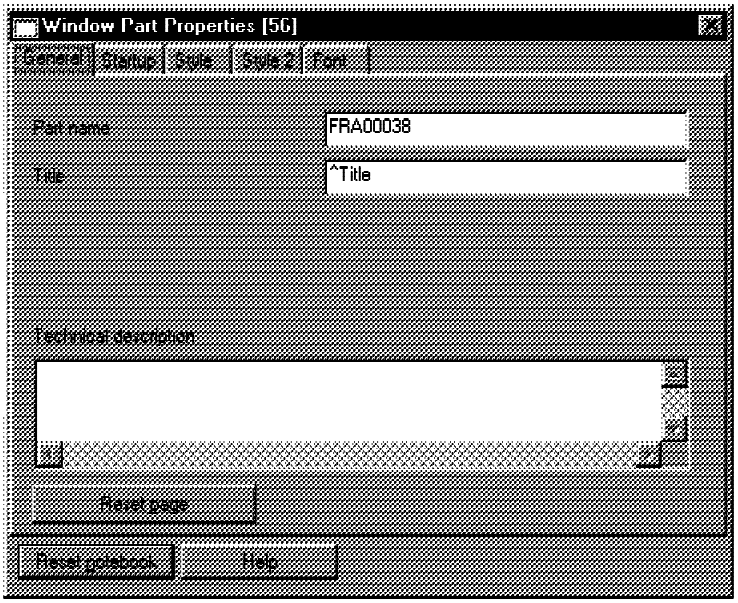


Figure 106. Window Part Properties Notebook, General Page

All parts that support TEXT and LABEL attributes and that allow you to specify them through their properties notebook such as windows, static text, push and radio buttons, check and group boxes, notebook pages, and menu items also allow you to use label substitutions instead.

Label Names

When coding for label substitution, always use label names that reflect the part they are used for. It is good coding style to use the real attribute value of a part as its label name. For example, the label substitution name for the *Close* push button could be ^Close. Figure 107 shows the Container example using label substitution for all TEXT and LABEL attributes.

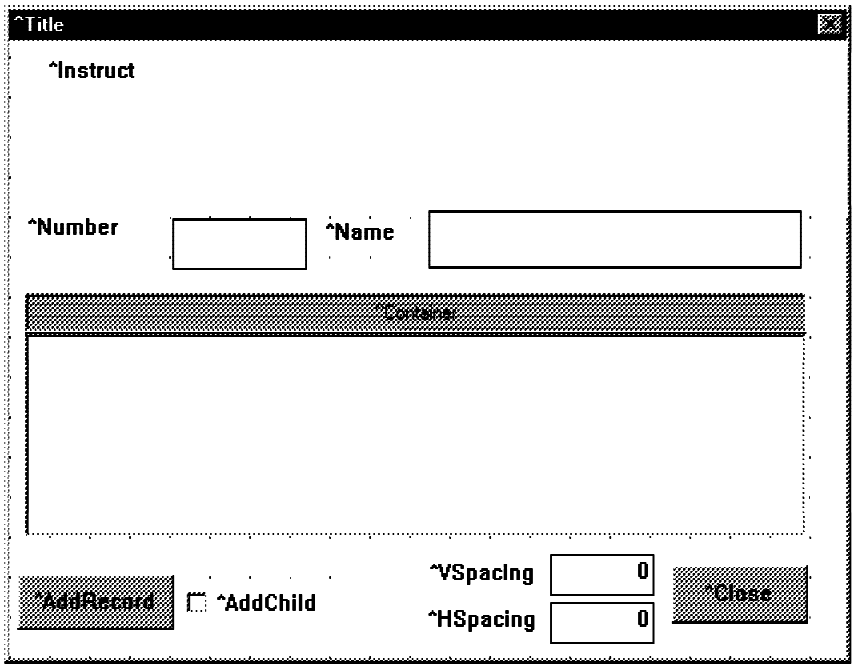


Figure 107. Container Example with Label Substitution

While designing the GUI, keep in mind that the size of the values you want to put into TEXT and LABEL attributes differ significantly from one language to another. Therefore, make sure that the parts are large enough to hold the entire text or label for any of the supported languages. However, as this can result in too large a distance between an entry field part and a describing static text part, such as the VSpacing and HSpacing static text parts in the Container example. Take the additional step of making sure that the TEXT attribute is aligned to the right of the part (Figure 108).

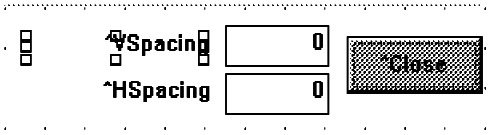


Figure 108. VSpacing and HSpacing Static Text Parts

Compiling and running this application now would result in a window showing the names of the label substitutes as their TEXT and LABEL attributes. To change this, open the *Define Messages* dialog. As shown in Figure 109, a label message has been generated for each of the label substitutions specified in the GUI.

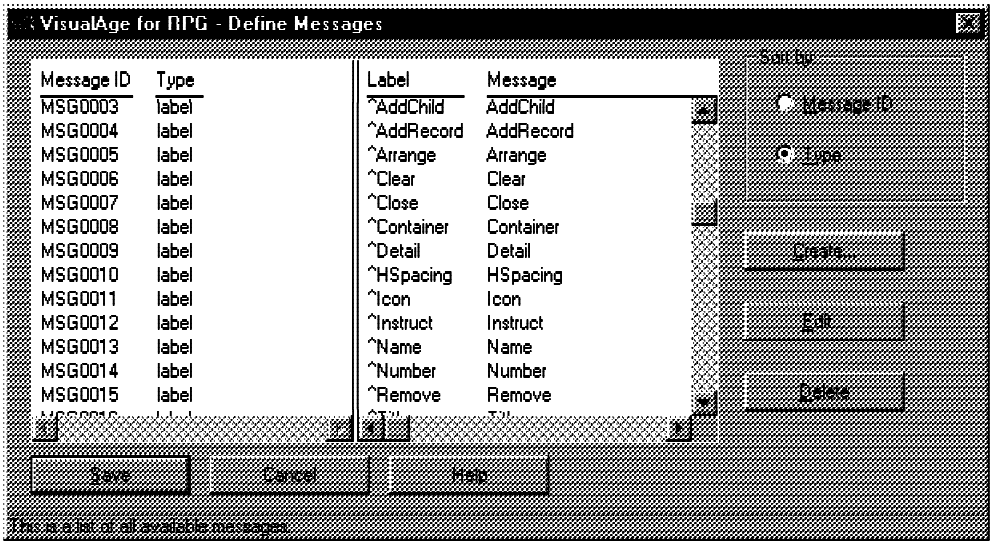


Figure 109. Define Messages Dialog

You can edit the message text for every label to show the correct value for the TEXT and LABEL attributes of all parts of the GUI. You don't need to go into any properties notebook to change these attributes. However, there are more benefits than this when using label substitution instead of just hardcoding static text, and they are discussed in “Message Versions.”

First, lets look into another area, where hardcoding might prevent you from developing a language-independent application: the RPG source code.

RPG Constants and Literals

It may sometime be necessary to change the TEXT and LABEL attribute for a certain part within your RPG source. For example, if you would like to insert an information area that gives the user help information for a part the mouse pointer is pointing to, you must code an action subroutine for the MOUSEMOVE event:

```
RPG
C     EXIT          BEGACT    MOUSEMOVE     MAIN
 *
C                   eval      %setatr('MAIN':'INFOLINE':'Label')
C                              = 'Press this button to exit the '
C                                + 'application.'
 *
C                   ENDACT
```

This is burdensome in an NLS environment. You can easily replace the literal by a constant or variable name defined in the D-Specs, but you must still maintain a different version of your RPG source for every supported language.

Using Messages

You can get around this difficulty by using messages. First, you define an appropriate message through the *Define Messages* dialog and then assign it to the attribute of the part as follows:

```
RPG
C     EXIT          BEGACT    MOUSEMOVE     MAIN
 *
C                   eval      %setatr('MAIN':'INFOLINE':'Label')
C                              = '*MSG0001'
 *
C                   ENDACT
```

Using this coding technique makes your RPG source independent of the language. You then have to change the message text through the *Define Message* dialog, where you already placed all your substitute labels.

The same approach should be used for the DSPLY opcode. Never use the MSGTEXT keyword when defining a message variable in the D-Specs:

```
RPG
D MsgTxt          M                   MSGTEXT('Value is invalid')
D OK              M                   STYLE(*HALT)
D                                     BUTTON(*OK)
 *
C     MsgTxt        dsply     OK            rc                9 0
```

Instead, define a message for that literal and specify the MSGNBR keyword:

```
RPG
D MsgTxt          M                   MSGNBR(*MSG0001)
D OK              M                   STYLE(*HALT)
D                                     BUTTON(*OK)
 *
C     MsgTxt        dsply     OK            rc                9 0
```

Constants

If you need to define constants in your program and have them appear to the user, place the D-specs into a /COPY member and prepare separate versions for every supported language. You can then use the conditional compile feature of VisualAge for RPG to have the proper /COPY member included. The coding is as follows:

```
RPG
D/IF DEFINED(ENGLISH)
D/COPY I:\VARPG\MyConst.ENU
 *
D/ELSEIF DEFINED(GERMAN)
D/COPY I:\VARPG\MyConst.DEU
 *
D/ELSE
D/COPY I:\VARPG\MyConst.DFT
 *
D/ENDIF
```

On the *Build Options* dialog (see Figure 110) you can now specify the appropriate condition name through the *User defines* entry field:

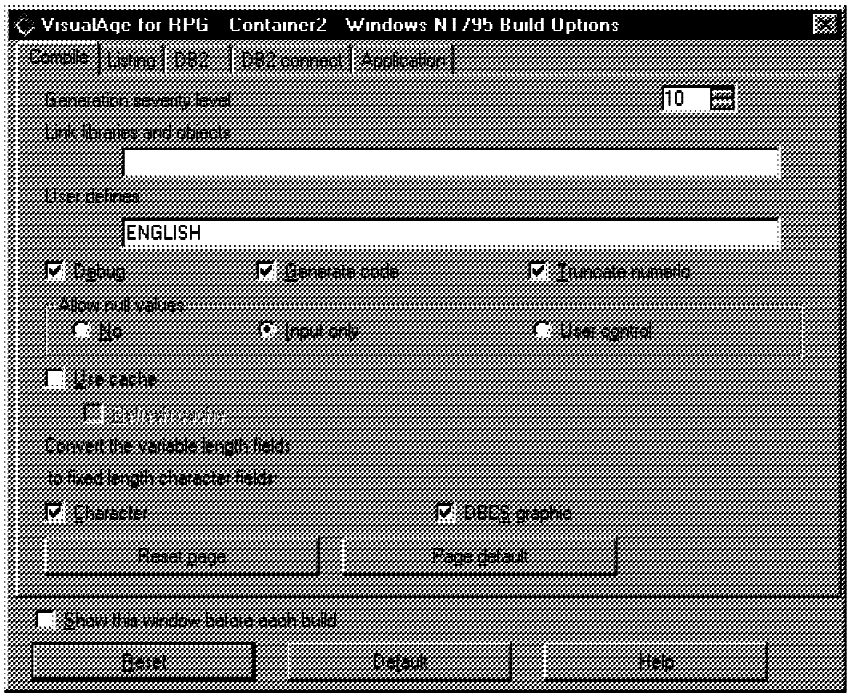


Figure 110. Build Options Dialog, Compile Page

Message Versions

Now that we have converted much of the former static text to messages, let's see how to get multiple versions of our messages.

All messages defined through the *Define Messages* dialog, regardless of whether they are used as a substitution label, for the DSPLY opcode, or for a message subfile, are stored in a single ASCII file in your project. The file, which has an extension of .TXM, can be found in the source directory of your application (see “Help Text” for information regarding the handling of the second-level help text for messages). For the modified Container example, the content of this message file looks like the list in Figure 111.

Flat file

```
MSG
MSG0001P: Employee Number must be specified
MSG0002P: Employee Name must be specified
MSG0003I: as Chil&d
MSG0004I: &Add Record
MSG0005I: &Arrange
MSG0006I: &Clear
MSG0007I: &Close
MSG0008I: VARPG Container Example
MSG0009I: &Detail
MSG0010I: HSpacing
MSG0011I: &Icon
MSG0012I: Enter data in the entry fields below, and press Add ...
MSG0013I: Name
MSG0014I: Employee Number
MSG0015I: &Remove
MSG0016I: VARPG - Container Example
MSG0017I: &Tree
MSG0018I: &Tree line
MSG0019I: VSpacing
MSG0020I: &View
```

Figure 111. Message File for the Container Example

To reproduce this message file for another language, create a copy of it first. You can then use an ASCII editor to key the translated message text into the original .TXM file. However, make sure that the sequence of the messages in the file remains unchanged.

It is probably a better idea to copy the .TXM file to something like xxxENU.TXM (to reflect it contains the messages in English language) and then modify the message text in the original .TXM file through the *Define Messages* dialog for whatever language you want to support (see Figure 112). You can make another copy of this changed version of the .TXM file afterward, such as xxxDEU.TXM if it contains the German version.

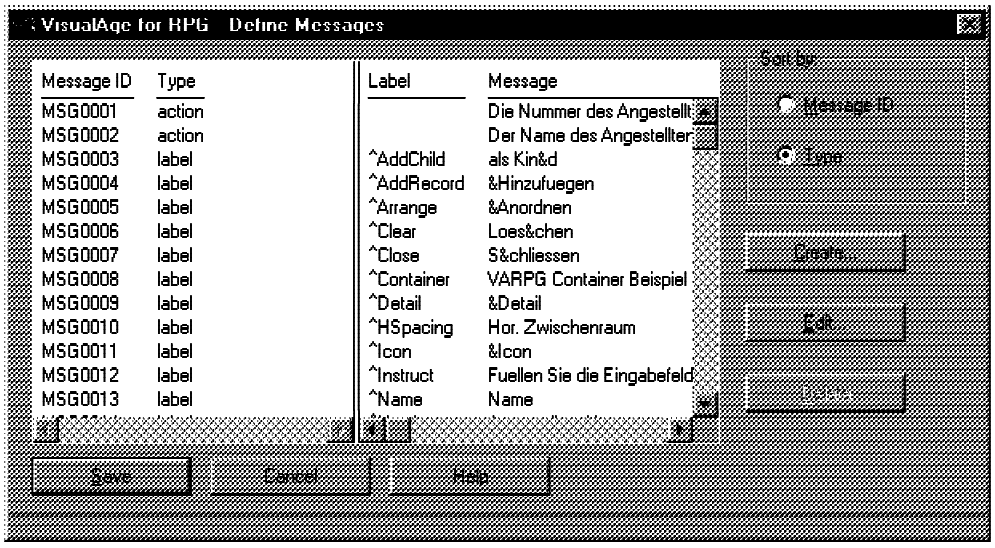


Figure 112. Defining German Versions of the Messages

While translating the label messages for parts that have a mnemonic key assigned (by specifying an & in front of the character representing the mnemonic), try to keep the same characters for all supported languages. This makes your application more consistent. This is especially important if you are developing a multinational application (refer to “Multilanguage Application” for further details).

During compilation, the only thing that happens to messages is that the .TXM file is copied from the source directory of your application to the RT_WIN32 or RT_WIN subdirectory. So, for example, if you compile the Container example with the English message file active (*ContaENU.TXM* renamed to *Containe.TXM*), the GUI looks like the original version shipped with VisualAge for RPG (refer to Figure 105).

However, copying the *ContaDEU.TXM* file with the German messages as *Containe.TXM* file into the RT_WIN32 subdirectory and calling the application again makes it look like Figure 113.

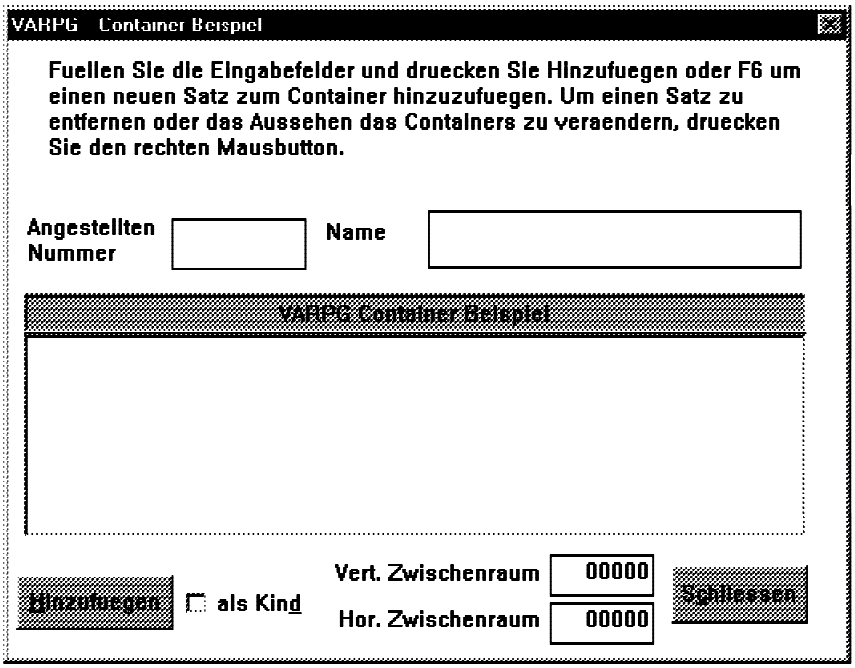


Figure 113. GUI in German Language for the Container Example

For any additional language you need to support, you need only add and maintain another .TXM file for your application.

Column Headings

There are two parts, the subfile and container part, that can have column headings for the contained columns. The only place where these column headings can be changed is the properties notebook for each of the two parts. Achieving an NLS-sensitive solution is still doable.

Let's look at the detail view of the Container example as shown in Figure 114. This view can be selected by the user through the pop-up menu of the container. Doing so adds a line of column headings below the container's title but above the records in the container.

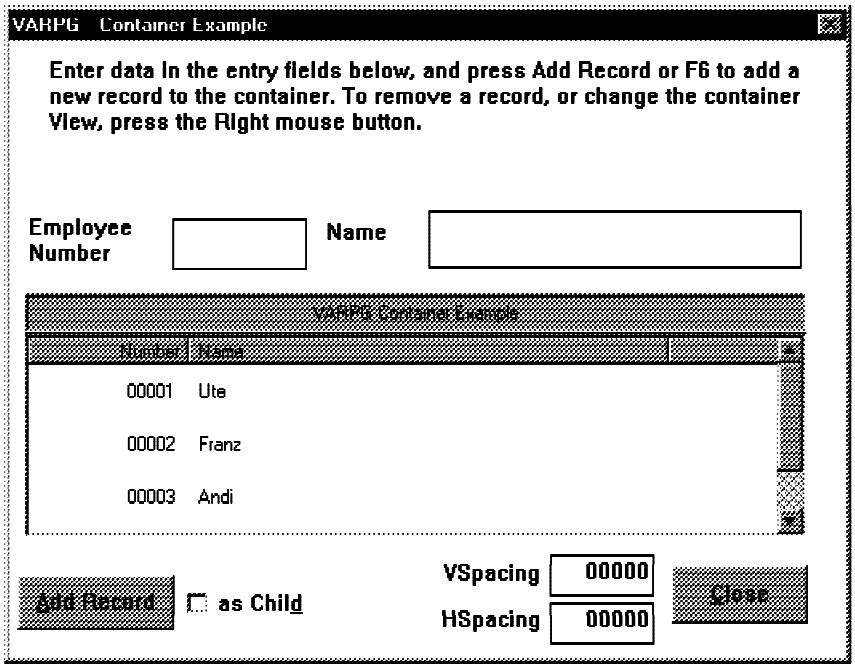


Figure 114. Detail View of Container Part

These headings always show the information specified on the *Properties* notebook of the container part. Figure 115 shows the properties for the second column of the container part in the Container example.

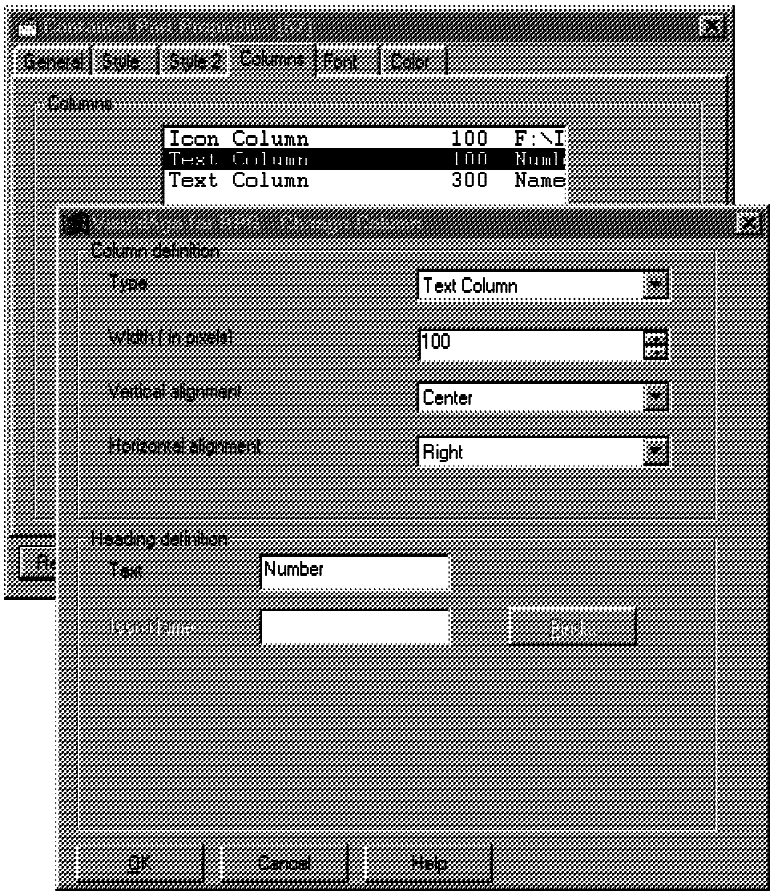


Figure 115. Properties of Column Number

To solve this problem, we add another static text part to the GUI of our application (Figure 116). We place it, where the column headings of the detailed view of the container part appear. The background and font have been changed, so that it resembles the original column headings.

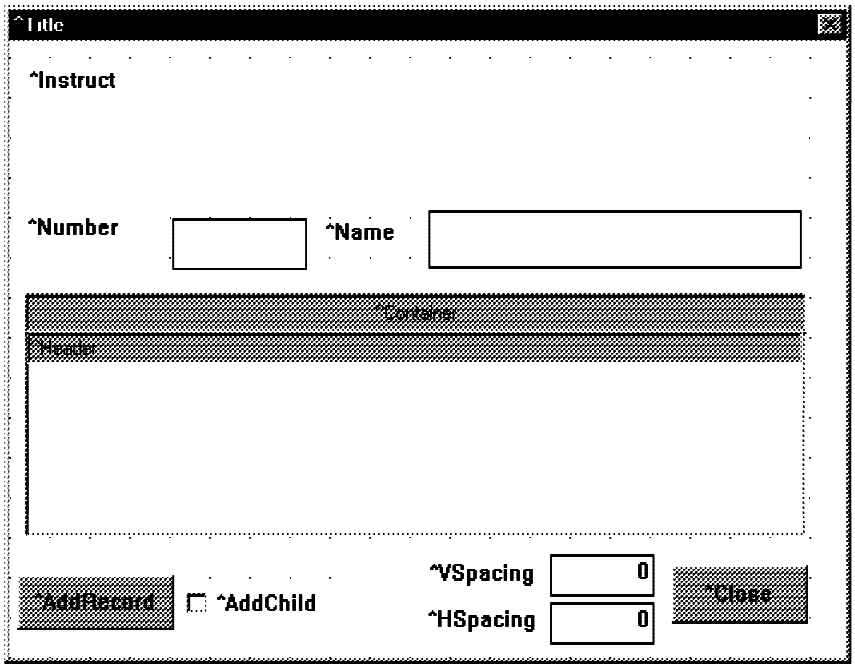


Figure 116. Adding a Static Text Part

Its LABEL attribute contains the label substitution ^Header, so that we will be able to maintain different headers for each language through the message file (see Figure 117 which shows the German version).

Note the vertical bar (|) character as the first character of the message. This prevents VisualAge for RPG from removing the leading blanks needed to adjust the column headings.

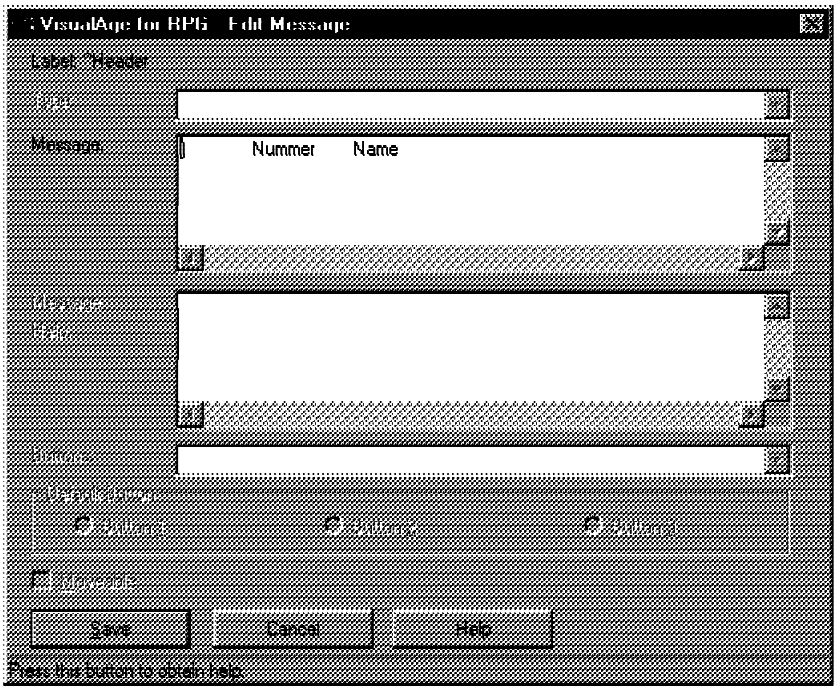


Figure 117. Change in the ^Header Substitution Label—Edit Message Window

There is still some work left. The column headings are displayed only for the detail view of the container. We need to deselect the VISIBLE attribute on the *Properties* notebook of the static text part and make it visible only if the user selects the *Detail* menu item from the pop-up menu:

```
RPG
C     MNI00043      BEGACT    MENUSELECT    FRA00038
 *
C     'CT1'         Setatr    3             'View'
C     'CT1HDR'      Setatr    1             'Visible'
 *
C                   ENDACT
```

It is important to have the VIEW attribute of the container-changed first. Otherwise, the static text part with the column headings will be overlaid by the container part. Additionally, we need to make sure that another event doesn't overlay the container the static text part. For example, selecting the *Remove* menu item from the pop-up menu of the container results in an update of the container part, which would overlay the static text. Therefore, at the end of these events, we need to update the static text part as well:

```
RPG
C     MNI00048      BEGACT    MENUSELECT    FRA00038
 *
C     'CT1'         Getatr    'FirstSel'    TmpID            6 0
C     'CT1'         Setatr    TmpID         'RemoveRcd'
C                   if        %getatr('FRA00038':'CT1HDR':
C                                       'Visible') = 1
C     'CT1HDR'      Setatr    0             'Visible'
C     'CT1HDR'      Setatr    1             'Visible'
C                   endif
 *
C                   ENDACT
```

Finally, if the user selects another view, we will switch the VISIBLE attribute off again.

Figure 118 shows the detail view of the modified Container example with the new column headings in German.

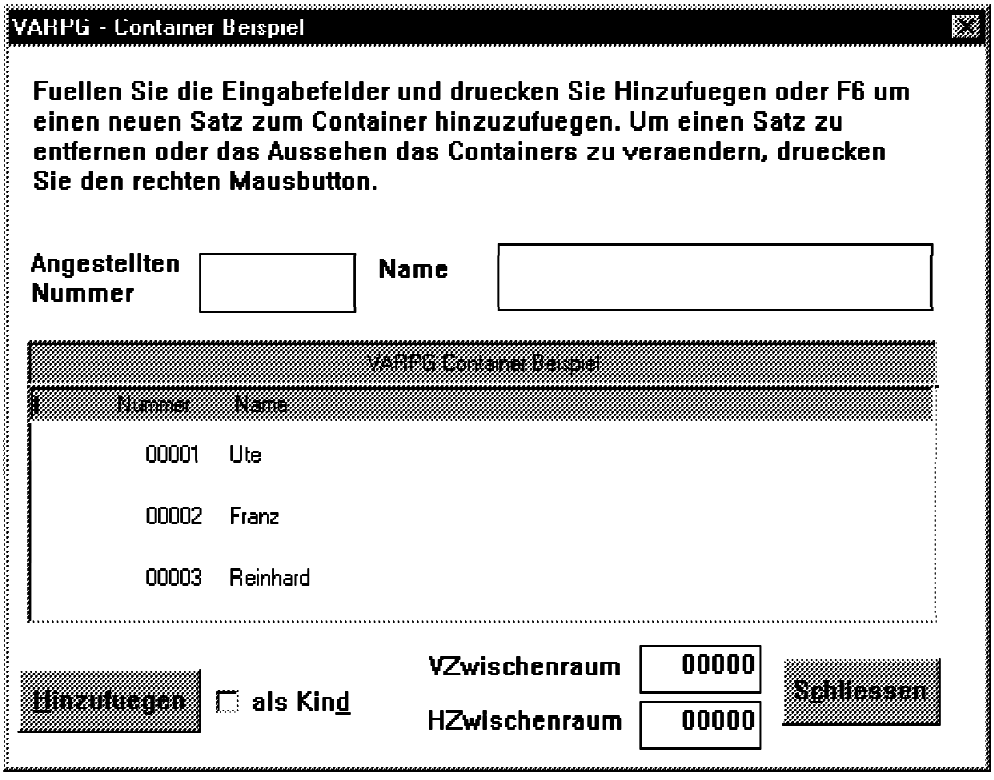


Figure 118. Modified Dialog View, German Version

Help Text

An integral part of an application is the help feature. Let's see how we can achieve to have help text in the same national language as the application itself.

If you have added some help text through the pop-up menus of the different parts of your window or the *Define Messages* dialog, VisualAge for RPG stores it into three different files, that are used to create a .HLP file:

.IPF The file with this extension contains all control information needed to create the .HLP file.

.VPF This file contains the stored help text that has been specified for the parts help.

.IPM This file stores the second-level help text stored for messages that were defined through the *Define Messages* dialog.

To create separate versions of help text for every supported language, make a copy of these three files that are stored in the source directory of your application (for example, use ContaENU to save the English version of these files).

Now, through the GUI Designer, change the help information of the original files to another language. Use the *Help text* menu items from the pop-up menus of the parts to change the help text to another language. For the second-level help text go to the *Define Messages* dialog.

After saving the project, make another copy of the .IPF, .VPF, and .IPM files to preserve the translated versions of these files.

You can also use an ASCII editor to make the necessary changes, but if you do, be careful to preserve the references of the different help text paragraphs to their corresponding parts of the GUI. In the .IPM file, do not change any information in header lines such as

IPF

```
:h1 res=99.MSG0001
```

A change would corrupt the connection between the message identifier and the second-level text associated with it.

In the .VPF file, do not change or remove the heading tags or the resource ID. These are essential if VisualAge for RPG is to find the correct help for parts of the graphical user interface.

An English IPF entry and the corresponding German text IPF entry is shown here:

IPF

```
.*
:h1 res=57.List of added items
.*
:p.Press the right-hand mouse button to get a pop-up menu with all
available operations you are allowed to perform on the container and
its items.
```

IPF

```
.*
:h1 res=57.Liste der hinzugefuegten Saetze
.*
:p.Druecken Sie den rechten Mausbutton um ein Menu mit allen
Funktionen zu bekommen, die Sie auf dem Container bzw. den
enthaltenen Saetzen ausfuehren duerfen.
```

The text immediately following the `res=` tag should be translated, as it gets presented to the user as the header information for the help text (refer to Figure 119).

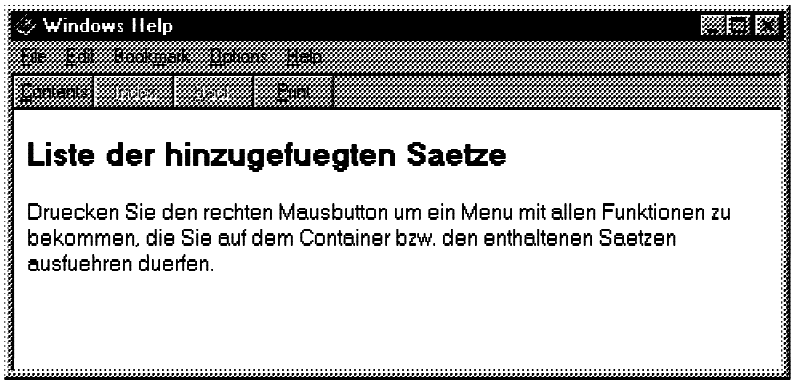


Figure 119. Help Text for the Container Part—German version

Building the Help File

Now, where we have edited the sources needed to build the .HLP file, how do we actually get it created?

There are two options to choose from:

- Change the names of the .IPF, .IPM, and .VPF files of the language for which you want to create the .HLP file back to their original names and invoke the build from the GUI designer.
- Perform only those steps of the build process manually, that are needed to create the .HLP file.

For the second option, consider the following sample batch file,

Batch file

```
d:\adtswin\system\ipfxlate.exe imbed %1.ipf    imbed.ipf
d:\adtswin\system\ipfxlate.exe rtf   imbed.ipf %1.rtf
d:\adtswin\system\hcrtf.exe    -xn   %1.hpj
```

Here, d: is the drive on which you installed VisualAge for RPG and %1 is the name of your application. You need to rename the language-specific .IPF, .IPM, and .VPF files to the applications defaults before calling this batch file. A new .HLP file will be created. Copying it to the RT_WIN32 subdirectory will make it the active .HLP file during run time.

Multilanguage Application

Up to now, we have learned how to create an application for a specific language, depending on the source files we specify for the build process. But what about a multilanguage application?

Assume you want to create an application that first displays a window in which the user can choose a language and then shows information to the user in the chosen language. For example, an airport information terminal that serves travelers from all over the world should be able to direct travelers to different facilities of the airport in any of several languages.

To make our Container example a multilanguage application, we first create this starting window. It could, for example, look like Figure 120, where flags of the different countries are used to allow the user to identify the preferred language.

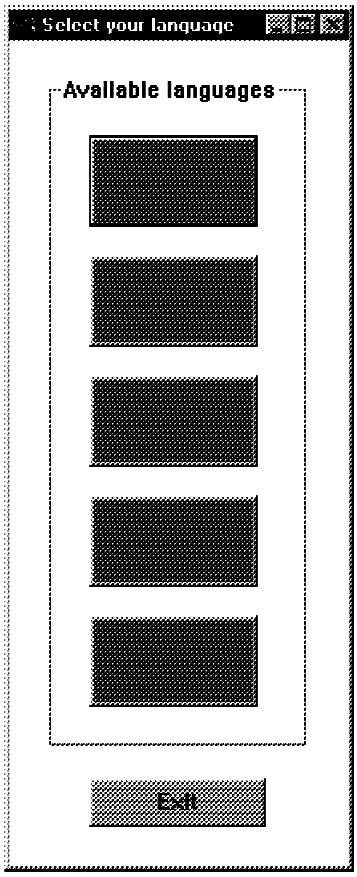


Figure 120. Window to Select a Language

If one of the graphic push buttons is pressed, the window should disappear from the screen, while Container example window *FRA00038* is shown, displaying all information in the chosen language. To achieve this, we begin by deselecting the *Open Immediately* check box from the *Properties* window of the Container example. We will then add the following action subroutine for the PRESS event of all graphic push buttons (the English version is shown here):

```
RPG
C     ENGLISH       BEGACT    PRESS         MAIN
 *
C                   eval      %setatr('*COMPONENT':'*COMPONENT':
C                                     'MsgFile') = '.\\ContaENU.TXM'
 *
C                   callp     ChgHlpFile('ENU')
 *
C                   eval      %setatr('MAIN':'MAIN':'Visible') = 0
 *
C                   showwin   'FRA00038'
 *
C                   ENDACT
```

The MsgFile attribute of the *COMPONENT part is used, to have the VisualAge for RPG runtime environment search in the specified message file for messages that couldn't be found in the .TXM file that has the same name as the application.

As we want the VisualAge for RPG runtime to find all messages in the specified alternate .TXM file, we provide an empty original .TXM file. This needs to hold just one line containing the keyword MSG. For every supported language, we also provide an alternative .TXM file with all messages used by the application.

Depending on the pressed image push button, we then switch among these different alternate .TXM files.

We also prepare different versions of the .HLP file, one for every supported language. To make sure the help texts are displayed from the correct .HLP file, invoke the procedure *ChgHlpFile* with the following prototype:

```
RPG
D ChgHlpFile      PR                  CLTPGM('.\\CHGHLPF.BAT')
D   PartName                    3A    VALUE
```

Behind this procedure stands a simple batch file, which deletes the current version of the application's .HLP file and recreates it using the version in the chosen language:

Batch file

```
@echo off
@del CONTAINE.HLP >nul
@copy CONTA%1.HLP CONTAINE.HLP >nul
@echo on
```

Finally, if the *Close* push button is selected here, the application should not end, but return to the language selection window. Therefore, we do not set LR to *ON, but close the Container example window and make the menu selection window visible again, as follows:

RPG

```
C     PSB0003B      BEGACT    PRESS         FRA00038
 *
C                   clswin    'FRA00038'
 *
C                   eval      %setatr('MAIN':'MAIN':'Visible') = 1
 *
C                   ENDACT
```

Chapter 8. Managing Projects

In this chapter, we show you more about VisualAge for RPG projects, their structure, and how they can be manipulated using the project organizer and the available tools. We also show you how to deal with nonvisual components. Finally, we offer you some guidelines you can use if you are developing applications together with other programmers and you want them to share their work.

Project Structure

Regardless of whether you are developing a VisualAge for RPG application or a component, VisualAge for RPG always refers to the development as a *project*. This term is basically a virtual folder, which holds all the associated files logically together and defines the actions and tools available for the entire project or parts of it.

Creating a Project

A project is created whenever you save the development environment for a VisualAge for RPG application for the first time using the *Save as Application* dialog (see Figure 121). You assign a name to it and specify whether you want to have a VisualAge for RPG application created or a VisualAge for RPG component.

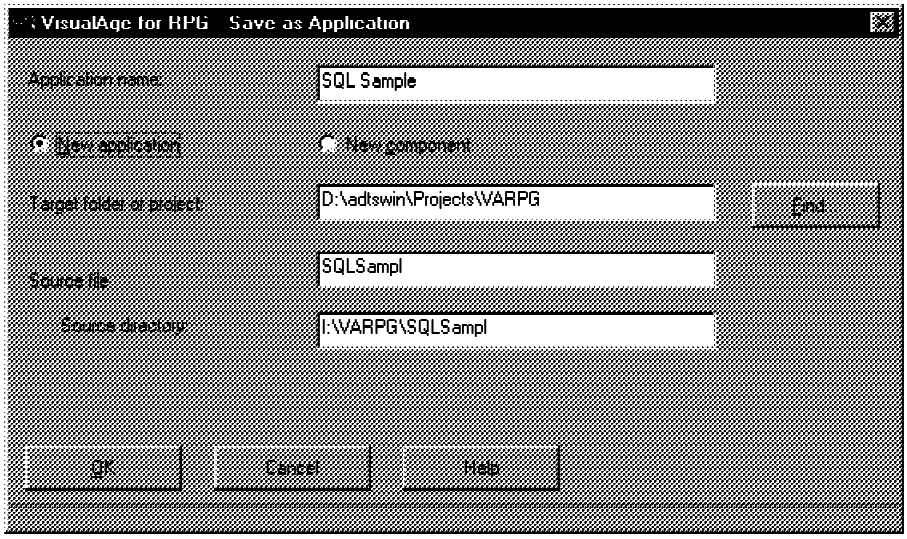


Figure 121. Save as Application Dialog

If you select the *New component* check box, a VisualAge for RPG dynamic link library is created as the target object of the build process, which may be called by other VisualAge for RPG components. However, if you select the *New application* check box, an EXE file will be created in addition to the VisualAge for RPG DLL, allowing you to invoke the DLL and pass parameters to it.

Source Files

To hold all the different source files of the project, a source directory with the name specified in this dialog is created. The following list shows the possible file extensions and a brief description of their content: (All files have the same source file name as specified in the corresponding entry field of the *Save as Application* dialog.)

.VPG This file contains all VisualAge for RPG source code you have coded.

.TXM This file contains the predefined messages.

.ODF This file contains the information about the windows and the contained parts of your application. It also contains the link information to the associated action subroutines and is used to create the .ODX file.

.TXC This file contains the text that was added as technical description in the properties notebook of the different GUI parts.

.IPF This file contains the control information needed to create the .HLP file using the .IPM and the .VPF files.

.IPM This file contains the second-level help text specified for the predefined messages.

.VPF This file contains the context-sensitive help information for the different parts of the GUI.

.TRC Whenever a problem occurred during the last build process, this file contains the executed commands and all resulting messages.

.EVT This file contains feedback information about the compiler, which is displayed in the *Error list* window.

.LST This file contains the compiler listing created on the last build.

.RC There can be several of these resource files. They are used to create the links between the context-sensitive help and the different parts of the application or component.

Runtime Files

Additionally, two subdirectories are added to the project's source directory to hold the runtime files of the application or component. Subdirectory RT_WIN32 holds the files created by a Windows 95 or Windows NT build, while RT_WIN contains the Windows 3.1 version of the files, after the corresponding build is performed. The following list shows the possible file extensions and their meaning:

.DLL This file contains the executable code for the VisualAge for RPG application or component.

.EXE The .EXE file is created for a VisualAge for RPG application only and allows invoking the application's DLL.

.ODX The .ODX file contains all the information about the GUI of the application or component and the links to the associated action subroutines.

.HLP This file contains the context-sensitive help information as well as the second-level help text for predefined messages.

.TXM This file contains the information on all messages defined through the *Define messages* dialog.

.RST The .RST file contains all the server aliases, file overrides, data area overrides, program overrides, and lock level information you define for your application using the *Define AS/400 information* notebook.

.BND The .BND file is created if your application contains embedded SQL statements and the *Bind file* check box on the *DB2* page of the *Build options* notebook is checked. It contains all information necessary to perform a bind to the DB2 database to be accessed.

Project File

All the information specified in the *Save as Application* dialog is saved into a project file with the project name as file name and an extension of IVG. This is now used as an entry point to the project and all of the available VisualAge for RPG utilities require this file in order to find their way to the different parts of the project. For example, if you would like to load a certain VisualAge for RPG application or component into the GUI designer, the *Open Component* dialog expects you to specify the corresponding project file (see Figure 122).

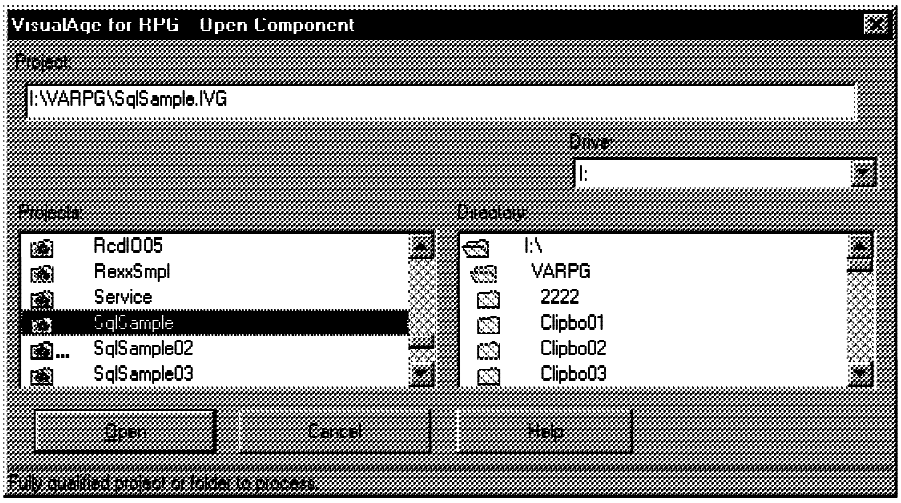


Figure 122. Open Component Dialog

As you can see in the sample IVG file below, the build options for Windows 95 or Windows NT *buildOpts* and Windows 3.1 *buildWinOpts* are also included in this file. Therefore, do not touch this file. If you should corrupt its content, the VisualAge for RPG utilities may be unable to read it any more. For example, if you split the build option lines, so that they span more than one line, the continuations are ignored when loading the project. The sample IVG file is as follows:

Project (IVG) file

```
-VRPG300ProjectInfo
sourceDir=I:\VARPG\SqlSampl
targetPgm=SQLSampl.EXE
buildOpts=0 /GL 10 /Ti /RN /RT /SVC /SVG /L /LX /LV /LC /LD /LE ...
buildWinOpts=0 /TW /GL 10 /Ti /RN /RT /SVC /SVG /L /LX /LC /LD ...
```

Where the IVG file is physically stored depends on the type of object you have specified in the *Target folder or Project* entry field of the *Save as Application* dialog. If an ordinary Windows 95 or Windows NT folder is selected, the IVG file is put into the corresponding drive or directory.

However, if you specify another project here, the IVG file is stored in the source directory of that project. The saved project then shows up as a subproject of the enclosing project.

Using this method of saving a project allows you to group different projects together. You can organize complex applications into project hierarchies,

which give you a visual perspective of how your code is structured (see Figure 123). It also enables the different VisualAge for RPG utilities (refer to Table 6) to take care of these subprojects, if they perform an action on the enclosing project.

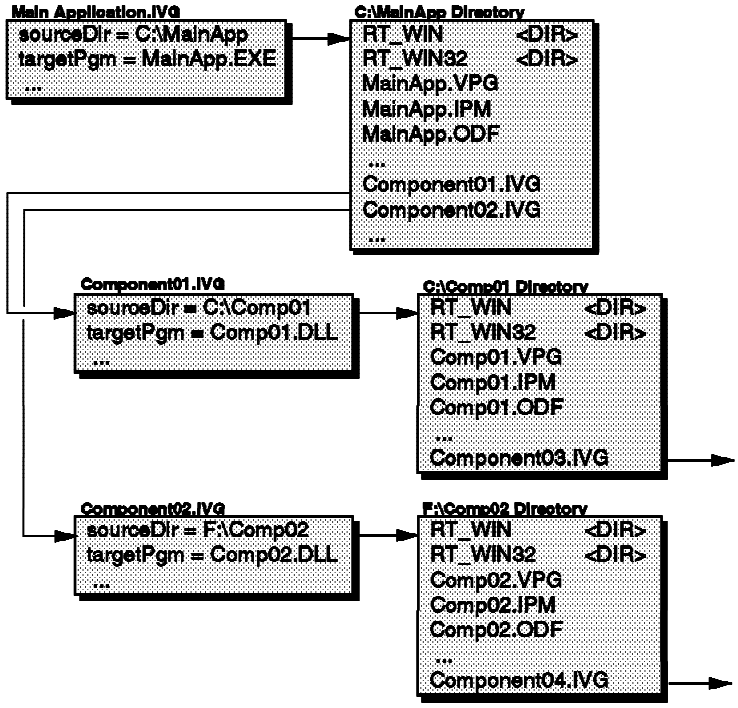


Figure 123. Complex Project Hierarchy

Only the project file of the component is stored into the source directory of the enclosing application or component. The source as well as the runtime files of the subcomponent remain in their own source directory structure.

Utilities and Actions

VisualAge for RPG offers a large variety of utilities, that allow to perform actions on the different parts of your projects. They all have in common that they know how to read and interpret the content of a project's IVG file and keep the structure of a project consistent when they work with its parts.

For example, if you would like to rename the source directory of your project, it is not sufficient to use the *Rename* function provided in the Windows 95 or Windows NT explorer, or execute the RENAME command in a DOS prompt. As these tools won't update the content of the project's IVG

file, the project structure would be destroyed. The GUI Designer would no longer be able to locate the files of the project. The *Rename Utility* of VisualAge for RPG, however, takes care of these dependencies and makes all necessary changes.

Utility Description

In Table 6, all available VisualAge for RPG utilities as well as the actions they perform are listed. Additionally, hints are provided about their usage.

Table 6 (Page 1 of 5). VisualAge for RPG Utilities

Utility	Action(s)	Additional Considerations
GUI Designer	This is the central utility when developing VisualAge for RPG applications. It can be used to design the GUI of your application and incorporates many other utilities to create and manipulate the different parts of a VisualAge for RPG project.	The *Save As* item of the *Project* menu can be used to combine a duplicate and a rename of the currently loaded project.
Project Organizer	The Project Organizer gives you a graphical view of your project structure. It also allows the invocation of many of the other VisualAge for RPG utilities.	
LPEX Editor	The LPEX editor is used to edit or browse the source code of your application or component. Additionally, you can browse the compiler listing and maintain the IPF source of your help information.	
Build	All runtime files of your application/component are created. The build utility invokes the VisualAge for RPG as well as the IPF compiler. If all parts have been created successfully, the files are placed into the proper runtime subdirectory of the project.	The runtime files are also copied to the TEST subdirectory of the ADTSWIN directory. Refer to the *Run* and *Debug* utility entries for additional information.

Table 6 (Page 2 of 5). VisualAge for RPG Utilities

Utility	Action(s)	Additional Considerations
Error list	The .EVT file of the project, which was created by the VisualAge for RPG compiler, is read and its content is displayed. A double-click on a listed message displays the corresponding line in the LPEX editor.	
Run	This utility allows the execution of a VisualAge for RPG application on a development workstation without having the VisualAge for RPG runtime installed.	The runtime files in the project's RT_WIN32 (or RT_WIN) subdirectory are used by this utility. For any component that is started by the application, the runtime files in the TEST subdirectory of the ADTSWIN directory are used.
Debug	The selected VisualAge for RPG application can be debugged. Other components that get started by the application can be added by the active utility.	The runtime files in the project's RT_WIN32 (or RT_WIN) subdirectory are used by this utility. For any component that is started by the application, the runtime files in the TEST subdirectory of the ADTSWIN directory are used.
Duplicate	The project's IVG file is duplicated into the specified folder or project and all source and runtime files are copied to a new source directory.	Only the project and source directory name are altered for the duplicated project. The file names remain unchanged.
Rename	Any of the following attributes of a VisualAge for RPG projects can be changed with this utility: • the project name • the file name • the source directory name	Do not rename a project that is currently loaded into the GUI Designer.

Table 6 (Page 3 of 5). VisualAge for RPG Utilities

Utility	Action(s)	Additional Considerations
Delete	All source and runtime files of a project as well as its IVG file are deleted.	This includes the IVG file of any subproject. However, the associated source and runtime files are not deleted. Therefore, in this case, it is better to delete the subprojects, first. Not included are the versions of the runtime files that were copied to the TEST subdirectory of the ADTSWIN directory. Do not delete a project that is currently loaded into the GUI Designer.
Check-In	A VisualAge for RPG project is checked in as a part into an Application Development Manager/400 project hierarchy. Any ADM/400 part lock is released.	See “Share Your Work” for further details.
Check-Out	A VisualAge for RPG project is checked out from the Application Development Manager/400 project hierarchy it was previously checked into. The corresponding ADM/400 part is locked to prevent other users from checking out the same ADM/400 part as well.	See “Share Your Work” for further details.
Extract	A VisualAge for RPG project is checked out from the Application Development Manager/400 project hierarchy it was previously checked into without being locked.	See “Share Your Work” for further details.

Table 6 (Page 4 of 5). VisualAge for RPG Utilities

Utility	Action(s)	Additional Considerations
Backup	The source and runtime files of a project are compressed and saved as members of either a database file or a source physical file in a specified library on the AS/400.	The name for both files corresponds to the name specified for the files of the project with an additional ending of 'DF' for the database file (binary format) and 'SF' for the source physical file (text format). All subprojects included in a project selected for backup are also saved into the same library on the AS/400. Each will result in a separate set of the two AS/400 files.
Restore	A previously saved VisualAge for RPG project is received from the AS/400 and its entire structure is restored on the workstation.	The project might be restored to a different workstation or to a different directory on the same workstation. The new location can be specified during the restore. Possible subprojects must be restored separately.
Application Packaging	The files in the RT_WIN32 or RT_WIN subdirectory of a project as well as the runtime files in the corresponding subdirectory for any subproject are packaged to a diskette or into a directory. Optionally, the VisualAge for RPG runtime code is packaged as well.	It is also possible to package only the VisualAge for RPG runtime. In this case, it is okay if no project name is specified in the appropriate entry field.
Define Server Logon	This utility allows defining user ID and password for different AS/400 servers. Each time an application needs to connect to one of the defined AS/400 servers, the specified user ID and password are used instead of prompting the user.	This utility is not only available on the development workstation, but is also shipped as part of the VisualAge for RPG runtime.

Table 6 (Page 5 of 5). VisualAge for RPG Utilities

Utility	Action(s)	Additional Considerations
Define AS/400 Information	This utility allows editing the content of the .RST file of a VisualAge for RPG application or component.	The utility is shipped as part of the VisualAge for RPG runtime and, therefore, might also be used to configure the application according to the environment it will run in.
Migration	The source files of a V3R6M0 or V3R1M1 project are migrated to a VisualAge for RPG project for V3R2M0.	Before migrating the source files, make sure they are all located in one directory. The migration utility will not convert files contained in any subdirectory. The directory must contain the .ODF file of the project.
Component Packaging	The files of a VisualAge for RPG project are compressed and packaged to the specified target diskette or directory. You can select, via check boxes, whether the runtime, the source, or both types of files should be included into the package.	See “Share Your Work” for further details.
Component Installation	The IVG file of a project previously packaged using the *Component Packaging* utility is added in the specified target folder or project. The source files are installed into the also-specified source directory.	See “Share Your Work” for further details.
User-defined Part Packaging	The selected user-defined part is packaged to the specified target diskette or directory.	See “Share Your Work” for further details.
User-defined Part Installation	The selected previously packaged user-defined part is added to the parts catalog as well as the parts palette of the GUI Designer.	See “Share Your Work” for further details.

Now that we know what utilities are available, let's see how they can be invoked.

Using Utilities

The first place to go is into the *Start* menu of Windows 95 or Windows NT (see Figure 124). A menu item *ADTS Client Server for AS/400* is added to the *Programs* menu during installation of the product. This in turn contains menu items for the different parts of the Application Development Toolset Client Server/400 license program. Selecting the menu item *VisualAge for RPG* opens a list of many of VisualAge for RPG's utilities.

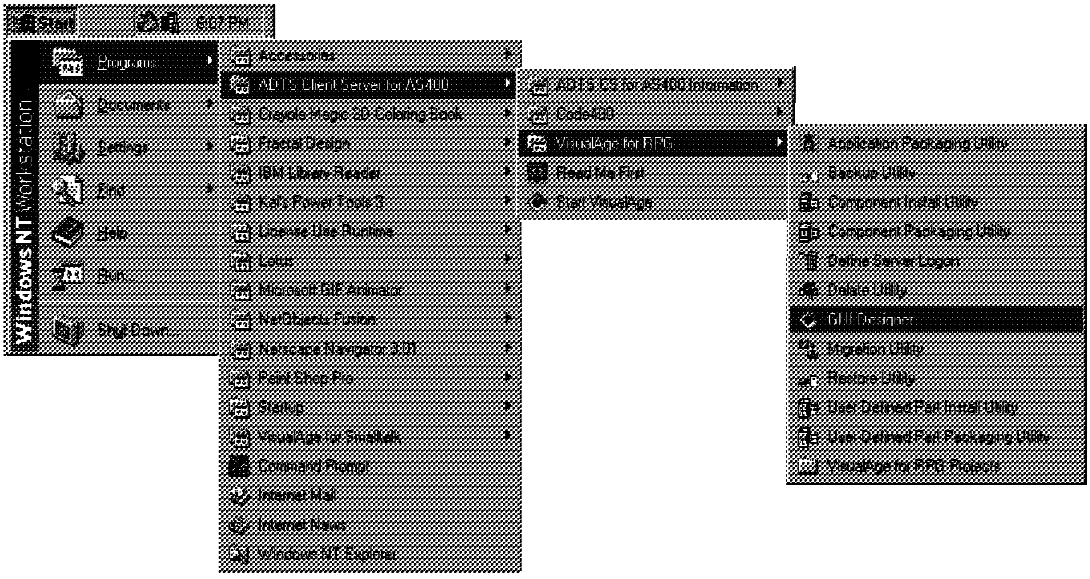


Figure 124. Start Menu Chain

This is the place, where you most probably will start the GUI Designer for the first time.

Taskbar

It is a fairly long way from the initial *Start* button to this last menu, so the Application Development Toolset Client Server/400 offers its own *Application bar* (see Figure 125), which has a look and feel similar to those of the taskbar of Windows 95 or Windows NT. Its *VisualAge for RPG* button opens a list with the same utilities.

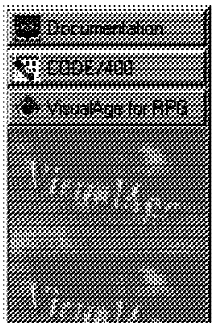


Figure 125. ADTS CS/400 Application Bar

To invoke this bar, select the *Start VisualAge* menu item from the *ADTS Client Server for AS/400* menu (see Figure 124) or start the BAR400.EXE program. Like the Windows 95 or Windows NT taskbar, you can move it to another side of your screen by selecting it with the left-hand mouse button and dragging it to the new location. Through its context menu, you can also specify whether the bar is to always be on top of any other window and whether it disappears as soon as it loses the focus.

Most of the utilities can also be invoked by selecting an action from the context menu of a project's IVG file (see Figure 126). When a menu item is selected, the corresponding utility is started, passing it the name of the IVG file. The *Edit* action, for example, starts the GUI Designer immediately loading the project it was selected for.

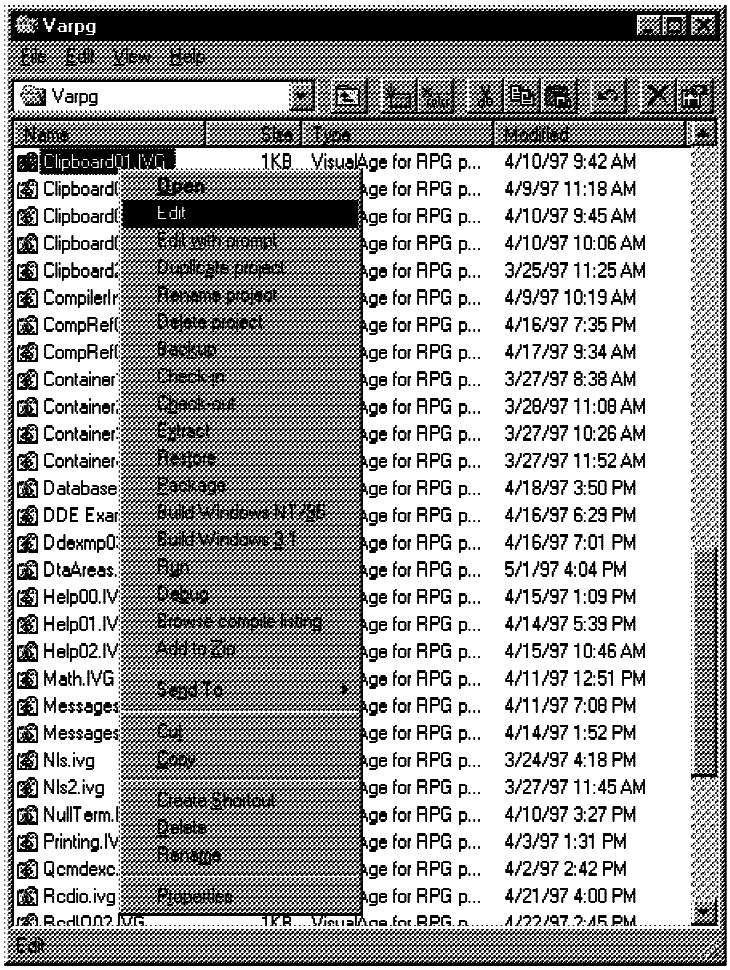


Figure 126. Context Menu of an IVG File

If you look at the *File Types* page of the *Options* notebook, which can be invoked via the corresponding menu item in the *View* menu of a folder or Windows 95 or Windows NT Explorer. The file type *VisualAge for RPG project* was added during the installation of VisualAge for RPG and represents the files with an extension of IVG.

Editing one of the actions defined for this file type shows you its command line interface (see Figure 127).

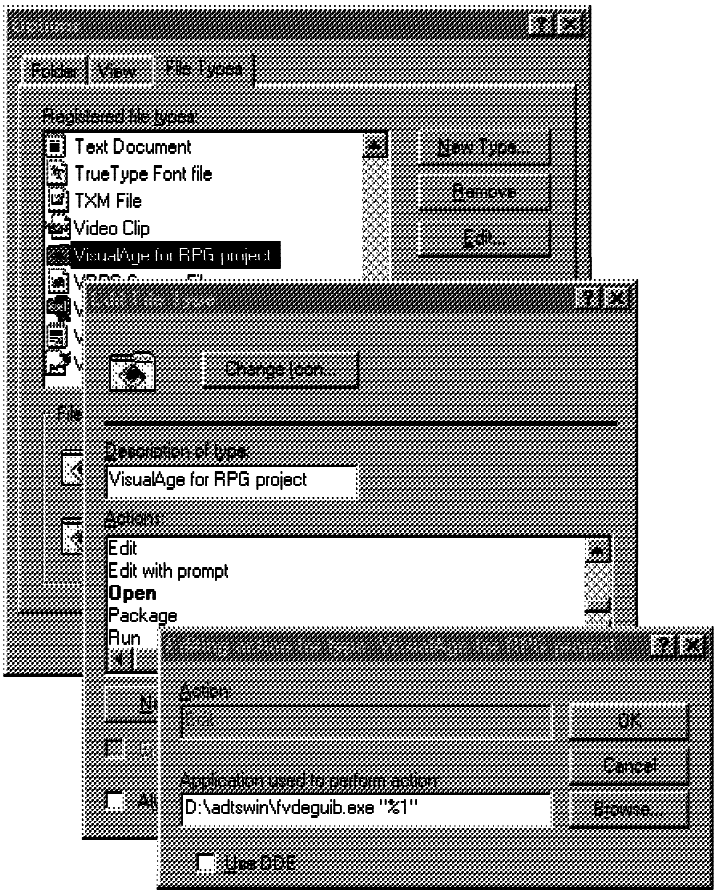


Figure 127. Edit an Action of the VisualAge for RPG Project File Type

If you select the *Open* action from the context menu of a IVG file, VisualAge for RPG's *Project Organizer* is started, which allows you to invoke many of the utilities through its *Project* menu.

Despite the fact that it also gives you a good overview of the structure of your project, it offers an option available nowhere else. The *Target type* menu item of its *Settings* menu (Figure 128) allows you to change the type of the project: VisualAge for RPG application or component.

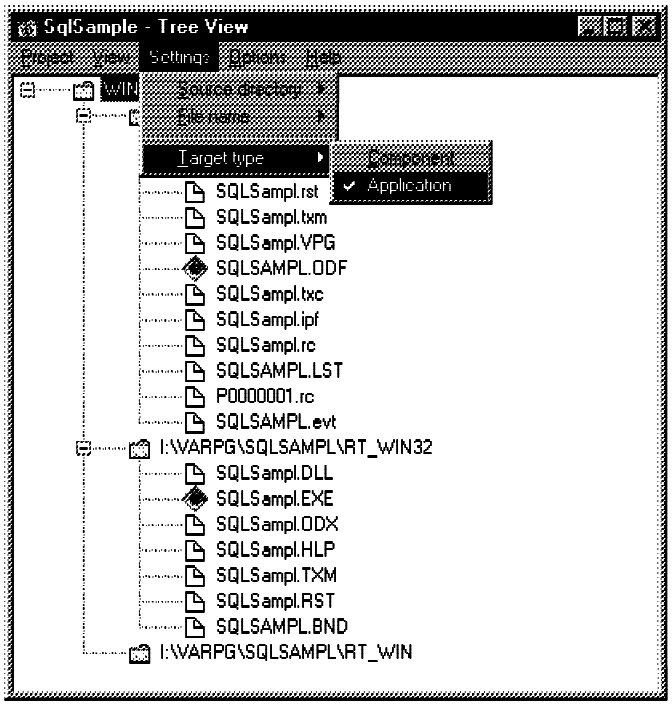


Figure 128. Changing the Type of a Project

If this or any other option has been grayed out, check whether the project has been loaded into the GUI Designer. As the project's IVG file is updated when the type is changed, the menu item is available only for a project not currently loaded.

Last but not least, the GUI Designer allows you to invoke the different utilities. Some are invoked implicitly, such as the editor, if an action subroutine should be coded. Others, such as the build, run, or debug utilities are executed explicitly by selecting the corresponding menu item from the *Project* menu or the toolbar item. Its *Window* menu even offers a *Project Organizer* option to invoke the *Project Organizer* for the currently loaded project.

There are many different places from which to invoke the VisualAge for RPG utilities. However, Table 7 offers you an overview.

Table 7 (Page 1 of 2). Invoking VisualAge for RPG Utilities

Utility	Start or Appl. Bar Menu	GUI Designer	Project Organizer	IVG Context Menu	Command line interface
GUI Designer	X	X	X	X	FVDEGUIB ProjName.IVG
Project Organizer		X		X	FVDEPORG ProjName.IVG
LPEX Editor	X	X	X		CODEEDIT FileName.Ext
Build 95/NT		X	X	X	FVDECPL ProjName.IVG
Build Win 3.1		X	X	X	FVDECPL ProjName.IVG /WIN
Error list		X	X	X	CODEEVNT FileName.EVT
Run		X	X	X	FVDEAPLN ProjName.IVG
Debug		X	X	X	FVDEAPLN ProjName.IVG /DEBUG
Duplicate			X	X	FVDEDUP ProjName.IVG
Rename			X	X	FVDEREN ProjName.IVG
Delete	X		X	X	FVDEDEL ProjName.IVG
Check-In			X	X	FVDEBKUP ProjName.IVG /ADMIN
Check-Out			X	X	FVDEREST ProjName.IVG /ADMOUT
Extract			X	X	FVDEREST ProjName.IVG /ADMEXT
Backup	X		X	X	FVDEBKUP ProjName.IVG
Restore	X		X	X	FVDEREST ProjName.IVG
Application Packaging	X		X	X	FVDIPACK ProjName.IVG
Def. Server Logon	X	X			FVDEPW
Def. AS/400 Information		X			FVDERST FileName.RST
Migration	X				FVDEMIG
Component Packaging	X				FVDEKAPK

Table 7 (Page 2 of 2). Invoking VisualAge for RPG Utilities					
Utility	**Start or Appl. Bar Menu**	**GUI Designer**	**Project Organizer**	**IVG Context Menu**	**Command line interface**
Comp. Install	X				FVDEKAIN
User-defined Part Packaging	X				FVDEKUPK
User-defined Part Install	X				FVDEKUIN

Nonvisual Components

When you are developing a nonvisual component and you have added either the EXE or the NOMAIN keyword in the control specification definitions of your VisualAge for RPG source file, then you can use the VisualAge for RPG utilities for a visual component.

Start the GUI Designer for a new project as usual, remove the predefined window part through its context menu (Figure 129), add the VisualAge for RPG source using the LPEX editor, and save the project for the first time. The source directory and IVG file are created, enabling you to perform the build process and invoke the *Run*, *Debug* or any other available utility.

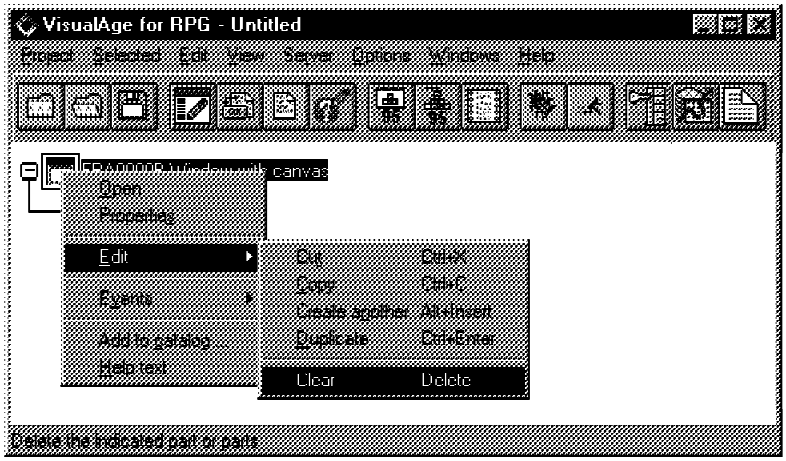


Figure 129. Removing the Default Window Part

The only problem when developing a nonvisual component in this environment is the build process. As you don't have a GUI, you need the .EXE file (keyword EXE) or the .DLL and .LIB files (keyword NOMAIN) only, so the compile step of the build process would be enough. Instead, the build creates the .ODX and .HLP files for you as well; this, though not an insuperable impediment, costs some extra performance.

To avoid the extra overhead, consider invoking VisualAge for RPG's compiler using its command interface. The file to be executed is called FVDFNFE.EXE, which resides in the SYSTEM subdirectory of the ADTSWIN directory. It expects the name of the .VPG file as a parameter, followed by the list of compiler options. Table 8 shows the available options and their meaning.

Table 8 (Page 1 of 2). Compiler Options

Option	Description
General compile options	
/GL nn	Specify the generation severity level. The range of errors is 1 to 99.
/BL filename	Specify the .LIB and .OBJ files, which contain all functions called by the component.
/D DefName	Define DefName as user; define before compilation.
/d or /TI	Generate debug information.
/RF	Fix invalid numeric data.
/RT	Truncate numeric values, if an overflow occurs.
/RN	Allow null-capable fields from AS/400 files for input-only access. This option mutually excludes /RNU. Omitting both means null-capable fields are not allowed.
/RNU	Allow null-capable fields from AS/400 files for read, write, and update access. Null values must be controlled by the program code. This option mutually excludes /RN. Omitting both means null-capable fields are not allowed.
/HCU	Enable caching.
/HCR	Enable cache refresh.
/SVC	Convert variable-length characters to fixed-length character variables.
/SVG	Convert DBCS graphic variables to fixed-length character variables.
/Tw	Generate a Windows 3.1 object.
Listing options	

Table 8 (Page 2 of 2). Compiler Options

Option	Description
/L	Create a compiler listing.
/LX	Create a field cross-reference list in the compiler listing.
/LV	Create a visual cross-reference list in the compiler listing.
/LC	Expand /COPY in the compiler listing.
/LS	Show skipped lines in the compiler listing.
/LD	Expand DDS in the compiler listing.
/LE	Show external references in the compiler listing.
/LM2	Show second-level message text in the compiler listing.
/LI '**'	Use this option to specify up to two character for indentation in the compiler listing.
/LP nn	Specify the lines per page for the compiler listing.
DB2 options	
/SN DBName	Specify the SQL database name
/SB [BndName]	Generate a SQL bind file (called BndName.BND).
/SP [PkgName]	Generate a SQL package (called PkgName).
/SF xxx	SQL format for date/time columns (EUR, USA, ISO, or JIS)
/SI xx	SQL isolation level (RR, CS, or UR)
/SR xxx	Perform SQL record blocking (NO, ALL, UNAMBIG).
DB2 connect options	
/SU UserID	User ID used to connect to the SQL database.
/SUP Password	Password used to connect to the SQL database.

Please note that the SYSTEM subdirectory of the ADTSWIN directory must be added to the LIB environment variable to be able to invoke the compiler from the command line. You get an error message LNK1094 about a missing FVDRNRT.LIB, if this has not been done.

For example, to generate the .LIB and .DLL file as well as a compiler listing for a NOMAIN utility DLL project *Service*, the command to be executed would look like this:

Command

```
D:\ADTSWIN\SYSTEM\FVDFNFE I:\VARPG\SERVICE\SERVICE.VPG /L
```

All files are created in the directory, the command is executed from. To avoid the path specification, you might add the SYSTEM subdirectory to your PATH environment variable.

The compile operation always ends without issuing any messages in the *MS-DOS Prompt* window. You have to look into the project's .EVT file for compiler feedback. However, as the .EVT file is the file that is read by the *Error list* utility (see Figure 130), you can look at its content using this utility. Beside the options mentioned in table Table 7, you can also invoke it from the *Windows* menu of the LPEX editor.

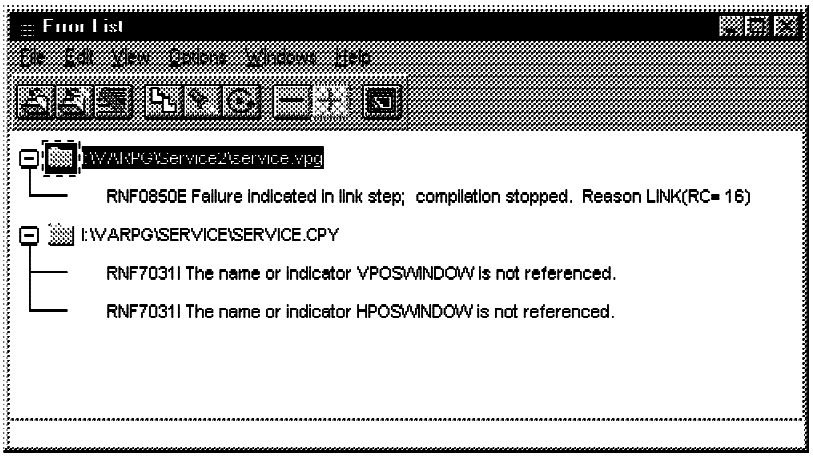


Figure 130. Error List Utililty

With LPEX, we have a highly programmable editor. Therefore, we can add the command-line invocation of the compiler as an additional item to one of its menus. The *Actions* menu might be the right place for this (see Figure 131).

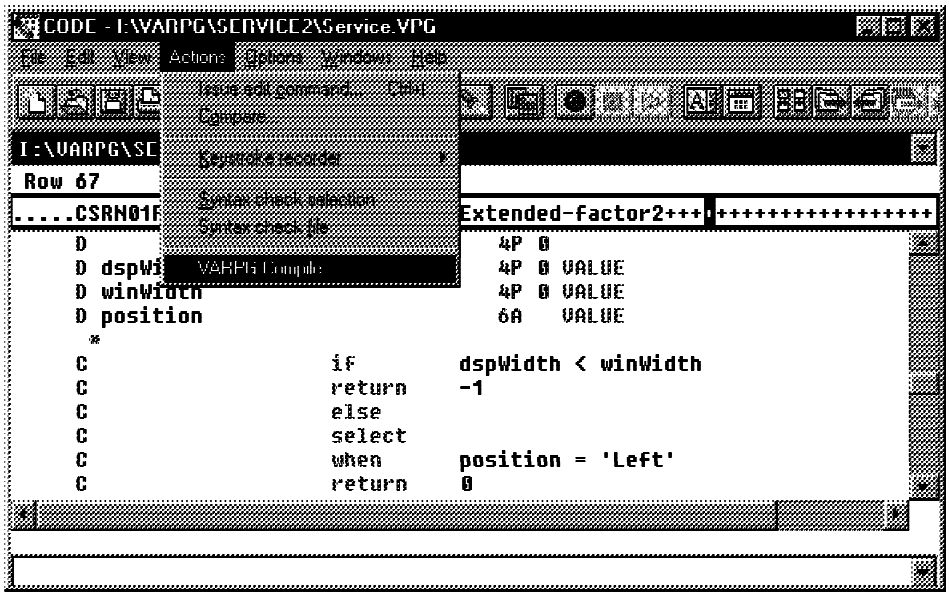


Figure 131. New Menu Item in the Actions Menu

To get this menu item included into the *Actions* menu, we added the following two lines to the VRPG400.LXL macro in the MACRO subdirectory of the ADTSWIN directory:

```
LPEX
'SET ACTIONBAR.LP_ACTIONS.SEPARATOR'
'SET ACTIONBAR.LP_ACTIONS.VARPG_Compile CMPVARPG'
```

Whenever the *VARPG Compile* item is selected, another LPEX macro CMPVARPG is invoked, which queries the user about the compiler options, starts the compiler, and invokes the *Error list* to show compiler feedback:

```
LPEX
'SET LINEREAD.TITLE Compiler Options'
'SET LINEREAD.PROMPT Specify compiler options:'
'LINEREAD 255'
'EXTRACT LASTLINE INTO MyOptions'

'EXTRACT NAME INTO FileName'
'START FVDFNFE 'FileName MyOptions

FileName = substr(FileName,1,length(FileName)-3) || "EVT"
'START CODEEVNT 'FileName
```

Share Your Work

If you are not the only programmer in your company, you probably need your colleagues to share parts or components you have developed. Additionally, you may need to maintain different versions of your applications or group a VisualAge for RPG application (or component) together with the AS/400 objects accessed by it.

Sharing Parts

If you were the programmer who created the *DBLOGON* component used in "SQL Support" to get the logon information needed to connect to a DB2 database from the user. It uses an entry field called *DONE* to signal its successful completion. If another programmer wants to use this component, he or she will need to define a component reference part that monitors for the CHANGE event of this entry field.

It would be handy to provide a user-defined part to your colleagues, where the information about the component name, window name, part name, and event name of the entry field part is already preset. If this user-defined part could be added to the part catalog of the colleagues' workstations, they would just have to drag and drop this part onto the GUI of their application and code its NOTIFY event.

The question is how to transfer the user-defined parts from your workstation to that of a colleague. VisualAge for RPG offers a packaging and install utility for user-defined parts for this purpose. If you invoke the *Packaging* utility, it shows you all user-defined parts of your workstation (see Figure 132).

Select the parts you want to be packaged, specify the target diskette or directory where the package should be placed, and press the *OK* push button. For each of the selected parts, an ODF file is created and, together with the ICO file, is copied into the specified directory. Additionally, a file called *VRPGUDPW.###* is created which contains control information for the package.

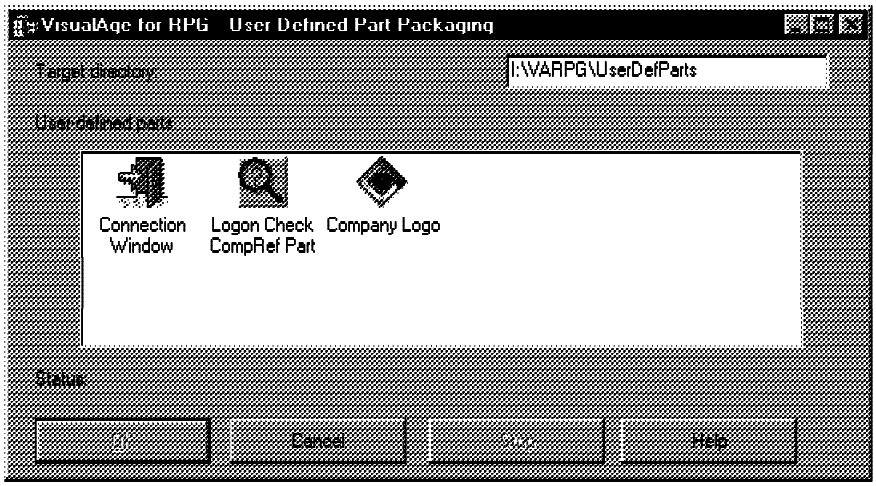


Figure 132. User-Defined Part Packaging Utility

You can now ask all colleagues who want to use the *DBLOGON* component to install this package onto their workstation using the *User-defined Part Install* utility (see Figure 133). To see the parts of the package, your colleagues must specify the diskette or directory you saved it to and press the *Fill* push button.

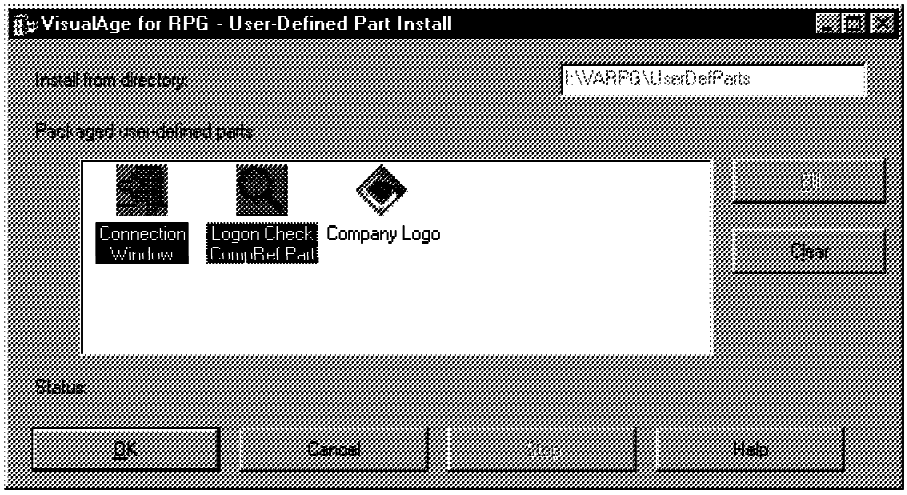


Figure 133. User-Defined Part Install Utility

Please note, it is not necessary to install the entire package. Instead, it is possible to select only those parts that are of interest. The selected parts are then added to the parts catalog as well as to the parts palette.

Sharing Components

Exchanging specialized parts for your graphical user interface is one thing. But you may also want to share program logic and complete components. One way to do that is to save your project in a directory on a drive shared by all company (or department) programmers. If you also place the .IVG files on a shared drive, the projects should be accessible from any workstation.

If you do that, however, make sure that the shared drive containing the source directories of the projects has the same drive letter assigned on every workstation. This is necessary because the .IVG file refers directly to the drive letter the project's files were saved on.

Chapter 9. Importing Display Files

In this chapter, we demonstrate how you can use the import display file function of VisualAge for RPG. Because there are no data definition specifications (DDS) on the workstation, the migration from a 5250 screen to a GUI is usually not a 1:1 mapping of texts and entry fields. Also, you will most likely change the appearance of the format. Some of your display files may not be worth migrating; it would be faster to recreate them again from scratch.

Screens and Windows

Applications that run on hosts such as AS/400 or S/390, with their *green screen* 5250 or 3270 interface, function in a fundamentally different way from workstation-based applications with GUIs. It's not only the appearance that makes them different, or the fact that users trade function keys for mouse clicks, it's also how a user navigates through the application.

In most cases, one GUI window contains more information and lets a user do more different things than does a 5250 window. Consider the omnipresent tool bar; by clicking on any of its icons, you can launch additional applications, open new windows, run macros, print lists, and much more using very little window real estate. Think about how much text you would need to put on a screen to describe the same functions.

Before you start importing all display files from your AS/400 application and converting them to VisualAge for RPG windows, step back and look at branches of your application and search for ways to merge display formats. You will gain a clearer interface and also save performance by omitting the opening and closing of windows while you navigate through your application. In the 5250 world, application users may consider it standard practice that screens change entirely after certain keys are pressed. GUI users usually stay in a few windows (sometimes only a single window) and use pull-down and pop-up menus to invoke additional functions.

Redesigning Screens

If you are an AS/400 developer, you have no doubt created menu screens and their corresponding menu programs. They may have looked like Figure 134.

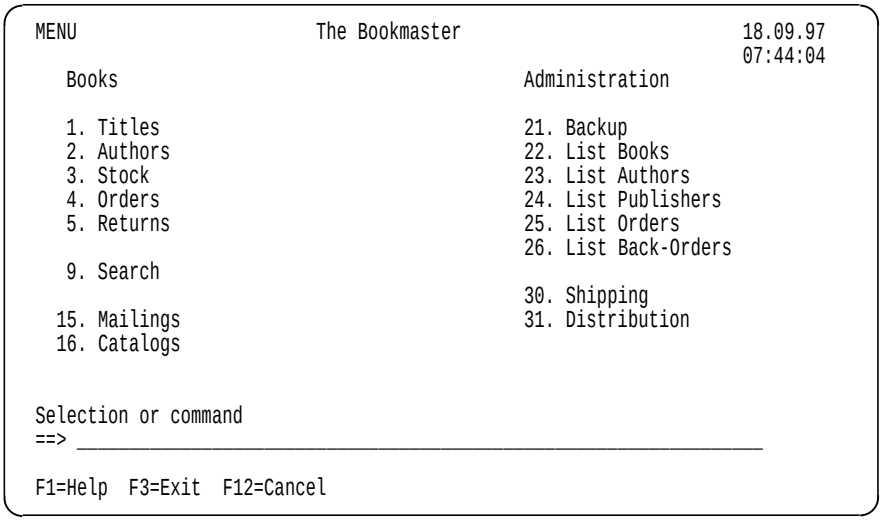


Figure 134. Traditional AS/400 Menu

You select an item from the menu, enter the number and press Enter. According to your selection, the entire screen changes. The original design of the application probably entailed 20 different formats to accommodate all possible menu selections and their application screens.

Using a GUI, your application design follows a different approach. After a possible splash screen to welcome the user (and, more important, to initialize the application in the background, such as opening the database or connecting to a server), the main window opens. This is where users should spend most—if not all—of their time. This window closes only when the user exits the application.

It is the responsibility of the application designer to find the GUI items (such as database records, containers with icons, or sets of radio buttons or check boxes) that are the most important in the application scenario. These items act as anchors for all subsequent actions from pull-down or pop-up menus. Instead of navigating through a series of screen formats, from a starting menu to, for example, a maintenance screen for books, the user should be able to initiate the required action instantly.

The first thing to get rid of in the original application design is the menu format. Instead of using a full screen with menu choices, the possible selections should be grouped in pull-down menus (see Figure 135).

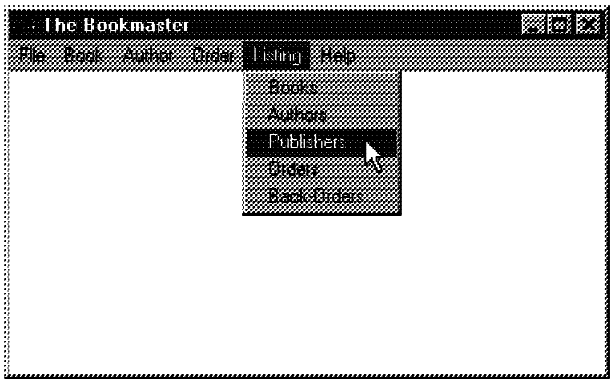


Figure 135. Example of a GUI Menu

Without using up more than a little screen real estate, we have already accommodated all possible menu selections, and we didn't even mention submenus and toggles as menu items.

Look at every single application screen and assess whether or not it contains enough information to justify giving it its own window.

Sample Import

The 5250 screens that are good canidates for a GUI window are the record maintenance formats that require static text and many entry fields. You need not keep the original entry fields. Depending on the type of entry field and its allowed values, you should consider GUI elements such as check boxes (for binary values), radio buttons (for small selection groups), drop-down list boxes (for selections with long texts), or spin buttons (for numeric ranges).

Figure 136 is an example of a 5250 screen that we will import into VisualAge for RPG. The format consists of static text, an output-only field (customer number), various input fields, and date and time fields.

```
CUSTFM1              Customer Maintenance                    19.09.97
                                                             20:32:55

     Customer Number . . . .   1234567

     Company Name  . . . . .   ____________________________________

     Rep Number  . . . . . .   ____
     Contact . . . . . . . .   ____________________________

     Address . . . . . . . .   ____________________________________
     City, ZIP . . . . . . .   ____________________________  _______
     Country . . . . . . . .   ____________________

     Telephone . . . . . . .   ________________
     Fax . . . . . . . . . .   ________________

     Zip Location  . . . . .   _


F3 = Exit
```

Figure 136. Display Format to be Imported

To import the format into VisualAge for RPG, use the *Import Display File* option from the *Server* pull-down menu of the VisualAge for RPG main window (Figure 137).

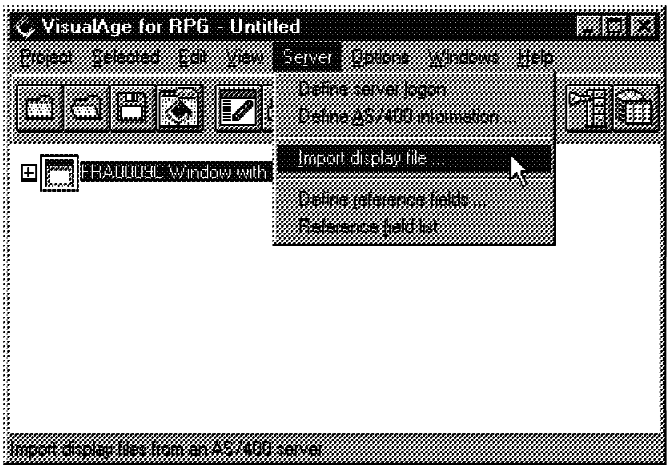


Figure 137. Import Display File Menu Option

The *Import Display File* window opens (Figure 138). From the *File hierarchy* pane, select the AS/400 server of your choice and click on the + icon to expand and show all libraries on that server. Notice that the *File name* entry field is updated along with your selections in the pane below. The libraries

listed depend on your AS/400 job description that your VisualAge for RPG job is running under. If your library list does not contain the library with the display file to be imported, simply type the library name into the *File name* entry field and press Enter.

In our example, the library GUIDES2 is not part of *LIBL, so it is not shown. After typing in

```
<server>GUIDES2
```

the appropriate information is extracted from the AS/400 and the hierarchy is expanded. Because this is the import function for display files, only display files are listed under the library name, even though the library on the AS/400 may contain many more objects.

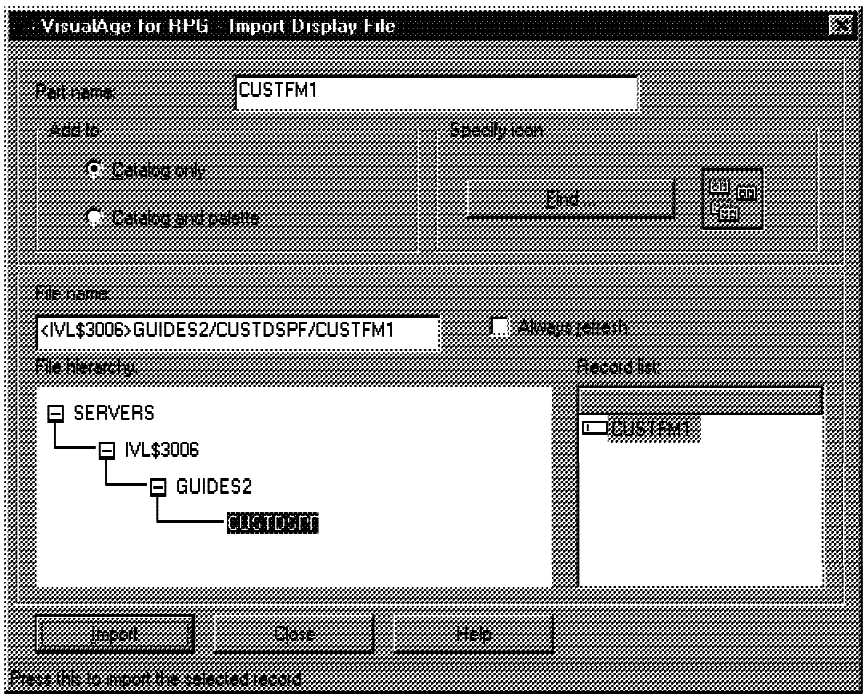


Figure 138. Import Display File Window

To see all formats that are defined in the display file, double-click on the display file name. After the information is again extracted from the AS/400 object, the record list is updated accordingly. Select the format you want to import (in our case, only one format is defined) and click on the *Import* button. A *Process List Request* window appears briefly, informing you that the system is importing the record. After the process window disappears, your imported format is added to the parts catalog. If you selected the

Catalog and palette radio button in Figure 138, your format is also placed on the palette. However, this is rarely necessary because the imported format is mostly used only once.

Open the *Parts Catalog* window and locate the new format on the *Imported* page (Figure 139). By default, the part name is the same as the format name. You can override the default by changing the name during the import process or by modifying the part in the catalog.

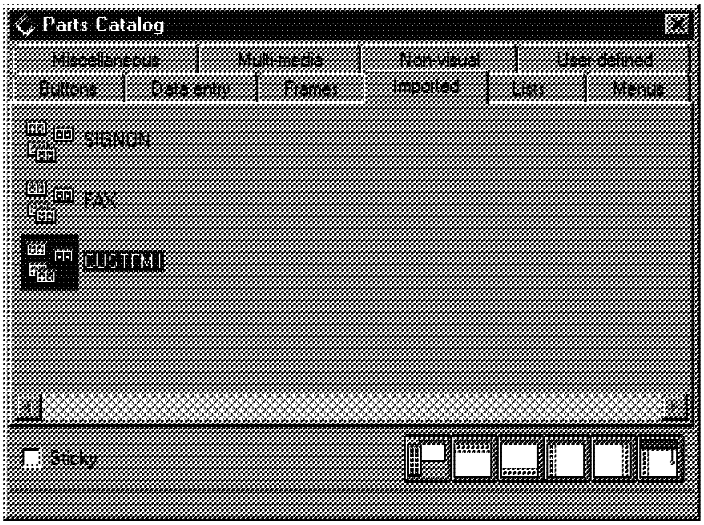


Figure 139. Parts Catalog Window with Imported Format

To create a window from the newly imported format, drag the new part to the project window and drop it on a *Window with Canvas* part in the project window. If you do not have an unused window with canvas, drag a new window with canvas from the parts catalog's *Frames* page to the project view first. The project view now contains a window with its canvas and, as subparts, all GUI elements that were identified on the original display format. The expanded view of the window may look like Figure 140.

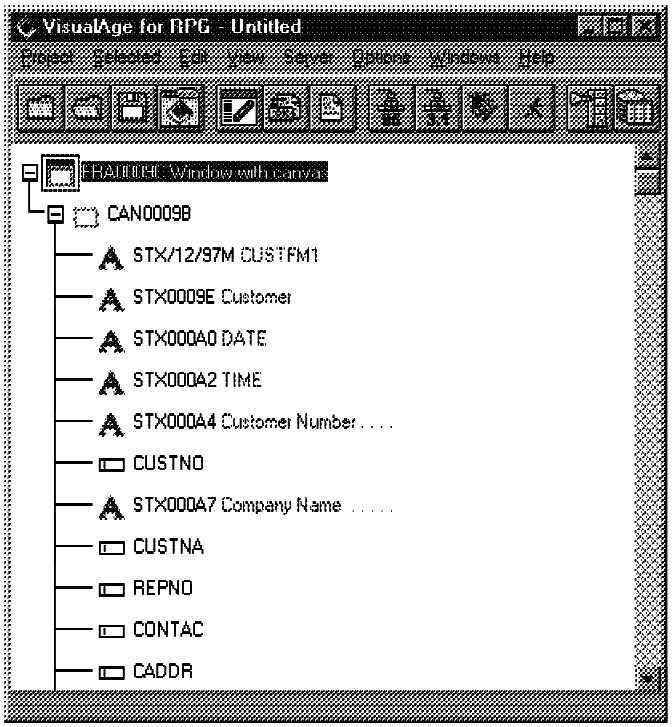


Figure 140. Imported Format Parts

To see how the format actually looks in a window, double-click on the window part to open it. Be ready for a surprise (Figure 141).

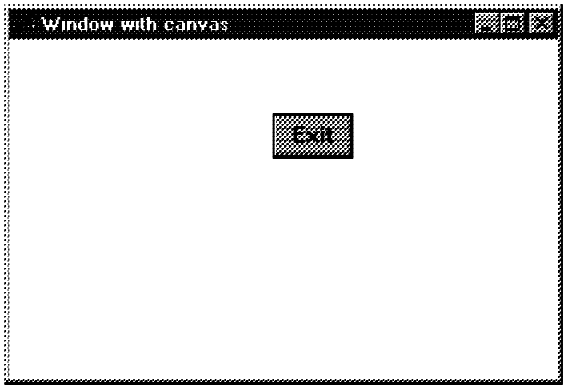


Figure 141. Imported Format as Window

“All I get is one button?” Don't worry. All entry fields and texts are there, however, they may have fallen off the window. Resize or maximize the

window so that all fields are visible and rearrange them. If you find only one static text field, try moving it a little; you will notice that they overlap each other. Expose all fields by dragging them to different locations and soon you have a window that you can actually start designing (Figure 142).

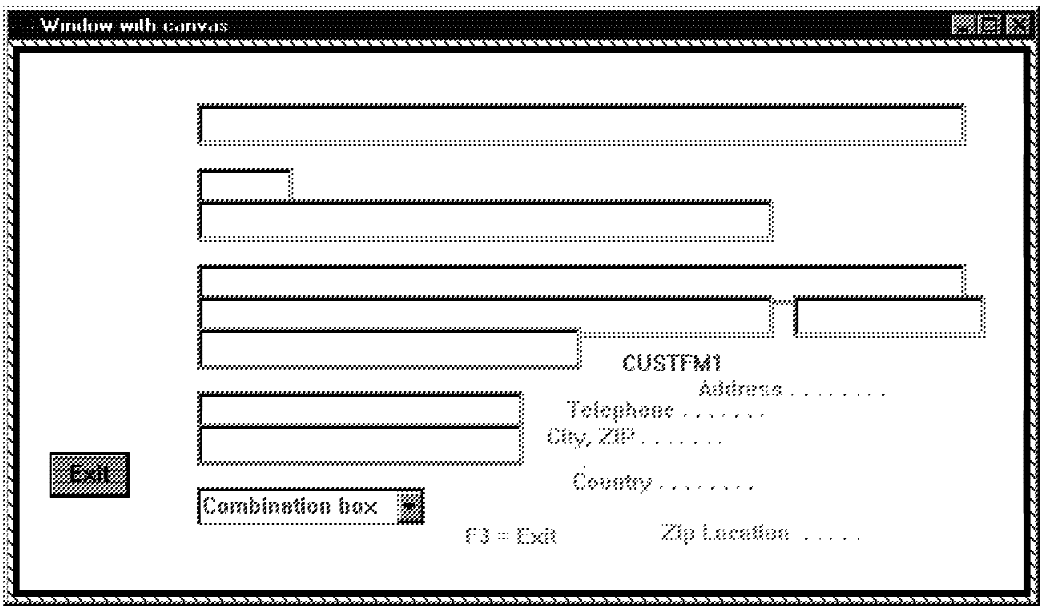


Figure 142. Imported Format Adjusted to Fit

After aligning static texts and entry fields (try the grid), adjusting the colors, and other cosmetic enhancements, your new window is ready to be used (Figure 143). All entry fields on the window have the same definitions (such as length) as in the imported display file, as you can verify on the *Data* page of the properties window. However, the *Reference* page does not contain a link to an AS/400 definition, so if you choose to change a field on the AS/400, your VisualAge for RPG window is not notified unless you define a reference after the import.

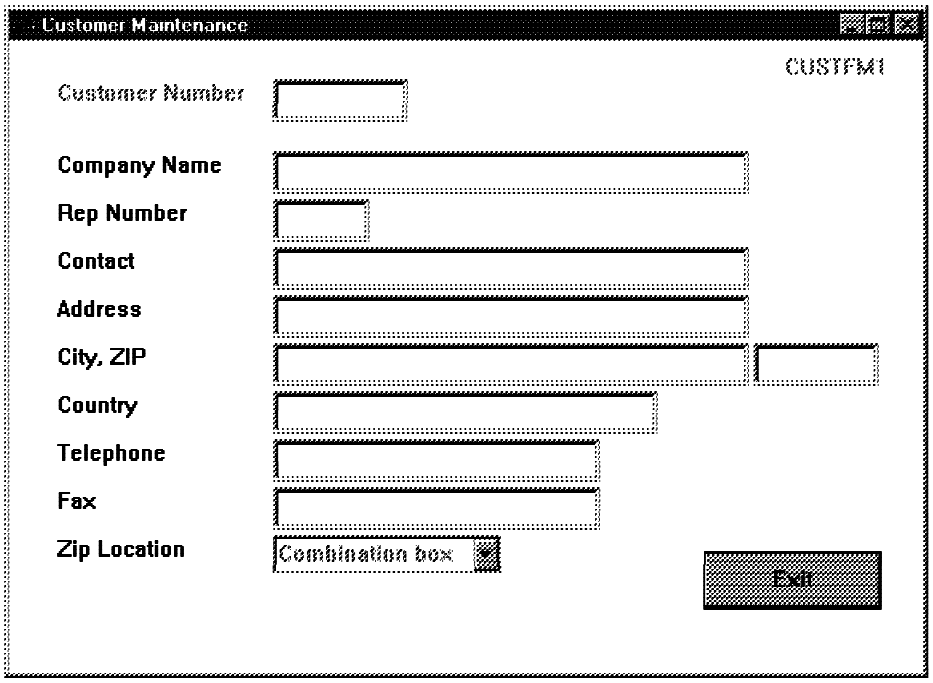


Figure 143. Finished Window from the Imported Format

Part 2. Exercises

Exercise 1. Creating a Simple VisualAge for RPG Application

The exercises in Part 2 of this book are derived from an education class on VisualAge for RPG.

Exploring the GUI Designer and Creating Action Subroutines

During this exercise, you learn more about the VisualAge for RPG GUI Designer as we walk you through the following steps:

1. Exploring the VisualAge for RPG GUI Designer:
   - Starting the VisualAge for RPG GUI Designer
   - Getting to know the components of the VisualAge for RPG GUI Designer
   - Using parts and GUI Designer components to create a graphical user interface
   - Saving the GUI
2. Creating action subroutines:
   - Selecting events
   - Working with the LPEX Editor
   - Creating action subroutines to respond to events
   - Building the application
   - Running the application

Objective

As a result of this exercise, you can create the first window of your GUI application. In doing so, you can:

- Customize the VisualAge for RPG GUI Designer.
- Create a window with parts.
- Create action subroutines.
- Save your application.
- Build the application.
- Run the application.

Introduction

You need a workstation with Windows 95 or Windows NT and VisualAge for RPG installed with the minimum configuration described in the product manuals.

If you are using this book in class, user IDs are given to you by the instructor. However, in this exercise, you are not working on an AS/400 system, so you do not need a user ID.

Starting the VisualAge for RPG GUI Designer

To start the VisualAge for RPG GUI Designer for the first time:

1. Locate and double-click on the *Application Development Toolset Client Server/400* icon on the desktop. This opens up the *Application Development Toolset Client Server/400* folder and makes it the active window.
2. Double-click on the *VisualAge for RPG* icon; the *VisualAge for RPG* folder is shown.
3. Double-click on the *GUI Designer* icon in the *VisualAge for RPG* folder to start the GUI Designer.

The VisualAge for RPG Product Information dialog is briefly shown, followed by a status window (Figure 144) indicating progress in the initialization process of the VisualAge for RPG GUI Designer.

Figure 144. VisualAge for RPG Status Indicator

The GUI Designer, including the Parts Palette and the first design window, is shown in Figure 145.

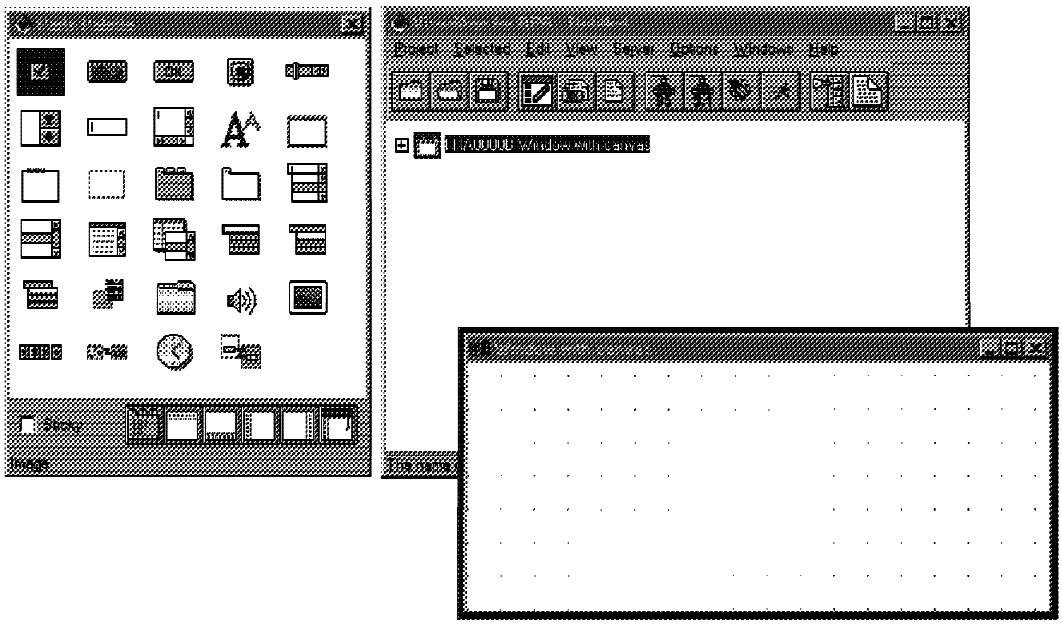


Figure 145. VisualAge for RPG GUI Designer

If necessary, rearrange the windows so that all are visible. Also, if needed, resize the Parts Palette so that all parts are visible.

Components of the VisualAge for RPG GUI Designer

The VisualAge for RPG GUI Designer consists of these major components:

- Menu bar
- Tool bar
- Project view
- Parts palette

Menu Bar

The menu bar is located below the title bar of the *VisualAge for RPG* window. It provides a variety of tasks that you can use to create and modify your GUI application, customize the VisualAge for RPG GUI Designer, and get online help. To learn more about the menu bar:

1. Press F10 to highlight the leftmost menu bar choice. Then, press F1 while the menu-bar item is highlighted. A pull-down menu with menu choices is opened.
2. Press the right arrow key to scroll through the choices of the pull-down menu. While doing this, look at the bottom of the *VisualAge for RPG*

window. This is the information area; it briefly describes the action performed when you select one of the menu choices or one of the items contained in the pull-down menu of a menu bar choice.

The Up and Down arrow keys can be used to navigate the menu choices. Detailed help for each choice is available by pressing F1 (see Figure 146).

To become familiar with the available menu choices:

1. Press F10 to highlight the *Project* menu bar choice.
2. Use the Up, Down, Left, and Right arrow keys to scroll through all the menu choices. Look at the information area at the bottom of the VisualAge for RPG window to see what action the choices perform.

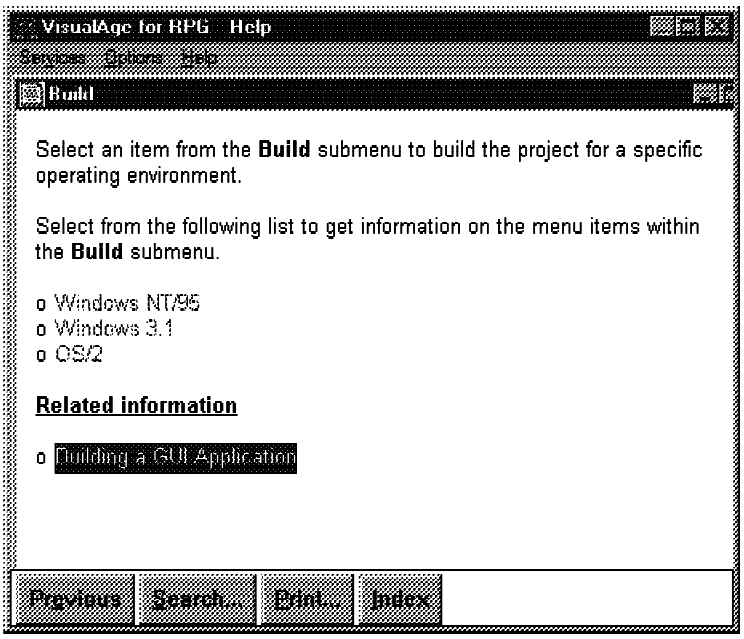


Figure 146. Displaying Menu Help

Tool Bar

The tool bar is a menu of icons. It provides a fast-path access to some of the menu items on the menu bar. Invoking the action from an icon on the tool bar allows for quicker access to commonly used menu choices. To learn more about it,

1. Move the mouse arrow cursor over the icons of the tool bar, one at a time. A short description of each icon's function is displayed in the information area.
2. Click on the *Font Palette* icon. This is one of the rightmost icons on the tool bar. The Font Palette window is shown.
3. Double-click on the *System* icon (at the upper left corner of the *Font Palette* window) to close it.

Project View

The project view is used to hold all of the parts you create during your VisualAge for RPG GUI Designer session. The project view can be changed to display the parts of a project in different views. This is demonstrated in a later exercise.

Parts Palette

The Parts Palette contains parts that you use to create the screen layout for your GUI application. These parts act as a template to create parts from. To learn more about it,

1. Move the mouse pointer over the parts on the *Parts Palette* window one at a time. The information area at the bottom of the Parts Palette displays a short description of each part. To see a detailed description of a part on the *Parts Palette* window, select a part with the left-hand mouse button, and press F1.
2. Click on the icon in the upper right corner of the *Parts Palette* window. The *Parts Catalog* window is shown. The Parts Catalog contains all the parts of the VisualAge for RPG GUI Designer, categorized by function, as indicated by the tabs. Click on the icon in the upper right corner of the *Parts Catalog* to return to the *Parts Palette* window.

Creating a Graphical User Interface

During this exercise, you learn how to create a GUI by creating windows and adding parts to windows. The GUI you are creating now is the first window of a Customer Inquiry application. It is used to prompt for a customer number.

When the GUI Designer is started for a new project, it creates the first design window for you. Notice also that a Window part has been added to the project view. We use this window as the first window in our Customer Inquiry application. As with all parts, the initial part name is generated by the GUI Designer, with the first few characters indicating the part type (for example, FRA indicates a Frame Window part).

A Note About Notebooks

As mentioned earlier, the Parts Palette contains parts that act as templates. Once the part has been created in the project view, its attributes, such as the part name, can be changed. These changes are made by invoking the *Parts Properties* properties notebook. A properties notebook consists of pages as indicated by the labeled *tabs*. You go to a properties notebook page by clicking on its tab with the left-hand mouse button. To close a properties notebook, double-click on its *System* icon in the upper left corner or click on the icon in the upper right corner.

Let's change some of the settings for the initial design window:

1. Double-click on the *title bar* of the design window. The *Window Part Properties* notebook is shown.

> **Tip**
>
> A window with canvas is made up of two parts: the window, which consists of the title bar and frame, and the canvas, which is the white area enclosed by the window. Thus, if you double-click on the white *canvas*, you get the properties notebook for the canvas part.

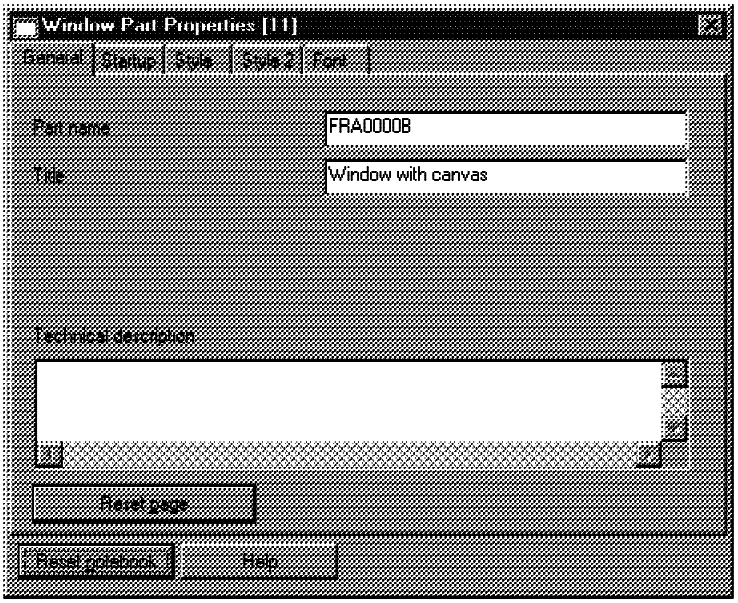


Figure 147. Changing the Window Part Name and Title

2. On the *General* page of the Window Part Properties window (Figure 147), change the *Part name* setting to CUSTINQ and the *Title* to Customer Inquiry.

   Note

   Although you can type the part name in upper or lower case, it is always folded to upper case by the GUI Designer.

3. Click on the *Style* tab of the properties notebook. The *Style* page is shown.
4. Deselect the *Minimize* button and *Maximize* button check boxes.
5. Select *Thick* for the style of the window border.
6. Double-click on the *System* icon of the *Window Part Properties* notebook to close it.

Notice the changes to the window. The new window title is displayed, the border style has changed, and *Minimize* and *Maximize* icons no longer appear on the title bar.

Tip

To change the label and default text of parts, you can also press the Alt+left-hand mouse button instead of using the properties notebook. When you have made your change, click the left-hand mouse button again. This is called *direct editing*.

The next step is to add a static text part to the window to prompt for the customer number to be entered:

1. Find the static text part in the *Parts Palette* window.

   Tip

   As you move the cursor over the icons on the Parts Palette, a short description of the part type is shown in the Parts Palette information area.

2. Drag and drop a static text on to the Customer Inquiry design window by clicking on it with the right-hand mouse button, dragging the part to where you want it on the design window with the mouse, and releasing the mouse button.
3. Invoke the properties notebook for this part by double-clicking on it.
4. Change the Text setting on the *General* page to `Enter Customer Number`.

5. Double-click on the *System* icon of the *Static Text Parts Prperties* notebook to close it.

You may notice that not all of the changed text is displayed. This is because of the default size setting of the static text part. To change the size:

1. Click on the *Static Text* part with the left-hand mouse button to select it. The borders of the part are marked by placing small squares, called *sizing handles*, around it.
2. Move the mouse pointer to the middle of the rightmost three sizing handles. The pointer changes to a double-arrow (pointing to the left and right).
3. Click and hold the right-hand mouse button.
4. Move the mouse pointer to the right until the entire text of the static text part is visible, then release the mouse button.

To add the entry field that receives the customer number:

1. Find the Entry Field part on the Parts Palette, drag it to the *Customer Inquiry* window and drop it to the right of the Static Text part.
2. Invoke the properties notebook for this part by double-clicking on the new entry field.

   Tip

   You can also open the properties notebook for a part by selecting the *Properties* choice from the pop-up menu for the part, or by selecting *Selected* and *Properties* from the GUI Designer menu bar when the part is selected.

3. Change the Part name setting on the General page to `CUSTNO`. This is the variable name by which the customer number can be accessed later, within VisualAge for RPG code.
4. Click on the *Data* tab to get to the *Data* page.
5. Change the length to 7.
6. Go to the *Validation* page.
7. Select the *Range* radio button.
8. Enter `0000000` as the Minimum value and `9999999` as the Maximum value.
9. Double-click on the *System* icon of the *Entry Field Part Properties* notebook (Figure 148) to close it.

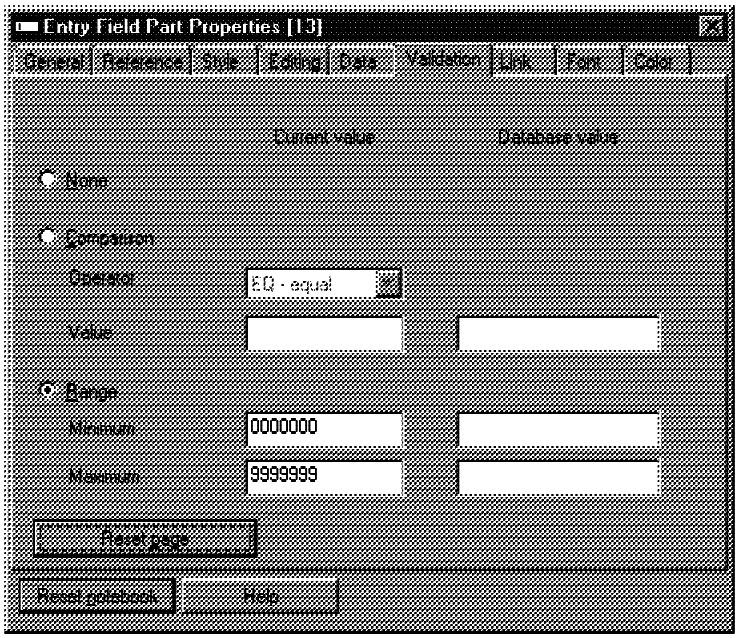


Figure 148. Entry Field Part Properties Window

The static text and entry field parts are probably not aligned with one another. To align them, do this:

1. Move the mouse pointer above and to the left of the static text part.
2. Click and hold the left-hand mouse button.
3. Drag the mouse down and to the right so that the two parts are covered by a gray rectangle (Figure 149). This gray rectangle is referred to as a *selection rectangle*.

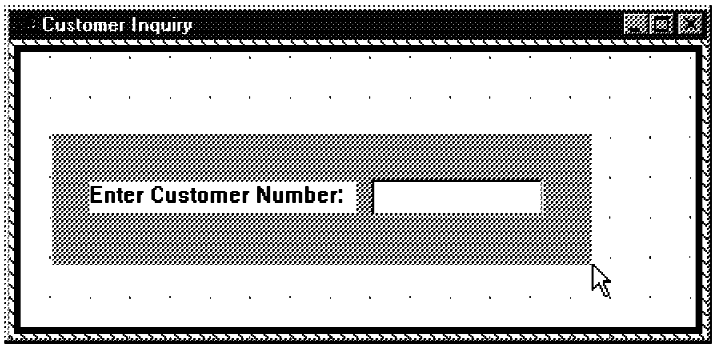


Figure 149. Selection Rectangle

4. Release the left-hand mouse button. The two selected parts are highlighted, with a dark border.
5. Move the mouse pointer to the static text part.
6. Click the right-hand mouse button. The pop-up menu for this part is shown.

   Tip

   When more than one part is selected, invoking the pop-up menu for one of the selected parts makes that part the anchor, meaning that other items are aligned relative to that part.

7. Choose the *Align...* choice on the pop-up menu. A submenu is shown.
8. Choose the second alignment choice offered. As depicted by the chosen icon, this aligns the two parts to an imaginary horizontal axis running through the static text part (Figure 150).

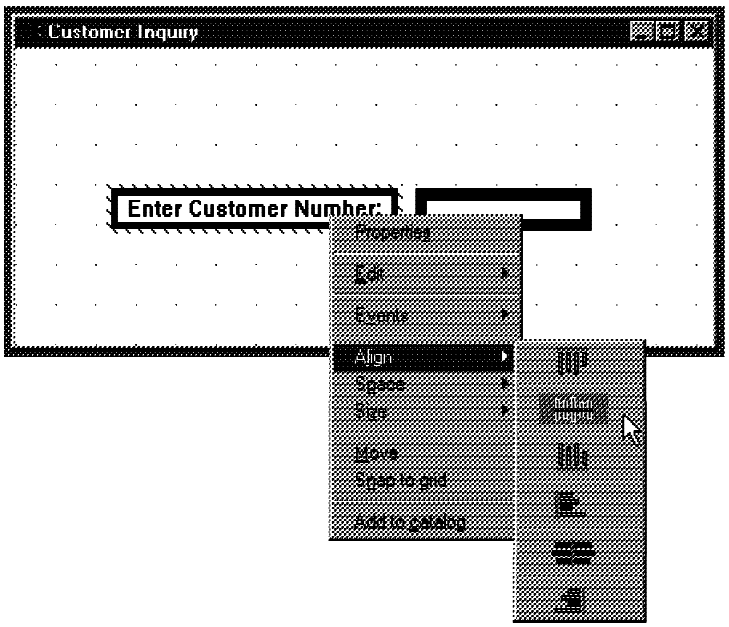


Figure 150. Alignment Tools

To complete this window, we need to add two push buttons, an *OK* push button to process the customer number and an *Exit* push button to end the application. To add these push buttons:

1. Find the push button part on the Parts Palette.
2. Drag a push button part to the *Customer Inquiry* window and drop it below the static text part.
3. Invoke the properties notebook for the push button.
4. Change the *Part name* setting on the *General* page to PSBOK and the Text setting to `OK`.
5. Go to the *Style* page.
6. Select the check box for the *Default* option. This makes it the part that gets selected when the Enter key is pressed.
7. Double-click on the *System* icon of the *Push Button Part Properties* notebook to close it.

To create the second (*Exit*) push button, create a duplicate of the *OK* push button:

1. Move the mouse pointer to the *OK* push button part and click the right-hand mouse button to get the pop-up menu for this part.
2. Choose the *Edit* menu choice and the *Duplicate* submenu choice. A second *OK* push button is created.
3. Move the mouse pointer to this new push button part.
4. Press and hold the right-hand mouse button and drag the push button part below the entry field part.
5. Release the mouse button.

To change the text of the push button:

1. Press and hold the ALT key and select the *push button* part with the left-hand mouse button. The text of the push button part becomes editable (Figure 151).
2. Type in `Exit`.
3. Move the mouse pointer outside the push button and deselect the part by pressing the left-hand mouse button.

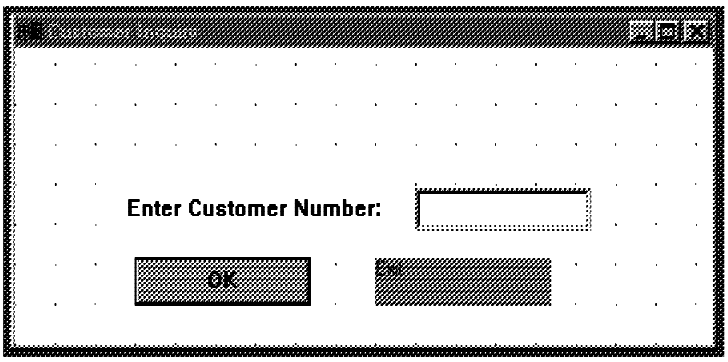


Figure 151. Changing Push Button Label

Because this push button part has been created as a duplicate of another push button, it adopted the same attributes with the exception of the part name which is generated by the GUI Designer to avoid name conflicts in this window. To change the name for the *Exit* push button part:

1. Invoke the properties notebook for the *Exit* push button part by double-clicking on the button.
2. Change the *Part name* setting on the *General* page to PSBEXIT.

   Note: The text is already changed to *Exit*.
3. Go to the *Style* page.
4. Deselect the check box for the *Default* option.
5. Double-click on the *System* icon of the properties notebook to close it.

Now, the two push button parts need to be aligned. The left border of the *OK* push button must be aligned to the left border of the static text, while the right border of the *Exit* push button needs to be aligned to the right border of the entry field. To do the alignment:

1. Move the mouse pointer above and to the left of the static text part.
2. Press and hold the left-hand mouse button.
3. Drag the mouse so that the static text part and the *OK* push button part are covered by the selection rectangle.
4. Release the mouse button. The two selected parts are highlighted.
5. Move the mouse pointer to the static text part, click the right-hand mouse button to get the pop-up menu, and choose the *Align...* menu choice from the pop-up menu.

6. Choose the fourth choice offered. This aligns the *OK* push button part to the left border of the static text part.

To align the *Exit* push button with the entry field:

1. Move the mouse pointer above and to the left of the entry field part.
2. Press and hold the left-hand mouse button.
3. Drag the mouse so that the entry field part and the *Exit* push button are covered by the selection rectangle.
4. Release the mouse button. The two selected parts are highlighted.
5. Move the mouse pointer to the entry field, click the right-hand mouse button to get the pop-up menu, and choose the *Align...* menu choice from the pop-up menu.
6. Choose the sixth choice offered. This aligns the *Exit* push button part to the right border of the entry field part.

Now, align the two push button parts to each other horizontally in the same way you aligned the static text and entry field parts.

Finally, the size of the window has to be changed to contain the parts within it:

1. Move the mouse pointer over the upper right corner of the window; the pointer becomes a double arrow.
2. Press and hold the right-hand mouse button.
3. Drag the mouse to resize the window until all of the parts are in the middle of the window.

Note: None of the parts created within the *Customer Inquiry* window are moved when the window is resized. This is because parts remain relative to the left and bottom edges of the window. You need to be aware of this since some resizing operations move parts so they are no longer visible.

The resulting window should look like in Figure 152:

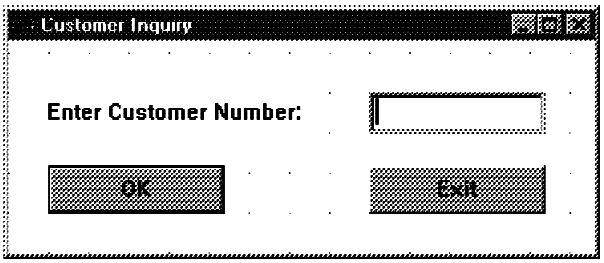


Figure 152. Completed Customer Inquiry Window

Saving Your Project

Now that you have finished your first graphical user interface, you need to save your project:

1. Move the mouse pointer to the *Save Project* icon on the tool bar.

   Tip

   As you move the mouse pointer over the tool bar, the information area is updated to indicate the function of each button.

2. Click on this icon and save your project. Since this is a new project, and has not yet been named (the title bar on the GUI Designer says *Untitled*), the *Save as Application* window is displayed. This window is used to name your project, indicate in which folder it should be saved, and where the project files should be saved.

   Make the following entries on this window:

   - In the *Application name* entry field, type a title for your application (for example, `Customer Inquiry`).
   - Leave the folders selection as *Desktop*.
   - In the Source file entry field, type `GuiDes2`.
   - Replace the value of the *Source directory* entry field with `x:\GUIDES2`.

     Note

     Replace `x` with a drive letter valid on your workstation.

   - Press the *OK* push button to save your project. Notice that after the project has been saved, the GUI Designer title bar is updated to show the project name.

This ends Part 1 of this exercise. You use this project in the next part of this exercise, so please do not exit the GUI Designer at this time.

Action Subroutines

You use the window that you created in the previous portion of the exercise. You are adding an action subroutine for the Press event of each push button. Recall that an action subroutine is the VisualAge for RPG logic that is invoked to handle an event on the graphical user interface.

Creating an Action Subroutine

Now that you have finished designing this window, you can add some code to give it some function. In this part of the exercise, you create the action subroutine for the *Exit* push button.

1. If the design window for *CUSTINQ Customer Inquiry* is not open, open it by double-clicking on its icon in the GUI Designer Project View (Figure 153).

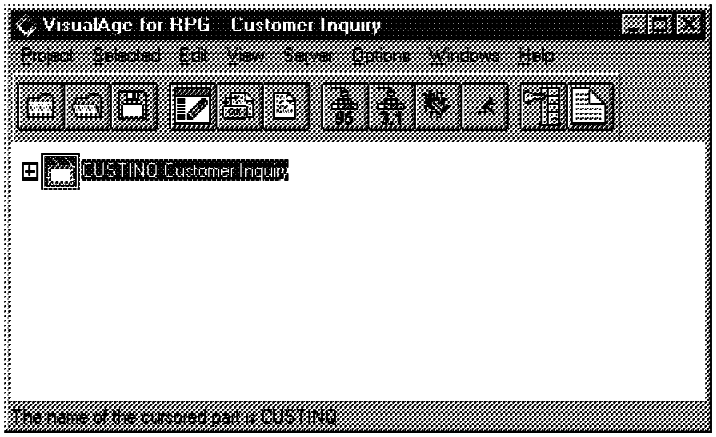


Figure 153. Opening a Window in the Project View

2. Select the *Exit* push button using left-hand mouse button. Click the right-hand mouse button to get the pop-up menu. Choose *Events* and choose *Press* from the submenu (Figure 154).
3. This starts the LPEX Editor (Figure 155).

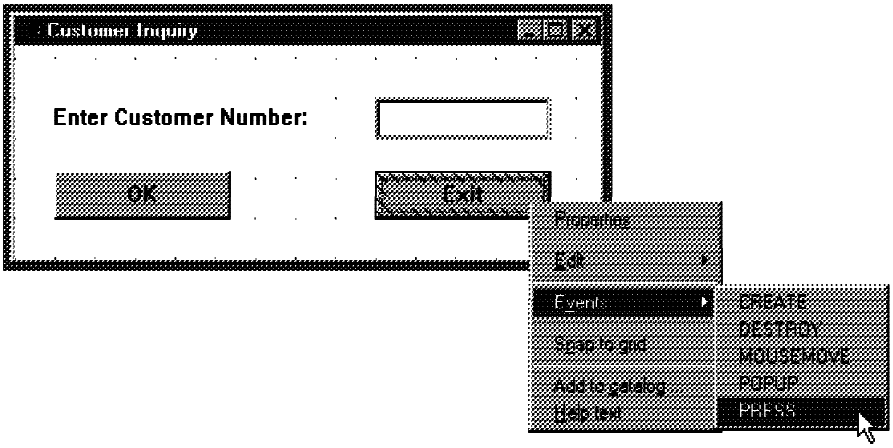


Figure 154. Selecting an Event

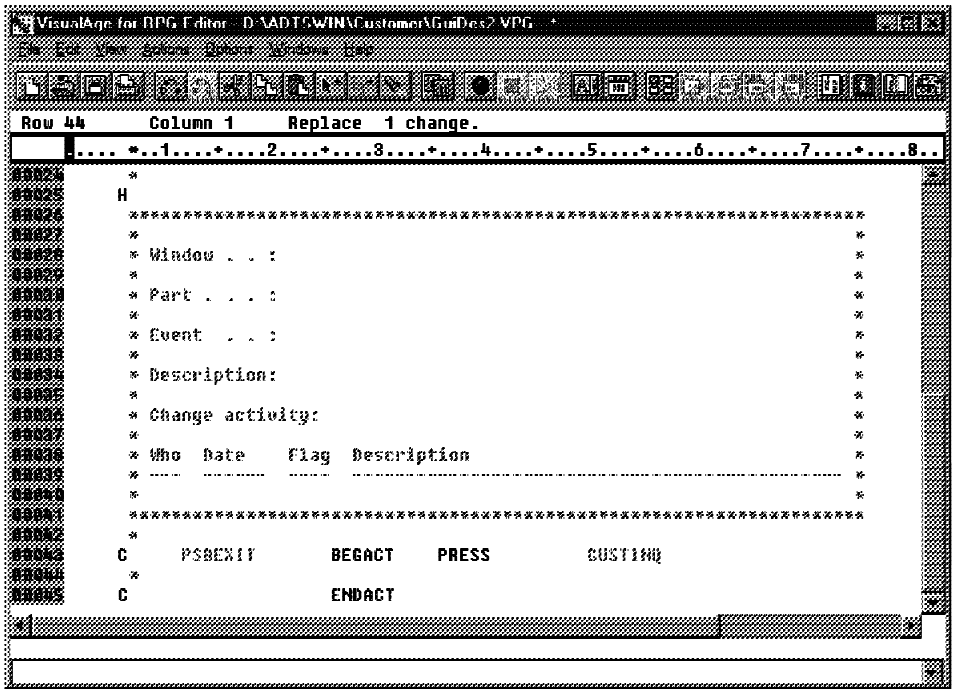


Figure 155. LPEX Editor for Exit Push Button Action Subroutine

When the LPEX Editor first appears, notice that the BEGACT and ENDACT statements have already been inserted into the action subroutine. A number of comment lines are also included to encourage

documentation. Change the comments to include any further information you consider necessary.

Although the framework for the RPG code has been built, the application logic is not there. It must be added. In this example, the code to terminate the application must be added.

4. Position the mouse pointer anywhere on the line of the BEGACT operation code and click the left-hand mouse button. This positions the cursor.
5. Choose the *Edit* option from the LPEX Editor menu bar and choose *Insert with Prompt* from the pull-down menu.

Tip

You can also perform an Insert with Prompt by pressing Shift+F4.

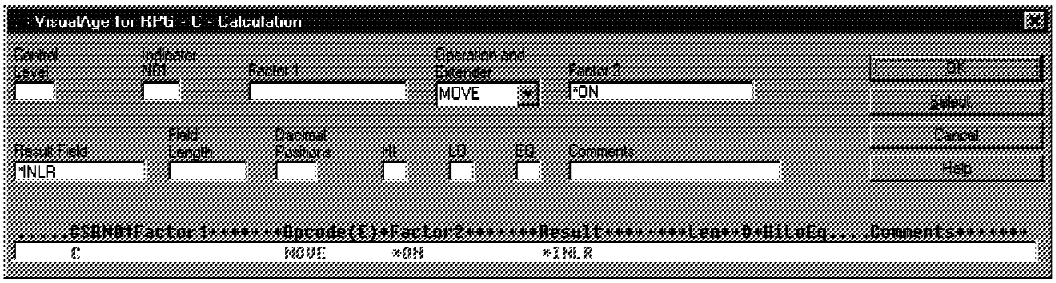


Figure 156. Example of an Insert with Prompt Window

6. A prompt window is displayed (Figure 156). Type in the VisualAge for RPG statement as shown below, which causes the program to terminate by setting the last record indicator, LR, to ON. Type the following:

RPG

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7..
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+Hi
 *
C                   MOVE      *ON           *INLR
```

7. Extensive online help is available for the VisualAge for RPG language. Before leaving the prompt window shown in Figure 156, position the cursor back into the *Operation and Extender* field. Press F1 (HELP). A *Help* window for the current value of the Operation code field is displayed. In this case, help for the MOVE operation code is shown. To close the *Help* window, click on the icon at the top right corner of the window.
8. Press the *OK* push button to add this line of code to the source.

9. To end prompting, press the *Cancel* push button on the *Prompt* window.

   Tip

   Most dialogs that have a *Cancel* push button can also be closed by pressing the Esc key.

   Notice that the line of code you added appears directly below the BEGACT statement on the editing window.

10. To save this change, choose the *File* option from the *LPEX Editor* menu bar and choose *Save*.
11. Close the LPEX Editor by double-clicking on its system menu.

Let's go over what an action subroutine is.

In the event-driven method of creating applications, what happens in the program is controlled by the events occurring on the GUI. An event can be linked to an action subroutine. An action subroutine is easily found in the code for the complete VisualAge for RPG source; it begins with a BEGACT operation code and ends with an ENDACT operation code. The logic between these two statements is the business logic that is executed when the event is generated.

Now, you can create an action subroutine for the OK push button.

Creating an Action Subroutine for the OK Push Button

You can add an action subroutine that changes the color of this push button when it is pressed. If the color of the push button is red, it is changed to gray; otherwise, it is set to red.

This code may not seem highly useful, but it does give you the opportunity to experiment with two of the new operation codes:

- SETATR (Set attribute)
- GETATR (Get attribute)

Later, you can add some more meaningful logic for the Press event of this push button. Now, however, start with this set of steps:

1. Select the *OK* push button with the left-hand mouse button. Use the right-hand mouse button to get the pop-up menu. Choose *Events* from the pop-up menu and choose *Press*, which invokes the LPEX Editor.
2. You want to write logic to get the background color of this push button. You do this by using the GETATR operation code. Your logic determines the new color to set it to; use the SETATR operation code to change it.

3. This logic sets the background color to red if it is not currently red; otherwise, it sets the background color to gray. Your logic should look like the code in Figure 157.

Tip

VisualAge for RPG statements can be entered in upper, lower, or mixed case including part names and attribute names.

RPG

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+.
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq
 *
C     PSBOK         BEGACT    PRESS         CUSTINQ
C     'PSBOK'       Getatr    'BackColor'   BCOLOR            2 0
C                   If        BCOLOR <> *RED
C     'PSBOK'       Setatr    *RED          'BackColor'
C                   Else
C     'PSBOK'       Setatr    *PALEGRAY     'BackColor'
C                   EndIf
C                   ENDACT
```

Figure 157. Example of Code to Change the Color of the OK Push Button

Tip

These are some of the basic LPEX Editor commands:

- ALT+L marks a line.
- ALT+U unmarks a line.
- ALT+D deletes a marked line.
- ALT+C copies a marked line.
- ALT+M moves a marked line.

Press Enter to insert a new line.

RPGIV: Base for VisualAge for RPG Language Definition

The VisualAge for RPG language is based on the RPGIV language syntax. Both upper and lower case letters are allowed in the source code. This greatly enhances the readability of the code, as well as the ease of coding. The second RPGIV feature that is notable in VisualAge for RPG is the 10 character names. Not only does this make the code more legible and the names more meaningful, it also means that the RPG names can be the same as those used in other AS/400 languages as well as in DDS without the need to rename them.

LPEX Editor

Now that you have added the second action subroutine, let's look at some of the features of the LPEX Editor so you can learn your way around and use them more easily.

LPEX Editor tool bar

This version of the LPEX Editor also has a tool bar to access the most commonly used LPEX Editor functions. By default, the tool bar, is not displayed. To use the tool bar:

1. If an LPEX Editor window is not currently open, open one. You can open an LPEX Editor window by choosing *Project* and *Edit source code* from the GUI Designer menu bar.
2. Choose *View* from the LPEX Editor menu bar and choose *Tool bar*.
3. The *LPEX Editor* tool bar is displayed. As you move the mouse pointer over the tool bar icons one by one, the LPEX Editor information area is updated to indicate the function of each item.
4. To remove the tool bar, choose *View* and choose *Tool bar*.

Sequence Numbers

If you prefer, the LPEX Editor can display a prefix area in the source view. This prefix area is similar to the one in SEU. It contains the statement number as well as supporting line commands. To enable the sequence numbers:

1. If an LPEX Editor window is not currently open, open one.
2. From the LPEX Editor menu bar, choose *View* and *Sequence numbers*.
3. The sequence numbers are displayed in the prefix area.
4. To execute a prefix area command, type the command in the prefix area and press ALT+Enter.
5. To remove the prefix area, choose *View* and *Sequence numbers*.

Token Highlighting

The LPEX Editor highlights the tokens (specification fields) of your VisualAge for RPG program source, providing a separate color for each to improve readability.

When you make changes to a line, the token colors are updated only after you move your cursor off the line.

To see how token highlighting works:

1. Move the cursor to the first calculation specification in the VisualAge for RPG source.
2. Position the cursor to Column 7 (right next to the 'C'). See Figure 158.

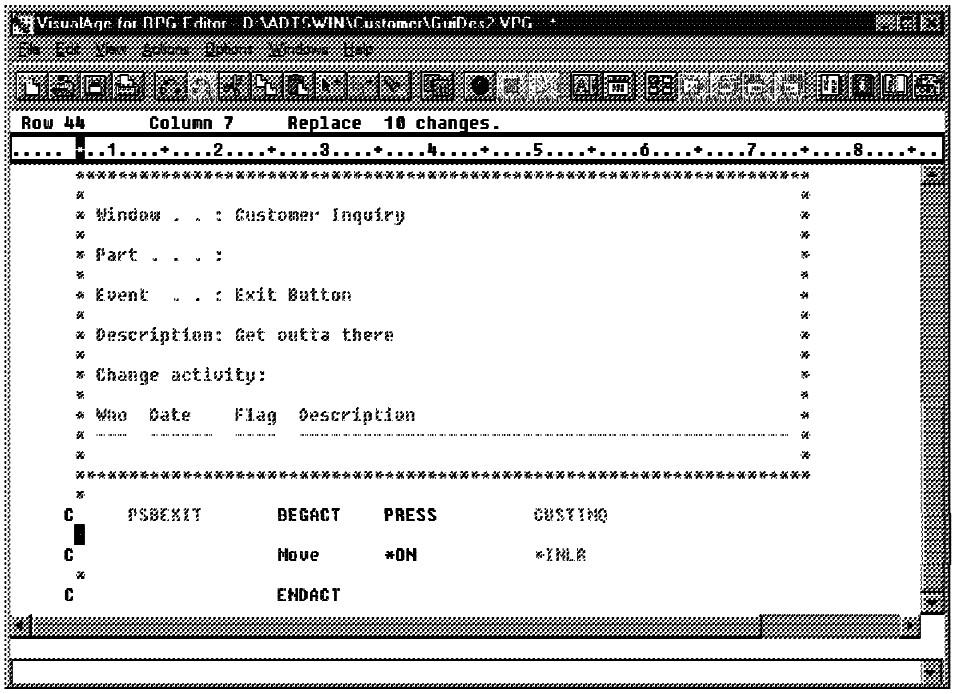


Figure 158. Positioning the Cursor to Column 7

3. Type an asterisk (*).
4. Move the cursor off the line and watch what happens. The line where the * was typed has become a comment line and its color changes accordingly.
5. Move the cursor back to Column 7 and remove the *. Move the cursor off that line of code.
6. The token highlighting changes to reflect that this is a noncommented "C" specification.

Displaying Types of Lines

Using the LPEX Editor, it is possible to have only particular types of source lines displayed at one time. To do this:

1. Choose *View* from the LPEX Editor menu bar. The options available on the pull-down menu lists the types of line selections that can be made.

2. Choose *Comments*. The LPEX Editor now shows only those VisualAge for RPG statements that are comments (see Figure 159).

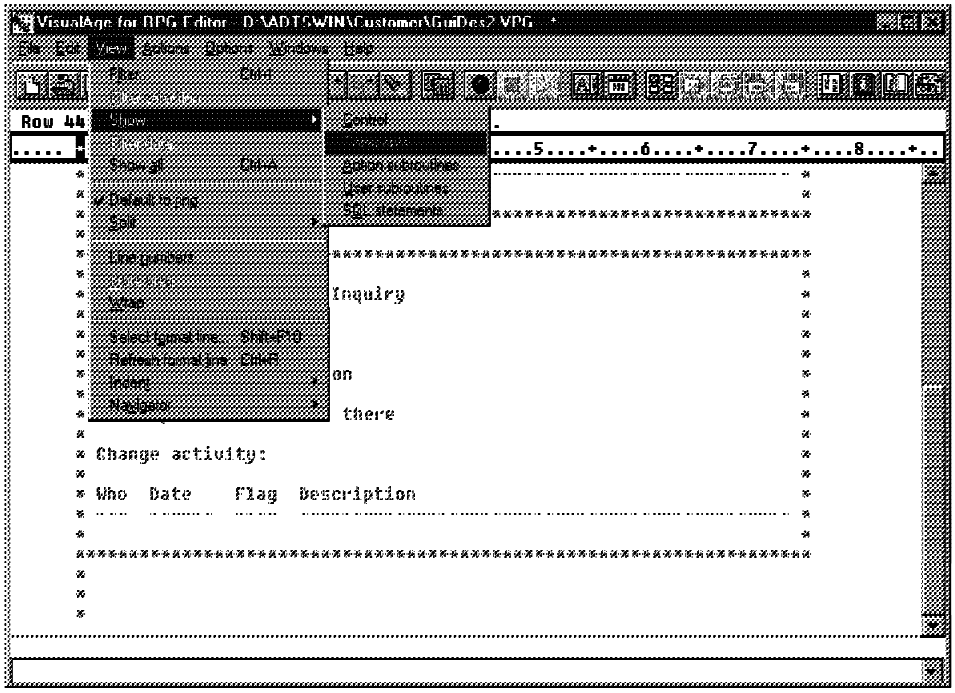


Figure 159. Only Comment Lines are Shown

3. Choose *View* from the LPEX Editor menu bar and choose *Include all*. All statement types are displayed.

   In addition, the choices in the *View* pull-down menu can be used to include only control specifications, user subroutines, and action subroutines. The *Include* choice allows the selection of only those lines containing a particular character string. As the LPEX Editor is fully programmable, it is possible to create macros to include or exclude all types of source statements based on specific criteria.

Syntax Checking

Syntax checking is available for VisualAge for RPG code and, by default, is active. The syntax of the VisualAge for RPG code is checked automatically when a change is made to a line and the cursor is moved off the line.

When errors are found, they are displayed in the message area at the bottom of the LPEX Editor.

To see how syntax checking works:

1. Move the cursor to the statement containing the MOVE operation code.

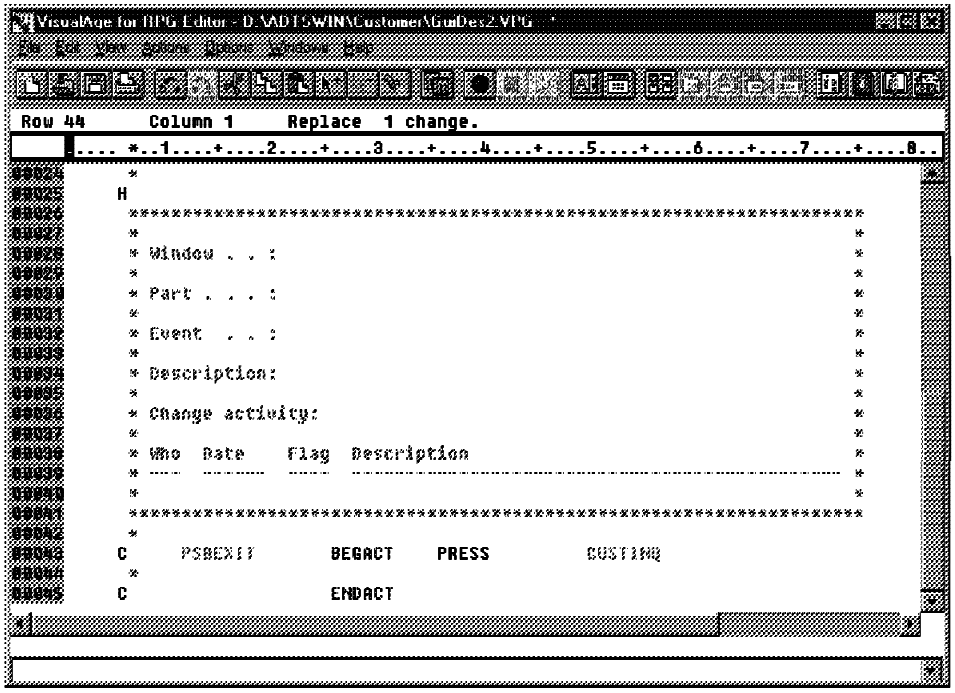


Figure 160. Edit Window for Syntax Checking Example

2. Make sure the LPEX Editor is in Replace mode. The mode indicator is directly to the right of the row and column information in Figure 160. Use the Insert key to change modes.
3. Create an error by removing the "E" from the MOVE operation code (see Figure 161).

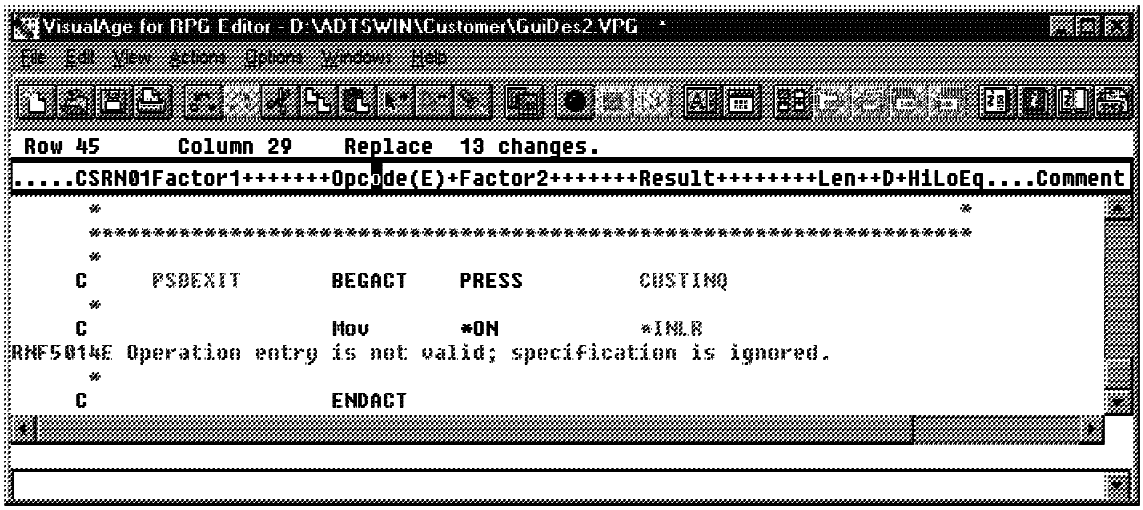


Figure 161. Creating a Syntax Error

4. Move the cursor off this statement. An error message is displayed in the message area at the bottom of the LPEX Editor window showing the syntax error (see Figure 161).
5. Correct the error by typing the character E back into the operation code MOVE.
6. Move the cursor to the next line. The message area is cleared.

Using Format Lines

The format line is at the top of the LPEX Editor window, just below the menu bar and status line. A format line is used to help keep track of the columns in a particular specification line. The content of the format line can vary to reflect the particular type of specification being keyed (for example: F specs, C specs, D specs, and so on).

To display a format line:

1. Position the cursor to any uncommented calculation specification line by clicking on the line with the left-hand mouse button, or by using the arrow keys and clicking the left-hand mouse button.
2. Choose the *View* menu bar choice and choose *Refresh format line*.
3. The format line changes to reflect a VisualAge for RPG calculation specification since the cursor was on a C-specification when the format line was requested.

Tip

The format line can also be refreshed by pressing Ctrl+R with the cursor on a statement.

4. You can move the cursor right or left with the arrow keys to go from character to character, or with the Tab key to go from field to field. An indicator on the format line moves with the cursor to show in which column the cursor is positioned. Try moving the cursor and watching the indicator on the format line.
5. Move the cursor to a comment line (an asterisk in Column 7).
6. Choose the *View* menu bar choice and choose *Refresh format line* (or press Ctrl+R). A format line for comment specifications is shown since the cursor was on that type of specification when you requested a refresh.
7. To select a format line for any specification you want, choose the *View* menu bar choice and choose *Select format line*. The VisualAge for RPG Format Line Selection window is shown.
8. Select the *C-Calculation* radio button and press the *OK* push button. The format line changes to reflect a calculation specification.

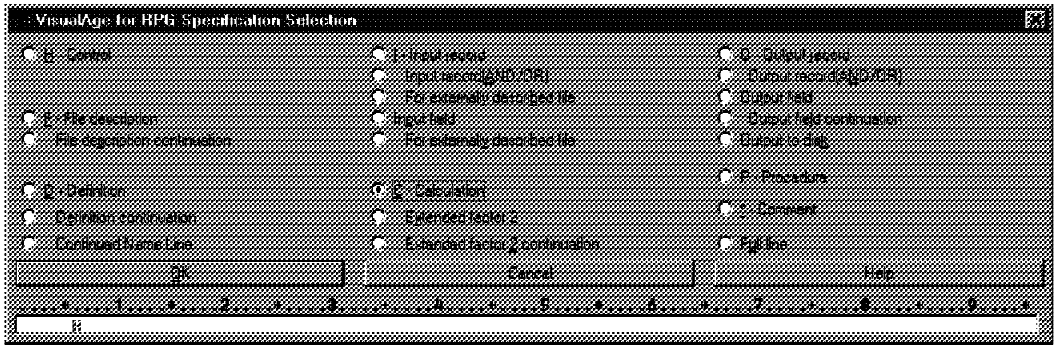


Figure 162. Format Line Selection

Now that you have created the push button logic and explored some of the features of the LPEX Editor, save your work by choosing *File* and *Save* from the LPEX Editor menu bar. Close the LPEX Editor by double-clicking on its system menu.

The next step is to build your application.

First close the *Customer Inquiry* design window by positioning the mouse pointer anywhere on its title bar and clicking with the right-hand mouse

button. From the pop-up menu, choose *Close*. The closure process is shown in Figure 163.

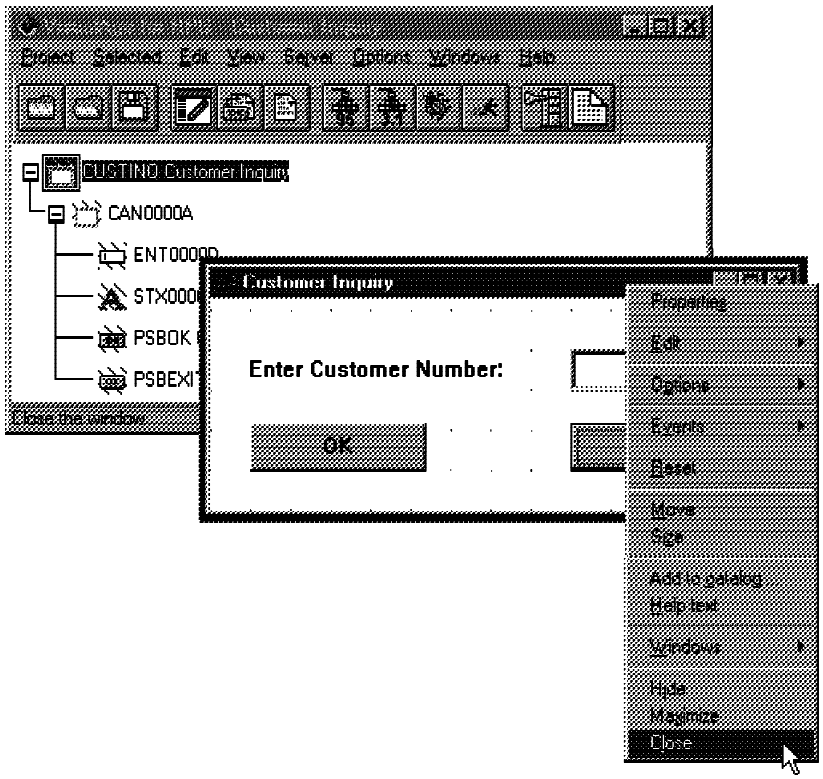


Figure 163. Closing the Design Window

Building the Application

Before the application can be run, it must be built.

With VisualAge for RPG, you can build an application that runs under Windows 95 and NT or under Windows 3.1. To indicate which program type you are going to create, you need to select the appropriate options. Before starting the build, let's have a look at the build options dialog that allows you to specify the options used to build this project (Figure 164).

1. Choose *Project* and *Build options* from the GUI Designer menu bar. Since we are building a Windows NT/95 application, choose *Windows NT/95* from the sub-menu.
2. The VisualAge for RPG *Windows NT/95 Build Options* dialog is shown

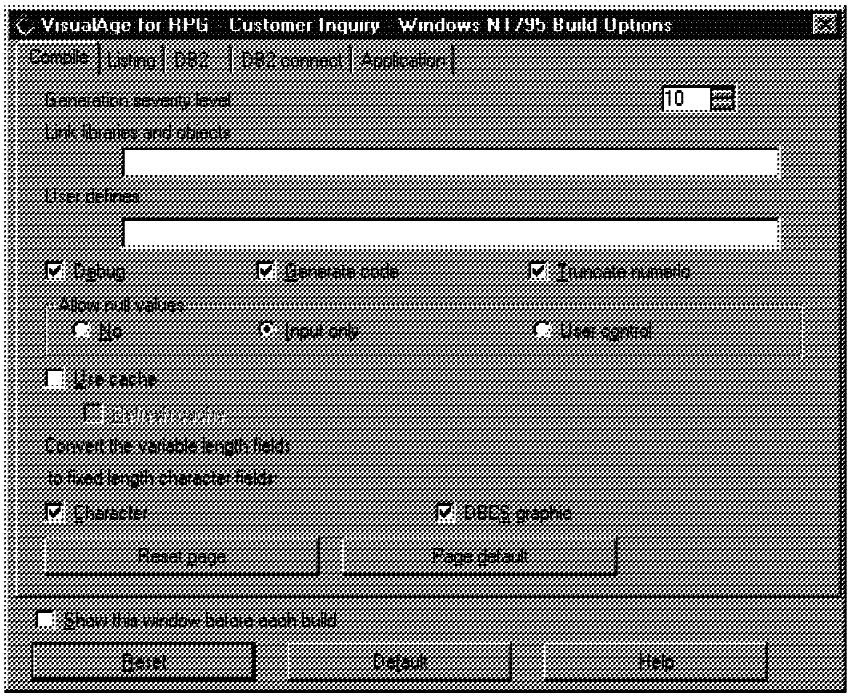


Figure 164. Build Options Dialog for the Customer Inquiry Example

3. Notice the check box at the bottom of this window, *Show this window before each build*. If this check box is selected, the *Build Options* dialog is displayed each time a project build is performed.
4. Go to the *Application* page of the *Build Options* dialog.
5. The icon shown on this page is the icon that represents the application executable on the desktop if the build is successful. For this exercise, we use the default icon.
6. Close the *Build Options* dialog.

Now, let's build the project:

1. Choose *Project* from the GUI Designer menu bar and choose *Build*.
2. The *VisualAge for RPG - Save project* window appears. Press the *Save* push button to save your project.
3. A status window is now shown to indicate that the project is being built. When the build is complete, another window is shown to indicate whether or not the build is successful.

Running the Application

If the build is successful, you are now ready to run the application. There are two ways to do this:

1. From the *Project* pull-down menu, choose *Run*.
2. Press the tool bar icon for *Run the project*.

The window you designed is displayed. Press the *OK* push button and check whether your logic is correct and the background color of the push button changes.

To terminate the program, press the *Exit* push button.

This concludes the first exercise. End the GUI Designer by double-clicking on its system icon.

Exercise 2. Exploring the GUI Designer and Accessing AS/400 Databases

During this exercise, you work on the application that you created in Exercise 1. The following tasks are covered:

- Starting the VisualAge for RPG GUI Designer for an existing application
- Customizing the VisualAge for RPG GUI Designer
- Creating a user-defined part and adding it to the Parts Palette and Parts Catalog
- Creating an instance of a user-defined part
- Referencing fields in a DB2/400 database
- Performing more alignment, sizing, and spacing

As a result of this exercise, you create a window that is used to display customer information using the input from the window you created in Exercise 1.

Objective

As a result of this exercise, you should be able to customize the VisualAge for RPG GUI Designer to suit your needs and access an AS/400 system to:

- Use existing DB2/400 field definitions.
- Use externally described AS/400 files in your VisualAge for RPG program.
- Read data from the AS/400 system.
- Read and write data from or to a window.
- Build a VisualAge for RPG application accessing AS/400 data.
- Run a VisualAge for RPG application accessing AS/400 data.

Starting GUI Designer for an Existing Project

When a VisualAge for RPG project has been saved, an icon that represents it is shown on the desktop with the text that was specified when the project was first saved.

Note

The icon is shown on the desktop if that is the location you chose on the *Save as* dialog, as is the case in Exercise 1. It is possible to save a project in another folder, in which case the icon is not visible until you open that folder.

There are two methods of opening a VisualAge for RPG project. The first involves dragging and dropping the project on the *VisualAge for RPG GUI Designer* icon. For this exercise, we use the second method, opening a project from its pop-up menu:

1. Locate the your project icon on the desktop.
2. Invoke its pop-menu by clicking the right-hand mouse button.
3. From the pop-up menu, choose *Edit*.
4. The VisualAge for RPG initialization window is displayed while GUI Designer opens the project.

Customizing Components in VisualAge for RPG GUI Designer

Many of the VisualAge for RPG GUI Designer components can be customized to suit your personal preferences. In the following tasks, you perform some customization.

Customizing the Project View

As mentioned, the project view shows you the parts that make up your project. By default, the project view is in tree view. To change the view, follow these steps:

1. Choose *View* from the GUI Designer menu bar.
2. Choose *Icons*. The project view changes to display all design windows with their name. Now let's return to the tree view.
3. Choose *View* from the GUI Designer menu bar and *Tree* followed by *Icons and Text*.

   The project view is refreshed and plus signs (+) are displayed next to the window parts. A plus sign in a tree view indicates more items, or in the case of the GUI Designer, parts in the window.
4. To display the additional parts, click on the plus sign with the left-hand mouse button to expand it. Another icon named CANxxxxx is shown. This is the *Canvas* part of the window.

5. Click on the + sign beside the CANxxxxx icon. The *Canvas* part is expanded to show all of the parts you have created so far on this window.

> **Tip**
>
> You can also change the project view by clicking with the right-hand mouse button on the *project view*. A pop-up menu is displayed from which you can choose a particular view.

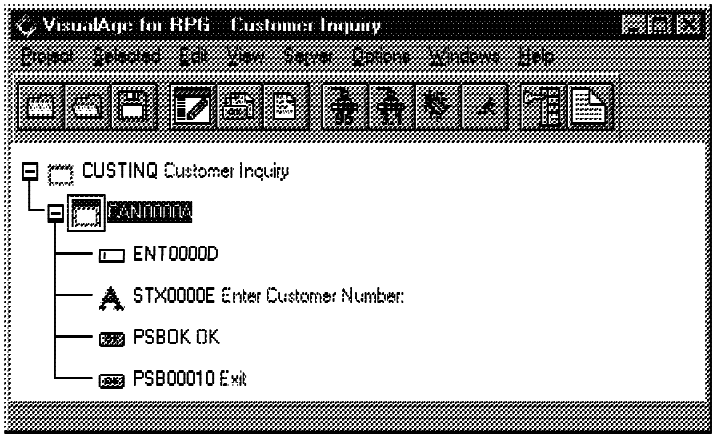


Figure 165. Tree View of the Customer Inquiry Project

Customizing the Tool Bar

The tool bar can be customized by adding and removing icons. To see how you can customize the tool bar, you add a new icon to it. These are the steps:

1. Choose *Options* from the GUI Designer menu bar and *Tool bar* followed by *Change...*. The *Customize Tool Bar* dialog is shown.

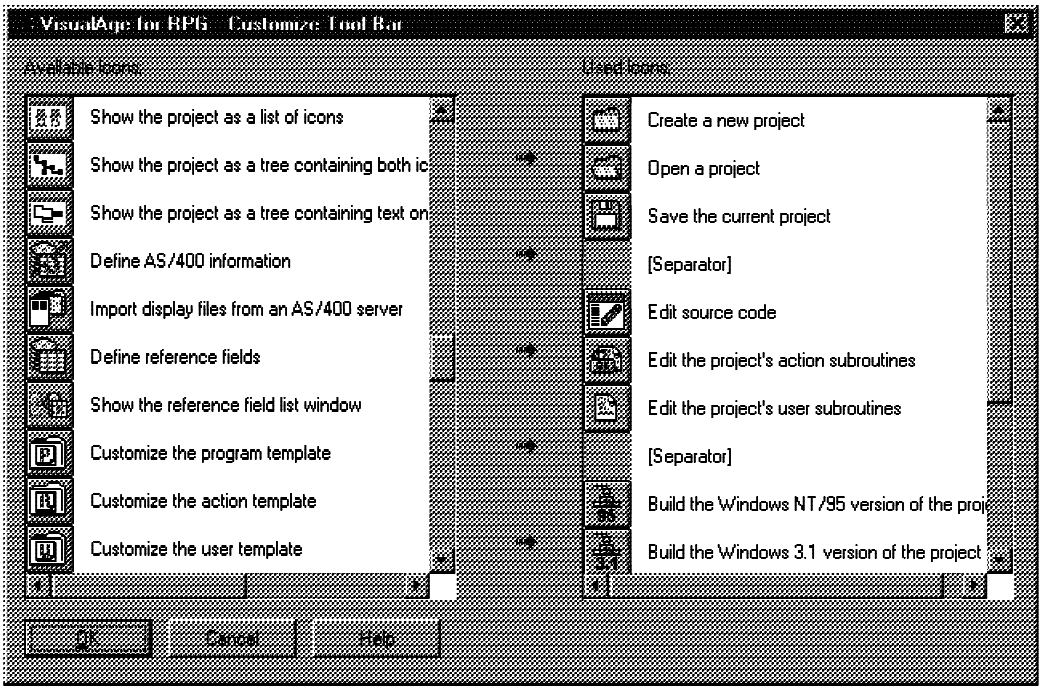


Figure 166. Customizing the Tool Bar

Figure 166 shows all available icons on the left and the icons currently selected for the tool bar on the right.

2. Move the mouse pointer over the *Define Reference Fields* icon in the left-hand box, and press and hold the right-hand mouse button.
3. Drag the icon to the right-hand box, place it below the *Define Logon Information* button and release the mouse button.
4. To place a separator between two icons on the tool bar, choose the *Separator* icon on the left and drag it to the right above the *Define Logon Information* icon. A space is added between the *Define Reference Field* icon and the *Define Logon Information* icon.
5. Press the *OK* push button to apply the changes to the tool bar.
6. Notice that the GUI Designer menu bar has been updated to reflect your changes (Figure 167).

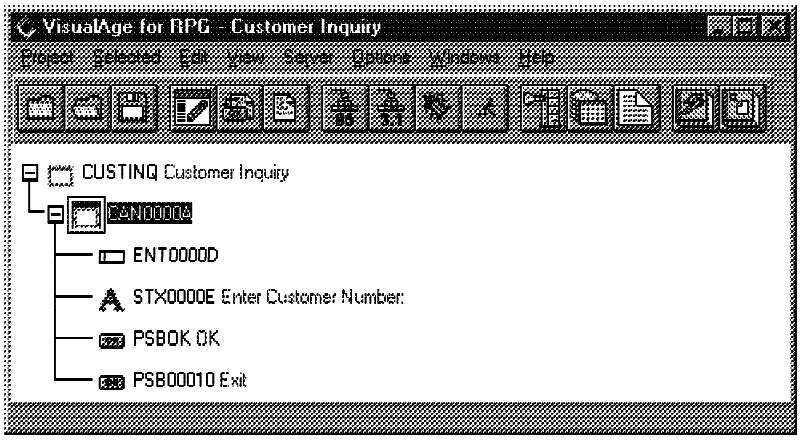


Figure 167. Moving a Tool Bar Push Button

Customizing the Parts Palette

You can customize the Parts Palette to contain only those parts that you use most often.

1. Find the *Group box* part on the Parts Palette.
2. Invoke the *Group box* pop-up menu by clicking on the right-hand mouse button while the mouse pointer is over it.
3. Choose *Remove* (Figure 168).

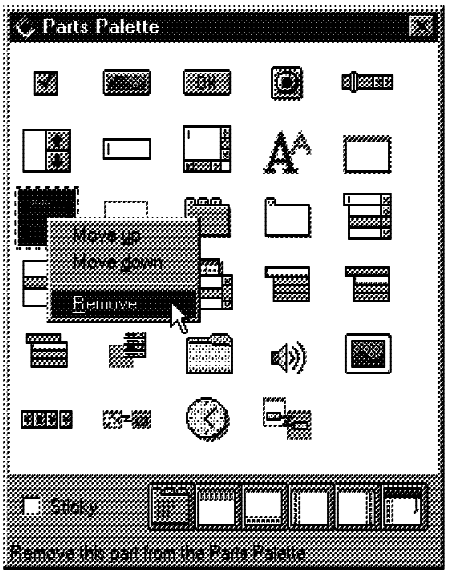


Figure 168. Customizing the Parts Palette by Removing a Part

The *Group box* part has been removed from the Part Palette. If you exit the VisualAge for RPG GUI Designer at this point, the Parts Palette does not contain the *Group box* part when the GUI Designer is invoked again.

The Group box part has been removed from the Parts Palette only, not from the GUI Designer. To return the *Group box* part to the Parts Palette, follow these steps:

1. Click on the left-most icon (showing a notebook) below the palette in the *Parts Palette* window.
2. The *Parts catalog* is displayed.
3. Select the *Frames* tab. As you can see, the *Group box* part is still present in the Parts catalog.

   Note

   As the Group box part is an IBM-supplied part, you cannot modify it (for example, change its name or icon) or delete it from the Parts catalog. You can perform those actions on user-defined parts and imported parts only.

4. Invoke the pop-up menu for the *Group box* part.
5. Choose *Add to palette*.
6. Click on the icon in the top right corner of the *Parts catalog* window.

7. The Parts Palette is displayed with the *Group box* in place.

Creating a User-Defined Part

When you are designing windows for your application, you are using parts supplied with VisualAge for RPG. In addition to these parts, you can create your own parts and add them to the Parts catalog and Parts Palette. These are called *user-defined parts*.

You are going to enlarge the Customer Inquiry window to accommodate some new parts:

1. Double-click on the *CUSTINQ* icon on the *Project View* to open the *Customer Inquiry* design window (if it is not currently open).
2. Move the mouse pointer over the upper border of the window so that it changes to a double arrow.
3. Press and hold the right-hand mouse button and size the window to double its vertical size so additional parts can be added.

In this case, we are adding a logo for a fictitious company. The logo consists of an image part and a static text part.

1. Find the *Image part* on the *Parts Palette* window and position the mouse pointer over it.
2. Press and hold the right-hand mouse button and drag and drop the Image part above the *Enter Customer Number:* static text in the *Customer Inquiry* window.
3. Release right-hand mouse button.
4. Invoke the properties notebook for this part by double-clicking on *Image Part*.

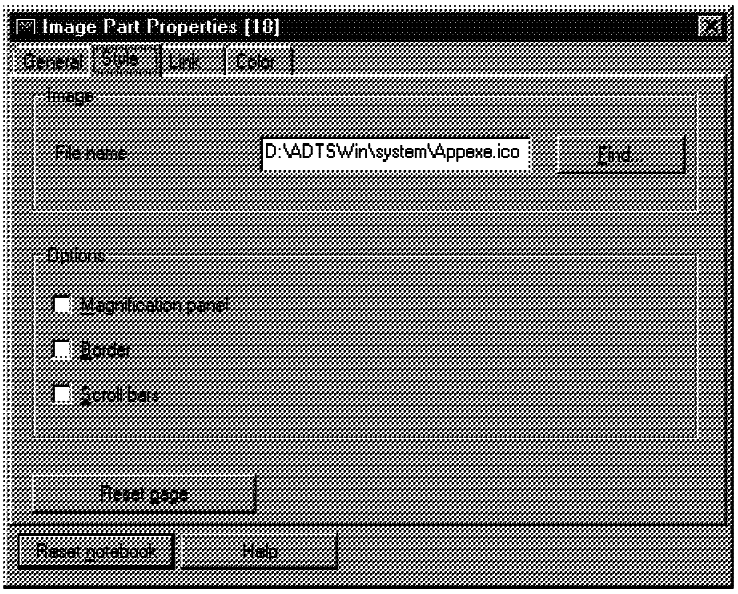


Figure 169. Setting the Image Part File Name

5. Go to the *Style* page of the properties notebook by selecting the *Style* tab.
6. Type `X:\ADTSWIN\SYSTEM\APPEXE.ICO` as the file name. This is an icon that is shipped in the VisualAge for RPG directory. Replace the `X` with the correct drive letter.
7. Deselect the *Magnification panel* as well as the *Scroll bars* setting by clicking on the appropriate check boxes.
8. Double-click on the *System* icon of the *Image Part Properties* notebook to close it.
9. Move the mouse pointer to the lower-right sizing handle of the Image part. The mouse pointer changes to a double arrow.
10. Press and hold the right-hand mouse button and size the Image part so that the icon fits into it.

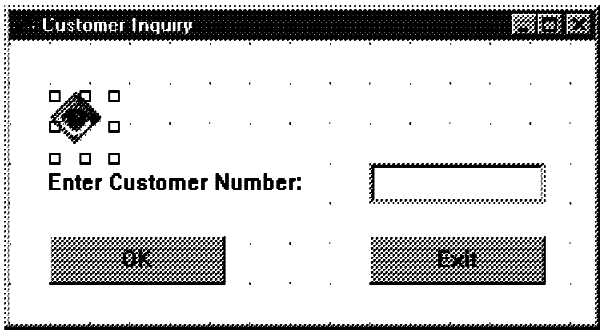


Figure 170. Sizing the Image Part

To add the company name:

1. Find the *Static Text* part on the *Parts Palette* window and position the mouse pointer on it.
2. Press and hold the right-hand mouse button, then drag the *Static Text* icon and drop it next to the *Image part* in the *Customer Inquiry* window.
3. Release right-hand mouse button.
4. Invoke the *Static Text Properties* notebook for this part by double-clicking on the icon.
5. Change the text setting to `Client Server Applications Inc.` (or any name you want to give your fictitious company).
6. Go to the *Font* page of the properties notebook.
7. Select the *Select font* radio button. The *Change font...* push button is enabled.
8. Press the *Change font* push button. The *VisualAge for RPG - Change Fonts* window is shown.
9. Click on the button on the right side of the *Name* drop-down combination box. A list of available fonts is shown (Figure 171).
10. Scroll through the list and select the *Times New Roman* font by clicking on it.
11. Change the font size to 14 in the same way.

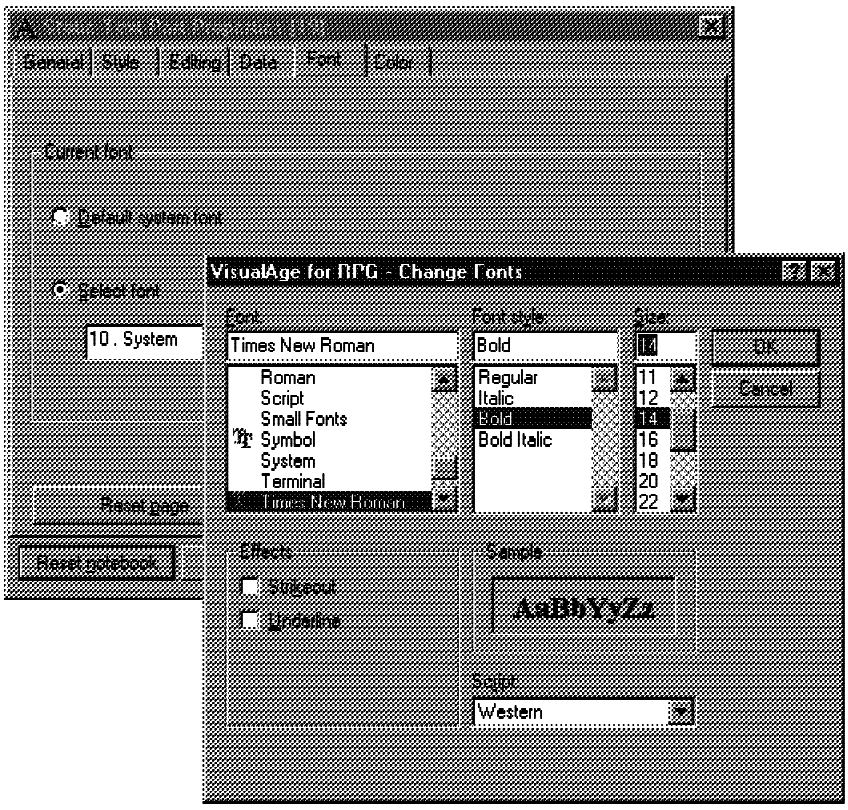


Figure 171. Change Fonts Window

12. Press the *OK* push button.
13. To change the part's color, go to the *Color* page of the properties notebook.
14. Select the *Apply to Foreground* radio button.
15. Select the *Predefined colors* radio button. The combination box containing all available colors is enabled.
16. Select *Red* from the combination box.
17. Go to the *Style* page.
18. From the *Vertical alignment* options, select *Center*.
19. Double-click on the *System* icon of the *Static Text Part Properties* notebook to close it.

Because the company name does not fit in the default size of the *Static Text Part* (Figure 172), you need to enlarge this part:

1. Click on the *static text* part holding the company name to make it the active part. The borders of the part are marked by placing the sizing handles around it.
2. Move the mouse pointer to the lower-right sizing handle so that the mouse pointer becomes a double arrow.
3. Press and hold the right-hand mouse button and move the mouse pointer to the right and down until the entire company name is visible.
4. Release the mouse button.

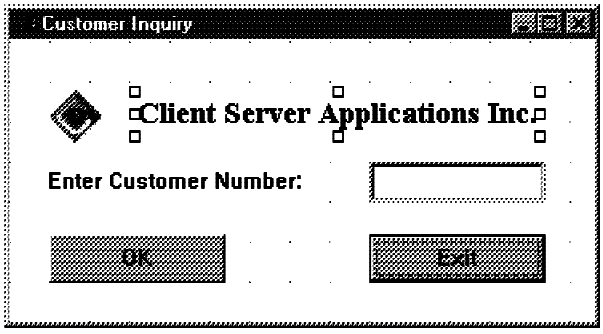


Figure 172. Before Aligning the New Parts

Now the image part and the company name should be aligned with each other. Using the mouse is the easiest way to select them, but in this exercise, you use the keyboard instead. It is important to know this method since there may be times when using the mouse selects parts you do not want to be aligned. For example, you may want to align two parts without affecting an intervening part.

To align the image with the static text:

1. Select the *Image* part with the mouse pointer.
2. Press the right-arrow key until the *Static Text* part with the company name is highlighted.
3. Press and hold the CTRL key and press the space bar. This selects the *Static Text* part without deselecting the *Image* part.
4. Invoke the pop-up menu of the *Image* part by pressing the right-hand mouse button while the mouse pointer is placed over the part.
5. Choose *Align* and the second alignment choice offered, to horizontally align the company name to the company logo.
6. If necessary, resize the window so that all of the contained parts reside in the middle of the window.

The intent of this exercise is to use these two parts as a logo that will be reused on other windows. To do this, you create a user-defined part from these parts and add it to the Parts Palette and the Parts catalog. You can then add your company's logo and name by just dragging the new user-defined part onto any design window and dropping it on the canvas.

To create the user-defined part:

1. Move the mouse pointer to the left and above the *Image part*.
2. Press and hold the left-hand mouse button and select the *Image Part* containing the company's logo and the *Static Text Part* part containing the company's name by dragging the mouse so that both parts are within the selection rectangle.
3. Release the mouse button.
4. Invoke the pop-up menu of either of the two parts.

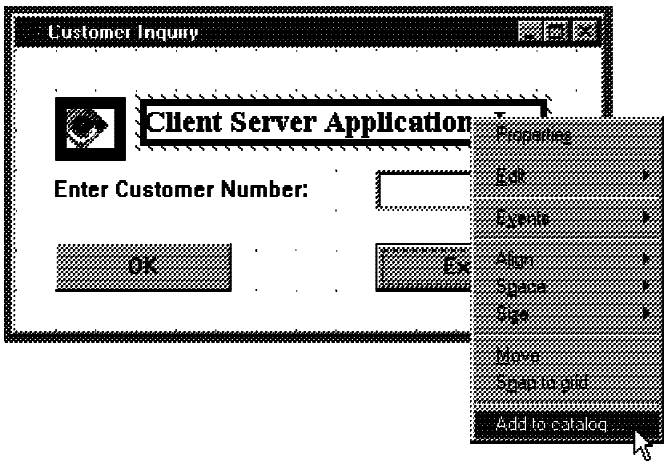


Figure 173. Adding a User-Defined Part to the Parts Catalog

5. Choose *Add to catalog*. The *Create User-Defined Part* window is shown.
6. Change the *Part name* to `Company`.
7. Press the *Find...* push button to locate an icon to represent this user-defined part. This icon is shown on the *Parts Catalog* and *Parts Palette*. The *Find an icon* for the part dialog is shown.
8. Type `x:\ADTSWIN\SYSTEM\APPEXE.ICO` in the file name entry field and press the Enter key. Replace the `x` with a valid drive letter. When you return to the *Create User Defined Part* window, the selected icon is displayed (Figure 174).

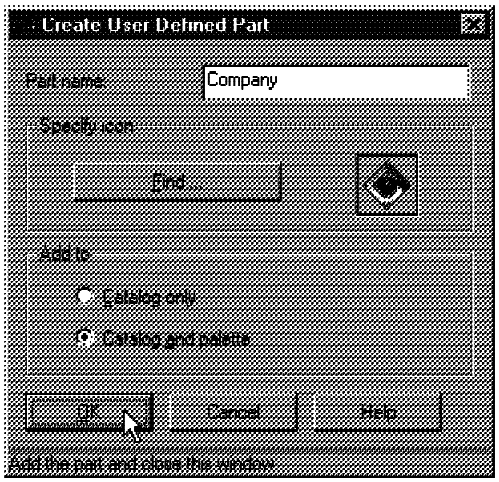


Figure 174. Create User Defined Part WiIndow

9. Press the *OK* push button to add the new part to the Parts Catalog and the Parts Palette.
10. Size the Parts Palette so that you can see the new user-defined part.

    Note: As you move the mouse pointer over the user-defined part on the Parts Palette, its name appears in the information area.

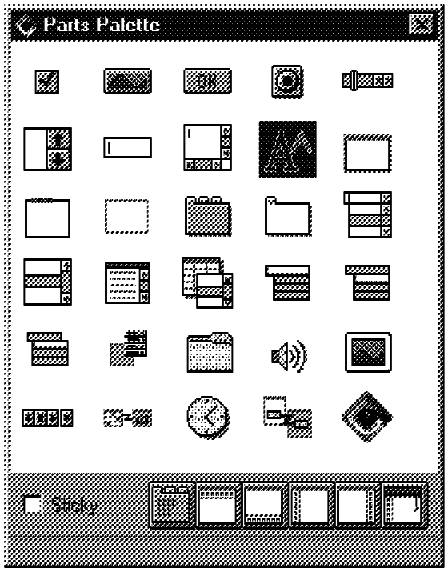


Figure 175. Parts Palette with a User-Defined Part

Close the *Customer Inquiry* window:

1. Invoke its pop-up menu by moving the mouse pointer to the title bar and pressing the right-hand mouse button.
2. Choose *Close*.

Adding a New Window to Your Project

During this part of the exercise, you create a new *Window with Canvas*, change its attributes, and add parts to it. This window shows the customer information for the customer whose number was typed in the CUSTINQ window:

To create the new window:

1. Find the *Window with Canvas* part on the *Parts Palette* and position the mouse pointer on it.
2. Press and hold the right-hand mouse button.
3. Move the mouse until the Window part is placed somewhere within the *Project View* of the VisualAge for RPG window.
4. Release the mouse button. A new design window is shown and an icon is placed in the Project View.
5. If necessary, move the window so that it fits on the desktop.

To change the settings for the new window:

1. Double-click on the Window's *title bar* to open its properties notebook.
2. Change the part name on the *General* page of the *Window Part Properties* notebook to `CUSTINFO` and its title to `Customer Information`.
3. Select the *Startup* tab.
4. Deselect the *Open immediately* option of the *Display style* settings (Figure 176). This allows your program to control when the window is displayed.

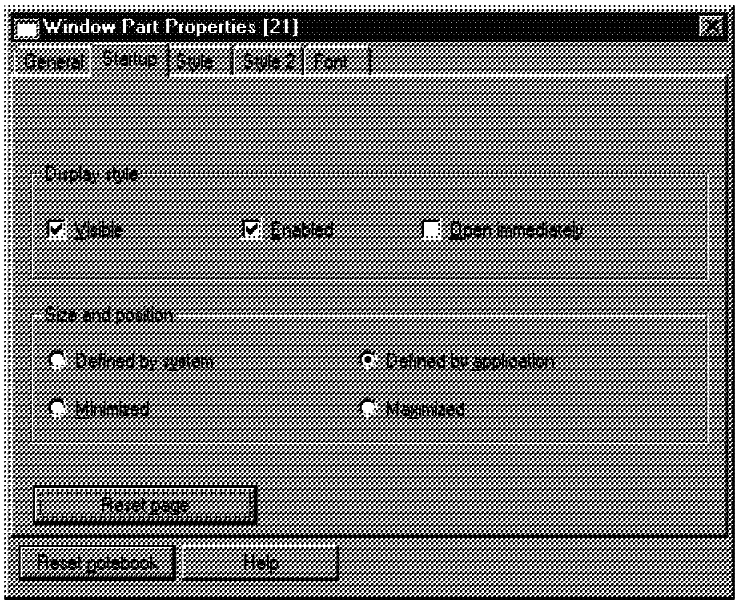


Figure 176. Window Part Properties Notebook

5. Double-click on the *System* icon of the *Window Part Properties* notebook to close the notebook.

Creating an Instance of a User-Defined Part

Now you add the company logo and name to this window as you did for the *Customer Inquiry* window at the beginning of this exercise. This time, however, use the user-defined part you previously created. These are the steps;

1. Find the user-defined Company part on the Parts Palette.
2. Drag and drop this part on the *Customer Information* window below the title bar.

Using an AS/400 File as a Field Reference File

During this part of the exercise, you learn how to add fields to the GUI using an AS/400 file as a field reference file and the Define Reference Field feature of VisualAge for RPG. You are using a file on the AS/400 system that contains the fields required for the *Customer Information* window.

The following entry fields are added to this window:

- Customer number
- Customer name
- Contact person

- Phone number
- Customer address consisting of address, ZIP code, and city
- ZIP location

Creating Field Headings

First, you create the field headings for these entry fields by using the static text part:

1. Find the static text part in the Parts Palette.
2. Drag it to and drop it at the left side of the window below the Company part.
3. Press and hold the ALT key and select the *Static Text* part with the left-hand mouse button. The text becomes editable.
4. Type `Customer Number` as the text of the static text part.
5. Apply this text to the static text part by clicking the left-hand mouse button anywhere outside of the part.
6. Select the *Static Text* part and resize it with the right mouse button so the entire text is displayed (Figure 177).

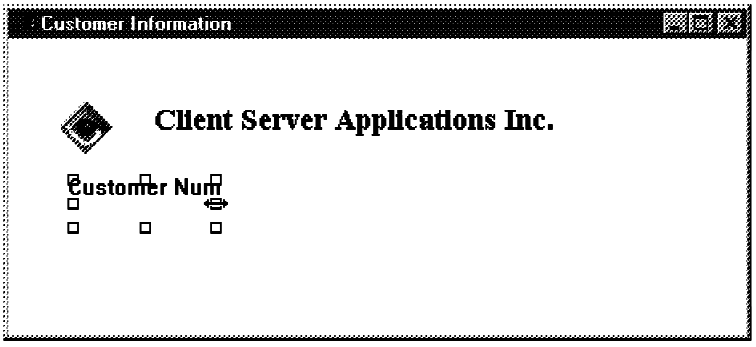


Figure 177. Resizing the Static Text Part

Repeat these steps for the other fields of the *Customer Information* window. Do not worry about the alignment of the parts at this point; alignment comes later. After you add the static text parts, the *Customer Information* window should resemble the example in Figure 178

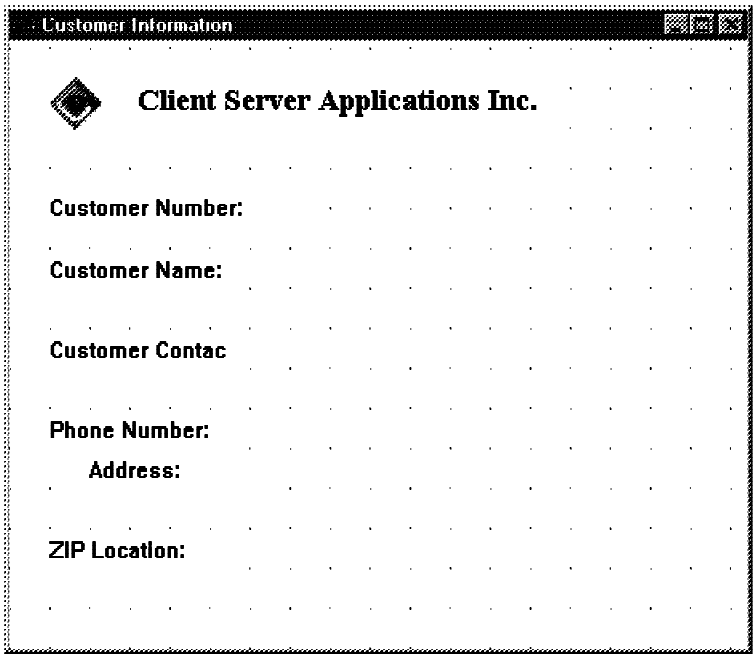


Figure 178. Design Window with Logo and Field Headings

Adding the Entry Fields

You now add the entry fields to the *Customer Information* window. This time, however, rather than using the Parts Palette, you create the entry fields using the Define Reference Fields feature of VisualAge for RPG. This feature allows you to create entry fields by direct reference to fields in a DB2/400 database file.

Defining the Server

Before you can use the Define Reference Fields feature, you have to tell the GUI Designer which server or AS/400 system you want to use:

1. Choose *Server* from the GUI Designer menu bar.
2. Choose *Define server logon...*.
3. Check for the AS/400 server name in your installation or define a new AS/400 system. Figure 180 shows two defined systems in VisualAge for RPG.

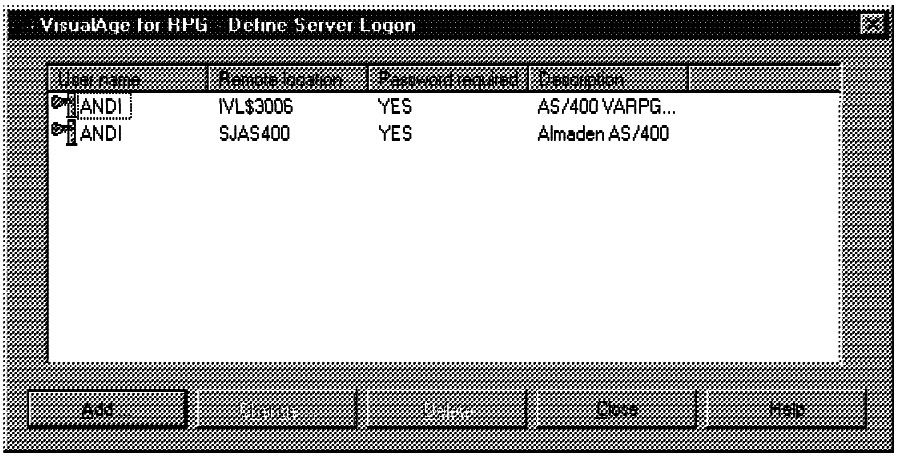


Figure 179. Define Server Logon

4. You should also have a defined router to the AS/400. Figure 180 shows a connected server system with three active conversations.

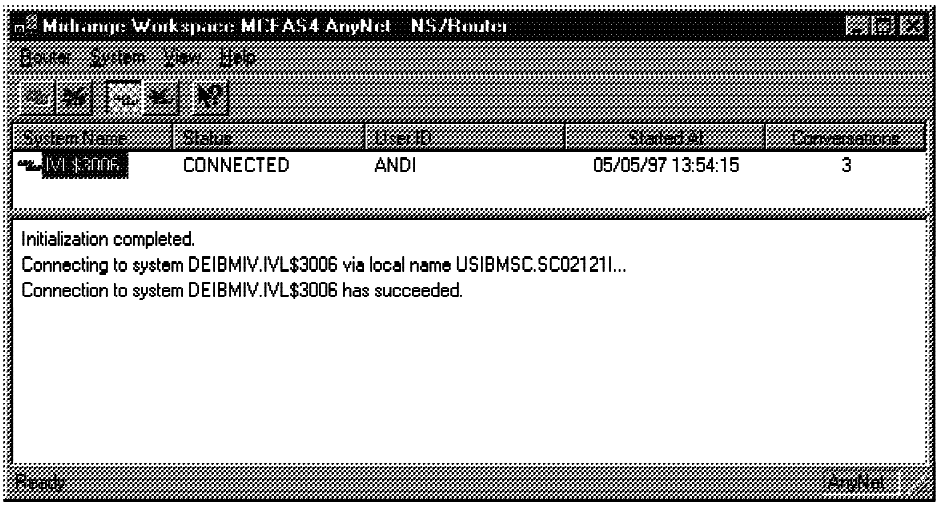


Figure 180. Midrange Workspace — NS/Router

The above system is defined in your system's environment. Make sure you have a working connection to your AS/400 as shown in Figure 181.

For the field reference file, a file named CUSTOML3 is used. It is located in library GuiDes2 on the AS/400 server you just specified. To obtain a list of the fields in this file:

1. Choose the *Server* menu item from the GUI Designer menu bar.

2. Choose the *Define reference fields...* menu item. The *Define Reference Fields* window is shown (Figure 182)

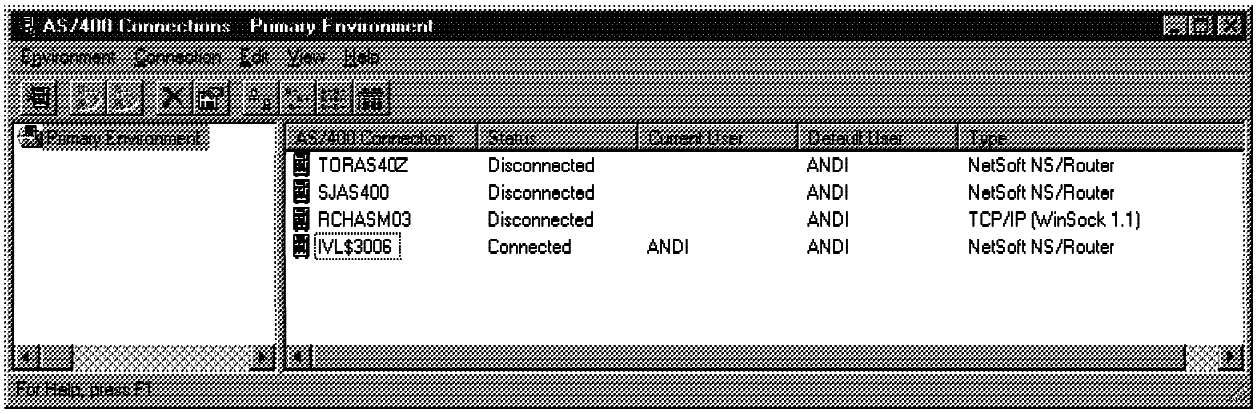


Figure 181. AS/400 Connections

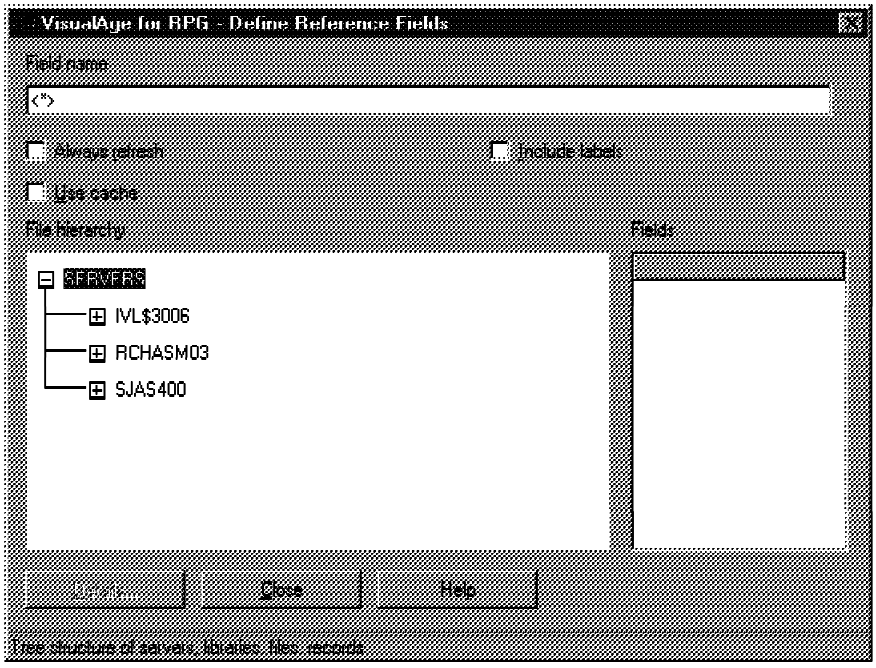


Figure 182. Define Reference Fields Window, Server List

Note

This list is different on your workstation. Select the server you specified earlier.

3. Expand the correct server name in your server tree by clicking on the plus (*+*) sign next to its name in the file hierarchy list. Since this may be the first time you are accessing this server, the *VisualAge for RPG—Logon* dialog may be shown (Figure 183) asking you for your AS/400 user ID and password.
4. Type a valid user ID and password in both fields to log on to the AS/400 (Figure 183).

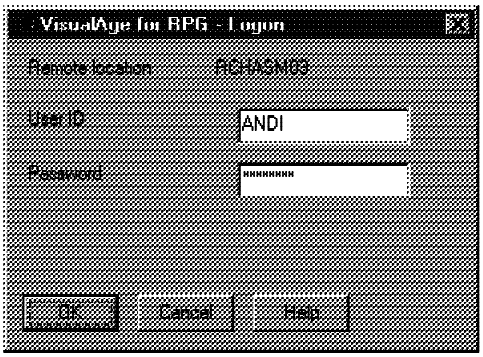


Figure 183. Logon Window

5. Press the *OK* push button to log on to the AS/400 system. If the logon is successful, the libraries of your initial library list are displayed (*LIBL).
6. Find the library GuiDes2 in this list and expand it by clicking on the + sign. The logical and physical files in this library are displayed.

Note

If the library GuiDes2 does not appear in the list of libraries, it may not be in your library list. In that case, type GuiDes2 in the entry field at the top of this window next to the server name and press Enter.

7. Click on the *+* next to CUSTOML3. A list of the record formats for this file is shown (Figure 184).
8. Double-click on the only available format name *CUSTOM01*. All fields in this record format are shown in the *Fields* list.

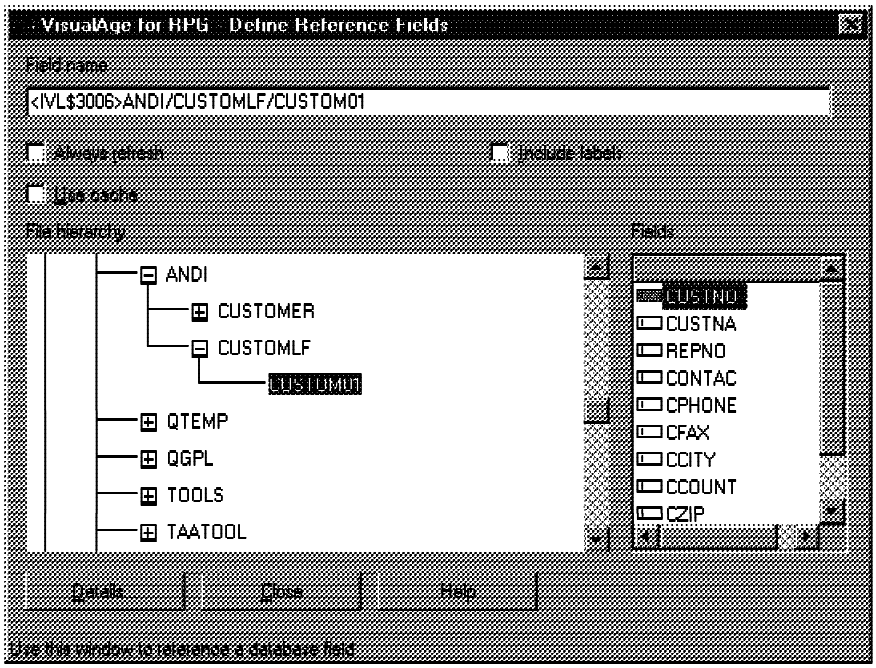


Figure 184. Define Reference Fields—Field List

9. Double-click on the first field, CUSTNO. The VisualAge for RPG—Database Field Description window is shown with detailed information about the field.
10. Press the *OK* push button to close this window.

You can now create the necessary entry field parts for the *Customer Information* window (Figure 185) using some of the fields of the CUSTOM01 record format as references:

1. Rearrange the *Customer Information* window and the *VisualAge for RPG—Define Reference Field* window so that you can see them both.
2. Move the mouse pointer over the *CUSTNO* entry in the fields list.
3. Press and hold the right-hand mouse button and drag the mouse pointer next to the *Customer Number* static text part on the *Customer Information* window.

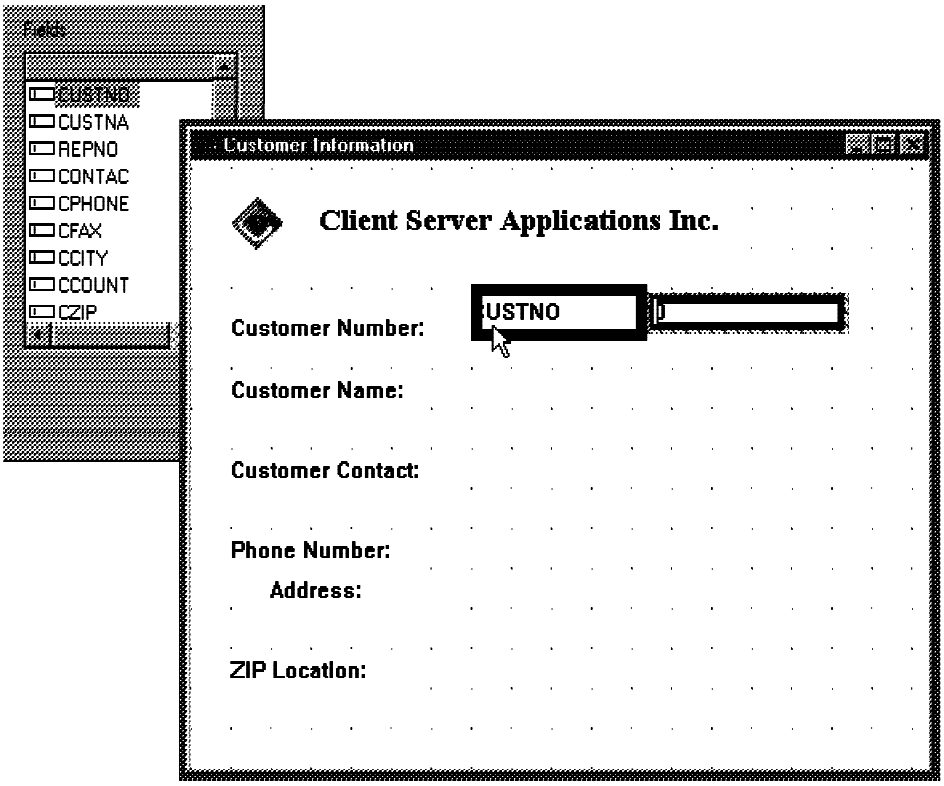


Figure 185. Dragging a Reference Field

4. Release the mouse button. An entry field part is created.
5. Double-click on this new entry field part to invoke its properties notebook (Figure 186).
6. Go to the *Reference* page of the *Entry Field Part Properties* notebook. This page shows the AS/400 database file this field is taking its data from.

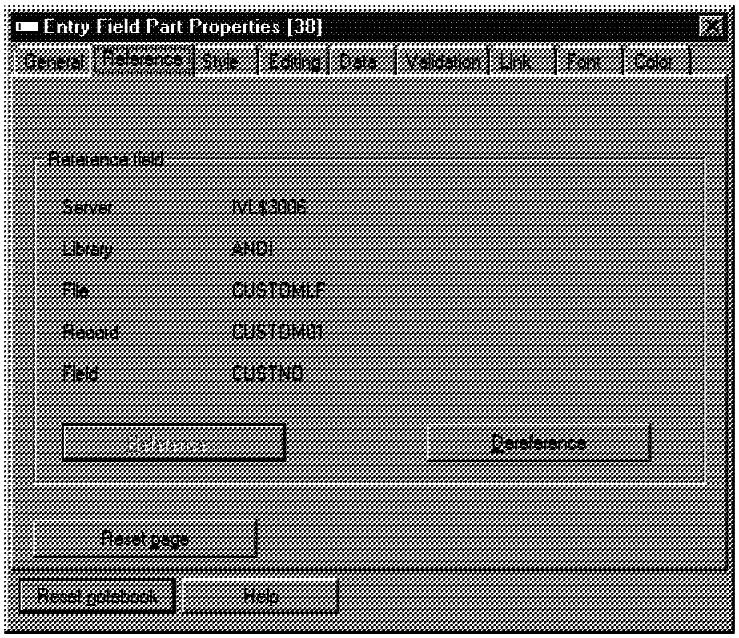


Figure 186. Reference Field Page

7. Go to the *Style* page of the properties notebook and select the *Read only* check box to prevent the customer number from being changed.
8. Go to the *Data* page of the properties notebook and verify that these attributes have been adopted from the referenced field.
9. Double-click on the *System* icon of the properties notebook to close it.

Create the other entry field parts for the *Customer Information* window in the same way, using the following AS/400 fields as a reference:

- CUSTNA
- CONTAC
- CPHONE
- CADDR
- CZIP
- CCITY
- CZIPLO

An example of the completed window is shown in Figure 187.

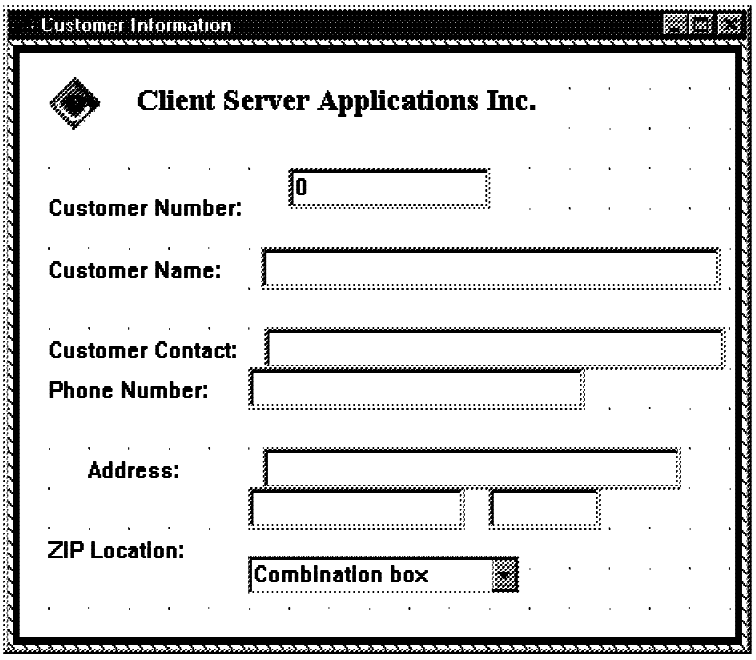


Figure 187. Window with Entry Fields

Note that the part created from the *ZIP location* field CZIPLO is not an entry field part, but a combination box part. This is because the VALUES DDS keyword is specified for this field. You can check this by looking at the details for this field (double-click on *CZIPLO* in the *Fields* list of the *Database Reference Field* window.

Close the *Database Reference Field* window by pressing the *Close* push button.

More Alignment, Sizing, and Spacing

Note

Feel free to skip some of the alignment and spacing exercises suggested here if you feel comfortable with using these tools.

Now that all the parts required to make up the *Customer Information* window are created, they still need to be arranged within the window. First, you align all static text parts containing the description of the information shown in the Customer Information window to the same vertical axis:

1. Move the mouse pointer above the static text part *Customer Number* and to the left of all of the static text parts.

2. Press and hold the left-hand mouse button and drag to create a selection rectangle that covers all static text parts.
3. Release the mouse button. All static text parts become highlighted.
4. Get the pop-up menu of the *Customer Number* static text part by clicking on the right-hand mouse button with the mouse pointer located somewhere over this part.
5. Choose *Align* and the fourth alignment choice offered. All static text parts are aligned to the left.

Tip

In case you selected the wrong alignment option and the selected parts overlay each other, you can use the Undo function in the GUI Designer to restore the parts to their original position. To undo the alignment, choose *Edit* from the menu bar and choose *Undo*.

The next step is used to even out the spaces between the static text parts *Customer Name*, *Customer Contact*, *Phone Number* and *Address*.

1. Select these static text parts in the same way you just did for vertical alignment.
2. Invoke the pop-up menu of one of the parts.
3. Choose *Space* and the fourth menu item.

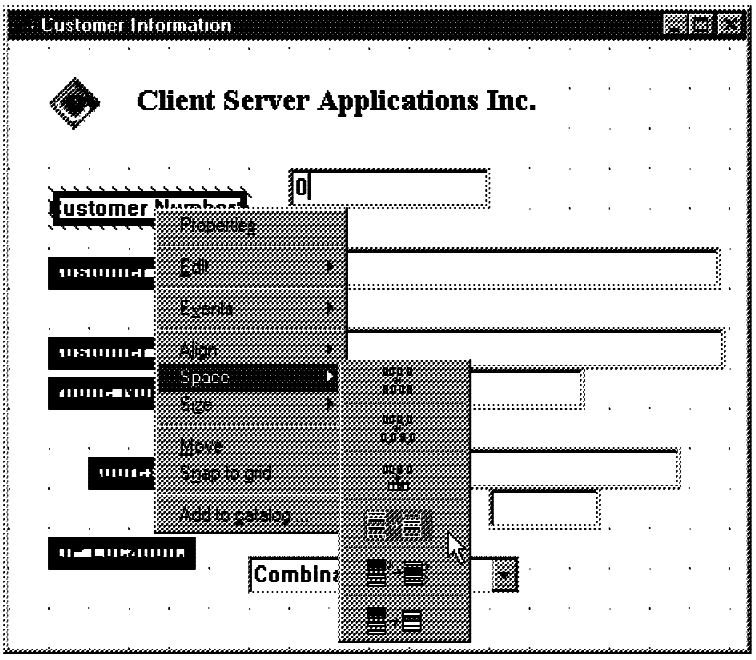


Figure 188. Spacing Static Text Parts

This evens out the spaces between the selected parts.

Now you align each entry field part with its corresponding static text part:

1. Draw a selection rectangle around the *Customer Number* static text part and the *CUSTNO* entry field part.
2. Invoke the pop-up menu for the *Customer Number* static text part.
3. Choose *Align* and the second alignment choice. The selected parts are aligned to an imaginary horizontal line running through the center of the Customer Number static text part.
4. Repeat these steps for the *Customer Name* static text part and the *CUSTNA* entry field part as well as for the *Address* static text part and the *CADDR* entry field.
5. For the *ZIP Location* static text part and the *CZIPLO* combination box part, you have to select the first alignment choice offered to get them aligned properly (Figure 189). This is because a combination box part consists not only of an entry field, but also of the value list, which makes it vertically larger than a static text part.

Note

You may need to drag the selection rectangle outside the *Customer Information* window to get the combination box selected.

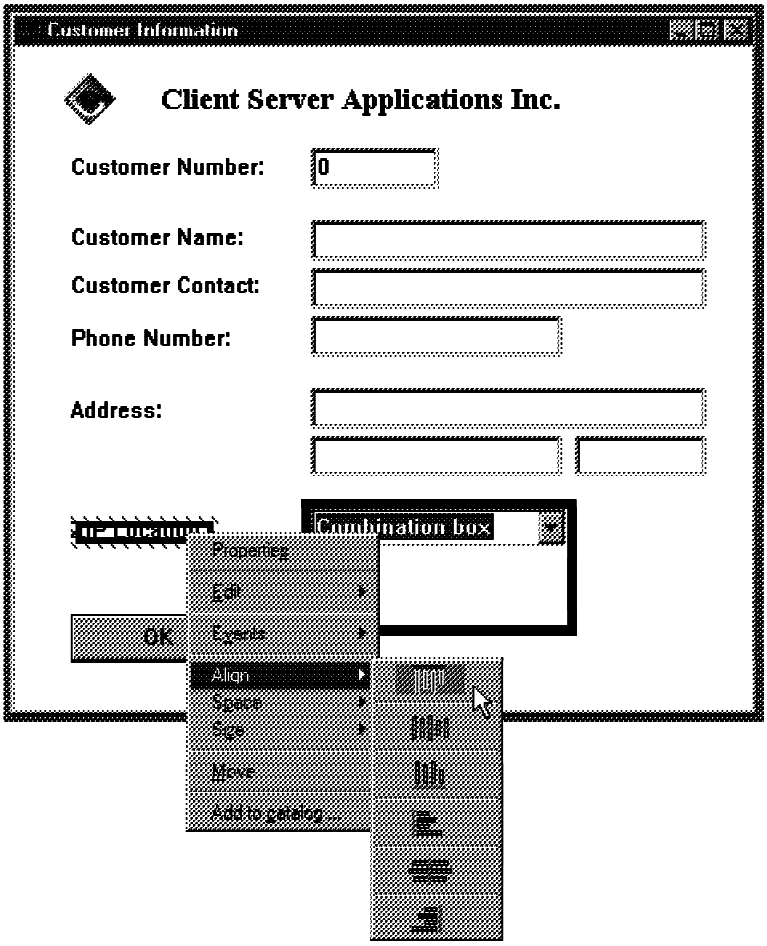


Figure 189. Alignment

To align the *CONTAC* entry field part as well as the *CPHONE* entry field part to their corresponding static text parts, you can use the spacing tool again:

1. Draw a selection rectangle around the entry field parts *CUSTNA*, *CONTAC*, *CPHONE*, and *CADDR* parts.

2. Call the pop-up menu of any of the selected parts, choose *Space*, and select the fourth choice offered (Figure 189). The spaces between the entry field parts are evened out.

Now align all entry field parts except *CCITY* to one vertical axis:

1. Draw a selection rectangle around all of the entry field parts with the exception of the *CCITY* part.

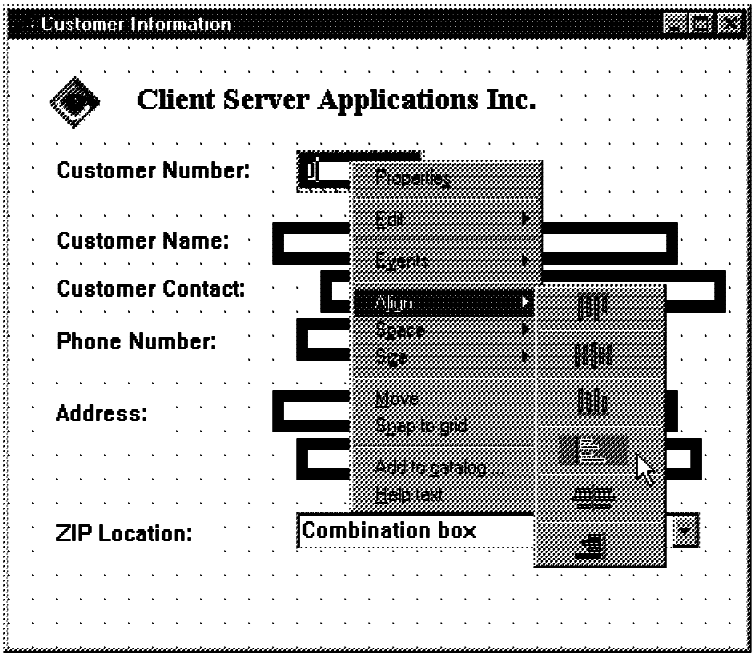


Figure 190. Left Align Entry Fields

2. Choose the fourth choice out of the *Align* menu item from the pop-up menu of the *CUSTNO* entry field part (Figure 190).
3. If the *CCITY* entry field part became overlapped by the *CZIP* entry field part, move the *CCITY* entry field part to the right of the *CZIP* entry field part.

Change the horizontal spaces between the entry field part *CZIP* and the combination box part *CZIPLO* on the one hand and the entry field part *CADDR* on the other hand to be equal to the space between the entry field parts *CPHONE* and *CADDR*. To do so,

1. Draw a selection rectangle around the entry field parts *CPHONE*, *CADDR*, *CZIP*, and *CZIPLO*.

2. Bring up the pop-up memu of any of the selected parts and choose *Space* and the fifth choice offered. This uses the space between the entry field parts *CPHONE* and *CADDR* and applies it to the other selected parts so they are evenly spaced.

Finally, the entry field part *CCITY* and the static text part *ZIP Location* need to be horizontally aligned to the entry field parts *CZIP* and *CZIPLO*.

1. Draw a selection rectangle around the entry field parts *CZIP* and *CCITY*.
2. Invoke the pop-up menu of the *CZIP* entry field part.
3. Choose *Align* and the second alignment choice offered.
4. Repeat these steps to align the *ZIP Location* static text part to the *CZIPLO* entry field part.

The aligned window is shown in Figure 191.

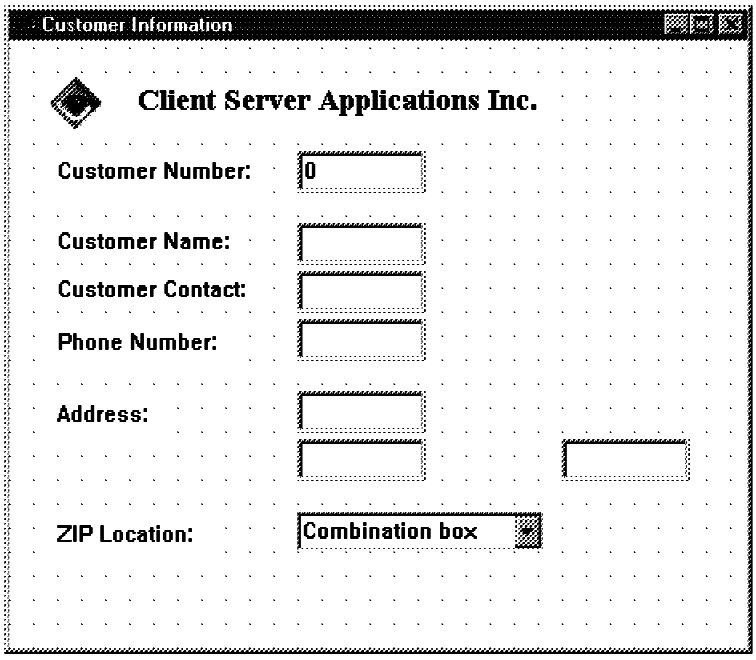


Figure 191. Customer Information Window After Alignment

Changing the Size of Some Fields

You may also want to change the size of some entry field parts that are supposed to display strings but are too long to be displayed using the

default size of the entry field part on the window. You can do this now for the entry field parts *CUSTNA*, *CONTAC*, and *CADDR*.

Even though you use the Define Field Reference feature that sizes the fields according to their definition, you may want to go through this exercise to become familiar with the sizing tools:

1. Click on the *CUSTNA* entry field part to make it the active part.
2. Drag the middle of the right-side sizing handles until the field size is enlarged to the right border of the *CCITY* entry field part (Figure 192).

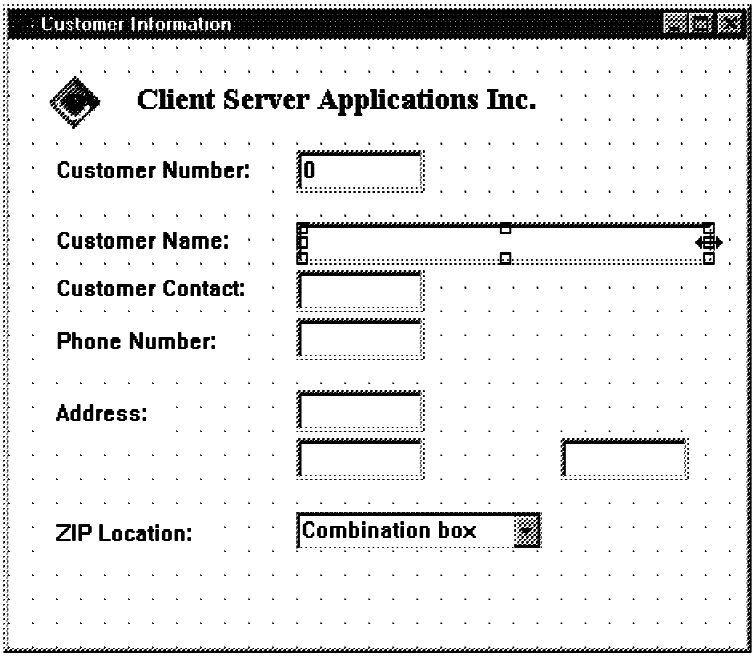


Figure 192. Sizing the CUSTNA Entry Field

3. Use the arrow keys to get to the *CONTAC* entry field part.
4. Press and hold the CTRL key and press the space bar to select the part.
5. Use the arrow keys to get to the *CADDR* entry field part.
6. Select this part also by using the CTRL key and the space bar.
7. Invoke the pop-up menu for the *CUSTNA* entry field part.
8. Choose *Size* and the first choice offered. The size of the entry field parts *CONTAC* and *CADDR* are enlarged to the size of the *CUSTNA* entry field part.

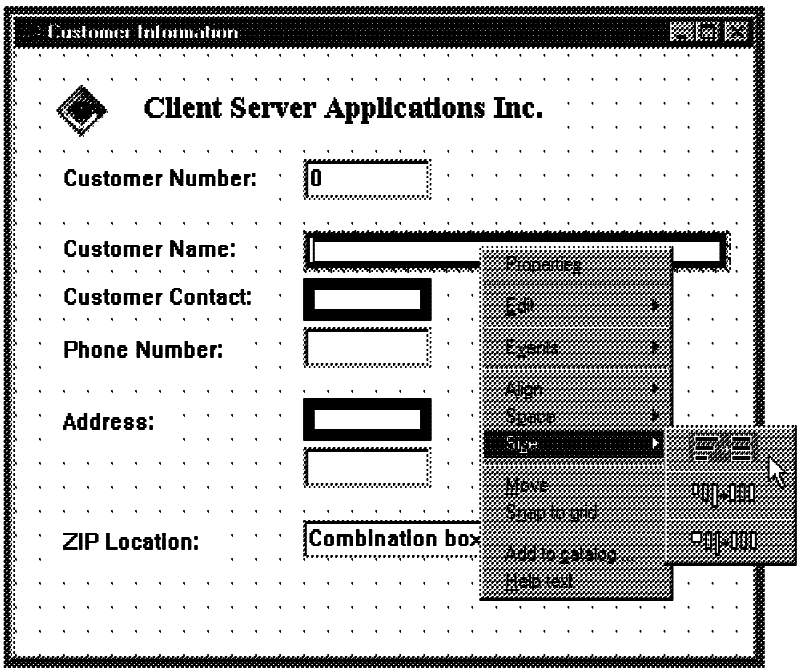


Figure 193. Resize Entry Fields CONTAC and CADDR

9. Deselect the *Grid* menu item of the *Customer Information* window's pop-up menu.

Now you add a push button part to the *Customer Information* window that allows the user to close this window and return to the *Customer Inquiry* window:

1. Move the mouse pointer to the bottom border of the *Customer Information* window so that it becomes a double arrow.
2. Press and hold the right-hand mouse button and drag the window border so that it can hold the additional push button part (Figure 194).

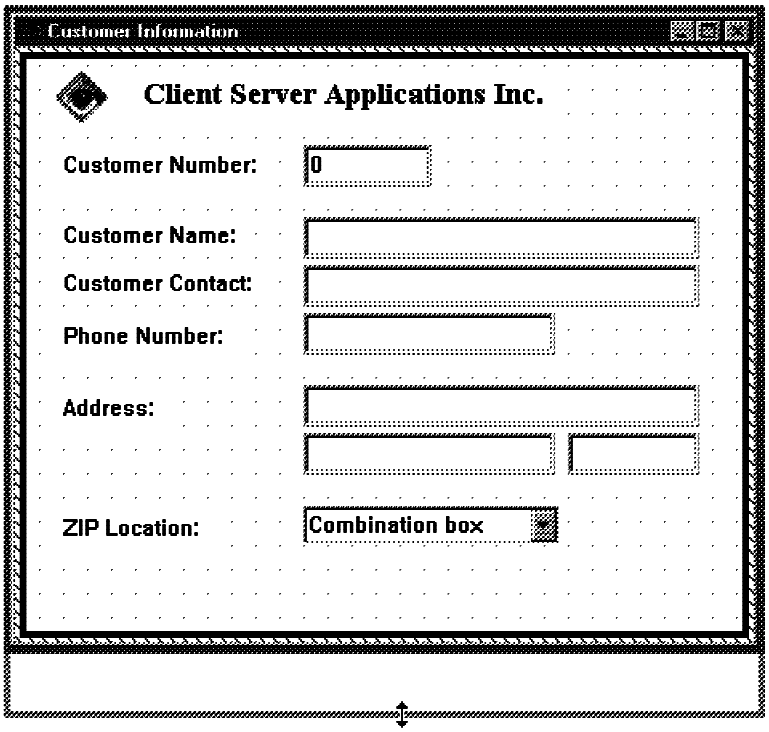


Figure 194. Resize Customer Information Window

All parts contained in the *Customer Information* window have also been moved to the bottom of the window. This happens because the focal point for all parts in a window is the lower left corner of the window.

3. Draw a selection rectangle around all of the parts on the *Customer Information* window.
4. Move the mouse pointer over the static text part that holds the company name Client Server Applications Inc.
5. Press and hold the right-hand mouse button and drag the company name static text part to just below the title bar of the *Customer Information* window.
6. Release the mouse button. All of the selected parts have been moved.
7. Find the push button part on the Parts Palette.
8. Press and hold the right-hand mouse button and drag the push button part below the *ZIP Location* static text part, then release the mouse button.

9. To align the push button with the static text part above, select the *ZIP Location* static text part and the push button.
10. Invoke the pop-up menu of the static text part.
11. Choose *Align* and the fourth alignment choice offered. The push button part is vertically aligned to the static text parts of the window.
12. Double-click on the *push button* part to get the properties notebook for this part.
13. Change the part name to INFOOK and the text to OK.
14. Go to the *Style* page of the properties notebook.
15. Select the *Default* check box. This causes the push button's Press event action subroutine to be invoked when the Enter key is pressed.
16. Double-click on the *System* icon of the properties notebook to close it.

The window should resemble Figure 195.

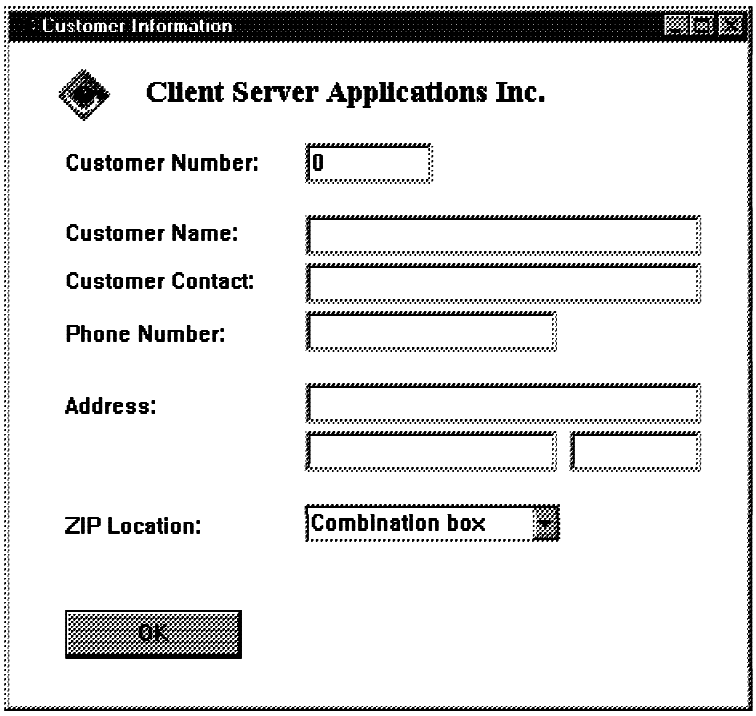


Figure 195. Final Window Design

To save the changes to your project and exit the GUI Designer, click on the *Save Project* icon on the tool bar (the third icon if you have not customized it) and your project is saved.

Adding More Logic to Your Project

The objective of this part of the exercise is to add the necessary logic to use the *Customer Information* window you have just designed with customer data.

There are several steps to follow:

1. Add a file definition to your program.
2. Change the code for the *OK* push button's Press event on the *Customer Inquiry* window to display the *Customer Information* window.
3. Create the code for the *OK* push button's Press event to close the *Customer Information* window.
4. Build the application.
5. Run the application.

The first step is to define the DB2/400 file in your program:

1. Choose the *Project* option from the GUI Designer menu bar and choose *Edit source code*.
2. Scroll to the *H* specification.
3. Position the mouse cursor on the *H* specification.
4. Choose *Edit* from the LPEX Editor menu bar and choose *Insert Prompt*.
5. Press the *Select* push button and select the *F* specification radio button. Press the *OK* push button and you see the prompt window for an *F* specification.
6. Fill in the following values:
   - File name: `CUSTOML3`
   - File type: `I`
   - File designation: `F`
   - File Format: `E`
   - Record address type: `K`
   - Device: `DISK`
   - Keywords: `REMOTE`

Note

With the exception of the REMOTE keyword, all entries are the same as RPGIV on the AS/400 system.

7. To save this input, choose *File* from the LPEX Editor menu bar and choose *Save*.
8. Close the LPEX Editor by double-clicking with the left-hand mouse button on its system menu.

Changing the OK Button Action Subroutine

You now add the logic for the Press event of the *OK* push button on the *Customer Inquiry* design window.

Note

Before you go ahead and code the real logic for this action subroutine, you may want to remove the three lines of code from Exercise 1. That code was created only to demonstrate events and action subroutines. If you prefer the push button to change color, leave the code in.

The RPG logic you are coding now should perform the following functions:

1. Get the customer number from the entry field on the *Customer Inquiry* window.
2. Retrieve customer detail from the AS/400 database file.
3. Display the second window (*Customer Information*).
4. Put data from the database file onto the second window.

Let's get started:

1. Using the pop-up menu from the *OK* push button, bring up the action subroutine for the Press event. If you did not remove it, you should see the code you created in Exercise 1 that changed the color of the push button.
2. The first step is to get the customer number entered in the *CUSTNO* entry field.
3. There are two methods for getting this value, but code only one in your action subroutine:
   a. Using the READ operation code:

```
RPG
*...1....+....2....+....3....+....4....+....5....+....6....+...
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++
C                   READ      'CUSTINQ'
```

b. Using the GETATR operation code:

```
RPG
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++
C     'CUSTNO'      GETATR    'TEXT'        CUSTNO
```

4. With the value of the field *CUSTNO*, you now have to chain to the database file CUSTOML3.

   Add the CHAIN statement with an indicator in the HI value column ("record not found").

   You may want to use F4 for prompting to get the right column for this indicator, or try out the syntax checker in the LPEX Editor.

   Your CHAIN statement should look like this:

```
RPG
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+Hi
C     CUSTNO        CHAIN     CUSTOML3                          50
```

5. Add the following statement, which is true if the CHAIN was successful:

```
RPG
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+Hi
C                   If        *IN50 = *OFF
```

6. Now you have to code a SHOWWIN operation code to get your second window displayed:

```
RPG
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+Hi
C                   SHOWWIN   'CUSTINFO'
```

7. You also have to get the data from the RPG program storage to this *Customer Information* window, which is similar to a WRITE to a record format in an AS/400 display file. So code this:

```
RPG
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+Hi
C                   WRITE     'CUSTINFO'
```

8. Now code the following to complete the IF condition:

```
RPG
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+Hi
C                   ENDIF
```

Your completed code should look like the following:

```
RPG
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq
C     PSBOK         BEGACT    PRESS         CUSTINQ
C                   Read      'CUSTINQ'
C     CustNo        CHAIN     CUSTOML3                          50
C                   If        *IN50 = *OFF
C                   ShowWin   'CustInfo'
C                   Write     'CustInfo'
C                   EndIf
C                   ENDACT
```

This completes this action subroutine for now. You can add some more function after we look at the runtime behavior. To save your changes, choose *File* from the LPEX Editor menu bar and choose *Save*. Close the LPEX Editor by double-clicking with left-hand mouse button on its system menu.

Adding an Action Subroutine to the Customer Information Window

In this part of the exercise, you add some code to make the *Customer Information* window invisible because the window should disappear when the *OK* button is pressed.

The instructions in the following tasks are not as detailed because you are now familiar with the VisualAge for RPG GUI Designer development environment.

Perform the following steps:

1. Create an action subroutine for the Press event of the OK push button on the *Customer Information* window.
2. Add a statement to set the Visible attribute for the window to *OFF*.

3. Save this action subroutine.

You can close the design windows at this point. To close a design window, position the mouse pointer on the window's title bar and click the right-hand mouse button. From the pop-up menu, choose *Close*.

Building the Application

Before you build the application, you have to specify where the compiler can find the AS/400 information it needs at compile time. In this case, you have to define on which server (AS/400 system) the file your program is using is located and which file you are using.

To enter this information, choose *Server* and *Define AS/400 information...* from the GUI Designer menu bar. The *Define AS/400 Information* notebook is displayed. This notebook has several pages on which you define information about your server and files (Figure 196).

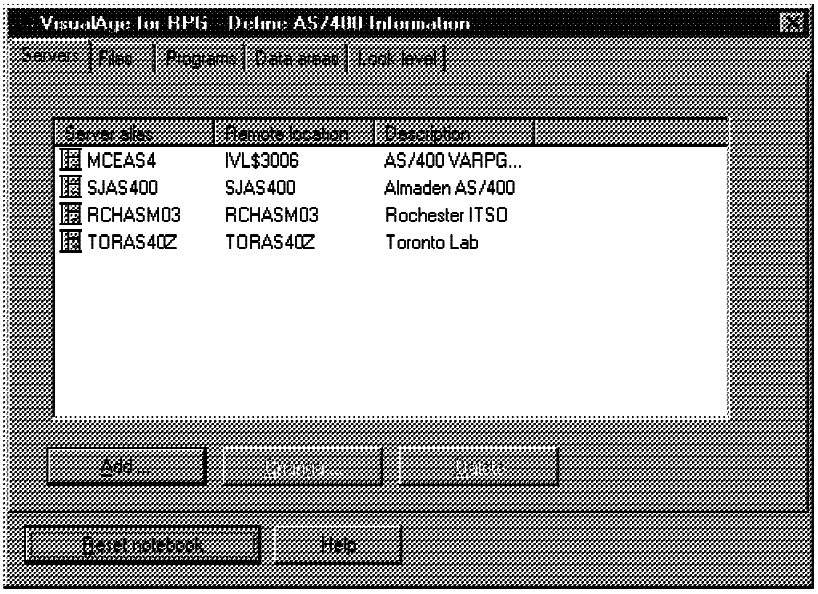


Figure 196. Define AS/400 Information Window

For this task, you need to define a server and a file:

1. On the *Servers* page, press the *Add...* push button. The Add Server Alias Name dialog is shown. Type an alias name in the *Server alias* entry field. From the drop-down combination box, select the same server you used when using the Define Reference Fields function.

2. Press the *OK* push button to add this server alias to the list on the *Servers* page.
3. Go to the *Files* page.
4. Press the *Add...* push button to add a new file definition. The *Add File Alias* Name dialog is shown.
5. In the *File Alias* entry field, type the name of the file you specified on the File specification in your program (CUSTOML3).
6. In the *Remote File* entry field, type the actual file name (GUIDES2/CUSTOML3).
7. From the *Server Alias* drop-down combination box, select the server alias name you defined in a previous step.
8. Press the *OK* push button to add this file definition.
9. Close the *Define AS/400 Information* dialog by double-clicking on its system menu.

With this information set, you can now build the project.

1. To build (compile) the project, choose *Project* from the GUI Designer menu bar and choose *Build*.
2. If you made changes to any of the design windows, a message window is displayed prompting you to decide if the project should be saved first. Answer "Yes" to this message.
3. A status window is displayed indicating that the build is in progress.
4. Once the build is complete, a completion window is displayed, indicating whether the build was successful or not.

Viewing the Compile Listing

Recall that the VisualAge for RPG compiler is resident on the workstation. Therefore, the compile listing is also resident on the workstation. At any time, it is possible to view the files that make up your project, including the compile listing.

To display the compile listing for this latest build:

1. Choose *Windows* on the GUI Designer menu bar.
2. From the pull-down, choose *Project Organizer* (Figure 197).

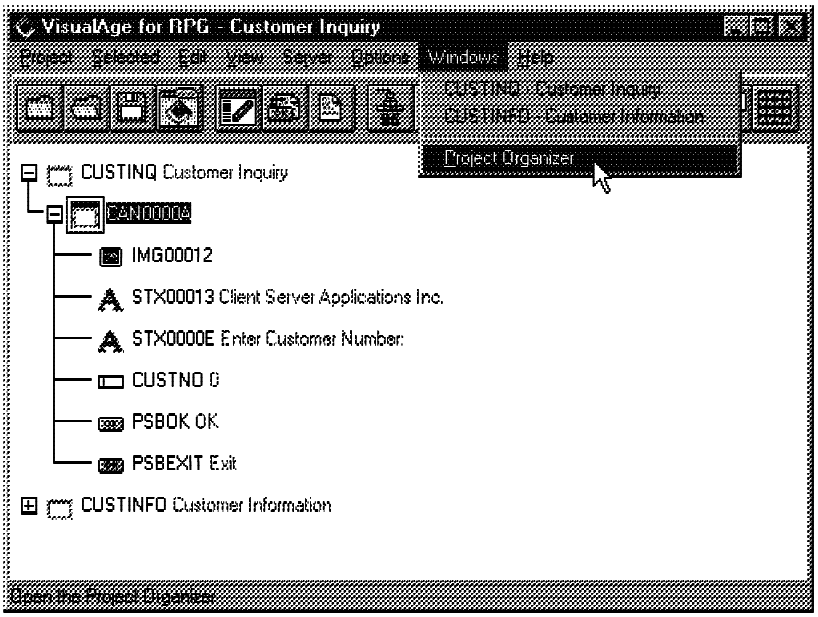


Figure 197. Displaying the Project Files

3. Figure 197 shows an icon view of the files that make up a project. Find the one that has a .LST file extension and select it with the left-hand mouse button. Click with the right-hand mouse button to get a pop-up menu for the actions allowed for this file. Choose *Browse*.

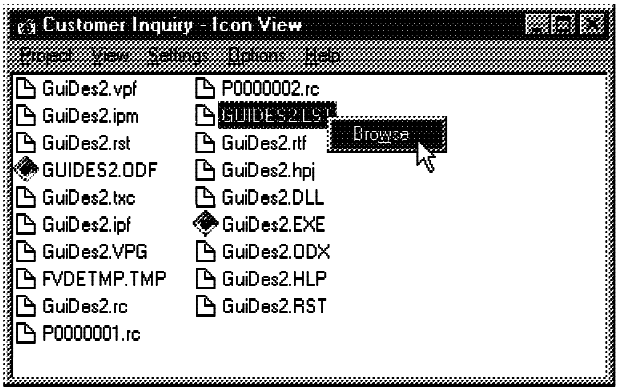


Figure 198. Icon View of the Project

4. The compile listing is now displayed in Figure 198. Note the similarities between the VisualAge for RPG compile listing and the compiler output of the AS/400 RPG compilers.

Testing Your Application

1. If the compile is successful, you can run your program. There are two ways to do this:
   a. Choose *Project* from the GUI Designer menu bar and choose *Run*.
   b. Click on the *Run* icon on the tool bar (the running man).
2. The first window should be displayed. Type `0010100` as the customer number. Notice that you have to move your cursor to the field before typing in the number. In the next exercise, you will improve this.
3. Press the *OK* push button. This brings up the second window with the customer information filled in.
4. If you type in a wrong customer number, nothing happens since you did not code any error handling.
5. Go to the second window again and use the *OK* button; the window is invisible.
6. Now try to get to the second window again with a valid customer number.
7. You receive a runtime error because the SHOWWIN operation code is trying to create a window that has already been created.
8. The next exercise shows why this is happening and how to handle it.
9. Now press the *Exit* button to terminate the application.

Exercise 3. Error Handling, Action Links, and Messages

In this exercise, you learn to handle error conditions and deal with the runtime behavior of the VisualAge for RPG environment. You also learn about action links and VisualAge for RPG messages. You do this by:

1. Removing the runtime error when using SHOWWIN twice
2. Positioning the cursor on an entry field
3. Linking two events to one action subroutine
4. Creating messages and displaying message boxes
5. Debugging your application
6. Building the application
7. Running the application

Objective

As a result of this exercise, you can:

- Handle runtime errors in your application.
- Understand how to position the cursor.
- Link multiple events to one action subroutine.
- Use messages and message boxes.
- Use the VisualAge for RPG debugger.

Enhancing Runtime Behavior

In this exercise, you are enhancing the application that you built during the previous exercises. The runtime error and cursor positioning problems are fixed.

Handling a Runtime Error

When you ran your application, a runtime error was displayed when you clicked on the *OK* button in the *Customer Inquiry* window the second time. This is because the SHOWWIN operation code is executed again that attempts to create the *Customer Information* window again. Since it has already been created, the runtime error occurs. The runtime error is displayed because you have not coded any error handling in the application.

As is typical in RPG, you use an error indicator on the operation code to deal with the error, and have a conditional statement to handle the error

depending on the state of the error indicator. The logic inside the IF condition uses the SETATR operation code to make the *Customer Information* window visible if the SHOWWIN operation fails. Remember that clicking on the *OK* push button on the *Customer Information* window made this window invisible.

Let's update the logic to handle the runtime error:

1. Edit the Press action subroutine for the *OK* button in the *Customer Inquiry* window.
2. Add an error indicator to the SHOWWIN operation code statement. This traps the runtime error. You now have to make the window visible without having to create it again.
3. Add a conditional statement for checking the error indicator. If the indicator is on, you want to make the window visible and bring it to the front by giving it the focus.
4. Add code to set the Visible attribute for the *Customer Information* window to 1.
5. Now you need to add a line of code to give the window focus (that is, make it the top window), by setting its Focus attribute to 1, as shown in Figure 199.

RPG

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+.
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq
 *
C     PSBOK         BEGACT    PRESS         CUSTINQ
C                   Read      'CustInq'
C     CustNo        Chain     CustomLF                          50
C                   If        *IN50 = *OFF
C                   ShowWin   'CustInfo'                          51
C                   If        *IN51 = *ON
C                   Eval      %Setatr('CustInfo':'CustInfo':'Visible')=1
C                   Eval      %Setatr('CustInfo':'CustInfo':'Focus')=1
C                   EndIf
C                   Write     'CustInfo'
C                   EndIf
C                   ENDACT
 *
```

Figure 199. Example of Error Handling Code

6. Do not forget to add the ENDIF operation code to complete the IF condition.

7. Save your changes.

Now you add code to position the cursor in the CUSTNO entry field (Figure 200). For this task, you use the Activate event to move the cursor to the entry field. This event occurs when the user clicks on a window with the mouse. The steps are these:

1. Create an action subroutine for the *Customer Inquiry* window Activate event.
2. Add a statement that uses the SETATR operation code to set the Focus attribute of the CUSTNO entry field to 1.

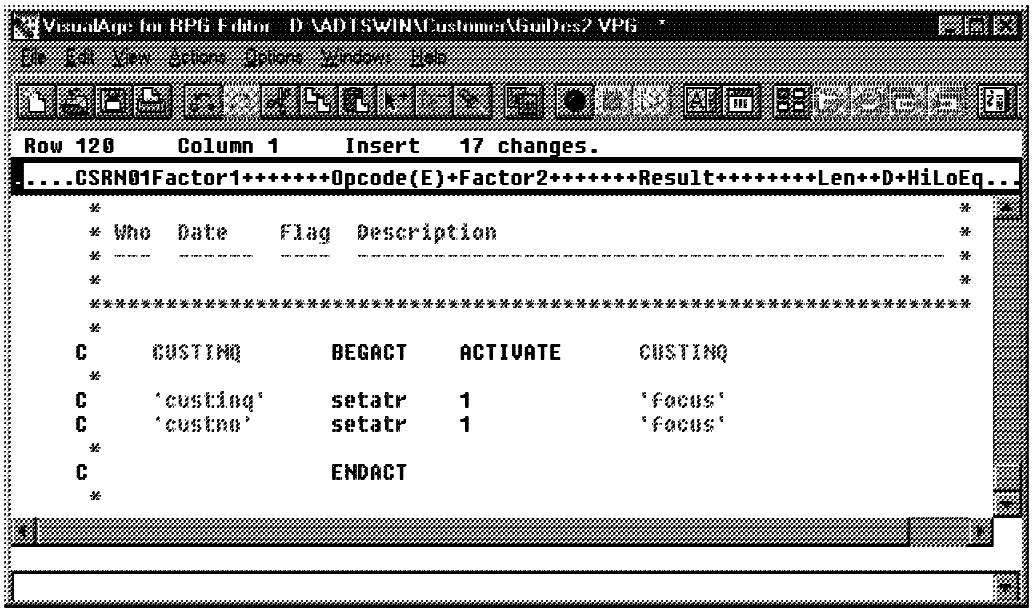


Figure 200. Code to Position Cursor

3. Save your changes and build the project.

Running the Improved Application

To see the improvement, do the following:

1. After successfully building your application, run it again.
2. Enter the customer number 0010200 and click on *OK* to display the second window with the customer information.
3. Now click on the *OK* button on the *Customer Information* window; this brings you back to *Customer Inquiry* but the cursor is not positioned on the entry field.

4. Click on the *OK* button on the *Customer Inquiry* window again. The runtime error message that you got before does not appear, so your error-handling code is working.

Working with Action Links

We still have the problem of the cursor not going to the customer number entry field when the first *Customer Inquiry* window is first displayed. The Activate event for the window already has the logic to move the cursor. Rather than rewriting this logic, use the Action Link feature of VisualAge for RPG. This feature allows more than one event to invoke the same action subroutine.

You use this feature to have the *Customer Inquiry* windows Create event call the action subroutine already defined for the Activate event.

First, let's look at the Action Link window. If an LPEX Editor window is not already opened, open one by choosing *Project* and *Edit source code* from the GUI Designer menu bar.

From the LPEX Editor menu bar, choose *Edit* and *Action subroutines...*. The VisualAge for RPG - *Action subroutines : Events View* window is displayed. Enlarge this window so you can see all of the information. Then,

1. From the *Windows* list box, select *CUSTINQ* by clicking on it with the left-hand mouse button.
2. This fills the *Parts* list box with all of the parts on that window.

   Since you are interested in the events for the *Customer Inquiry* window, select CUSTINQ from the *Parts* list box.
3. You now see the *Events* list box filled with all valid events for a window.

   Choose the *Create* event in the *Events* list box and select *CUSTINQ+ACTIVATE+CUSTINQ* in the *Action subroutines* list box (Figure 201).

   The *Create link* push button is now enabled.

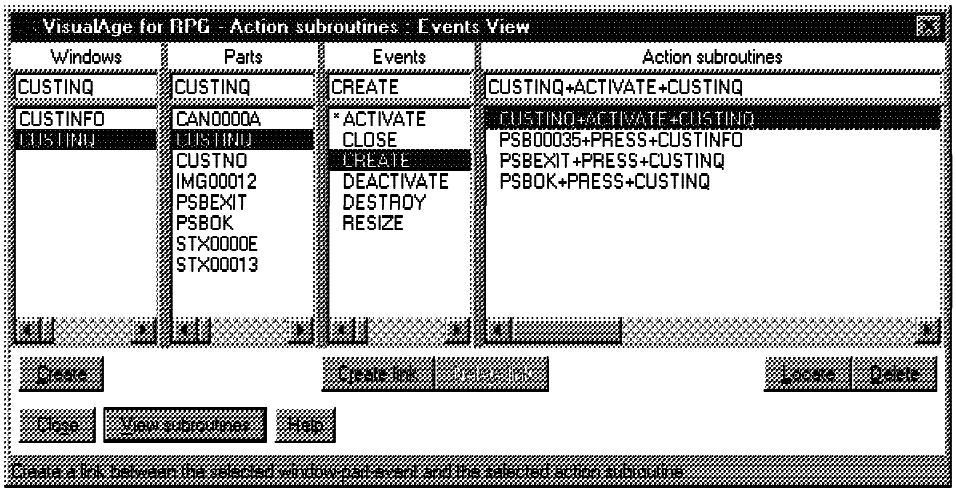


Figure 201. Creating an Action Link

4. Click on the *Create link* push button. This links the Create event to the Activate event action subroutine.
5. Close this window by clicking on the *Close* push button.
6. Build your application and test it by noting that the cursor is positioned in the CUSTNO field when the window is first displayed.

Using Message Boxes

As the application is now, if the user enters a customer number that is not in the database, no action is performed and no message is displayed. In the next exercise task, you add logic that causes a message box to be displayed to inform the user that the customer number is invalid.

There are two methods of defining the message text for messages in VisualAge for RPG:

1. Define the message text and message box style with D specifications in your VisualAge for RPG source.
2. Define a message in the GUI Designer.

In either case, the DSPLY operation code is used to display the message in a message box.

In this exercise task, you define the message in the GUI Designer and add a DSPLY opcode to your VisualAge for RPG logic to display a message if the customer number is not found in the database.

Let's start by defining the message:

1. Choose *Project* from the GUI Designer menu bar and choose *Define messages...*.
2. In the *Define Messages* window, click on the *Create...* push button. The window in Figure 202 is displayed, which allows you to define a new message.

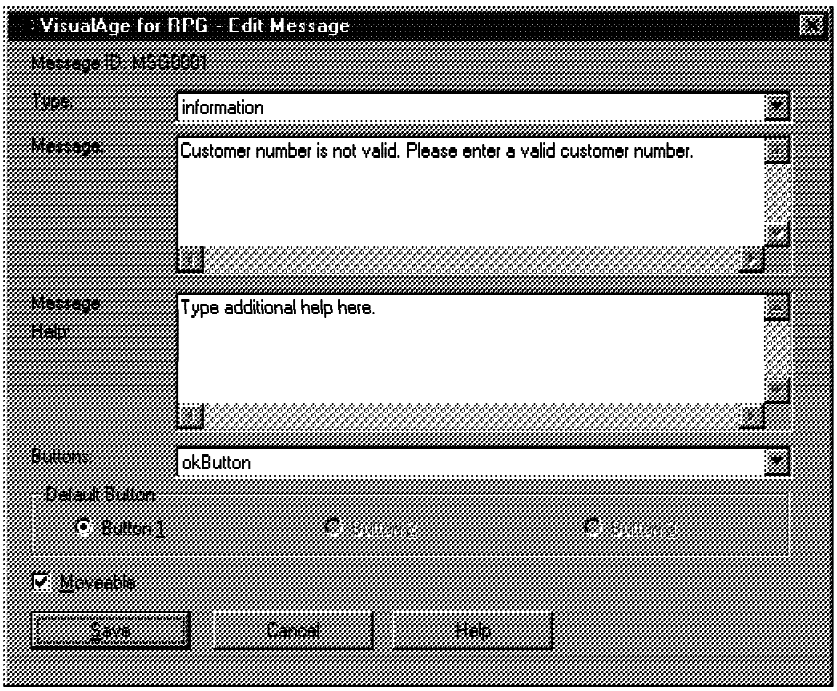


Figure 202. Creating a Message

3. Scroll through the different types of messages available and select any one. The message type determines which icon is displayed on the message box.
4. Type some message text in the *Message* entry box. This is the text that is shown on the message box.
5. You may also type some additional message text in the *Message Help* entry box. This text is shown when the Help push button on the message box is clicked.
6. Select the *Movable* check box. If this is not checked, the user cannot move the message box.

7. Select the push buttons to appear on the message box from the *Buttons* drop-down combination box. For this exercise, choose just the *OK* button.
8. Click on *Save* to save your message definition.
9. MSG0001 now is shown in the *Define Messages* window.

Note

VisualAge for RPG assigns message numbers. Message numbers cannot be changed by the user.

10. Click on the *OK* push button on the *Define Messages* window.

Now, you need to add the code to your action subroutine to display the message you have just created.

1. Open the *Customer Inquiry* window if it is not already opened and choose the *PRESS* event for the *OK* push button to edit its action subroutine.
2. Add the following statements after the CHAIN statement to display the message if the record is not found:

RPG

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7..
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+Hi
C                   ELSE
C     *MSG0001      DSPLY                   REPLY             9 0
```

3. Now add a statement to change the background color of the entry field to red using the SETATR operation code.

Tip

The part name goes in Factor 1, the attribute value in Factor 2, and the attribute name in the Result field. Do not forget the quotes.

All of the online manuals including the Parts Reference are also available from the LPEX Editor Help menu.

To see a list of attributes that are allowed for a specific part type, select the part on the Parts Palette by selecting it with the left-hand mouse button and press F1. Help is displayed for the part along with hyperlinks to that part's attributes.

4. Build and run the application. Enter an invalid customer number, such as 9999999. A message box should be displayed with the message text you defined earlier.

   Notice that a *Help* push button has been added to the message box. This is because you specified second-level message text. Click on the *Help* push button to see the second-level help you typed when you created the message.

This is an example of how messages are created in VisualAge for RPG and how they can be displayed with the DSPLY operation code. Messages are used elsewhere in VisualAge for RPG, including the Message Subfile part and label substitution. Those areas are not covered in this exercise.

Optional Exercise: More Message Box Handling

The first part illustrated how to display a message using the DSPLY operation code and specifying the message number in Factor 1. There are other ways of displaying a message or a text with the DSPLY operation code.

The next method is similar to the one in the previous exercise task. However, instead of specifying a message number in Factor 1, you define the message on a Definition specification. The following steps demonstrate how to do this:

Tip

D specifications must follow any File (F) specifications and come before any Calculation (C) specifications.

- Enter the following specifications in your program:

  (Notice that the Definition type is M. This is unique for VisualAge for RPG, and indicates this definition specification is defining a message.)

  RPG

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++
 *
DInfoBox          M                   STYLE(*INFO)
D                                     BUTTON(*RETRY: *ABORT: *ENTER)
 *
DMSG1             M                   MSGNBR(*MSG0001)
```

- Replace your current DSPLY operation code with the following code:

```
RPG
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+Hi
C     MSG1          DSPLY     InfoBox       Reply             9 0
```

- Build your application and run it. When the message box is displayed, notice that the push buttons and the icon are different from the ones you specified when you created the message. By using this method of displaying a message, you can override the attributes defined when the message was created.

Using Message Substitution

VisualAge for RPG messages also support message substitution. Using message substitution, the value of a program variable can be displayed in the message text.

For this exercise task, you create a new message that shows the invalid customer number in the message box.

- Create a new message as you did for the first message. This time, for the message text, use message substitution. Type the following information as the message text:

  ```
  Customer number %1 is not valid.  Please enter a valid
  customer number.
  ```

  Note the difference. The message substitution variable %1 has been added. This variable is replaced at runtime with the value of a variable in your program.

- Add the following D specifications to your program. The keyword MSGDATA is used to define which program variable should be used as the message substitution text. In this case, you are using the customer number entered by the user. Enter this:

```
RPG
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++
 *
DMSG2             M                   MSGNBR(*MSG0002)
D                                     MSGDATA(CUSTNO)
 *
```

- Replace the current DSPLY operation code with the following code:

```
RPG
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+Hi
C     MSG2          DSPLY                   Reply             9 0
```

Build your application and run it. Now, when the message box is displayed, the customer number should be displayed in the message box when you enter an invalid number.

Defining Message Text in the Program

An additional variation for displaying a message is to define the text of the message in your VisualAge for RPG program. This is done as follows:

- Add the following D specifications to your program. The TEXT field defines the message text to be displayed. Enter this:

```
RPG
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++
 *
DWarnBox          M                   STYLE(*WARN)
D                                     BUTTON(*YESBUTTON:*NOBUTTON)
 *
DTEXT             M                   MSGTEXT('Number NOT found')
 *
```

- Replace the current DSPLY operation code with the following code:

```
RPG
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+Hi
C     TEXT          DSPLY     WarnBox       Reply             9 0
```

- Build your application and run it. Observe the message text on the message box.

Using Action Links to Navigate Your Logic

The Action link window can also provide you with some help in navigating through your application logic. This window is accessed from the LPEX Editor:

1. If an *LPEX Editor* window is not currently open, choose *Project* and *Edit source code* from the GUI Designer menu bar. The *LPEX Editor* window is shown.
2. From the *LPEX Editor* menu bar, choose *Edit* and *Action subroutines...*. The *Action subroutines : Events View* window is shown.

3. Click on the *View subroutines* push button. This gives you a different view of all the action links in your program (Figure 203). Click on the *CUSTINQ+ACTIVATE+CUSTINQ* action subroutine and the second list box shows all of the action links for this action subroutine.

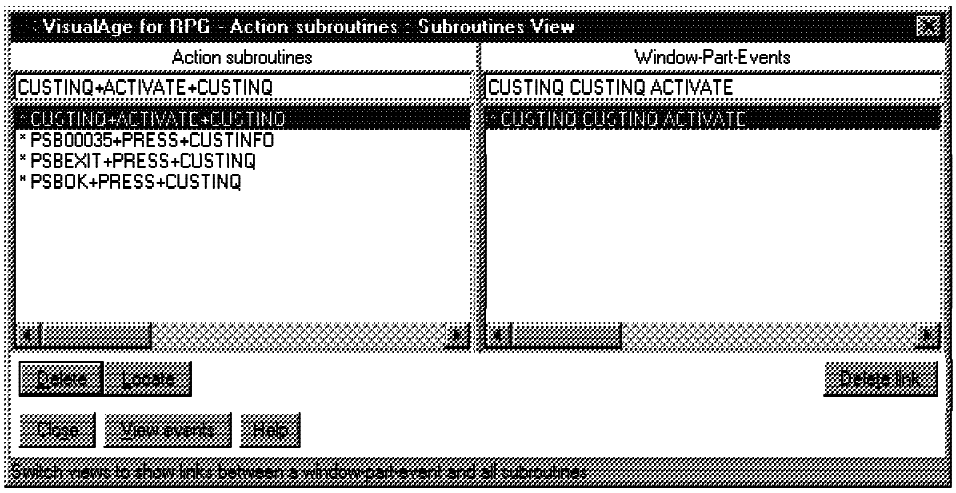


Figure 203. Navigate with Action Link Window

4. To view the logic for an action subroutine, double-click on an action subroutine name. The LPEX Editor positions the source to that action subroutine.

Debugging a VisualAge for RPG Application

VisualAge for RPG includes a powerful debugger to help in locating logic errors in your program. This short exercise task illustrates some of the basic debugger features.

The debugger can be started by clicking on the *debug tool bar* icon or from the GUI Designer menu bar by choosing *Project* and *Debug*.

1. The debugger *Debug Session Control* dialog is displayed, followed by another window with the source view of your program.
2. In the source view (Figure 204), scroll to the Activate action subroutine.

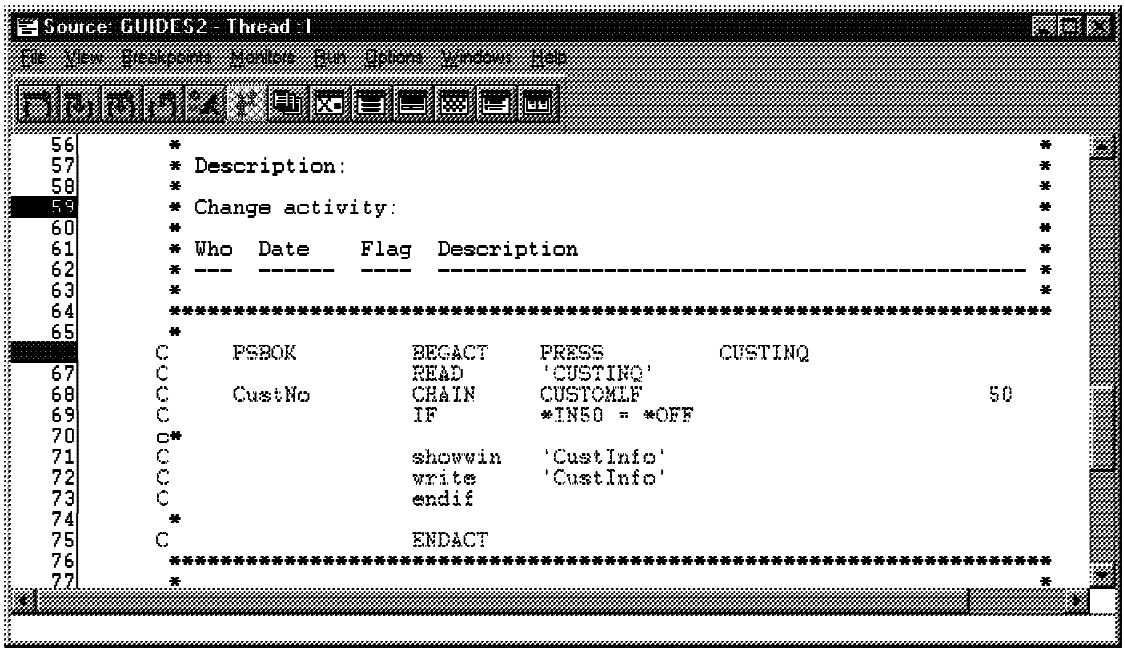


Figure 204. Debug Source View

3. Set a breakpoint on the statement that sets the focus attribute by double-clicking on the statement number.
4. Run your application by clicking on the *Run* icon on the source view tool bar. The *Run* icon is the one with the green circle and running person.

   Tip

   As you move the mouse pointer over the Debugger tool bar icons one by one, a short description of each icon is displayed.

5. When the break point is reached, the statement is highlighted.
6. To examine the contents of a program variable, scroll to any statement that contains a variable.

   Note

   This technique does not work on the EVAL operation code.

7. Double-click on the variable name. The *Program Monitor* window is shown with the variable contents.
8. Allow the program to run by clicking on the *Run* icon on the *Debugger* tool bar.

9. Click on the *Exit* button on the *Customer Inquiry* window to end the debug session and your program.

This concludes this exercise. In the next exercise, you make more enhancements to your application by adding a Subfile part and some Application Help.

End the GUI Designer by double-clicking on its system menu.

Exercise 4. Creating a Component with Subfile and Application Help

In this exercise, you enhance an existing application by creating a new component with a Subfile part. In addition, you add some help to your application. You are also making changes to your current application. Currently, the *Customer Inquiry* window has two push buttons:

- DETAIL
- EXIT

You'll by adding a LIST push button that starts the new subfile component that you are about to create in this exercise.

During this exercise, you:

1. Create a new component in a project.
2. Create a subfile and add fields.
3. Write the VisualAge for RPG logic to write records to the subfile part.
4. Write the VisualAge for RPG logic to get data from a subfile part.
5. Add application help to your application.
6. Build the application.
7. Run the application.

Objective

As a result of this exercise, you will know how to write applications containing multiple components, subfile parts, and application help by:

- Adding a new component to an existing project.
- Creating subfiles with multiple fields.
- Writing VisualAge for RPG logic to display data in a subfile part.
- Writing VisualAge for RPG logic to respond to subfile events.
- Writing application help using the IPF tag language.

Creating a Subfile

In this exercise, you are enhancing an application that was created earlier. As the application is now, the user must know ahead of time what a valid customer number is.

The enhancement you make is to add to the first window, *Customer Inquiry*, a push button that starts a new component to show another window. That window contains a subfile part that lists all of the customers in the database. When the user selects a customer from the list, the customer number is placed in the *CUSTNO* entry field on the *Customer Inquiry* window.

In this exercise, you create only the subfile; later you will have an opportunity to use a component reference part to fill the customer number in the *Customer Inquiry* window.

Note

The instructions in this exercise are not as detailed as in the earlier labs since you are expected to have had some experience using the VisualAge for RPG GUI Designer. Remember that online help is always available from the Help menu item on the GUI Designer.

First, you create a new component called COMPLIST. These steps guide you through the process of creating a new component in an existing application:

1. Open the GUI Designer, click on the *Project* menu bar choice and select *New* from the menu pull-down.

   You now have to tell it that you want to create a component and also identify the component name. Select the *Save Project* icon on the tool bar. You see a window that allows you to specify the component name COMPLIST.

2. Type the name COMPLIST.
3. Select the radio button *New Component*.
4. Select the *Customer Inquiry* project from the *Folders and Projects Combination* box.
5. Make sure the source file has COMPLIST as an entry.
6. Click on the *OK* push button.

You have now set up the component for the subfile window and can go ahead with creating a subfile.

You now create a subfile that is used to display records from a logical file on the AS/400 system. This file is named CUSTOML1 and is in library GUIDES2. The record format name is CUSTOM01. The subfile you create shows data from the following database fields:

- CUSTNO
- CUSTNA

Perform the following steps to create the new window with the subfile part:

1. Change the title of the design window to *Select Customer* (or something equivalent).
2. Name the design window CUSTLIST.
3. Locate the subfile part on the Parts Palette, drag it and drop it on the design window. Position and size the subfile as required.
4. Double-click on the subfile part to bring up its properties notebook.
5. On the *General* page, change the name of the subfile by changing the value in the *Part Name* entry field to SFL1.
6. Close the properties notebook.
7. Now you add fields to the subfile. There are two ways to add fields to a subfile part:
   a. By opening the Subfile parts properties notebook and going to the *Field list* page.
   b. By dragging and dropping fields from a database Reference File.

For this exercise, we use the database reference method.

Defining Reference Fields

Before you can use the Define Reference Fields feature, you have to define the AS/400 systems. This was already handled in an earlier exercise.

1. Use *Define reference fields* from the GUI Designer menu bar to display the field list for file CUSTOML1 in library GUIDES2.
2. Drag and drop the following fields on the subfile part:
   - CUSTNO
   - CUSTNA

Note

If you add the wrong fields, or want to change the order of the fields in the subfile, open the properties notebook for the subfile and go to the *Fields* page.

Once you have completed these two steps, you should have a design window that looks similar to Figure 205.

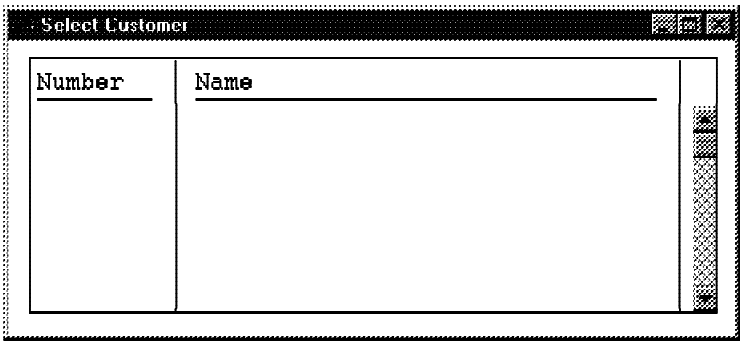


Figure 205. Completed Window with Subfile

Now you need to add code to fill the subfile. In this example, you are going to fill the subfile when the window is created. Therefore, you need to create an action subroutine for the window's Create event.

To create the action subroutine, use the right-hand mouse button to invoke the pop-up menu for the window. Choose *Events* and choose *CREATE*.

When the skeleton action subroutine is displayed, add the VisualAge for RPG statements shown in Figure 206 between the BEGACT and ENDACT statements to fill the subfile. Then, save your code.

```
RPG
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+.
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq
      *
C     FRA00014      BEGACT    CREATE        FRA00014
 *
 * Read a record from database
C                   read      customna                               99
 *
 * Do until end of file
C     *in99         doweq     *off
 *
 * Add a record to the subfile
C                   write     SFL1
 *
 * Read a record from database
C                   read      customna                               99
C                   enddo
 *
C                   ENDACT
 *
```

Figure 206. Code to Fill the Subfile from the Database

Now you must add a file specification to your program to fill the subfile.

Adding the File Specification

Take these steps:

1. If an LPEX Editor window is not already open, choose *Project* and *Edit source code* from the GUI Designer menu bar.
2. Enter the following file specification for the customer physical file:

```
RPG
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+.
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++++++
FCustomer  IF   E           K DISK    REMOTE
F                                     Rename(CUSTOM01:CUSTOMNA)
```

Tip

File specifications must follow any H specifications and come before any D specifications.

The following values were used to define this F specification:

- Customer - The file name used by the program.
- I - File is input only.
- F - File is being processed as full procedural.
- E - The file is externally described.
- K - The file is being processed by key.
- DISK - This is a disk file.
- REMOTE - The file resides on the AS/400 system.
- RENAME - Rename the record format since it is already used in file CUSTOMLF.

Tip

You may want to add the file keyword BLOCK(*YES) in applications that use SETTLL or CHAIN. This allows the RPG compiler to define blocking even if these operation codes are used to access the file.

Selecting a Record from the Subfile

Now you need to add the logic that reads the record selected by the user when the user double-clicks on a subfile record. When the user double-clicks on the subfile part, an Enter event is signalled. Therefore, you need to create an action subroutine for the subfile's Enter event. The steps are these:

1. From the *Subfiles* pop-up menu, create an action subroutine for the Enter event.
2. In this action subroutine, use the READS operation code to get the selected record from the subfile. READS moves the value for the customer number into the RPG variable CUSTNO. The READS operation code requires the subfile part name in Factor 2 (without the quotes).
3. You must move the customer number to the CUSTNO field on the *Customer Inquiry* window in application Customer Inquiry. Since this field resides in a different component, you must use a component reference part to exchange information. (You can skip this step now and add this function later.)
4. Finally, you want this subfile window to disappear when a selection has been made. Add the logic required to make this window not visible.

This is all of the logic needed for this action subroutine. Save your code changes by choosing *File* and *Save* from the *LPEX Editor* menu bar.

Defining AS/400 Information

Before you can build the application, you must indicate which file on the AS/400 system is being used, and on which AS/400 system or server the file resides. Do this by entering information in the Define AS/400 information notebook. The server is still the same as the one used before, so no changes are required here.

Defining the AS/400 File

Now you must define the file that is being used to fill the subfile. Display the Define AS/400 information window by choosing *Project* and *Define AS/400 information* from the GUI Designer menu bar. Go to the *File* page and click the *Add* push button to add the file CUSTOMER.

The override value is the name we called the file in our program (in this case, CUSTOMER).

The remote-name value is the qualified name of the file on the AS/400 system. For this example, the remote file name is GUIDES2/CUSTOML1.

The server alias name is the alias of the server that was defined on the *Server* page.

Building the Subfile Component

Now you can build the component. Since the build environment has been changed, you may want to have a look at the build options.

Under the project menu, select *Build options*; this brings up the new *Build options* dialog. Specify any changes you want to make in here; you are not prompted at build time for changing the options.

You should note that if you attempt to run this component, it will fail. This is because you have created a VisualAge for RPG component, not an application. VisualAge for RPG components can be invoked only by starting them from another VisualAge for RPG component or application. In the following section, you return to the Customer Inquiry application to add logic that starts this subfile component.

Adding Logic to the Customer Inquiry Component

Since you want to show the subfile for selecting a customer when the user does not know the customer number, you have to add some capability to the *Customer Inquiry* window in application Customer Inquiry to do this. Adding a *Find* push button and an action subroutine with some logic to start the component COMPLIST gets you there.

Select the *Open a Project* icon on the tool bar.

In the open dialog window, select *Customer Inquiry* and click on the *Open* push button.

You are now editing the *Customer Inquiry* application.

1. Open the *Customer Inquiry* design window and add a push button part.
2. Change the label of the push button to *Find....*
3. Change the name to *PSBFIND*.
4. Create an action subroutine for the Press event of this push button to START component *COMPLIST*:

```
RPG
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+.
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq
     *
C                   START     'COMPLIST'
```

Save your code changes by choosing *File* and *Save* from the LPEX Editor menu bar, and then build this project.

Running the Enhanced Application

Run the Customer Inquiry application. You now have an additional push button that lists customers from the database. Try it out. When the *Customer Inquiry* window is displayed, press the *Find...* push button to display the list of customers. In a later exercise, you see how to retrieve the selected customer number.

Creating Help for Your Application

In this exercise, you create online help for your application. Help in VisualAge for RPG is written for each part. In this example, you create the help that describes the function of the CUSTNO entry field on the *Customer Inquiry* window.

To begin, invoke the pop-up menu for the CUSTNO field by clicking on it with the right mouse button.

1. On the pop-up menu, choose *Help text*.
2. The LPEX Editor is shown with statements similar to the following:

```
:h1 res=171.EF00001A
:p.Help
```

The information up to the period on the first statement is used by VisualAge for RPG when invoking help, so this information should not be changed. The string following the period, EF00001A, is the heading that is shown in the *Help* window. Change it to something similar to the following:

```
:h1 res=191.Help for Customer Number Field
```

3. The text on the following line is the body of the help text, and begins with the paragraph tag, (:p.). Change this text to something similar to the following:

```
:p.Enter a customer number in this field and click on OK.
If the customer number is found, the
:hp9.Customer Info:ehp9.
window is displayed.
To find a valid customer number, press the
:hp9.Find...:ehp9. pushbutton.
```

   The :hp9. and :ehp9. tags are called *highlight phrase tags* and cause the text between them to be displayed in a different color.

   Tip

   A description of all IPF tags can be found in the IPF online reference manual in the VisualAge for RPG folder. You can view this manual by choosing *Help* from the GUI Designer menu bar.

4. Save your help and close the LPEX Editor window.
5. In the GUI Designer menu bar, choose *Project* and *Build* to build this project.

To test your help, invoke the program by selecting the *Run* icon on the tool bar. When the *Customer Inquiry* window (CUSTINQ) is shown, move the cursor to the customer number field by clicking on it with the mouse and press the F1 key. The help you created is shown in a separate window.

This short example illustrates how easy it is to add online help to your VisualAge for RPG applications. There are many other IPF tags you can use to enhance your help, including adding bitmap images.

Exercise Summary

This exercise guided you through:

1. Creating a component
2. Creating a subfile
3. Creating online help

Exercise 5. Using the Component Reference Part

Component reference parts are used to monitor for events in other components and to get and set attribute values from parts in other components.

In this exercise, you enhance your application using a component reference part to pass the value of the selected customer number from the subfile component COMPLIST back to the component Customer Inquiry.

You are making the following changes:

- For component COMPLIST:
  - Add an entry field part that contains the selected customer number. It is this part that the component reference part is referencing.
- For component Customer Inquiry:
  - Add a component reference part.
  - Change the settings for this part so it monitors for the Change event of the entry field added to component COMPLIST.
  - Change the settings for this part so it can retrieve the Text attribute for the entry field.
  - Add logic for the Notify event of the component reference part to get the changed data.

During this exercise, you:

1. Create a hidden field containing the selected customer number.
2. Create a component reference part.
3. Use the component reference part to react to an event and get an attribute value from another component.
4. Write the VisualAge for RPG logic to fill the hidden field with the subfile selection.
5. Write logic to get the customer number value from a different component.
6. Build the application.
7. Run and test the application.
8. Improve it.
9. Test it.

Objective

In this exercise, you learn about the behavior of components by writing applications containing multiple components that use component reference parts to communicate with each other. In doing this, you will be

- Creating and using component reference parts.
- Writing VisualAge for RPG logic to monitor for events across components.
- Writing VisualAge for RPG logic to get attribute values from other components.

Creating the Hidden Field

In this exercise, you add function to the subfile application that you started in Exercise 4.

The enhancement you make first is to add a hidden entry field to the CUSTLIST window of component COMPLIST. The Text attribute of this is set to the customer number value when the user double-clicks on a record in the subfile. The hidden field, in turn, triggers a Change event.

The following steps guide you through the process of creating the hidden field in the existing component COMPLIST:

1. Double-click on the *Customer Inquiry* project on your desktop.
2. You see all of the parts belonging to the Customer Inquiry in the project window, including the component COMPLIST.
3. Double-click on the *COMPLIST* icon, which brings up the project window for this component.
4. Locate the *EDIT* icon on the tool bar and select it. This brings up the GUI Designer.
5. You can now double-click on the *CUSTLIST* window icon to bring up this window.
6. Select an entry field from the Parts Palette and put it on the CUSTLIST window. Since it is hidden, its position is not important.
7. Change the field name to EF2.
8. Deselect the visible check box so it does not appear at runtime.
9. Invoke the pop-up for EF2 and choose *EVENTS*. From the Events list, choose *CHANGE*. When the editor window opens, you can just close it.

The component reference part requires that an action subroutine exist even though it contains no logic.

10. Invoke the pop up menu for the subfile part SFL1.
11. Select *Events* and *Enter*.
12. In the editor for the Enter action subroutine, set the Text attribute of the hidden field EF2 to CUSTNO. This is the customer number from the selected subfile record.

Save your code and build the component. After the component is built successfully, you can now open project Customer Inquiry and make changes to it.

Creating and Using the Component Reference Part

Click on the *OPEN* icon on the tool bar and select project *Customer Inquiry*.

Now add a component reference part to the *Customer Inquiry* window.

1. Open the *Customer Inquiry* window.
2. Drag a component reference part from the Parts Palette and drop it on this window.

Open the properties notebook for the component reference part and rename it to MONCHG. Now go to the *Style* page. The information you need to give a component reference part is twofold:

1. The upper section defines the part attribute being referenced.
2. The lower section defines the part event to monitor.

On the *Style* page, specify the component name that you want to reference (in this case, COMPLIST).

Since you are interested in getting the Value attribute and monitoring for the change event, you must fill out both sections:

1. Specify the reference window name CUSTLIST.
2. Specify the reference part name EF2.
3. Specify the reference attribute name TEST. This concludes the definition for the upper part of the *Style* page for this component reference part.
4. Now do the lower part to define the event you want to monitor. In this case, you want to monitor part EF2 for the Change event.
5. Use the same preceding information for window name and part name.
6. Then specify CHANGE for the event to be monitored.

The next step involves writing an action subroutine that is invoked when component reference part MONCHG gets notified by a Change event from entry field EF2. In the VisualAge for RPG component reference part, this is called a *Notify* event.

7. Write an action subroutine for the MONCHG Notify event that gets the attribute value (ATTRVALUE) from the MONCHG part and puts it into the variable CUSTNO:

```
RPG
*...1....+....2....+....3....+....4....+....5....+....6....+....7..
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+Hi
C     'MONCHG'      Getatr    'AttrValue'   CustNo
```

8. Copy this value to the CUSTNO field in the *Customer Inquiry* window with a SETATR statement:

```
RPG
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+Hi
C     'CustNo'      Setatr    CustNo        'Text'
```

9. Save everything you have done.
10. Build the application.
11. Run and test the application. When you double-click on a subfile record, the corresponding customer number should appear in the CUSTNO entry field.

Visibility and Focus

You notice that when the CUSTLIST window is displayed, it does not have focus—that is, does not appear in front of other windows.

To give this window focus, we use another component reference part in component Customer Inquiry to set the Visible and Focus attributes for window CUSTLIST.

Open the GUI Designer for project Customer Inquiry and open the *Customer Inquiry* window. Create another component reference part on this window and name it *CRP1*. Open the properties notebook, go to the *Style* page and set:

1. Component to COMPLIST.
2. Reference window name to CUSTLIST.
3. Reference part name to CUSTLIST.

This completes the definition of the component reference part. Now let's add the necessary logic.

First, we add an indicator that tells the application Customer Inquiry that the component COMPLIST is already started and the only thing left to do is to make visible the window CUSTLIST in component COMPLIST and give it focus.

You have to add some code to the action subroutine that is linked to the *Find* push button on the *Customer Inquiry* in component Customer Inquiry:

1. Use the LPEX Editor and locate the Press event action subroutine for the Find push button.
2. Add code to turn indicator 60 on when component COMPLIST is started to indicate that component COMPLIST is started.
3. Add code to skip starting component COMPLIST if *IN60 is on.
4. Add the following code to set window CUSTLIST in component COMPLIST visible:

```
RPG
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+Hi
C     'CRP1'        SETATR    'VISIBLE'     'REFATTR'
C     'CRP1'        SETATR    '1'           'ATTRVALUE'
```

5. To give the window focus, add the following two statements:

```
RPG
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+Hi
C    'CRP1'         SETATR    'FOCUS'       'REFATTR'
C    'CRP1'         SETATR    '1'           'ATTRVALUE'
```

6. Finally, build and test your application. When the *Find* push button is pressed, the *Subfile* window should be visible and have focus.

Exercise Summary

This exercise guided you through:

1. Creating and using a component reference part.
2. Using the two different functions of a component reference part.
3. Writing VisualAge for RPG logic for a component reference part.

Exercise 6. Pop-Up Menus and Notebooks

In this exercise, you enhance the subfile component with a pop-up menu. The pop-up menu is shown when the user clicks the right-hand mouse button on the subfile.

To show details of a selected customer, you also add a notebook part.

During this exercise, you:

1. Create a pop-up menu.
2. Create a notebook containing customer information.
3. Write the VisualAge for RPG logic to handle pop-up menus.
4. Write the VisualAge for RPG logic to fill a notebook page.
5. Run the application.

Objective

As a result of this exercise, you can write applications containing pop-up menus and notebooks by:

- Adding a pop-up menu to a VisualAge for RPG application.
- Creating a notebook in a noncanvas window.
- Writing VisualAge for RPG logic to react to the pop-up event.
- Writing VisualAge for RPG logic to respond to menu select events.
- Writing VisualAge for RPG logic for notebook parts.

Creating a Pop-up Menu

You are enhancing your subfile component with a pop-up menu and a notebook. The user can click with the right-hand mouse button on a selected subfile record and your VisualAge for RPG application shows a pop-up menu. The pop-up menu gives you the choice of viewing the details of the selected customer or deleting the selected customer record.

The additional enhancement you make pertains to the presentation of the user data. If the user chooses the *Detail* item in the pop-up menu, you show the customer data in a notebook.

Working with Pop-up Menus

These steps guide you through the process of creating the pop-up menu:

1. Double-click on the *COMPLIST* component in your GUIDES2 project window.
2. You see all of the parts belonging to the component COMPLIST in the project window.
3. Click on the *EDIT* button on the tool bar to bring up the GUI Designer for this project.
4. Locate the pop-up icon in the Parts Palette and drag it onto your design window. Where you drop it on the window does not matter. At runtime, the pop-up menu shows where the mouse cursor is located when the right mouse button was clicked.
5. In the GUI Designer, open the properties notebook for the pop-up menu and name the menu POPUP1.
6. Name the pop-up menu item POPDETAIL.
7. Now you have to add one more menu item to this pop-up part. Select a menu item from the parts palette and drop it onto the POPDETAIL menu item. Selecting this menu item shows a Notebook with customer information.
8. Name this menu item POPDELETE.

Adding Logic to Support Pop-up Menus

1. Select the *Popup* event on subfile SFL1.
2. Add the following code to the action subroutine for the Popup event:

```
RPG
*...1....+....2....+....3....+....4....+....5....+....6....+....7..
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+Hi
C     'POPUP1'      SETATR    1             'VISIBLE'
```

   This piece of code tells the runtime to make the pop-up part visible. Pop-up menus are always displayed at the mouse cursor position.

3. Close this window.

Creating the Notebook

You create a notebook with two pages to show the customer data of the selected customer record in the subfile. One page contains name and address information; the other page contains contact information.

To introduce the notion of a window without canvas, you are creating a notebook that fills the entire window:

1. Go to the Parts Catalog and locate the Window part. It is in the *Frames* page.
2. Drag it to the *Project View* of the GUI window.
3. Name this window WINNOTE.
4. Change the title to `Customer Detail Notebook`.
5. Drag a notebook icon onto the WINNOTE window and drop it.

   Using a window without canvas causes the notebook to fill the entire window frame.
6. Name this notebook NBKCUST and deselect *VISIBLE*.
7. Drag two *Notebook* pages with canvas parts onto the notebook and drop them.

   Tip

   You can reorder notebook pages by opening the properties notebook for the notebook part and going to the *Page List* page.
8. Change the tab text on each of the pages to:
   - Address
   - Contact
9. Now use the database reference function to create the following fields on the *Address* page:
   - Customer number
   - Address
   - City
   - Phone
10. Get the *CUSTNO* and *Contact* field, drag them to the *Contact* notebook page, and drop them.

    Note: The *CUSTNO* field on the second page gets a new name. Only one unique name per window per part is allowed, so VisualAge for RPG renamed this field when it was placed on the window.
11. Save your project.

Adding Logic to Update the Notebook Pages

It takes little effort to write the logic for this piece of the application:

1. Write an action subroutine that reacts to the MenuSelect event of the POPDetail menu part.
2. Get the selected record from the subfile (READS operation code).
3. Use the *CUSTNO* field to chain to the AS/400 file.
4. Write to window WINNOTE.
5. Make window WINNOTE visible and set focus.

   Tip

   Do not forget to set the text attribute for the customer number on the *Contact* page. Check the manuals to determine whether to use setatr or %setatr.

   If you have time, you may want to implement the delete function for a customer record by creating an action subroutine for the MenuSelect event of the POPDELETE menu item and deleting the selected subfile record.

   You may also want to add some code to make the WINNOTE window invisible if the Close event code is received.

   Tip

   You can specify on Factor 2 of the ENDACT operation code whether default processing should take place or not. By setting this value to *NODEFAULT, you can prevent the window from actually closing.

   This is useful if the end user double-clicks on the *system* icon of a window and you want to prevent the window from being closed.

6. Save and build your application.
7. Run and test your application.

Exercise Summary

This exercise guided you through:

1. Creating a pop-up menu.
2. Creating a notebook with notebook pages.
3. Writing logic to deal with these parts.

Exercise 7. Using the Container Part

In this exercise, you create an application that uses the container part. The application also reads records from a local file to add records to the container. During this exercise, you:

1. Create a container part and add columns.
2. Write the VisualAge for RPG logic to read records from a local file.
3. Write the VisualAge for RPG logic to add records to the container part.
4. Write the VisualAge for RPG logic to change the container view.
5. Build the application.
6. Run the application.

Objective

As a result of this exercise, you can write applications that use program described files to access local files and also use the container part by:

- Creating containers with multiple columns.
- Writing VisualAge for RPG logic to add records to a container.
- Writing VisualAge for RPG logic to change a container view.
- Writing VisualAge for RPG logic to read records from a local file.

Container Overview

The container part allows you to display records in three different views:

- Tree view
- Icon view
- Details view

In addition, records are added in a hierarchical manner (that is, a record can have one or more subrecords). A record that has subrecords is called the *Parent* record. Records added to this record are referred to as *Child* records.

Application Description

In this exercise, you use a container part that displays records stored in a database on the workstation. The database is a list of employee records and has the following record layout:

```
 1 -  3   Department number
 5 -  5   Employee type
 7 - 29   Name
30 - 42   Phone
```

If Column 1 contains an asterisk (*), it is considered a comment record and should be ignored.

If the field *Employee type* is an M, the record is considered a manager record and, therefore, has a ParentID value of 0. An *Employee type* of E indicates this employee works for a manager and should be added as a child record to the appropriate manager record using the department number as the ParentID.

Creating the Graphical User Interface

If the GUI Designer is currently active, choose *Project* and *New* to create a new application.

Perform the following steps to create the container part:

1. Use direct editing or the window properties notebook to change the title of the window to something appropriate, such as Departments.
2. Open the properties notebook for the window, go to the *Style* page, and select *Sizable* as the border style.
3. Close the windows properties notebook.
4. For this exercise, we want the container part to always fill the window part. Therefore, we need to delete the canvas part from the initial design window. Invoke the pop-up menu for the Canvas part and choose *Cut* to delete the Canvas part.
5. Locate the Container part, drag it onto the design window, and drop it. Notice that the container completely fills the window frame.
6. Open the properties notebook for the Container part by double-clicking on it.
7. Set the name to CT1.
8. Select the *Style* tab to go to the *Style* page. On the Style page, select *Details* as the view. Also, deselect the *Visible* check box for the title.
9. Now you define the columns for the container. Select the *Columns* tab to go to the *Columns* page and click on the *Add after...* push button.
10. On the *Add column* window, select *Object Icon Column* as the type.
11. Change the width value to 64 pixels and select *Center* for Horizontal alignment.

12. Click on the *OK* push button to add this column and return to the *Columns* page.
13. Click on the *Add after...* push button to add a second column to the container.
14. On the *Add column* window, select *Object Text Column* as the column type. Change the width value to 200 pixels.
15. In the Heading definition section, type `Name` as the Text value.
16. Click on the *OK* push button to add this column to the container and return to the *Columns* page.
17. Click on the *Add after...* push button to add a third column to the container. This column contains the phone number.
18. On the Add column window, select *Text column* as the column type and change the width value to 200 pixels.
19. Click on the *OK* push button to add this column to the container and return to the *Columns* page.
20. Add the code for the pop-up event.
21. Close the properties notebook for the container part.

Defining the Local File

Since the database to be read is on the workstation, you must define a program described file in your program. Open an Editor session by choosing *Project*, *Edit source code* from the GUI Designer menu bar, and add the following file specification:

RPG

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+.
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords++++++++++++++++++++++++++
FContacts  IF   F  256        DISK    USROPN EXTFILE(file)
F                                     RCDLEN(Field1)
DField1           S              5  0 INZ(256)
 *
```

The USROPN keyword indicates that this file is opened at program execution. The EXTFILE keyword specifies that the field named *file* contains the actual file name to be processed.

Defining the Input Specifications

Now you must define the record layout for the employee record department file using Input specifications. The first record definition is for comment records and the second is for employee records.

```
RPG
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+.
IFilename++SqNORiPos1+NCCPos2+NCCPos3+NCC.............................
IContacts  NS  02    1 C*
 *
IContacts  NS  01
I                                  1    3  DeptA
I                                  1    3 0DeptN
I                                  5    5  Type
I                                  7   29  Name
I                                 30   42  Phone
```

Defining Program Variables

Add the D specifications shown in Figure 207 to your program. These specifications define variables and constants that are used by the program.

```
RPG
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+.
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++
 *
 * Define constants used by the program
DManager          C                   'M'
DEmployee         C                   'E'
DMgrIcon          C                   '.\\MANAGER.BMP'
DEmpIcon          C                   '.\\EMPLOYEE.BMP'
 *
 * Define variables used by the program
DFile             S            256    inz('.\\CONTACTS.TXT')
DRecordID         S              6  0
DParent           S              6
DRecord           S             64
DIconFile         S             64
```

Figure 207. Defining Definition Specifications

The two bitmap files and the CONTACTS.TXT file need to be in your RT_WIN32 directory. They are included in the CONTEXMP.ZIP file from the ITSO FTP server at ftp.almaden.ibm.com/redbooks/sg242222.

Adding Records to the Container

To read records from the workstation database to add to the container, use the Create event for the window part. Create an action subroutine for the window's Create event by invoking the pop-up menu for the window part and choose *Events* and *CREATE*.

Type the logic shown in Figure 208 for the Create event:

RPG

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+.
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq
 *
C     FRA0000B      BEGACT    CREATE        FRA0000B
 * Open employee file and get the first record
C                   Open      Contacts
C                   Read      Contacts                               99
 * Process all records in the employee file
C                   DoW       *IN99 = *OFF
 * Ignore comment records
C                   If        *in01 = *on
 * Handle Manager records
C                   If        Type = Manager
C                   Eval      RecordIDC = DeptA
C                   Eval      IconFile = MgrIcon
C                   Eval      Parent = '0'
C                   Eval      Name = %Trim(Name) + '(' + DeptA + ')'
 * Handle Employee records
C                   Else
C     'CT1'         GetAtr    'GetNewID'    RecordID
C                   Eval      Parent = DeptA
C                   Move      RecordID      RecordIDC         6
C                   Eval      IconFile = EmpIcon
C                   EndIf
```

Figure 208 (Part 1 of 2). Logic to Add Records to the Container

RPG (continued)

```
 * Since AddRcd uses spaces as a delimiter, change spaces
 * to underscores
C     ' ':'_'       Xlate     Name          Name
C                   Eval      Record = RecordIDC + ' ' +
C                                      %Trim(Name) + ' ' +
C                                      %Trim(IconFile) + ' ' +
C                                      %Trim(Parent) + ' ' +
C                                      Phone
C     'CT1'         Setatr    Record        'AddRcd'
C                   EndIf
C                   Setoff                                      0102
C                   Read      Contacts                            99
C                   EndDo
 *
C                   Close     Contacts
C                   Seton                                       LR
 *
C                   ENDACT
 *
```

Figure 208 (Part 2 of 2). Logic to Add Records to the Container

Changing the Container View

To allow the user to change the container view, add a pop-up menu part to the design window. This pop-up menu is displayed when the user presses the right-hand mouse button when the mouse pointer is over the container.

To create the pop-up menu:

1. Locate the pop-up menu part on the Parts Palette and drag and drop it on the design window. This also creates the initial menu item.
2. Drag two more menu items from the Parts Palette and drop them onto the first menu item.
3. Use the properties notebook for the menu items, or use direct editing to change the text of the menu items to one of the following views:
   - Tree View
   - Icon View
   - Details View
4. For each menu item, create an action subroutine to change the view of the container by setting the View attribute of the container. Refer to the *Parts Reference* manual, which is online and can be invoked from the

LPEX Editor or GUI Designer menu bar, to see what values the View attribute must be set to for each view.

Testing the Container Application

Compile and run the container application. Change the container view by using the pop-up menu and notice the difference in the views. Notice that the Details view is the only view that displays all of the records.

Also notice that, in the Icon view, the *record* icon is not always visible. To fix this, you need to add a statement in the Icon View action subroutine to change the container's Arrange attribute. Again, refer to the *Parts Reference* manual to see how this is done. Make this change to your program, recompile, and build to test your changes.

This completes the Container exercise.

Appendix A. File Descriptions

The following sections contain the description of files used during the exercises.

Customer File DDS

The DDS example in Figure 209 is for the file CUSTOMER/CUSTOM01 in library GUIDES2.

DDS

```
...A*. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+ ...7
************** Beginning of data ************************************
    A* CELDIAL CUSTOMER FILE
    A          R CUSTOM01
    A            CUSTNO         7               COLHDG('Customer')
    A            CUSTNA        40               COLHDG('Company name')
    A            REPNO          5               COLHDG('Rep identifier')
    A            CONTAC        30               COLHDG('Name')
    A            CPHONE        17               COLHDG('Telephone')
    A            CFAX          17               COLHDG('Fax')
    A            CADDR         40               COLHDG('Address')
    A            CCITY         30               COLHDG('City')
    A            CCOUNT        20               COLHDG('Country'
    A            CZIP          10               COLHDG('Postal Code')
    A            CZIPLO         1               COLHDG('PC location')
    A                                           VALUES('1' '2' '3')
```

Figure 209. DDS for Customer File

The following DDS example is for the file CUSTOMLF/CUSTOM01 in library GUIDES2:

DDS

```
...A*. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+ ...7
************** Beginning of data ************************************
    A          R CUSTOM01                PFILE(GUIDES2/CUSTOMER)
    A          K CUSTNO
```

The following DDS example is for the file CUSTOML3/CUSTOM01 in library GUIDES2.

DDS

```
...A*. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+ ...7
************** Beginning of data ************************************
     A          R CUSTOM01                  PFILE(GUIDES2/CUSTOMER)
     A          K CUSTNO
```

Customer File on AS/400 System

The example shown in Figure 210 is a partial sample of fields and data from file CUSTOML3/CUSTOM01 in library GUIDES2. The instructor can use the data files from the diskette, create the files and enter sample data from the following example, or come up with unique data. The file on the diskette contains 72 records.

File

```
CustomerCompany        Name           Telephone    Address        Postal  City
number  name                                                     Code
0010100 Meridien Electr Alfredo Bayonne 206-865-4027 10423 S.E. 30th 98007   Bellevue, WA
0010200 Royal Hardware  Arnie Podell    905-619-2045 Maple View Plaz L8D 4S6 Ajax, Ontario
0010300 Webster Applian Bob Wolfstadt   619-549-5212 7350 Miramar Ro 92121   San Diego, CA
0010400 ProLine Buildin Bud Dobbs       905-403-4055 73 Marchwood Ro D8G 3V6 Burlington, Ontario
0010500 Donnamora Const Byron Goeds     905-805-2295 Woodbridge Plaz F7V 5S7 Woodbridge, Ontario
0010700 Universal Commu Dave Franken    415-545-5055 720 Harrison St 94107   San Francisco, CA
0010800 Baker Electroni Dave Matthison  818-715-2045 17150 Koll Cent 94566   Pasadena, CA
0010900 Village Telepho Dayle Swigger   707-367-4530 2752 Betsy Ross 95054   Santa Clara, CA
0011000 Bayview Direct  Claus Weiss     416-448-3987 201 Bloor Stree M8B 7F5 Toronto, Ontario
0011100 BelAir Communic Doreen Coin     914-765-8021 302 Washington  71530   White Plains, NY
0011300 Burnham Trading Efrem Helassie  613-225-0753 91 Baseline Roa C6B 9S3 Trenton, Ontario
0011400 Calderone Impor Elsie Pons      407-392-7077 702 S.W. 15th S 33486   Boca Raton, FL
0011500 The Communicati Esther Varrick  904-599-0377 3011 E. Georgia 33830   Jacksonville, FL
0011600 Sudbury Radio a Garry Morehouse 705-522-5044 8 North Road    P7G 5A3 Sudbury, Ontario
0011700 Christies Elect George Baccxrat 818-707-6767 70223 Agoura Ro 91354   Westlake Village, CA
0011900 Conroy Communic Guy Lewis       918-825-4545 701 S. Adair St 74361   Pryor, OK
```

Figure 210. Data in CUSTOML3 File

Contacts.TXT File

The example shown in Figure 211 contains the layout of the CONTACTS.TXT file used in Exercise 7.

File

```
* Department 817
817 M Eli Javier            (416)448-9999
817 E Larry Schweyer        (416)448-3344
817 E Hans Koert            (416)448-3411
817 E Phil Coulthard        (416)448-2199
*
* Department 814
814 M Enoch Ng              (416)448-9944
814 E Alan Chao             (416)448-7733
814 E Larry Keeling         (416)448-8831
814 E Bobby Siu             (416)448-9900
814 E Paul Kao              (416)448-3388
*
* Department 059
059 M Mark Changfoot        (416)448-0311
059 E Brian Farn            (416)448-9931
059 E Andrew Kerr           (416)448-9911
059 E Derek Lewsey          (416)448-2711
059 E Vincent Suen          (416)448-3992
*
* Department 546
546 M George Farr           (416)448-9221
546 E Dave Cheng            (416)448-9211
546 E Sarah Ettritch        (416)448-8211
546 E John Fellner          (416)448-8222
546 E Scott Ripley          (416)448-2234
```

Figure 211. Data in Contacts.TXT File

Appendix B. VisualAge for RPG Source

This appendix contains listings for the final VisualAge for RPG application from Exercise 1 through Exercise 6.

Source for GUIDES2

The following code represents the GUIDES2 program.

```
*********************************************************************
* Program ID . . :                                                  *
*                                                                   *
* Description  . : Sample VisualAge for RPG program to demonstrate  *
*                  the Component Reference Part.                    *
*                                                                   *
*                  When the Find push button is pressed on this     *
*                  component, a second component, COMPLIST is       *
*                  invoked which displays a subfile containing all  *
*                  customer records.  When the user double-clicks   *
*                  on a customer record, the customer number is     *
*                  set into the CUSTNO entry field in this component*
*                                                                   *
* Function . . . :                                                  *
* Messages . . . :                                                  *
* Files  . . . . :                                                  *
* Input  . . . . :                                                  *
* Output . . . . :                                                  *
* Change activity:                                                  *
*                                                                   *
* Who  Date    Flag  Description                                    *
* ---  ------  ----  ---------------------------------------------- *
*                                                                   *
*********************************************************************
 *
H
Fcustomlf  if   E           k disk    remote
Dinfobox          m                   style(*info)
D                                     button(*retry: *abort: *enter)
 *
Dmsg1             m                   msgnbr(*msg001)
 *
Dmsg2             m                   msgnbr(*msg002)
D                                     msgdata(custno)
 *
```

Figure 212 (Part 1 of 4). GUIDES2 Source

```
Dwarnbox          m                   style(*warn)
D                                     button(*yesbutton: *nobutton)
 *
Dtext             m                   msgtext('Number Not Found')
 ***************************************************************************
 *                                                                         *
 * Window . . :                                                            *
 * Part . . . :                                                            *
 * Event  . . :                                                            *
 * Description:                                                            *
 * Change activity:                                                        *
 *                                                                         *
 * Who  Date    Flag  Description                                          *
 * ---  ------  ----  ------------------------------------------------ *
 *                                                                         *
 ***************************************************************************
 *
C     PSBEXIT       BEGACT    PRESS         CUSTINQ
C                   move      *on           *inlr
 *
C                   ENDACT
 ***************************************************************************
 *                                                                         *
 * Window . . :                                                            *
 * Part . . . :                                                            *
 * Event  . . :                                                            *
 * Description:                                                            *
 * Change activity:                                                        *
 *                                                                         *
 * Who  Date    Flag  Description                                          *
 * ---  ------  ----  ------------------------------------------------ *
 *                                                                         *
 ***************************************************************************
C     PSBOK         BEGACT    PRESS         CUSTINQ
C     'psbok'       getatr    'Backcolor'   bcolor           2 0
C                   if        bcolor <> *RED
C     'psbok'       setatr    *RED          'backcolor'
C                   else
C     'psbok'       setatr    *PALEGRAY     'Backcolor'
C                   end
C     'custno'      getatr    'text'        custno
 *  read the customer file, if no customer found
```

Figure 212 (Part 2 of 4). GUIDES2 Source

```
C     custno        chain     customlf                        50
C                   if        *in50 = *off
C                   showwin   'custinfo'                        51
C                   if        *in51 = *on
C                   eval      %setatr('custinfo':'custinfo':'visible')=1
C                   eval      %setatr('custinfo':'custinfo':'focus')=1
C                   endif
C                   write     'custinfo'
C                   else
C     msg1          DSPLY     infobox       reply            9 0
C     'custno'      setatr    *red          'backcolor'
C                   endif
 *
C                   ENDACT
 ***********************************************************************
 *                                                                     *
 * Window . . . . :                                                   *
 * Part . . . . . :                                                   *
 * Event  . . . . :                                                   *
 * Description  . :                                                   *
 * Change activity:                                                   *
 *                                                                     *
 * Who  Date    Flag  Description                                     *
 * ---  ------  ----  ------------------------------------------- *
 *                                                                     *
 ***********************************************************************
C     INFOOK        BEGACT    PRESS         CUSTINFO
 *
C     'CUSTINFO'    setatr    0             'visible'
C                   ENDACT
 ***********************************************************************
 *                                                                     *
 * Window . . . . :                                                   *
 * Part . . . . . :                                                   *
 * Event  . . . . :                                                   *
 * Description  . :                                                   *
 * Change activity:                                                   *
 *                                                                     *
 * Who  Date    Flag  Description                                     *
 * ---  ------  ----  ------------------------------------------- *
 *                                                                     *
 ***********************************************************************
```

Figure 212 (Part 3 of 4). GUIDES2 Source

```
 *
C     CUSTINQ       BEGACT    ACTIVATE      CUSTINQ
C     'CUSTINq'     setatr    1             'focus'
C     'CUSTno'      setatr    1             'focus'
 *
C                   ENDACT
 ********************************************************************
 *                                                                  *
 * Window . . . . :                                                *
 * Part . . . . . :                                                *
 * Event  . . . . :                                                *
 * Description  . :                                                *
 * Change activity:                                                *
 *                                                                  *
 * Who  Date    Flag  Description                                   *
 * ---  ------  ----  ------------------------------------------- *
 *                                                                  *
 ********************************************************************
C     PSBFIND       BEGACT    PRESS         CUSTINQ
 *  put up the customer list window, if the window is all ready up
 *  just set focus to it.
C                   start     'complist'                          65
C                   if        *in65 = *on
C                   eval      %setatr('custinq':'custinq':'focus')=0
C                   endif
C                   ENDACT
 ********************************************************************
 *                                                                  *
 * Window . . . . :    CUSTINQ                                     *
 * Part . . . . . :                                                *
 * Event  . . . . :    NOTIFY                                      *
 * Description  . :                                                *
 * Change activity:                                                *
 *                                                                  *
 * Who  Date    Flag  Description                                   *
 * ---  ------  ----  ------------------------------------------- *
 *                                                                  *
 ********************************************************************
 *
C     MONCHG        BEGACT    NOTIFY        CUSTINQ
C     'MONCHG'      GETATR    'attrvalue'   CUSTNO
C     'CUSTNO'      SETATR    CUSTNO        'TEXT'
C                   ENDACT
```

Figure 212 (Part 4 of 4). GUIDES2 Source

Source for COMPLIST

The following is the source code for the customer list component.

```
FCustomer  IF   F  186        DISK    usropn extfile(file)
 *
DFile             S             15      Inz('.\\CUSTOMER.DAT')
 *
ICustomer  NS  01
I                                    1    7  CustNo
I                                    8   47  CustNa
I                                   53   82  Contac
I                                   83   99  CPhone
I                                  117  156  CAddr
I                                  157  186  CAddr2
 ******************************************************************************
 *                                                                            *
 * Window . . :                                                               *
 * Part . . . :                                                               *
 * Event  . . :                                                               *
 * Description:                                                               *
 *                                                                            *
 ******************************************************************************
 *
C     CUSTLIST      BEGACT    CREATE        CUSTLIST
 *
C                   Open      Customer
C                   Read      Customer                               99
 *
C     *in99         DowEq     *off
C                   Write     SFL1
C                   Read      Customer                               99
C                   EndDo
 *
C                   Close     Customer
 *
C                   EndAct
 ******************************************************************************
 *                                                                            *
 * Window . . :                                                               *
 * Part . . . :                                                               *
 * Event  . . :                                                               *
 * Description:                                                               *
 *                                                                            *
 ******************************************************************************
```

Figure 213 (Part 1 of 3). COMPLIST Source

```
C     SFL1          BEGACT    ENTER         CUSTLIST
 *
C                   ReadS     SFL1                                  91
C     'EF2'         Setatr    CustNo        'Text'
C*    'CustList'    Setatr    0             'Visible'
 *
C                   EndAct
 ************************************************************************
 *                                                                      *
 * Window . . :                                                         *
 * Part . . . :                                                         *
 * Event  . . :                                                         *
 * Description:                                                         *
 *                                                                      *
 ************************************************************************
 *
C     EF2           BEGACT    CHANGE        CUSTLIST
 *
 *
C                   EndAct
 ************************************************************************
 *                                                                      *
 * Window . . :                                                         *
 * Part . . . :                                                         *
 * Event  . . :                                                         *
 * Description:                                                         *
 *                                                                      *
 * Change activity:                                                     *
 *                                                                      *
 * Who  Date    Flag  Description                                       *
 * ---  ------  ----  ------------------------------------------------- *
 ************************************************************************
 *
C     SFL1          BEGACT    POPUP         CUSTLIST
 *
C     'POPUP1'      Setatr    1             'Visible'
 *
C                   ENDACT
```

Figure 213 (Part 2 of 3). COMPLIST Source

```
                                                                      *
 ************************************************************************
 *                                                                      *
 * Window . . :                                                         *
 * Part . . . :                                                         *
 * Event  . . :                                                         *
 * Description:                                                         *
 *                                                                      *
 ************************************************************************
 *
C     PU_DETAIL     BEGACT    MENUSELECT    CUSTLIST
 *
C                   Reads     SFL1                                   98
C                   Movel     CustNo        Tmp               7
C                   Open      Customer
C                   Read      Customer                               99
C                   DoW       CUSTNO <> Tmp
C                   Read      Customer                               99
C                   EndDo
C                   ShowWin   'Detail'
C                   Write     'Detail'
C                   Close     Customer
 *
C                   ENDACT
 ************************************************************************
 *                                                                      *
 * Window . . :                                                         *
 * Part . . . :                                                         *
 * Event  . . :                                                         *
 * Description:                                                         *
 *                                                                      *
 ************************************************************************
C     CUSTLIST      BEGACT    CLOSE         CUSTLIST
 *
C                   Move      *on           *inlr
 *
C                   ENDACT
 ************************************************************************
 *                                                                      *
 *                                                                      *
 *                                                                      *
 ************************************************************************
 *
```

Figure 213 (Part 3 of 3). COMPLIST Source

Source for Container Example

The following is the source code for the container application for Exercise 7.

```
 ********************************************************************
 *                                                                  *
 * Program ID . . : CONTAIN                                         *
 *                                                                  *
 * Description  . : VisualAge for RPG Container example             *
 *                                                                  *
 ********************************************************************
 *
 * Define the employee record file
FContacts  IF   F  256        DISK    usropn extfile(file)               Employee file
F                                     RCDLEN(Field1)
DField1           S              5  0 INZ(256)
 *
 * Define constants used by the program
DManager          C                   'M'                              Manager record
DEmployee         C                   'E'                              Employee record
DMgrIcon          C                   '.\\MANAGER.ICO'                 Manager icon file
DEmpIcon          C                   '.\\EMPLOYEE.ICO'                Employee icon file
 *
 * Define variables used by the program
DFile             S             12    inz('CONTACTS.TXT')              File to open
DRecordID         S              6  0                                   Container record ID
DParent           S              6                                      Parent ID
DRecord           S             64                                      Container record
DIconFile         S             64                                      Icon file name
 *
 *
IContacts  NS  02    1 C*                                              Comment record
 *
 * Record layout for employee file
IContacts  NS  01
I                                1    3  DeptA                         Department - char
I                                1    3 0DeptN                         Department - num
I                                5    5  Type                          Employee type
I                                7   29  Name                             Name
I                               30   42  Phone                         Phone number
 *
 ********************************************************************
 *                                                                  *
 * Window . . : FRA0000B                                            *
 *                                                                  *
 * Part . . . : FRA0000B                                            *
 *                                                                  *
 * Event  . . : Create                                              *
 *                                                                  *
 * Description: Add records from the employee database to the       *
 *              Container part when the window is created.          *
 *                                                                  *
 ********************************************************************
```

Figure 214 (Part 1 of 3). Container Example Source

```
C     FRA0000B      BEGACT    CREATE        FRA0000B
 *
 * Open employee file and get the first record
C                   Eval      File = %Trim(File)
C                   Open      Contacts                                    Open file
C                   Read      Contacts                              99    Read first record
 *
 * Process all records in the employee file
C                   DoW       *IN99 = *OFF                                Do until EOF
 *
 * Ignore comment records
C                   If        *in01 = *on                                 If not comment
 *
 * Handle Manager records
C                   If        Type = Manager                              Manager record?
C                   Eval      RecordIDC = DeptA                           Use Dept as ID
C                   Eval      IconFile = MgrIcon                          Set icon file name
C                   Eval      Parent = '0'                                Parent record
C                   Eval      Name = %Trim(Name) + '(' + DeptA + ')'
 *
 * Handle Employee records
C                   Else                                                  Employee record
C     'CT1'         GetAtr    'GetNewID'    RecordID                      Get a new ID
C                   Eval      Parent = DeptA                              Set parent ID
C                   Move      RecordID      RecordIDC         6           Make it character
C                   Eval      IconFile = EmpIcon                          Set icon file name
C                   EndIf
 *
 * Since AddRcd uses spaces as a delimiter, change spaces
 * to underscores
C     ' ':'_'       Xlate     Name          Name                          Convert spaces
C                   Eval      Record = RecordIDC + ' ' +                  Construct new record
C                                      %Trim(Name) + ' ' +
C                                      %Trim(IconFile) + ' ' +
C                                      %Trim(Parent) + ' ' +
C                                      Phone
C     'CT1'         Setatr    Record        'AddRcd'                      Add the record
C                   EndIf                                                 End-not comment
 *
C                   Setoff                                      0102
C                   Read      Contacts                              99    Get next record
C                   EndDo                                                 End-Not EOF
 *
C                   Close     Contacts                                    Close file
 *
C                   ENDACT
 ************************************************************************
 *                                                                      *
 * Window . . : FRA0000B                                                *
 *                                                                      *
 * Part . . . : MI_TREE                                                 *
 *                                                                      *
 * Event  . . : MenuSelect                                              *
 *                                                                      *
 * Description: Change the Container view to Tree view                  *
 *                                                                      *
 ************************************************************************
 *
C     MI_TREE       BEGACT    MENUSELECT    FRA0000B
 *
C     'CT1'         Setatr    2             'View'                        Set to Tree view
 *
C                   ENDACT
```

Figure 214 (Part 2 of 3). Container Example Source

```
 ********************************************************************
 *                                                                  *
 * Window . . : FRA0000B                                            *
 *                                                                  *
 * Part . . . : MI_ICON                                             *
 *                                                                  *
 * Event  . . : MenuSelect                                          *
 *                                                                  *
 * Description: Change the Container view to Icon view              *
 *                                                                  *
 ********************************************************************
 *
C     MI_ICON       BEGACT    MENUSELECT    FRA0000B
 *
C     'CT1'         Setatr    1             'View'                    Set to Icon view
C     'CT1'         Setatr    1             'Arrange'                 Arrange icons
 *
C                   ENDACT
 ********************************************************************
 *                                                                  *
 * Window . . : FRA0000B                                            *
 *                                                                  *
 * Part . . . : MI_DETAIL                                           *
 *                                                                  *
 * Event  . . : MenuSelect                                          *
 *                                                                  *
 * Description: Change the Container view to Details view           *
 *                                                                  *
 ********************************************************************
C     MI_DETAIL     BEGACT    MENUSELECT    FRA0000B
 *
c     'ct1'         setatr    3             'view'                    Set to Details view
 *
C                   ENDACT
 ********************************************************************
 *                                                                  *
 * Window . . : FRA0000B                                            *
 *                                                                  *
 * Part . . . : CT1                                                 *
 *                                                                  *
 * Event  . . : PopUp                                               *
 *                                                                  *
 * Description: Display the popup menu for the container part       *
 *                                                                  *
 ********************************************************************
 *
C     CT1           BEGACT    POPUP         FRA0000B
 *
C     'PMN00010'    Setatr    1             'Visible'                 Make popup visible
 *
C                   ENDACT
```

Figure 214 (Part 3 of 3). Container Example Source

Appendix C. Sample Code

You can download the sample code developed in this book from the ITSO FTP server at ftp://ftp.almaden.ibm.com/redbooks/sg242222. In particular, you find there the following files:

UPSAVF.ZIP

This ZIP archive contains multiple files in the UPSAVF directory. The UPSAVF VisualAge for RPG application allows you to create the AS/400 environment for the exercises in Part 2 of this book:

- UPLOAD project files
- CUSTOMER physical file
- QDDSSRC source for customer file (physical and logical)

SAMPLES.ZIP

This ZIP archive contains the various components that are developed throughout Part 1 of this book. You will find sample projects about:

- Containers
- Clipboards
- National Language Support
- Data queues and data areas
- Remote command execution
- Printing
- REXX procedure invocation
- Mathematical functions
- Message handling
- Online help
- Record I/O
- SQL support

CONTEXMP.ZIP

This ZIP archive contains the project files, the exectuables, and the following local PC files and bitmaps used in Exercise 7.

- Contacts.TXT
- Manager.BMP
- Employee.BMP

AS/400 Upload Program

The program UPSAVF uploads the DDS and data files required for the VisualAge for RPG exercises. It creates the following files in the library that has already been created on the AS/400 system.

- QDDSSRC
  Contains DDS source for creating the physical and logical files.
- CUSTOMER
  This physical file contains customer records.

Note

Before the UPSAVF program is executed, the .RST file must be modified to reference the correct AS/400 system. To do this, use the following steps:

- Go to the project's run-time directory RT_WIN32.
- Use an editor to edit the .RST file.
- Change the value of the REMOTE_LOCATION_NAME to reflect the name of the AS/400 system.
- Save the .RST file.

To upload the exercise files, do the following:

- On the AS/400 system, create a library to receive the files. For the VisualAge for RPG exercises, this library should be named *GUIDES2*.
- Run the UPSAVF program (click with the right mouse button on project *UPSAVF* and select *RUN* from the pop-up menu).
- In the window that is shown, type library name GUIDES2 in the *Library* field if it is not already GUIDES2.
- Click on the *Upload* push button to begin the upload. Do not select any of the check boxes. They are used as status indicators.
- As each step completes, the corresponding check box is checked.
- When the upload is complete as indicated by the *Done* check box, go to the AS/400 system and compile the two logical files.

Source for AS/400 Upload Program

The following example contains a sample of VisualAge for RPG source code that can be used to upload the AS/400 files for the exercises.

```
 ********************************************************************
 *                                                                  *
 * Program ID . . :                                                 *
 *                                                                  *
 * Description  . :  This program will upload the DDS and data      *
 *                   file required for the VisualAge exercises. It  *
 *                   will put up a window for the user to the enter *
 *                   the AS/400 library which has been created      *
 *                   to contain the lab files. GUIDES2 is the       *
 *                   library which should be created.               *
 *                   Files QDDSSRC - contains DDS source            *
 *                         CUSTOMER- This physical file contains    *
 *                                   customer records               *
 ********************************************************************
 *
FDDSIN     IF   F  128        Disk    EXTFILE(PCFILE) usropn rcdlen(Recln1)
FPCCUST    IF   F  217        Disk    EXTFILE(cust) usropn rcdlen(Recln2)
FQDDSSRC   O    E             Disk    Remote usropn Rename(QDDSSRC:FMT01)
FCUSTOMER  O    E             Disk    Remote usropn Rename(CUSTOM01:FMT02)
 *
DRcdln1           S              5  0 INZ(128)
DRcdln2           S              5  0 INZ(217)
DSaveFile         S             14A
DCMD              S            128A
DMEMBER           S             10A
DCUSTFILE         S             21A   INZ('TESTLIB/CUSTOMER')
DTOFILE           S             21A   INZ('TESTLIB/QDDSSRC')
DPCFILE           S             15A
DCUST             S             15A   INZ('.\\CUSTOMER.DAT')
DQCMDDDM          S              7A   INZ('QCMDDDM') LINKAGE(*SERVER)
DCMDLEN           S             15P 5 INZ(%SIZE(CMD))
DLC               S             26    INZ('abcdefghijklmnopqrstuvwxyz')
DUC               S             26    INZ('ABCDEFGHIJKLMNOPQRSTUVWXYZ')
 *
IDDSIN     NS  01
I                                  1   80  SRCDTA
```

Figure 215 (Part 1 of 6). Source for AS/400 Upload Program

```
IPCCust    NS  01
I                                1    7  CustNo
I                                8   47  CustNa
I                               48   52  RepNo
I                               53   82  Contac
I                               83   99  CPhone
I                              100  116  CFax
I                              117  156  CAddr
I                              157  186  CCity
I                              187  206  CCount
I                              207  216  CZip
I                              217  217  CRegion
 ********************************************************************
 *                                                                  *
 * Window . . . . :                                                 *
 * Part . . . . . :                                                 *
 * Event  . . . . :                                                 *
 * Description  . :                                                 *
 *                                                                  *
 ********************************************************************
 *
C     UPLOAD        BEGACT    PRESS         MAIN
 *
C     'LIBRARY'     Getatr    'Text'        LIBRARY
C                   If        Library = *Blanks
C     *MSG0001      Dsply                   rc               9 0
 *
C                   Else
c     LC:UC         Xlate     library       library
C                   Eval      tofile = %Trim(library) + '/QDDSSRC'
C                   Eval      custfile = %Trim(library) + '/CUSTOMER'
 *
C                   Eval      CMD = 'CRTSRCPF ' + tofile
C                   Exsr      IssCmd
 *
C                   Eval      CMD = 'ADDPFM ' + tofile + ' ' + 'CUSTOMER' +
C                                   ' SRCTYPE(PF)'
C                   Exsr      ISSCMD
 *
C                   Eval      CMD = 'ADDPFM ' + tofile + ' ' + 'CUST   OMLF' +
C                                   ' SRCTYPE(LF)'
C                   Exsr      ISSCMD
 *
C                   Eval      CMD = 'ADDPFM ' + tofile + ' ' + 'CUSTOML3' +
C                                   ' SRCTYPE(LF)'
C                   Exsr      ISSCMD
```

Figure 215 (Part 2 of 6). Source for AS/400 Upload Program

```
C     'CRTDDS'      Setatr    1             'Checked'
C                   Eval      CMD    = 'OVRDBF QDDSSRC ' + tofile +
C                                       MEMBER + ' ' +
C                                       'OVRSCOPE(*JOB)'
C                   Exsr      ISSCMD
 *
C                   Eval      member = 'CUSTOMER'
C                   Exsr      CPYDDS
C     'CUSTOMER'    Setatr    1             'Checked'
C                   Eval      CMD = 'CRTPF ' + CUSTFILE + ' ' + tofi   le
C                   Exsr      IssCmd
C     'CRTPF'       Setatr    1             'Checked'
 *
C                   Exsr      CpyDta
C                   Eval      member = 'CUSTOMLF'
C                   Exsr      CPYDDS
C     'CUSTOMLF'    Setatr    1             'Checked'
 *
C                   Eval      member = 'CUSTOML3'
C                   Exsr      CPYDDS
 *
C     'CUSTOML3'    Setatr    1             'Checked'
 *
C     'DONE'        Setatr    1             'Checked'
 *
C                   EndIf
 *
C                   ENDACT
 ***************************************************************************
 *                                                                         *
 * Subroutine . . :                                                        *
 * Description  . :                                                        *
 *                                                                         *
 ***************************************************************************
 *
C     *INZSR        BEGSR
 *
C                   Z-Add     10            SRCSEQ
 *
C                   ENDSR
 ***************************************************************************
 *                                                                         *
 * Subroutine . . :                                                        *
 * Description  . :                                                        *
 *                                                                         *
 ***************************************************************************
```

Figure 215 (Part 3 of 6). Source for AS/400 Upload Program

```
C     CPYDDS        BEGSR
 *
C                   Eval      CMD    = 'OVRDBF QDDSSRC ' + tofile +
C                                      MEMBER + ' ' +
C                                      'OVRSCOPE(*JOB) OPNSCOPE(*JOB)'
C                   Exsr      ISSCMD
 *
C                   Eval      PCFile = '.\\' + %Trim(member) + '.DDS'
C                   Open      DDSIN
C                   Open      QDDSSRC
C                   Read      DDSIN                                  99
C                   Eval      SRCSEQ = 10
 *
C                   Dow       *in99 = *off
C                   Write     FMT01
C                   Eval      SRCSEQ = SRCSEQ + 10
C                   Read      DDSIN                                99
C                   EndDo
 *
C                   Close     DDSIN
C                   Close     QDDSSRC
 *
C                   ENDSR
 ********************************************************************
 *                                                                  *
 * Subroutine . . :                                                 *
 * Description  . :                                                 *
 *                                                                  *
 ********************************************************************
 *
C     ISSCMD        BEGSR
 *
C                   CALL      QCMDDDM                            8   0
C                   PARM                    CMD
C                   PARM                    CMDLEN
 *
C                   ENDSR
 ********************************************************************
 *                                                                  *
 * Subroutine . . :                                                 *
 * Description  . :                                                 *
 *                                                                  *
 ********************************************************************
```

Figure 215 (Part 4 of 6). Source for AS/400 Upload Program

```
C     CPYDTA        BEGSR
 *
C                   Eval      CMD    = 'OVRDBF CUSTOMER ' + custfile +
C                                      ' CUSTOMER OVRSCOPE(*JOB) +
C                                        OPNSCOPE(*JOB)'
C                   Exsr      ISSCMD
C                   Open      Customer
C                   Open      PCCust
C                   Read      PCCust                                 99
 *
C                   DoW       *in99 = *off
C                   Write     FMT02
C                   Read      PCCust                                 99
C                   EndDo
 *
C     'Data'        Setatr    1             'Checked'
C                   Close     Customer
C                   Close     PCCust
 *
C                   Endsr
 ************************************************************************
 *                                                                      *
 * Window . . . . :                                                    *
 * Part . . . . . :                                                    *
 * Event  . . . . :                                                    *
 * Description  . :                                                    *
 *                                                                      *
 ************************************************************************
 *
C     CLOSE         BEGACT    PRESS         MAIN
 *
C                   Move      *on           *inlr
 *
C                   ENDACT
 ************************************************************************
 *                                                                      *
 * Window . . . . :                                                    *
 * Part . . . . . :                                                    *
 * Event  . . . . :                                                    *
 * Description  . :                                                    *
 *                                                                      *
```

Figure 215 (Part 5 of 6). Source for AS/400 Upload Program

```
 ********************************************************************
 *
C     MAIN          BEGACT    CREATE        MAIN
 *
C     'Library'     Setatr    1             'Focus'
 *
C                   ENDACT
```

Figure 215 (Part 6 of 6). Source for AS/400 Upload Program

Appendix D. Special Notices

This publication is intended to help AS/400 RPG programmers to learn how to use VisualAge for RPG to develop AS/400 client/server applications. The information in this publication is not intended as the specification of any programming interfaces that are provided by VisualAge for RPG. See the PUBLICATIONS section of the IBM Programming Announcement for VisualAge for RPG for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

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 the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594 USA.

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 IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers

attempting to adapt these techniques to their own environments do so at their own risk.

Any pointers in this publication to external Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.

The following document contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples contain the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Reference to PTF numbers that have not been released through the normal distribution process does not imply general availability. The purpose of including these reference numbers is to alert IBM customers to specific information relative to the implementation of the PTF when it becomes available to each customer according to the normal IBM PTF distribution process.

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

AnyNet
Application System/400
AS/400
Client Access
Client Access/400
CUA
DATABASE 2 OS/400
DB2
DB2/400
IBM
Integrated Language Environment
OS/2
OS/400
Personal System/2
RPG/400
SQL/400
VisualAge
VRPG CLIENT
400

The following terms are trademarks of other companies:

PC Direct is a trademark of Ziff Communications Company and is used by IBM Corporation under license.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Microsoft, Windows, and the Windows 95 logo
are trademarks or registered trademarks of Microsoft Corporation.

Java and HotJava are trademarks of Sun Microsystems, Inc.

Other trademarks are trademarks of their respective companies.

Appendix E. Related Publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

International Technical Support Organization Publications

For information on ordering these ITSO publications see “How to Get ITSO Redbooks.”

- *Inside AS/400 Client Access for Windows 95/NT*, SG24-4748
- *Inside Client Access/400 for Windows 3.1*, SG24-4429
- *AS/400 Applications: A Fast an Easy Way to Install, Set Up and Work with VRPG and CODE/400*, SG24-4841

Redbooks on CD-ROMs

Redbooks are also available on CD-ROMs. **Order a subscription** and receive updates 2-4 times a year at significant savings.

CD-ROM Title	Subscription Number	Collection Kit Number
System/390 Redbooks Collection	SBOF-7201	SK2T-2177
Networking and Systems Management Redbooks Collection	SBOF-7370	SK2T-6022
Transaction Processing and Data Management Redbook	SBOF-7240	SK2T-8038
Lotus Redbooks Collection	SBOF-6899	SK2T-8039
Tivoli Redbooks Collection	SBOF-6898	SK2T-8044
AS/400 Redbooks Collection	SBOF-7270	SK2T-2849
RS/6000 Redbooks Collection (HTML, BkMgr)	SBOF-7230	SK2T-8040
RS/6000 Redbooks Collection (PostScript)	SBOF-7205	SK2T-8041
RS/6000 Redbooks Collection (PDF Format)	SBOF-8700	SK2T-8043
Application Development Redbooks Collection	SBOF-7290	SK2T-8037

Other Publications

These publications are also relevant as further information sources:

- *Getting Started with VisualAge for RPG*, SC09-2448
- *Programming with VisualAge for RPG*, SC09-2449
- *VisualAge for RPG Parts Reference*, SC09-2450
- *VisualAge for RPG Language Reference*, SC09-2451

How to Get ITSO Redbooks

This section explains how both customers and IBM employees can find out about ITSO redbooks, CD-ROMs, workshops, and residencies. A form for ordering books and CD-ROMs is also provided.

This information was current at the time of publication, but is continually subject to change. The latest information may be found at http://www.redbooks.ibm.com/.

How IBM Employees Can Get ITSO Redbooks

Employees may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- **PUBORDER** —to order hardcopies in United States
- **GOPHER link to the Internet** - type GOPHER.WTSCPOK.ITSO.IBM.COM
- **Tools disks**

  To get LIST3820s of redbooks, type one of the following commands:

  ```
  TOOLS SENDTO EHONE4 TOOLS2 REDPRINT GET SG24xxxx PACKAGE
  TOOLS SENDTO CANVM2 TOOLS REDPRINT GET SG24xxxx PACKAGE (Canadian users only)
  ```

  To get BookManager BOOKs of redbooks, type the following command:

  ```
  TOOLCAT REDBOOKS
  ```

  To get lists of redbooks, type the following command:

  ```
  TOOLS SENDTO USDIST MKTTOOLS MKTTOOLS GET ITSOCAT TXT
  ```

  To register for information on workshops, residencies, and redbooks, type the following command:

  ```
  TOOLS SENDTO WTSCPOK TOOLS ZDISK GET ITSOREGI 1998
  ```

  For a list of product area specialists in the ITSO: type the following command:

  ```
  TOOLS SENDTO WTSCPOK TOOLS ZDISK GET ORGCARD PACKAGE
  ```

- **Redbooks Web Site on the World Wide Web**

  http://w3.itso.ibm.com/redbooks/

- **IBM Direct Publications Catalog on the World Wide Web**

  http://www.elink.ibmlink.ibm.com/pbl/pbl

  IBM employees may obtain LIST3820s of redbooks from this page.

- **REDBOOKS category on INEWS**
- **Online** —send orders to: USIB6FPL at IBMMAIL or DKIBMBSH at IBMMAIL
- **Internet Listserver**

  With an Internet e-mail address, anyone can subscribe to an IBM Announcement Listserver. To initiate the service, send an e-mail note to announce@webster.ibmlink.ibm.com with the keyword subscribe in the body of the note (leave the subject line blank). A category form and detailed instructions will be sent to you.

How Customers Can Get ITSO Redbooks

Customers may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- **Online Orders** —send orders to:

	IBMMAIL	**Internet**
In United States:	usib6fpl at ibmmail	usib6fpl@ibmmail.com
In Canada:	caibmbkz at ibmmail	lmannix@vnet.ibm.com
Outside North America:	dkibmbsh at ibmmail	bookshop@dk.ibm.com

- **Telephone orders**

United States (toll free)	1-800-879-2755
Canada (toll free)	1-800-IBM-4YOU
Outside North America	(long distance charges apply)
(+45) 4810-1320 - Danish	(+45) 4810-1020 - German
(+45) 4810-1420 - Dutch	(+45) 4810-1620 - Italian
(+45) 4810-1540 - English	(+45) 4810-1270 - Norwegian
(+45) 4810-1670 - Finnish	(+45) 4810-1120 - Spanish
(+45) 4810-1220 - French	(+45) 4810-1170 - Swedish

- **Mail Orders** —send orders to:

IBM Publications	IBM Publications	IBM Direct Services
Publications Customer Support	144-4th Avenue, S.W.	Sortemosevej 21
P.O. Box 29570	Calgary, Alberta T2P 3N5	DK-3450 Allerød
Raleigh, NC 27626-0570	Canada	Denmark
USA		

- **Fax** —send orders to:

United States (toll free)	1-800-445-9269
Canada	1-403-267-4455
Outside North America	(+45) 48 14 2207 (long distance charge)

- **1-800-IBM-4FAX (United States)** or **(+1)001-408-256-5422 (Outside USA)** —ask for:

  Index # 4421 Abstracts of new redbooks
  Index # 4422 IBM redbooks
  Index # 4420 Redbooks for last six months

- **Direct Services** - send note to softwareshop@vnet.ibm.com
- **On the World Wide Web**

Redbooks Web Site	http://www.redbooks.ibm.com/
IBM Direct Publications Catalog	http://www.elink.ibmlink.ibm.com/pbl/pbl

- **Internet Listserver**

  With an Internet e-mail address, anyone can subscribe to an IBM Announcement Listserver. To initiate the service, send an e-mail note to announce@webster.ibmlink.ibm.com with the keyword subscribe in the body of the note (leave the subject line blank).

IBM Redbook Order Form

Please send me the following:

Title	Order Number	Quantity

First name Last name

Company

Address

City Postal code Country

Telephone number Telefax number VAT number

□ Invoice to customer number

□ Credit card number

Credit card expiration date Card issued to Signature

We accept American Express, Diners, Eurocard, Master Card, and Visa. Payment by credit card not available in all countries. Signature mandatory for credit card payment.

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 (|) 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.
- *IBM Dictionary of Computing*, New York: McGraw-Hill, 1994.
- *Object-Oriented Interface Design IBM Common User Interface Guidelines*, SC34-4399-00, Carmel, IN: Que Corporation, 1992.

If the definitions in this glossary differ from those in the &bigsys. *&b4006.*, &b4006n., use the ones in this glossary.

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.

anchor. Any part that you use as a reference point for aligning, sizing, and spacing other parts.

application. A collection of software components used to perform specific user tasks on a computer.

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)

audio part. A part that gives a program the ability to process wave files (.WAV) and MIDI files (.MID).

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 VisualAge for RPG 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

canvas part. A part onto which you can drag and drop 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 for RPG, you drag a check box part from the parts palette or parts catalog and drop 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 VisualAge for RPG and VisualAge for RPG 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 for RPG, you can drag a combination box part from the parts palette or parts catalog and drop 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 VisualAge for RPG 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 a 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.

entry 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 for RPG, you drag an entry field part from the parts palette or parts catalog and drop it onto a design window.

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 for 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

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 for RPG, you drag a group box part from the parts palette or parts catalog and drop 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*.

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 VisualAge for RPG part. Contrast with *export*.

inactive window. A window that can not receive keyboard input at a given moment.

index. The identifier of an entry in VisualAge for RPG 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.

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 for RPG, you can drag the list box part from the parts palette or parts catalog and drop 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 VisualAge for RPG 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 an audio part.

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 for RPG, you can drag a menu bar part from the parts palette or parts catalog and drop 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.

MIDI file. Musical Instrument Digital Interface 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

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 95 or Windows NT, a notebook is sometimes referred to as a Windows 95 or Windows NT 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*.

O

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)

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 VisualAge for RPG application together for distribution.

parts. Objects that make up the GUI of a VisualAge for RPG application.

parts catalog. A storage space for all of the parts used to create graphical user interfaces for VisualAge for RPG 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.

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 drag a pop-up menu part from the parts palette or parts catalog and drop 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 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 drag a push button part from the parts palette or parts catalog and drop 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 drag a radio button part from the parts palette or parts catalog and drop 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*.

selection border. The visual border that appears around a VisualAge for RPG 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 drag a slider part from the parts palette or parts catalog and drop 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 VisualAge for RPG 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 drag a spin button part from the parts palette or parts catalog and drop it onto a design window.

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*.

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.

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 VisualAge for RPG application is stored after a build. Contrast with *target folder*.

target folder. The object in which the icon representing a VisualAge for RPG 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.

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 drag and drop this part onto the design window as you would any other VisualAge for RPG part.

utility DLL. See *NOMAIN module*.

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 drag a window part from the parts palette or parts catalog and drop 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.

workstation. A device that allows a user to do work. See also *programmable workstation*.

List of Abbreviations

CCSID	coded character set identifier
CL	command language
CODE	cooperative development environment
DDL	dynamic link library
DDM	distributed data management
DDS	data description specifications
EXE	executable
GUI	graphical user interface
IBM	International Business Machines Corporation
ILE	integrated language environment
IPF	information presentation facility
ITSO	International Technical Support Organization
LPEX	live parsing editor
NLS	national language support
ODP	open data path
PWS	programmable workstation
RPG	report program generator
SFL	subfile
SQL	structured query language

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

ITSO Redbook Evaluation

AS/400 Programming with VisualAge for RPG
SG24-2222-00

Your feedback is very important to help us maintain the quality of ITSO redbooks. **Please complete this questionnaire and return it using one of the following methods:**

- Use the online evaluation form found at http://www.redbooks.ibm.com
- Fax this form to: USA International Access Code + 1 914 432 8264
- Send your comments in an Internet note to redbook@us.ibm.com

Please rate your overall satisfaction with this book using the scale:
(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)

Overall Satisfaction _____

Please answer the following questions:

Was this redbook published in time for your needs? Yes____ No____

If no, please explain:

_____

_____

_____

_____

What other redbooks would you like to see published?

_____

_____

_____

Comments/Suggestions: **(THANK YOU FOR YOUR FEEDBACK!)**

_____

_____

_____

_____

_____

SG24-2222-00
Printed in the U.S.A.