RPC Broker 1.1

Developer's Guide (REDACTED)

September 2021

Department of Veterans Affairs (VA)

Office of Information and Technology (OIT)

Enterprise Program Management Office (EPMO)

Revision History

Documentation Revisions

Date	Revision	Description	Author
09/15/2021	6.0	Tech Edits based on the Broker Development Kit (BDK) release with RPC Broker Patch XWB*1.1*73 (Client-Side only; no VistA M Server updates): • Supports Delphi XE8, 10.0, 10.1, 10.2, 10.3, and Delphi/RAD Studio v10.4: Section 1.4.1. • Corrects the following issues: o Ensures the data placed into the **Brokerx.User.Division** field is correctly formatted. o Redesigned the method of certificate processing; it automatically selects the user's Authentication certificate, eliminating the need for the user to select from a list of certificates. • Added the **ShowCertDialog** property. Updated the following: o Table 4: TCCOWRPCBroker Component—All Properties (listed alphabetically). o Section 2.6.50. • Deleted Section 1.7, "Online Help," and references to **.chm**/online help, since the online help is *not* being released with RPC Broker Patch XWB*1.1*73. **RPC Broker 1.1; XWB*1.1*73 BDK**	RPC Broker Development Team
12/1/2020	5.0	Tech Edits based on the Broker Development Kit (BDK) release with RPC Broker Patch XWB*1.1*72 (Client-Side only; no VistA M Server updates): • Supports Delphi XE8, 10.0, 10.1, 10.2, 10.3, and Delphi/RAD Studio v10.4: Sections 1.3 and 1.4.2.	RPC Broker Development Team

Date	Revision	Description	Author

		• Corrects the following issues:
		   ○ Ensures the DIVISION field is properly set.
		   ○ Addresses Hints and Warnings along with many of the memory leaks.
		RPC Broker 1.1; XWB*1.1*72 BDK	

05/06/2020	4.0	Tech Edits based on the Broker Development Kit (BDK) release with RPC Broker Patch XWB*1.1*71 (Client-Side only; no VistA M Server updates): • Added Section 1.4.1, "XWB*1.1*71." • Updated supported Delphi versions to: XE8, 10.0, 10.1, 10.2, and 10.3: Sections 1.3, 1.4.3. • Corrected broken links in Section 3.7.8. • Updated "Caution" note for the reference **PType** in Table 11, Table 14, and Table 61. • Reformatted all references to file and field name numbers throughout. • Updated API formatting to synchronize with online APIs. • Updated document to follow current documentation standards and style guidelines. **RPC Broker 1.1; XWB*1.1*71 BDK**	RPC Broker Development Team
02/15/2017	3.0	Tech Edits based on release of RPC Broker Patch XWB*1.1*65: • Reformatted document to follow current documentation standards and style formatting requirements. • Added Section 1.1.1. • Updated Step 2 for 2-factor authentication (2FA) and BSE "GUI Developer Issues" in Section 1.1.2. • Updated Section 1.3 for IPv4/IPv6 support, 2-factor authentication, renamed XWBHash Unit, and	RPC Broker Development Team

Date	Revision	Description	Author
		current Delphi software version support. Also, added "**TXWBSSOiToken**" to the list of components.	
		• Restructured Section 1.4 to now list changes by BDK patch release. Updated the following content in those sections for Patch XWB*1.1*65:	
		o Caution Note for 2-factor authentication.	
		o New/Modified components.	
		o Current Delphi software support and 2-factor authentication.	
		o New library methods	
		o New properties.	
		o Modified type.	
		• Updated Section 1.5.2 for currently supported Delphi software version.	
		• Updated Section 1.6.5 for support of three types of silent login.	
		• Updated Section 2.1.1.3; clarified SSO/UC-aware and capable of CCOW single sign-on (SSO).	
		• Updated Section 2.1.3.4.	
		• Updated patch reference in Section 2.1.3.5.	
		• Changed "Using clause" to "Uses clause" in Section 2.2.6.2.	
		• Added the "XWBSSOi Unit" and renamed "Hash Unit" to "XWBHash Unit" in Section 2.3 and moved "XWBHash Unit" to Section 2.3.10.	
		• Added Section 2.3.11, "XWBSSOi Unit."	
		• Added the following properties to Table 4:	
		o **SecurityPhrase** Property	
		o **SSHHide** Property	
		o **SSHport** Property	

Date	Revision	Description	Author
		o **SSHpw** Property o **SSHUser** Property o **SSOiToken** Property o **SSOiSECID** Property o **SSOiADUPN** Property o **SSOiLogonName** Property o **UseSecureConnection** Property • Added the following properties to Table 6: o **SSOiToken** Property o **SSOiSECID** Property o **SSOiADUPN** Property o **SSOiLogonName** Property • Added Section 2.1.5, "TXWBSSOiToken Component." • Added "Caution" note for the reference **PType** in Table 11, Table 14, and Table 61. • Added the following new properties: o SSOiADUPN Property (TRPCBroker Component) o SSOiADUPN Property (TXWBSSOiToken Component) o SSOiLogonName Property (TRPCBroker Component) o SSOiLogonName Property (TXWBSSOiToken Component) o SSOiSECID (TRPCBroker Component) o SSOiSECID Property (TXWBSSOiToken Component) o SSOiToken Property (TRPCBroker Component) o SSOiToken Property (TXWBSSOiToken Component) • Removed Note reference to the example in Sections 2.6.32.3,	

Date	Revision	Description	Author
		2.6.71.3, 7.16.2.1, and 7.16.2.2; those sample files no longer distributed. • Modified/Renamed Section 4. o Added new RPC Broker APIs Section 4.1. o Moved content and added Section 4.2. • Updated the option parameter description to clarify it is an "encrypted name" in Section 4.1.5. • Added the following new RPCs to Section 4.2: o XWB CREATE CONTEXT o XWB GET BROKER INFO o XWB IM HERE • Updated Section 4.2.9.1 regarding Windows registry entries. • Added Section 5, "Broker Security Enhancement (BSE)." Consolidating all developer-related content from the standalone *Broker Security Enhancement (BSE) Supplement to Patch: XWB*1.1*45 & XU*8.0*404* document into the *RPC Broker Developer's Guide*. Specifically, content taken from Sections 3-6. **NOTE:** Once all content is transferred from the BSE standalone document into the appropriate RPC Broker documents, the *Broker Security Enhancement (BSE) Supplement to Patch: XWB*1.1*45 & XU*8.0*404* will be deleted and removed from the VDL. • Changed references from "Borland Delphi" to "Embarcadero Delphi," removed Note referring to CAPRI, and updated Steps 1-2, removed old Step 3 (including RPC Broker login components) and old Step 8 (recompiling	

Date	Revision	Description	Author
		application), updated new Steps 3-7 in Section 5.3. • Updated Section 6.2. • Deleted "BAPI32.DLL not able to support SSH" Note from Section 8.1. • Added the MySsoToken Function to Table 40. • Updated Section 8.3.2. Also, deleted Section 8.7; it was a duplicate of Section 8.3.2. • Added Section 8.7, "MySsoToken Function." • Added "SAML" and "XML" to the Glossary, Table 66. **RPC Broker 1.1; XWB*1.1*65 BDK**	
04/28/2016	2.0	Tech Edits based on release of RPC Broker Patch XWB*1.1*60 (released 06/11/2015): • Reformatted document to follow current documentation standards and style formatting requirements. • Updated the "Orientation" section. • Updated Section 1.3. • Updated Section 1.4.2.1 for deprecated (removed) components. • Updated Section 1.4.2.2 for added or modified components. • Updated Section 1.4.4 for added functionality. • Updated Section 1.4.5.2 for modified methods. • Updated Section 1.4.6.1 for deprecated (removed) properties. • Added Figure 1 caption. • Removed deprecated properties from Table 4, Table 6, Table 7, and Table 9. • Modified command line parameter in Section 2.1.3.4. • Updated Sections 2.4.1.4.1 and 2.4.1.4.2.	RPC Broker Development Team

Date	Revision	Description	Author
		• Updated Section 2.6.52.4. • Modified property references in Sections 2.6.55.3 and 2.6.56.3. • Updated Figure 53. • Removed malfunction note in Section 4.2.7.1. • Updated Figure 59. • Removed Winsock reference note from Section 5.4. • Removed caution note regarding writing to Windows registry in Section 6.17. • Updated Section 7.1. • Removed references to "**DSM**" and **ZDCEBUG** throughout. Also, deleted Section 5.7, "Identifying the Listener Process on the Server" and Section 5.8, "Identifying the Handler Process on the Server," since they referred to DSM commands and processes. • Updated help file references from "BROKER.HLP" to "**Broker_1_1.chm**" throughout. • Updated references to show RPC Broker Patch XWB*1.1*60 supports Delphi XE7, XE6, XE5, and XE4 throughout. **RPC Broker 1.1; XWB*1.1*60 BDK**	
04/16/2014	1.2	Tech Edits: • Added links to new properties added with Patch XWB*1.1*50 in Section Properties—Added or Modified. • Corrected sort order of properties in Table 6. • Added the "RunTime only" icon to the **Socket** property (read-only) and **User** property throughout. • Corrected the **Assign** procedure link in Section 2.2.3.4.	RPC Broker Development Team

Date	Revision	Description	Author
		• Added bullet list of Units described in this document in Section 2.3. • Deleted **TMult** class from the list in Section 2.3.8.1. • Made other minor format and content updates. **RPC Broker 1.1; XWB*1.1*50**	
04/14/2014	1.1	Tech Edits: • Updated the "Definitions" section for units, classes, components, and routines. • Updated the "About this Version of the RPC Broker" section. • Updated the order of the classes in the "Classes—Added" section. • Updated the "Components—Added or Modified" section. • Changed references from "TVCEdit Unit" to "VCEdit Unit" throughout. • Updated the "Properties—Added or Modified" section for properties added with XWB*1.1*50. • Updated Section 2.1.1.7 for reference to Sample directory. • Updated "TContextorControl Component" section. Added Parent class, Unit, and Description sub-sections. • Updated Table 6 with duplicate properties from TCCOWRPCBroker, because they have been made available within the TRPCBroker component. • Updated Section 2.1.3.8 for CCOW methods added to the TRPCBroker Component. • Changed references to correct Sample directory throughout: **BDK32\Samples\BrokerEx** Also, changed references to the BDK32\Samples\SharedRPCBrok	RPC Broker Development Team

Date	Revision	Description	Author
		er directory, since these were not included with XWB*1.1*50. • Updated Section 2.2.1.3. • Updated Section 2.2.1.4: o Changed **Assign** method (**TMult Class**) to **Assign** procedure (**TMult Class**). o Changed **Order** method to **Order** function throughout. o Changed **Position** method to **Position** function throughout. o Changed **Subscript** method to **Subscript** function throughout. • Updated Section 2.2.3.3 title and added the **ParamArray** property. • Updated Section 2.2.3.4 title and **ParamArray** property. • Updated Table 7; added the **NTToken** property. • Updated description in Section 2.2.6.2. • Added Caution note to Section 2.3. • Updated Encryption and Decryption Function links in Section 2.3.2.1. • Added description in Section 2.3.4. • Added description and Caution note in Section 2.3.5. • Added IsIPAddress Function to Section 2.3.5.1. • Added description to Section 2.3.6. • Added the following methods to Section 2.3.6.1: **GetSessionInfo** procedure, **GetUserInfo** procedure, **SilentLogIn** function, **ValidAppHandle** function, **ValidAVCodes** function, and **ValidNTToken** function. • Added description to Section 2.3.7.	

Date	Revision	Description	Author
		- Added "Procedure to library methods in Section 2.3.7.1. - Added the "Wsockc Unit" section. - Removed or modified references to the BDK32\Samples\SilentSignOn directory throughout. - Added the "SecurityPhrase Property" section. - Added the following properties/sections: **SSHHide** Property **SSHport** Property **SSHpw** Property **SSHUser** Property - Added the "UseSecureConnection Property" section. - Corrected sample app name in Section 3.7.8; added ".exe" and deleted the Note. - Added Caution and Note to Section 4.2.3. - As per Keith Cox, head of the ICR team, changed all XWB "public" RPCs to "Controlled Subscription throughout to improve VistA security. Added Notes where appropriate. - Updated Figure 53. - Added Windows 7 Note to Section 4.2.7.1. - Added Caution to Section 4.2.8.1. - Deleted first reference Note in Section 4.2.12.1, since repeated with the Example. - Updated Table 39: Added errors 20008 - 20112. - Updated Figure 74. - Updated example in Step 2 in the "Tutorial—Step 4: Routine to List Terminal Types" section. - Updated Figure 81.	

Date	Revision	Description	Author
		• Updated example in Step 3 in the "Tutorial—Step 8: Routine to Retrieve Terminal Types" section. • Updated Figure 90. • Updated Note in Section 6.16.4.1 and 6.16.4.2. • Added Caution note to Section 6.17. • Updated references to the VB5EGCHO sample application to have been distributed with an earlier BDK. **RPC Broker 1.1; XWB*1.1*50 BDK**	
04/10/2014	1.0	Initial document: • Content derived from the RPC Broker 1.1 online HTML help topics using RoboHelp utility. • Reformatted document and made sure it conforms to the current OIT National Documentations Standards. • Made other minor grammar and punctuation corrections throughout. **RPC Broker 1.1**	RPC Broker Development Team

Patch Revisions

For the current patch history related to this software, see the Patch Module on FORUM.

Table of Contents

List of Figures

List of Tables

Orientation

How to Use this Manual

Throughout this manual, advice and instructions are offered regarding the use of the Remote Procedure Call (RPC) Broker 1.1 Development Kit (BDK) and the functionality it provides for Veterans Health Information Systems and Technology Architecture (VistA).

Intended Audience

The intended audience of this manual is the following stakeholders:

- Enterprise Program Management Office (EPMO)—VistA legacy development teams.

- System Administrators—System administrators at Department of Veterans Affairs (VA) regional and local sites who are responsible for computer management and system security on the VistA M Servers.

- Information Security Officers (ISOs)—Personnel at VA sites responsible for system security.

- Product Support (PS).

Disclaimers

Software Disclaimer

This software was developed at the Department of Veterans Affairs (VA) by employees of the Federal Government in the course of their official duties. Pursuant to title 17 Section 105 of the United States Code this software is *not* subject to copyright protection and is in the public domain. VA assumes no responsibility whatsoever for its use by other parties, and makes no guarantees, expressed or implied, about its quality, reliability, or any other characteristic. We would appreciate acknowledgement if the software is used. This software can be redistributed and/or modified freely provided that any derivative works bear some notice that they are derived from it, and any modified versions bear some notice that they have been modified.

⚠️ **CAUTION: To protect the security of VistA systems, distribution of this software for use on any other computer system by VistA sites is prohibited. All requests for copies of this software for *non*-VistA use should be referred to the VistA site's local Office of Information and Technology Field Office (OITFO).**

Documentation Disclaimer

This manual provides an overall explanation of RPC Broker and the functionality contained in RPC Broker 1.1; however, no attempt is made to explain how the overall VistA programming system is integrated and maintained. Such methods and procedures are documented elsewhere. We suggest you look at the various VA Internet and Intranet Websites for a general orientation to VistA. For example, visit the Office of Information and Technology (OIT) VistA Development Intranet website.

> **DISCLAIMER: The appearance of any external hyperlink references in this manual does *not* constitute endorsement by the Department of Veterans Affairs (VA) of this Website or the information, products, or services contained therein. The VA does *not* exercise any editorial control over the information you find at these locations. Such links are provided and are consistent with the stated purpose of this VA Intranet Service.**

Documentation Conventions

This manual uses several methods to highlight different aspects of the material:

- Various symbols are used throughout the documentation to alert the reader to special information. Table 1 gives a description of each of these symbols:

Table 1: Documentation Symbol Descriptions

Symbol	Description
ℹ️	**NOTE / REF:** Used to inform the reader of general information including references to additional reading material
⚠️	**CAUTION / RECOMMENDATION / DISCLAIMER:** Used to caution the reader to take special notice of critical information

- Descriptive text is presented in a proportional font (as represented by this font).
- Conventions for displaying TEST data in this document are as follows:
  - The first three digits (prefix) of any Social Security Numbers (SSN) begin with either "**000**" or "**666**."
  - Patient and user names are formatted as follows:
    - *[Application Name]*PATIENT,*[N]*
    - *[Application Name]*USER,*[N]*

Where "*[Application Name]*" is defined in the Approved Application Abbreviations document and "*[N]*" represents the first name as a number spelled out and incremented with each new entry.

For example, in RPC Broker (XWB) test patient names would be documented as follows:

XWBPATIENT,ONE; XWBPATIENT,TWO; XWBPATIENT,14, etc.

For example, in RPC Broker (XWB) test user names would be documented as follows:

XWBUSER,ONE; XWBUSER,TWO; XWBUSER,14, etc.

- "Snapshots" of computer online displays (i.e., screen captures/dialogues) and computer source code is shown in a *non*-proportional font and may be enclosed within a box.

- User's responses to online prompts are in **boldface** and highlighted in yellow (e.g., **<Enter>**).

- Emphasis within a dialogue box is in **boldface** and highlighted in blue (e.g., STANDARD LISTENER: RUNNING).

- Some software code reserved/key words are in **boldface** with alternate color font.

- References to "**<Enter>**" within these snapshots indicate that the user should press the <**Enter**> key on the keyboard. Other special keys are represented within < > angle brackets. For example, pressing the **PF1** key can be represented as pressing **<PF1>**.

- Author's comments are displayed in italics or as "callout" boxes.

  NOTE: Callout boxes refer to labels or descriptions usually enclosed within a box, which point to specific areas of a displayed image.

- The following conventions are used with regards to APIs:
  - The following API types are documented:
    - **Supported:**

      This applies where any VistA application may use the attributes/functions defined by the Integration Control Registration (ICR); these are also called "Public". An example is an ICR that describes a standard API. The package that creates/maintains the Supported Reference *must* ensure it is recorded as a Supported Reference in the ICR database. There is no need for other VistA packages to request an ICR to use these references; they are open to all by default.

– **Controlled Subscription:**

Describes attributes/functions that *must* be controlled in their use. The decision to restrict the Integration Control Registration (ICR) is based on the maturity of the custodian package. Typically, these ICRs are created by the requesting package based on their independent examination of the custodian package's features. For the ICR to be approved the custodian grants permission to other VistA packages to use the attributes/functions of the ICR; permission is granted on a one-by-one basis where each is based on a solicitation by the requesting package.

Private APIs are *not* documented.

o Headings for developer API descriptions (e.g., supported for use in applications and on the Database Integration Committee [DBIC] list) include the routine tag (if any), the caret (^) used when calling the routine, and the routine name. The following is an example:

```
$$BROKER^XWBLIB
```

o For APIs that take input parameter, the input parameter is labeled "required" when it is a required input parameter and labeled "optional" when it is an optional input parameter.

o For APIs that take parameters, parameters are shown in lowercase and variables are shown in uppercase. This is to convey that the parameter name is merely a placeholder; M allows you to pass a variable of any name as the parameter or even a string literal (if the parameter is *not* being passed by reference). The following is an example of the formatting for input parameters:

```
XGLMSG^XGLMSG(msg_type,[.]var[,timeout])
```

o Rectangular brackets **[ ]** around a parameter are used to indicate that passing the parameter is optional. Rectangular brackets around a leading period **[.]** in front of a parameter indicate that you can optionally pass that parameter by reference.

o All APIs are categorized by function. This categorization is subjective and subject to change based on feedback from the development community. In addition, some APIs could fall under multiple categories; however, they are only listed once under a chosen category.

APIs within a category are first sorted alphabetically by Routine name and then within routine name are sorted alphabetically by Tag reference. The **$$**, ^, or **^%** prefixes on APIs is ignored when alphabetizing.

• This manual refers to the M programming language. Under the 1995 American National Standards Institute (ANSI) standard, M is the primary name of the MUMPS

programming language, and MUMPS is considered an alternate name. This manual uses the name M.

- All uppercase is reserved for the representation of M code, variable names, or the formal name of options, field/file names, and security keys (e.g., the XUPROGMODE security key).

> **NOTE:** Other software code (e.g., Delphi/Pascal and Java) variable names and file/folder names can be written in lower or mixed case.

Documentation Navigation

This document uses Microsoft® Word's built-in navigation for internal hyperlinks. To add **Back** and **Forward** navigation buttons to your toolbar, do the following:

1. Right-click anywhere on the customizable Toolbar in Word (*not* the Ribbon section).
2. Select **Customize Quick Access Toolbar** from the secondary menu.
3. Select the drop-down arrow in the "Choose commands from:" box.
4. Select **All Commands** from the displayed list.
5. Scroll through the command list in the left column until you see the **Back** command (circle with arrow pointing left).
6. Select/Highlight the **Back** command and select **Add** to add it to your customized toolbar.
7. Scroll through the command list in the left column until you see the **Forward** command (circle with arrow pointing right).
8. Select/Highlight the **Forward** command and select **Add** to add it to the customized toolbar.
9. Select **OK**.

You can now use these **Back** and **Forward** command buttons in the Toolbar to navigate back and forth in the Word document when selecting hyperlinks within the document.

> **NOTE:** This is a one-time setup and is automatically available in any other Word document once you install it on the Toolbar.

Commonly Used Terms

Table 2 is a list of terms and their descriptions that you may find helpful while reading the RPC Broker documentation:

Table 2: Commonly Used RPC Broker Terms

Term	Description
Client	A single term used interchangeably to refer to a user, the workstation (i.e., PC), and the portion of the program that runs on the workstation.
Component	A software object that contains data and code. A component may or may not be visible. **REF:** For a more detailed description, see the *Embarcadero Delphi for Windows User Guide*.
GUI	The Graphical User Interface application that is developed for the client workstation.
Host	The term Host is used interchangeably with the term Server.
Server	The computer where the data and the RPC Broker remote procedure calls (RPCs) reside.

REF: For additional terms and definitions, see the "Glossary" section in this manual and other RPC Broker manuals.

How to Obtain Technical Information Online

Exported VistA M Server-based software file, routine, and global documentation can be generated using Kernel, MailMan, and VA FileMan utilities.

NOTE: Methods of obtaining specific technical information online is indicated where applicable under the appropriate section.

REF: For further information, see the *RPC Broker Technical Manual*.

Help at Prompts

VistA M Server-based software provides online help and commonly used system default prompts. Users are encouraged to enter question marks at any response prompt. At the end of the help display, you are immediately returned to the point from which you started. This is an easy way to learn about any aspect of VistA M Server-based software.

Obtaining Data Dictionary Listings

Technical information about VistA M Server-based files and the fields in files is stored in data dictionaries (DD). You can use the **List File Attributes** [DILIST] option on the **Data Dictionary Utilities** [DI DDU] menu in VA FileMan to print formatted data dictionaries.

REF: For details about obtaining data dictionaries and about the formats available, see the "List File Attributes" chapter in the "File Management" section of the *VA FileMan Advanced User Manual*.

Assumptions

This manual is written with the assumption that the reader is familiar with the following:

- VistA computing environment:
  - Kernel—VistA M Server software
  - Remote Procedure Call (RPC) Broker—VistA Client/Server software
  - VA FileMan data structures and terminology—VistA M Server software

- Microsoft® Windows environment
- M programming language
- Object Pascal programming language.
- Object Pascal programming language/Embarcadero Delphi Integrated Development Environment (IDE)—RPC Broker

References

Readers who wish to learn more about RPC Broker should consult the following:

- *RPC Broker Release Notes*
- *RPC Broker Deployment, Installation, Back-Out, and Rollback Guide (DIBRG)*
- *RPC Broker Systems Management Guide*
- *RPC Broker Technical Manual*
- *RPC Broker User Guide*
- *RPC Broker Developer's Guide* (this manual)—This document provides an overview of development with the RPC Broker.
- RPC Broker VA Intranet website.

  This site provides announcements, additional information (e.g., Frequently Asked

Questions [FAQs], advisories), documentation links, archives of older documentation and software downloads.

VistA documentation is made available online in Microsoft® Word format and in Adobe Acrobat Portable Document Format (PDF). The PDF documents *must* be read using the Adobe Acrobat Reader, which is freely distributed by Adobe Systems Incorporated at: http://www.adobe.com/

VistA documentation can be downloaded from the VA Software Document Library (VDL) Website: http://www.va.gov/vdl/

The RPC Broker documentation is located on the VDL at: https://www.va.gov/vdl/application.asp?appid=23

VistA documentation and software can also be downloaded from the Product Support (PS) Anonymous Directories.

1 Introduction

The RPC Broker is a client/server system within Department of Veterans Affairs (VA) Veterans Health Information Systems and Technology Architecture (VistA) environment. It enables client applications to communicate and exchange data with VistA M Servers.

This manual describes the development features of the RPC Broker. The emphasis is on using the RPC Broker in conjunction with Delphi software. However, the RPC Broker supports other development environments.

The manual provides a complete reference for development with the RPC Broker. For an overview of development with the RPC Broker components, see the *RPC Broker User Guide*.

This manual is intended for the VistA development community and system administrators. A wider audience of technical personnel engaged in operating and maintaining VA software might also find it useful as a reference.

The following topics are discussed in this section:

- Broker Overview
    - o Broker Security Enhancement (BSE) Overview
    - o Broker Call Steps

- Definitions
- About this Version of the RPC Broker
- What's New in the BDK
- Developer Considerations
- Application Considerations

REF: For the latest RPC Broker product information, see the RPC Broker VA Intranet Website.

1.1   Broker Overview

The RPC Broker is a bridge connecting the application front-end on the client workstation (e.g., Delphi-based GUI applications) to the M-based data and business rules on the VistA M Server.

Table 3: Broker Client-side and Server-side Overview

Client Workstation: RPC Broker	VistA M Server: RPC Broker

• Manages the connection to the client workstation.
 	
    i **REF:** For details, see the *RPC Broker Systems Management Guide.*
• The RPC Broker components allow Delphi-based applications to make RPCs to the server.
• The Broker Dynamic Link Library (DLL) provides support for Commercial-Off-The-Shelf (COTS)/HOST client/server software.
	• Manages the connection to the client.
	
	    i **REF:** For details, see the *RPC Broker Systems Management Guide.*
	• Authenticates client workstation.
	• Authenticates user.
	• Manages RPCs from the client, executes the M code, and passes back return values.

The RPC Broker frees GUI developers from the details of the client-server connection and allows them to concentrate executing operations on the VistA M Server.

1.1.1   Broker Security Enhancement (BSE) Overview

Some VistA application users require access to data located at remote sites at which the users:

- Do *not* have assigned Access and Verify codes.
- Have *not* been entered into the NEW PERSON (#200) file by system administrators.
- Want to avoid having multiple Access/Verify code pairs.

Some applications use the Broker Security Enhancement (BSE) to obtain such access. BSE enters the user's information into the NEW PERSON (#200) file as a visitor, but it does *not* require an Access or Verify code for the user at the remote site. This process accomplishes the following:

- Enables RPC Broker applications to access remote VistA M Servers with increased security.

- Ensures correct information for user access to prevent the mistaken identification of an incorrect or *non*-existent user (spoofing) via unauthorized applications.

- Provides the ability for RPC Broker applications that have implemented BSE to specify their own context option.

BSE adds a step to the RPC Broker signon process to authenticate the connecting application. This involves passing a secret encoded phrase that is established on the VistA M Server via a patch and KIDS build. BSE also adds a step to the RPC Broker signon context on the remote VistA M Server to authenticate the user by connecting back to the authenticating VistA M Server to validate a token established during the authentication process and retrieve the user's information from the authenticating server.

1.1.2 Broker Call Steps

These steps present a basic outline of the events that go into an RPC Broker call, starting with the initial client-server connection. Once the client machine and user are authenticated, any number of calls (Steps 3-5) can occur through the open connection.

GUI developer issues are noted for each step.

1. Authentication of client workstation. When a client workstation initiates a session, the Broker Listener on the server spawns a new job. The server then calls the client back to ensure that the client's address is accurate.

   GUI Developer Issues:

   None. This process is built into the RPC Broker.

   REF: For more details, see the *RPC Broker Systems Management Guide* on the VDL at:
   http://www.va.gov/vdl/documents/Infrastructure/Remote_Proc_Call_Broker_(RPC)/xwb_1_1_sm.pdf

2. Authentication of user. After the server connects back to the client workstation, the user is asked for user credentials, either 2-factor authentication (Public Key Infrastructure [PKI] certificate and Personal Identification Number [PIN]) or an Access and Verify code. User authentication and identification is done with calls to VistA Kernel RPCs, including:

   - XUS SIGNON SETUP
   - XUS ESSO VALIDATE
   - XUS AV CODE

   GUI Developer Issues:

   Broker Security Enhancement (BSE)—BSE user authentication and identification on remote VistA M servers is performed by passing a token to the **XUS SIGNON SETUP** RPC, which the server then uses to validate the user's credentials on the authenticating VistA M Server.

   Creating user context—Applications *must* create a context for the user by calling the **XWB CREATE CONTEXT** RPC. This process checks the user's access to individual RPCs associated with the application.

Enabling Silent Login—Developers *must* decide whether to enable Silent Login.

3. Client makes a Remote Procedure Call.

GUI Developer Issues:

Connecting to VistA—Developers creating Delphi GUI applications can use the TRPCBroker Component to connect to VistA. For each transaction, the application *must* set parameters and execute a call. Issues include:

- Determining data types for input and return.

- Determining the kind of call to make.

In addition to the RPC Broker components, other components are available. The VA FileMan Delphi components (FMDC) encapsulate the details of retrieving, validating, and updating VA FileMan data within a Delphi-based application.

i

REF: For more information on the VA FileMan Delphi Components (FMDC), see the FMDC VA Intranet Website.

i

NOTE: In the future, components may become available to encapsulate other VistA functions.

4. RPC execution on server.

GUI Developer Issues:

A Remote Procedure Call (RPC) is a defined call to M code that runs on a VistA M Server.

i

REF: For RPC information, see the "RPC Overview" section.

Issues include:

- Determining the best RPC—The BDK provides some RPC Broker APIs.

  i

  REF: For more information on RPC Broker APIs, see the "Application Programming Interface (API)" section.

- Creating RPCs from scratch—In many cases, an existing M API can be wrapped into an RPC.

  i

  REF: For more information, see the "Creating RPCs" and "RPC Overview" sections.

- Registering RPCs. RPCs *must* be registered on the server, so users of the GUI VistA application have access to them.

REF: For more information on registering RPCs, see the "RPC Security: How to Register an RPC" section.

5. RPC returns information to the client.

GUI Developer Issues:

Handling the return values, including any error messages.

1.2 Definitions

The RPC Broker BDK includes:

- Units
- Classes
- Objects
- Components
- Types
- Methods
- Routines: Functions and Procedures

For each class, object, and component, this manual lists the unit, declaration, properties, methods, and a description of how to use the class, object, or component.

Some types and properties are public, some are private, and some are available only within the function or procedure in which they are defined:

Unit

Interface {specifies that this unit is an interface to a class}

Uses

{list of external units being referenced within this unit}

Type

{Class definition}

Private

{private (available within this unit) variable, type, property, method, function, and procedure definitions}

Public

{published (available to units using this unit) Variable, type, property, method, function, and procedure definitions}

Implementation

{Method, Function, and Procedure programming, which can contain their own Uses, Type, and property definitions within themselves}

1.2.1 Units

A Unit is a Pascal source-code file or program composed of classes, objects, and components containing all of the other elements. The BDK includes a number of units (e.g., **winsockc.pas**). This manual documents some of the units provided, and details what parts of the BDK are declared in each unit.

Sometimes it is helpful to know in which unit a particular item, such as a type or routine, is declared in the BDK. This is because if you use the item in your own code, you may need to include the corresponding unit in your own Pascal unit's Uses clause.

The BDK is *not* really a standalone program, but the units in the BDK are compiled with an application (e.g., Computerized Patient Record System [CPRS]) to make a program. The interfaces to those units are called components (well-defined and published to be used externally). For example, the wsockc unit in the BDK uses (references) other external units (i.e., BDK and Delphi Run Time Library: **AnsiStrings**, **SysUtils**, **WinSock2**, **XWBBut1**, **WinProcs**, **WinTypes**, **Classes**, **Dialogs**, **Forms**, **Controls**, **StdCtrls**, **ClipBrd**, **TRPCB**, **RpcbErr**) to make the functions and procedures in those units available to **wsockc**.

1.2.2 Classes

A class, or class type, defines a structure consisting of fields, methods, and properties.

1.2.3 Objects

An object is a specific instance of that class with associated values.

1.2.4 Components

A component as defined by this document is a self-contained object with a well-defined interface defined by properties, methods, and events that makes it suitable for specialized purposes. Embarcadero Delphi documentation uses a more generic definition of component to refer to the elements within a class.

REF: For a more detailed description, see the *Embarcadero Delphi for Windows User Guide*.

1.2.5　Types

A type defines the possible range of values for a property or a method. A number of types are declared in the BDK, which you may need to make use of in the code. Some types and properties are public, some are private, and some are available only within the function or procedure in which they are defined.

NOTE: For sections describing types in this manual, the unit and declaration for each type, as well as a description of the type is also provided.

1.2.6　Methods

Delphi's definition: "A method uses the same calling conventions as ordinary procedures and functions, except that every method has an additional implicit parameter "Self", which is a reference to the instance or class in which the method is called. For example, clicking on a button invokes a method which changes the properties of the button."

1.2.7　Routines: Functions and Procedures

Procedures and functions, referred to collectively as routines, are self-contained statement blocks that can be called from different locations in a program. Routines can either be functions or procedures. A function returns a value, and a procedure does not.

NOTE: For sections in this manual describing routines, the unit and declaration for each routine is listed, as well as a description of the routine is provided.

NOTE: In Delphi, routine is the generic term. It is *not* the same as a VistA M routine. In M, a routine is the file containing everything else, including functions and procedures. In Delphi, that would be called a Unit.

1.3　About this Version of the RPC Broker

RPC Broker 1.1 provides developers with the capability to develop VistA Client/Server software using the following RPC Broker Delphi components in a 32-bit environment (listed alphabetically):

- TCCOWRPCBroker Component
- TContextorControl Component
- TRPCBroker Component (original component)
- TXWBRichEdit Component
- TXWBSSOiToken Component

REF: For a complete list of patches released with RPC Broker 1.1, see the National Patch Module (NPM) on FORUM.

RPC Broker 1.1 supports IPv4/IPv6 dual-stack network addressing. It also supports 2-factor authentication (2FA) using Identity and Access Management (IAM) Secure Token Service (STS) delegated authentication.

To develop Delphi client VistA applications in a 32-bit environment you *must* have Delphi XE8 or newer. This version of the RPC Broker supports Delphi versions XE8, 10.0, 10.1, 10.2, 10.3, and Delphi/RAD Studio v10.4. This version of the RPC Broker does *not* allow you to develop new applications in older versions of Delphi or in a **16-bit** environment. However, the RPC Broker routines on the VistA M Server continue to support VistA applications previously developed in the 16-bit environment.

NOTE: Applications developed using previous versions of the RPC Broker Development Kit (BDK) can be adapted to use Delphi XE8 or newer by renaming references to the "Hash" Unit to "XWBHash" to resolve a conflict with a new "Hash" Unit provided in Delphi XE8. Current versions of the BDK use the renamed XWBHash Unit.

The default installation of the RPC Broker creates a separate Broker Development Kit (BDK) directory (i.e., BDK32) that contains the required RPC Broker files for development.

CAUTION: This statement defines the extent of support relative to use of Delphi. The Office of Information and Technology (OIT) only supports the Broker Development Kit (BDK) running in the currently offered version of Delphi and the immediately previous version of Delphi. This level of support became effective 06/12/2000.

Sites can continue to use outdated versions of the RPC Broker Development Kit but do so with the understanding that support is *not* available and that continued use of outdated versions do *not* afford features that can be essential to effective client/server operations in the VistA environment. An archive of old (no longer supported) Broker Development Kits is maintained in the VA Intranet Broker Archive.

1.4 What's New in the BDK

This section highlights the major changes made to the Broker Development Kit (BDK) 1.1. Changes are listed by BDK patch release in reverse order (latest to earliest):

1.4.1 XWB*1.1*73

As of BDK Patch XWB*1.1*73, the following changes were made to RPC Broker 1.1:

Functionality Added or Modified:

- Bug Fixes and Enhancements:

    o When a user assigned more than one division in VistA selects the division in which they will be working, the **Brokerx.User.Division** field is populated with the selected division. However, the division is only identified by its number; the correct return should be **IEN^SITENAME^SITEID**. This patch ensures the data placed into the **Brokerx.User.Division** field is correctly formatted.

    o Users are reporting that the default certificate for connection to VistA is *not* correct. Patches XWB*1.1*71 and XWB*1.1*72 addressed certificate processing; however, the user would have to ensure they were selecting the correct certificate from the list presented. This causes confusion if the list presented does *not* default to the certificate used for authenticating to VistA. This patch redesigned the method of certificate processing; it automatically selects the user's Authentication certificate, eliminating the need for the user to select from a list of certificates.

    o The ShowCertDialog Property (Published) was added to the **TRPCBroker** component.

- Support for Delphi Versions—BDK supports Delphi XE8, 10, 10.1, 10.2, 10.3, and Delphi/RAD Studio v10.4.

1.4.2   XWB*1.1*72

As of BDK Patch XWB*1.1*72, the following changes were made to RPC Broker 1.1:

Functionality Added or Modified:

- Bug Fixes and Enhancements:

    o A user having only one division set in the NEW PERSON (#200) file was causing the site's DEFAULT INSITUTION entry to be set in the DIVISION field of the **RPCBrokerx.User** class, *not* the entry from the NEW PERSON (#200) file. This patch ensures the DIVISION field is properly set.

    o Before this patch, there were many compiler Hints and Warnings, as well as some memory leaks. The Hints and Warnings have been addressed along with many of the memory leaks.

- Support for Delphi Versions—BDK supports Delphi XE8, 10, 10.1, 10.2, 10.3, and Delphi/RAD Studio v10.4.

1.4.3   XWB*1.1*71

As of BDK Patch XWB*1.1*71, the following changes were made to RPC Broker 1.1:

Functionality Added or Modified:

- Bug Fixes and Enhancements:

    o **Patient Safety Issue (HITPS-2387)**—Mental Health providers are unable to renew medications in the Computerized Patient Record System (CPRS) when a patient has more than one medication used by Mental Health providers. The issue arises when more than one medication is being renewed. The data received from First DataBank for drug<->drug interactions exceeds the allowable width of a string in the LPack function in the **wsockc.pas** code of the BDK. This patch widens the width from **999** characters to **99999** characters.

    o **Broker Security Enhancement (BSE) Issue**—Applications compiled with the BDK provided with XWB*1.1*65 could no longer connect to a remote VistA instance using the BSE without the user having to enter his/her Access/Verify codes at the remote site. This patch corrects this issue.

    o **Hidden Dialogue**—There was a hidden dialogue when a user entered incorrect Access/Verify codes; this only occurred after the first dialogue was presented. The first dialogue was visible to the user, but subsequent dialogues for invalid Access/Verify codes were hidden behind the application window. This patch remedies this issue by ensuring the dialogue is always in front of the main application.

    o **Inability to Restart Unattended Applications**—There was an inability to restart an unattended application, like the VistA Imaging Background Processor. If an unattended application was suddenly stopped by a VistA error, the application's context with VistA was removed; this prevented the application from reconnecting to

VistA in an unattended fashion. This patch corrects this issue by preserving the context option that was initially created by the execution of the application.

- o **Token Issues**—The way in which Identity and Access Management (IAM) security (Security Assertion Markup Language [SAML]) tokens were being retrieved was altering the way the token was presented to VistA, the original token structure was not sound. This patch changes the way the token is requested and how it is received from the IAM sever.

- o **Active Directory (AD) Credentials**—When a user is unable to log onto a workstation with their Personal Identity Verification (PIV) card, the user contacts the Enterprise Service Desk (ESD) to receive a PIV exemption to allow them to log on with their Active Directory (AD) credentials (username and password). This version of the BDK was enhanced to detect this condition and allow the user to use their AD credentials to secure a SAML token from IAM for logging onto VistA via applications compiled with this version of the BDK.

- o **Section 508 Issues**—When applications connecting to VistA were presented with the security banner, the text of the banner was *not* accessible to screen reader software, such as JAWS. This was caused by a **Tab Stop** *not* being set on the component that contains the banner text. This patch adds that **Tab Stop** and the text is readable by the screen reader software.

- Support for Delphi Versions—BDK supports Delphi XE8, 10, 10.1, 10.2, and 10.3.

1.4.4   XWB*1.1*65

As of BDK Patch XWB*1.1*65, the following changes were made to RPC Broker 1.1:

Functionality Added or Modified:

- Support for 2-Factor Authentication (2FA)—The [TRPCBroker Component](#) provides Windows client support for 2-factor authentication using an IAM STS token. The user is authenticated into IAM with smart-card credentials:

  - o Public Key Infrastructure (PKI) Certificate

    and

  - o Personal Identification Number (PIN)

  The credentials are exchanged for a digitally signed token that is forwarded to VistA to authenticate and identify the user.

- Support for Delphi Versions—BDK supports Delphi 10.2, 10.1, 10.0, and XE8, XE7, XE6, XE5, and XE4.

Components Added or Modified:

- TXWBSSOiToken Component—Added this component to explicitly obtain an Identity and Access Management (IAM) Secure Token Service (STS) token for 2-factor user authentication and identification. It is used by the **BAPI32.DLL** to make the STS token available to *non*-RPC Broker applications. It is *not* needed for TRPCBroker or TCCOWRPCBroker applications, as the STS token is obtained and consumed internally for user authentication and identification.

- TRPCBroker Component—Modified to support 2-factor authentication by obtaining and using an IAM STS token.

Library Methods Modified:

Silent Login Function—Used to authenticate into a VistA server without user interaction. It was modified to accept a new **lmSSOi** login mode. A silent login is used to authenticate into VistA with STS token credentials obtained from an earlier 2-factor authentication into IAM.

Properties Added or Modified (listed by component/class):

- TXWBSSOiToken Component Properties:
  - SSOiADUPN Property (TXWBSSOiToken Component) (Published)
  - SSOiLogonName Property (TXWBSSOiToken Component) (Published)
  - SSOiSECID Property (TXWBSSOiToken Component) (Published)
  - SSOiToken Property (TXWBSSOiToken Component) (Published)

- TRPCBroker Component Properties:
  - SSOiADUPN Property (TRPCBroker Component) (Public)
  - SSOiLogonName Property (TRPCBroker Component) (Public)
  - SSOiSECID (TRPCBroker Component) (Public)
  - SSOiToken Property (TRPCBroker Component) (Public)
  - Connected Property (Published)

Types Modified:

TLoginMode Type

1.4.5 XWB*1.1*60

As of BDK Patch XWB*1.1*60, the following changes were made to RPC Broker 1.1:

Functionality Added or Modified:

- Support for IPv4/IPv6 Dual-Stack—The TRPCBroker Component provides Windows client support for IPv4/IPv6 dual-stack environment using new Application Programming Interfaces (APIs). This patch also upgrades from WinSock 1.1 to WinSock 2.2. By using this BDK for development of a Delphi client application, the application will be protocol independent and will be able to connect to both IPv4 and IPv6 VistA servers.

- Support for Delphi Versions—BDK supports Delphi XE7, XE6, XE5, and XE4.

Components Deprecated (Removed):

- **TSharedBroker** component—Deprecated and removed the **TSharedBroker** component from RPC Broker 1.1. This component allowed applications to share a single Broker connection.

- **TSharedRPCBroker** component—Deprecated and removed the **TSharedRPCBroker** component from RPC Broker 1.1. This component allowed applications to share a single Broker connection.

CAUTION: The Shared Broker components have been deprecated and removed with RPC Broker Patch XWB*1.1*60. They were *not* widely used and had proven to be difficult to maintain. They will *not* be updated to support IPv6 functionality or 2-factor authentication. Application developers are encouraged to migrate their applications to the standard TRPCBroker component when adding support for IPv6 and 2-factor authentication.

Components Modified:

TRPCBroker Component:

- Modified to upgrade Windows Sockets (WinSock) from Version 1.1 to Version 2.2.

- Modified to support IPv4/IPv6 dual-stack addressing for connection to a VistA server. Applications compiled with this BDK will be protocol independent and will be able to connect to both IPv4 and IPv6 VistA servers.

- Modified to support lookup to the Windows Registry for Secure Shell (SSH) configuration for connection to a VistA server.

- Deprecated and removed the old-style Broker (where VistA calls back to a different port on the client workstation when making a connection). Applications compiled with this BDK will use the new-style Broker (where VistA calls back to the originating port on the client workstation).

- Deprecated and removed the old-style Broker (where VistA calls back to a different port on the client workstation when making a connection). Applications compiled with this BDK will use the new-style Broker (where VistA calls back to the originating port on the client workstation).

Library Methods Modified:

GetServerInfo Function—Used to select the desired Server name and ListenerPort (see ListenerPort Property). Added a new **SSH Username** field when adding a new Server/ListenerPort combination. This field can be used to identify the Attachmate® Reflection/Micro Focus® SSH Username for SSH connection to the specified server.

Properties Deprecated (Removed; listed by component/class):

- TRPCBroker Component Properties:

  The following TRPCBroker Component properties were deprecated, as the old-style Broker connection is no longer supported:

  o **IsBackwardCompatibleConnection** property (Deprecated)

  o **IsNewStyleConnection** property (read-only) (Deprecated)

  o **OldConnectionOnly** property (Deprecated)

- **TSharedBroker** component and **TSharedRPCBroker** component properties:

  The following **TSharedBroker** component and **TSharedRPCBroker** component properties were deprecated, as the Shared Broker itself has been deprecated:

  o **OnConnectionDropped** property (Deprecated)

  o **OnLogout** property (Deprecated)

1.4.6 XWB*1.1*50

As of BDK Patch XWB*1.1*50, the following changes were made to RPC Broker 1.1:

Functionality Added or Modified:

- Support for Secure Shell (SSH) Tunneling—The TRPCBroker Component enabled Secure Shell (SSH) Tunnels to be used for secure connections. This functionality is controlled by setting an internal property value (mandatory SSH) or command line option at run time. Support is provided for the Attachmate® Reflection/Micro Focus® terminal emulator software using SSH tunneling for clients within the VA, and support is provided for PuTTY Link (Plink) for secure channels for clients outside the VA.

- Support for Broker Security Enhancement (BSE)—The TRPCBroker Component enabled visitor access to remote sites using authentication established at a home site.

- Support for Delphi Versions—BDK supports Delphi XE5, XE4, XE3, and XE2.

Components Added or Modified:

TRPCBroker Component:

- Modified to include support for Secure Shell (SSH) tunneling using Attachmate® Reflection/Micro Focus® Reflection SSH or PuTTY Link (Plink).

- Modified to include support for Broker Security Enhancement (BSE) modifications introduced in patch XWB*1.1*45.

- Modified by wrapping CCOW User Context into the primary TRPCBroker Component so that if the Contextor Property is set, then CCOW User Context is used.

  NOTE: All of the CCOW functionality used by and for the TCCOWRPCBroker Component is still present, but it is now part of the regular TRPCBroker Component.

  This means that there is no longer a need to have the separate TCCOWRPCBroker Component. The TCCOWRPCBroker Component is included separately in XWB*1.1*50 for backward compatibility.

Properties Added or Modified to the TCCOWRPCBroker Component:

- SecurityPhrase Property (Published)

- SSHHide Property (Published)

- SSHport Property (Public)

- SSHpw Property (Public)

- SSHUser Property (Public)

- UseSecureConnection Property (Published)

1.4.7   XWB*1.1*40

As of BDK Patch XWB*1.1*40, the following changes were made to RPC Broker 1.1:

Functionality Added or Modified:

Supports Single Sign-On/User Context (SSO/UC)—As of RPC Broker Patch XWB*1.1*40, the TCCOWRPCBroker Component enabled Single Sign-On/User Context (SSO/UC) in CCOW-enabled applications.

  REF: For more information on SSO/UC, see the Single Sign-On/User Context (SSO/UC) Installation Guide and Single Sign-On/User Context (SSO/UC) Deployment Guide on the VA Software Document Library (VDL).

Class Added:

[TXWBWinsock Class](#)

Components Added or Modified:

- [TCCOWRPCBroker Component](#)—Allows applications to be CCOW-enabled and Single Sign-On/User Context (SSO/UC)-aware.

- [TContextorControl Component](#)—Communicates with the Vergence Locator service.

Library Methods Added to the [TCCOWRPCBroker Component](#):

- [GetCCOWtoken Method](#)

- [IsUserCleared Method](#)

- [IsUserContextPending Method](#)

- [WasUserDefined Method](#)

Properties Added or Modified (listed by component/class):

- [TCCOWRPCBroker Component](#) Properties:
  - [CCOWLogonIDName Property (read-only)](#) (Public)
  - [CCOWLogonIDValue Property (read-only)](#) (Public)
  - [CCOWLogonName Property (read-only)](#) (Public)
  - [CCOWLogonNameValue Property (read-only)](#) (Public)
  - [CCOWLogonVpid Property (read-only)](#) (Public)
  - [CCOWLogonVpidValue Property (read-only)](#) (Public)
  - [Contextor Property](#) (Public)

- [TVistaLogin Class](#) Properties:
  - [DomainName Property](#) (Public)
  - [IsProductionAccount Property](#) (Public)

- [TVistaUser Class](#) Property:
    [Vpid Property](#) (Public)

Types Added or Modified:

- [TLoginMode Type](#)
- **TShowErorMsgs** (see [ShowErrorMsgs Property](#))
- **TOnLoginFailure** (see [OnFailedLogin Property](#))
- **TOnRPCBFailure** (see [OnRPCBFailure Property](#))
- [TParamType](#)

1.4.8 XWB*1.1*35

As of BDK Patch XWB*1.1*35, the following changes were made to RPC Broker 1.1:

Functionality Added or Modified:

Supports Non-Callback Connections—The RPC Broker components are built with a UCX or *non*-callback Broker connection, so that it can be used from behind firewalls, routers, etc.

Properties Added or Modified in the [TRPCBroker Component](#):

- [BrokerVersion Property (read-only)](#) (Public)
- [CurrentContext Property (read-only)](#) (Public)
- **IsBackwardCompatibleConnection** property (Published; deprecated with [XWB*1.1*60](#))
- **IsNewStyleConnection** property (read-only) (Public; deprecated with [XWB*1.1*60](#))
- [KernelLogIn Property](#) (Published)
- [LogIn Property](#) (Public)
- **OldConnectionOnly** property (Published; deprecated with [XWB*1.1*60](#))
- [OnRPCBFailure Property](#) (Public)
- [RPCBError Property (read-only)](#) (Public)
- [ShowErrorMsgs Property](#) (Published)
- [User Property](#) (Public)

1.4.9   XWB*1.1*26

As of BDK Patch XWB*1.1*26, the following changes were made to RPC Broker 1.1:

Components Added or Modified:

- **TSharedBroker** component (component deprecated with XWB*1.1*60)—Added the **TSharedBroker** component to RPC Broker 1.1. This component allows applications to share a single Broker connection.

- **TSharedRPCBroker** component (component deprecated with XWB*1.1*60)—Added the **TSharedRPCBroker** component to RPC Broker 1.1. This component allows applications to share a single Broker connection.

1.4.10 XWB*1.1*23

As of BDK Patch XWB*1.1*23, the following changes were made to RPC Broker 1.1:

Properties Added or Modified (listed by component/class):

TSharedBroker component and **TSharedRPCBroker** component properties (components deprecated with XWB*1.1*60):

- **AllowShared** property (Public; deprecated with XWB*1.1*60)

- **OnConnectionDropped** property (Public; deprecated with XWB*1.1*60)

- **OnLogout** property (Published; deprecated with XWB*1.1*60)

1.4.11 XWB*1.1*14

As of BDK Patch XWB*1.1*14, the following changes were made to RPC Broker 1.1:

- Separate **RunTime** and **DesignTime** Packages:

  REF: For details and compiling instructions, see the "DesignTime and RunTime Packages" section in the "Developer Considerations" section.

- Broker Source Code Released:

  The source code is located in the following directory:

    BDK32\Source

  CAUTION: Modified BDK source code should *not* be used to create VistA GUI applications. For more details, see the "Developer Considerations" section.

  Not all methods and properties found in the source code are documented

at this time. Only those documented methods and properties are guaranteed to be made backwards compatible in future versions of the BDK.

1.4.12 XWB*1.1*13

As of BDK Patch XWB*1.1*13, the following changes were made to RPC Broker 1.1:

Functionality Added or Modified:

- Supports Silent Login—Provides functionality associated with the ability to make logins to a VistA M Server without the RPC Broker asking for Access and Verify code information.

- Documented Deferred RPCs and Capability to Run RPCs on a Remote Server:

  o Running RPCs on a Remote Server

  o Deferred RPCs

- Multi-instances of the RPC Broker—RPC Broker code was modified to permit an application to open two separate Broker instances with the same **Server**/**ListenerPort** (see Server Property and ListenerPort Property) combination, resulting in two separate partitions on the server. Previously, an attempt to open a second Broker instance ended up using the same partition. For this capability to be useful for concurrent processing, an application would have to use threads to handle the separate Broker sessions.

  CAUTION: Although there should be no problems, the RPC Broker is *not guaranteed to be thread safe*.

- Operates in a 32-bit Microsoft® Windows environment.

Classes Added:

- TVistaLogin Class
- TVistaUser Class

Component Added or Modified:

TXWBRichEdit Component—This component replaced the Introductory Text Memo component on the Login Form. It permits URLs to be identified and launched.

Library Methods Added to VCEdit Unit:

- ChangeVerify Function
- SilentChangeVerify Function
- StartProgSLogin Method

Library Methods Modified:

- CheckCmdLine Function—Changed from procedure to function with a Boolean return value.

- GetServerInfo Function—Used to select the desired Server name and ListenerPort (see ListenerPort Property). It was modified to allow users to add a new Server/ListenerPort combination to those available for selection. It also accepts and stores a valid IP address, if no name is known for the location. This permits those who have access to other Server/ListenerPort combinations that may not be available in the list on the current workstation to access them. However, they still need a valid Access and Verify code to log on to the added location.

- TParams Class—**Clear** procedure was moved from Private to Public.

- TRPCB Unit:
  - **TOnLoginFailure**: Changed from Object: **TObject**, since this is what should be expected by the procedure if it is called.
  - **TOnRPCBFailure**: Changed from Object: **TObject**, since this is what should be expected by the procedure if it is called.

Properties Added or Modified in TRPCBroker Component:

- BrokerVersion Property (read-only) (Public)
- CurrentContext Property (read-only) (Public)
- **IsBackwardCompatibleConnection** property (Published; deprecated with XWB*1.1*60)
- **IsNewStyleConnection** property (read-only) (Public; deprecated with XWB*1.1*60)
- KernelLogIn Property (Published)
- LogIn Property (Public)
- **OldConnectionOnly** property (Published; deprecated with XWB*1.1*60)
- OnRPCBFailure Property (Public)
- RPCBError Property (read-only) (Public)
- ShowErrorMsgs Property (Published)
- User Property (Public)

Types Added or Modified:

- [TLoginMode Type](#)
- **TShowErorMsgs** (see [ShowErrorMsgs Property](#))
- **TOnLoginFailure** (see [OnFailedLogin Property](#))
- **TOnRPCBFailure** (see [OnRPCBFailure Property](#))
- [TParamType](#)

1.5 Developer Considerations

1.5.1 Source Code

As of RPC Broker Patch XWB*1.1*14, the RPC Broker source code was released. The release of the source code does *not* affect how a developer uses the Broker Components or other parts of the BDK.

⚠️ **CAUTION: Modified BDK source code should *not* be used to create VistA GUI applications.**

Suggestions for changes, bugs, and enhancements to the BDK should be done via the Service Desk Manager (SDM) support system for review and possible inclusion in a future patch.

The source code is located in the following directory:

BDK32\Source

1.5.2 DesignTime and RunTime Packages

As of RPC Broker Patch XWB*1.1*14, the BDK has separate **RunTime** and **DesignTime** packages. There is no longer a VistA Broker package. The new packages are:

- **XWB_DXE***n*
- **XWB_R***XEn*

Where:

- "**D**"—**DesignTime**
- "**R**"—**RunTime**
- "***XEn***"—Delphi version with which it should be used

For example, XWB_D*XE8* is the **DesignTime** package for Delphi Version *XE8*.

Delphi 10.*n* uses the **XWB_RunTime** and **XWB_DesignTime** packages. The **RunTime** package should *not* be used to create executables that depend on a separate XWB_R*XEn*.bpl installed on client workstations. There is no procedure in place at this time to reliably install the correct version of the **RunTime bpl** on client workstations.

CAUTION: Do *not* compile your project so that it relies on dynamic linking with the BDK's RunTime package; that is, do *not* check the "Build with runtime packages" box on the "Packages" tab of the "Project Options' dialogue.

1.5.3   Resource Reuse

Developers should be aware of existing resources that may be of use. These resources may be available nationally or through a particular project. Possibilities include:

- Delphi components, such as the VA FileMan Delphi components (FMDC).

    REF: For more information on the VA FileMan Delphi components (FMDC), see the FMDC VA Intranet website.

- [RPC Broker: Developer Tools](#)
- [Using an Existing M API](#)

1.5.4   Component Connect-Disconnect Behavior

1.5.4.1   Connect

The first time one of the Broker components in your application connects, it establishes an actual connection with the server. The connection record is added to the list of all active connections for your application. This list is internal to the application and is completely under the control of the Broker component and is transparent to you. If another Broker component tries to connect to the same server/port, the existing connection record is found in the list and its socket is shared. The new connection is also added to this list. This process is repeated with each connection request.

1.5.4.2   Disconnect

When a Broker component disconnects, its connection record is removed from the internal list of active connections. If it happens to be the last record for the particular server/port combination, the connection is actually closed. This scheme provides the illusion of multiple connections without "clogging up" the server.

1.6　Application Considerations

1.6.1　Application Version Numbers

There may be a need to set or pass application version numbers. The suggested format is as follows:

VersionNumber_PatchNumber (3 digits)

For example, Patch 22 of Version 8.2 would be formatted as follows:

8.2_022

1.6.2　Deferred RPCs

In order to increase efficiency, applications can run RPCs in the background.

REF: For more information on Deferred RPCs, see the "Deferred RPCs" section.

1.6.3　Remote RPCs

In order to work with patient data across sites, applications can run RPCs on a remote server.

REF: For more information on running RPCs on a remote server, see the "Running RPCs on a Remote Server" section.

1.6.4　Blocking RPCs

Applications can install RPCs that should be used only in certain contexts. It is possible to block access to an RPC.

REF: For more information on blocking access to an RPC, see the "Blocking an RPC" section.

1.6.5　Silent Login

In special cases, applications can use one of three types of Silent Login to log in users *without* the RPC Broker prompting for login information.

2 RPC Broker Components, Classes, Units, Methods, Types, and Properties

2.1 Components

2.1.1 TCCOWRPCBroker Component

- Properties (All)

- Methods

- Example

2.1.1.1 Parent Class

TRPCBroker = class(TComponent)

2.1.1.2 Unit

CCOWRPCBroker.pas

2.1.1.3 Description

The **TCCOWRPCBroker** component (**CCOWRPCBroker.pas**) is derived from the existing TRPCBroker Component. The **TCCOWRPCBroker** component (**Trpcb.pas**) allows VistA application developers to make their applications CCOW-enabled and Single Sign-On/User Context (SSO/UC)-aware with all of the client/server-related functionality in one integrated component. Using the **TCCOWRPCBroker** component, an application can share User Context stored in the CCOW Context Vault.

When a VistA CCOW-enabled application is recompiled with the **TCCOWRPCBroker** component and other required code modifications are made, that application becomes SSO/UC-aware and capable of CCOW single sign-on (SSO).

REF: For more detailed information on the application developer procedures and code modifications needed to make CCOW-enabled RPC Broker-based applications SSO/UC aware, see the "RPC Broker-based Client/Server Applications" section in the "Making VistA Applications SSO/UC-aware" chapter in the *Single Sign-On User Context (SSO/UC) Deployment Guide.*

NOTE: Properties inherited from the parent component (i.e., **TComponent**) are *not* discussed in this manual (only those properties added to the parent component are described). For help on inherited properties, see Delphi's documentation on the parent component (i.e., **TComponent**).

REF: For help on inherited properties, see the parent component (i.e., TRPCBroker Component).

2.1.1.4 Properties (All)

- Properties (Unique)

Table 4 lists all properties available with the TCCOWRPCBroker Component (includes those properties inherited from the parent TRPCBroker Component):

Table 4: TCCOWRPCBroker Component—All Properties (listed alphabetically)

Read-only	RunTime only	Property
▶	▶	BrokerVersion Property (read-only)
▶	▶	CCOWLogonIDName Property (read-only)
▶	▶	CCOWLogonIDValue Property (read-only)
▶	▶	CCOWLogonName Property (read-only)
▶	▶	CCOWLogonNameValue Property (read-only)
▶	▶	CCOWLogonVpid Property (read-only)
▶	▶	CCOWLogonVpidValue Property (read-only)
		ClearParameters Property
		ClearResults Property
		Connected Property
▶		Contextor Property
▶	▶	CurrentContext Property (read-only)
		DebugMode Property
		KernelLogIn Property
		ListenerPort Property
	▶	LogIn Property
	▶	OnRPCBFailure Property
	▶	Param Property
		RemoteProcedure Property
		Results Property

Read-only	RunTime only	Property
▶	▶	RPCBError Property (read-only)
	▶	RPCTimeLimit Property
		RPCVersion Property
	▶	SecurityPhrase Property
		Server Property
		ShowCertDialog Property
		ShowErrorMsgs Property
▶	▶	Socket Property (read-only)
		SSHHide Property
	▶	SSHport Property
	▶	SSHpw Property
	▶	SSHUser Property
	▶	SSOiToken Property (TRPCBroker Component)
	▶	SSOiSECID (TRPCBroker Component)
	▶	SSOiADUPN Property (TRPCBroker Component)
	▶	SSOiLogonName Property (TRPCBroker Component)
▶	▶	User Property
	▶	UseSecureConnection Property

2.1.1.5   Properties (Unique)

- [Properties (All)](#)

[Table 5](#) lists the unique properties available with the [TCCOWRPCBroker Component](#):

Table 5: TCCOWRPCBroker Component—Unique Properties (listed alphabetically)

Read-only	RunTime only	Property
▶	▶	[CCOWLogonIDName Property (read-only)](#)
▶	▶	[CCOWLogonIDValue Property (read-only)](#)
▶	▶	[CCOWLogonName Property (read-only)](#)
▶	▶	[CCOWLogonNameValue Property (read-only)](#)
▶	▶	[CCOWLogonVpid Property (read-only)](#)
▶	▶	[CCOWLogonVpidValue Property (read-only)](#)
	▶	[Contextor Property](#)

NOTE: Since the [TCCOWRPCBroker Component](#) is a class derived from the [TRPCBroker Component](#), it contains all of the [Properties (All)](#), [Methods](#), etc., of its parent.

2.1.1.6   Methods

- [GetCCOWtoken Method](#)
- [IsUserCleared Method](#)
- [IsUserContextPending Method](#)
- [WasUserDefined Method](#)

2.1.1.7   Example

For examples, see the **Samples** directory on the use of the [TCCOWRPCBroker Component](#); located in the following directory:

BDK32\Samples\BrokerEx

2.1.2 TContextorControl Component

As of RPC Broker Patch XWB*1.1*40, the **TContextorControl** component was added to RPC Broker 1.1.

2.1.2.1 Parent Class

TRPCBroker = class(TOleServer)

2.1.2.2 Unit

[TRPCB Unit](#)

2.1.2.3 Description

The **TContextorControl** component provides Delphi developers with access to the CCOW Vergence Locator service.

2.1.3 TRPCBroker Component

- [Properties (All)](#)
- [Methods](#)
- [Example](#)

2.1.3.1 Parent Class

TRPCBroker = class(TComponent)

2.1.3.2 Unit

[TRPCB Unit](#)

2.1.3.3 Description

The **TRPCBroker** component provides Delphi developers with an easy, object-based access to the Broker. It is compatible with the Delphi object oriented (OO) environment. This component, when placed on a Delphi form, allows applications to connect to the VistA M Server and reference M data within Delphi's Integrated Development Environment (IDE). It makes a Delphi form and everything on it "data aware."

The **TRPCBroker** component (**Trpcb.pas**) provides VistA application developers with all of the client/server-related functionality in one integrated component. Using the **TRPCBroker** component, an application can connect to the VistA M Server by simply setting the Connected Property to **True**. Remote procedures on the server can be executed by preparing the Param Property and RemoteProcedure Property and invoking any of the following methods:

- Call Method
- strCall Method
- lstCall Method

The **TRPCBroker** component can be found on the **Kernel** tab in the component palette.

> **NOTE:** Properties inherited from the parent component (i.e., **TComponent**) are *not* discussed in this manual (only those properties added to the parent component are described). For help on inherited properties, see Delphi's documentation on the parent component (i.e., **TComponent**).

2.1.3.4 Support for Secure Shell (SSH) Tunneling

As of RPC Broker Patch XWB*1.1*50 support was added for a Secure Shell (SSH) tunneling service to provide secure data transfer between the client and the VistA M Server.

The Attachmate® Reflection/Micro Focus® Reflection terminal emulator software with SSH tunneling is used inside the VA to provide secure data transfer between the client and the VistA M Server. SSH tunneling is also supported for PuTTY Link (Plink) for those using VistA outside of the VA.

For SSH tunneling using Reflection, either set a command line option or a property within the application. SSH is enabled if the UseSecureConnection Property is set to "**secureAttachmate**". SSH is also enabled if either of the following command line parameters are set:

- SSHPort=portnumber (to specify a particular port number—If *not* specified, it uses the port number for the remote server).
- SSHUser=username (for the remote server, where username is of the form *xxxvista*, where the *xxx* is the station's three letter abbreviation).

For SSH tunneling using **Plink.exe**, either set a command line option or a property within the application. SSH is enabled if the **UseSecureConnection** property is set to "**securePlink**". SSH is also enabled if the following command line parameter is set:

```
SSHpw=password
```

2.1.3.5   Support for Broker Security Enhancement (BSE)

As of RPC Broker Patch XWB*1.1*45, the RPC Broker supports the Broker Security Enhancement (BSE). The **TRPCBroker** component was modified to enable visitor access to remote sites using authentication established at a home site.

2.1.3.6   CCOW User Context Wrapped into the Primary TRPCBroker Component

As of RPC Broker Patch XWB*1.1*50, the RPC Broker wraps CCOW User Context into the primary **TRPCBroker** component so that if the Contextor Property is set, then CCOW User Context is used. This means that there is no longer a need to have the separate TCCOWRPCBroker Component.

> **NOTE:** All of the functionality used by and for the TCCOWRPCBroker Component is still present, but it is now part of the regular **TRPCBroker** component.

2.1.3.7   Properties (All)

Table 6 lists all of the properties available with the TRPCBroker Component:

Table 6: TRPCBroker Component—All Properties (listed alphabetically)

Read-only	RunTime only	Property
▶	▶	BrokerVersion Property (read-only)
▶	▶	CCOWLogonIDName Property (read-only)
▶	▶	CCOWLogonIDValue Property (read-only)
▶	▶	CCOWLogonName Property (read-only)
▶	▶	CCOWLogonNameValue Property (read-only)
▶	▶	CCOWLogonVpid Property (read-only)
▶	▶	CCOWLogonVpidValue Property (read-only)
		ClearParameters Property
		ClearResults Property
		Connected Property
	▶	Contextor Property
▶	▶	CurrentContext Property (read-only)
		DebugMode Property
		KernelLogIn Property
		ListenerPort Property
	▶	LogIn Property

Read-only	RunTime only	Property
	▷	OnRPCBFailure Property
	▷	Param Property
		RemoteProcedure Property
		Results Property
▷	▷	RPCBError Property (read-only)
	▷	RPCTimeLimit Property
		RPCVersion Property
	▷	SecurityPhrase Property
		Server Property
		ShowErrorMsgs Property
▷	▷	Socket Property (read-only)
		SSHHide Property
	▷	SSHport Property
	▷	SSHpw Property
	▷	SSHUser Property
	▷	SSOiToken Property (TRPCBroker Component)
	▷	SSOiSECID (TRPCBroker Component)
	▷	SSOiADUPN Property (TRPCBroker Component)
	▷	SSOiLogonName Property (TRPCBroker Component)
▷	▷	User Property
	▷	UseSecureConnection Property

2.1.3.8 Methods

- Call Method
- CreateContext Method
- GetCCOWtoken Method
- IsUserCleared Method
- IsUserContextPending Method
- lstCall Method
- pchCall Method
- strCall Method

- [WasUserDefined Method](#)

2.1.3.9   Example

The following example demonstrates how a [TRPCBroker Component](#) can be used to:

1. Connect to the VistA M Server.
2. Execute various remote procedures.
3. Return the results.
4. Disconnect from the server.

The example in [Figure 2](#) assumes that a [TRPCBroker Component](#) already exists on the form as **brkrRPCBroker1**:

Figure 1: TRPCBroker Component—Example

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   try
      {connect to the server}
      brkrRPCBroker1.Connected := True;
      //assign RPC name
      brkrRPCBroker1.RemoteProcedure := 'SOME APPLICATION RPC';
      {make the call}
      brkrRPCBroker1.Call;
      {display results}
      ListBox1.Items := brkrRPCBroker1.Results;
      {disconnect from the server}
      brkrRPCBroker1.Connected := False;
   except
      //put error handling code here
   end;
end;
```

REF: For more examples, see the **Samples** directory on the use of the [TRPCBroker Component](#); located in the following directory:

BDK32\Samples\BrokerEx

2.1.4   TXWBRichEdit Component

[Property](#)

2.1.4.1   Parent Class

TXWBRichEdit = class(TComponent)

2.1.4.2   Unit

XwbRich20

2.1.4.3   Description

The **TXWBRichEdit** component replaces the Introductory Text Memo component on the Login Form. **TXWBRichEdit** (**XwbRich20.pas**) is a version of the **TRichEdit** component that uses Version 2 of Microsoft's **RichEdit Control** and adds the ability to detect and respond to a Uniform Resource Locator (URL) in the text. This component permits developers to provide some requested functionality on the login form. As an **XWB** namespaced component, it was required to be put on the **Kernel** tab of the component palette; however, it rightly belongs on the **Win32** tab.

> ℹ️ **NOTE:** Properties inherited from the parent component (i.e., **TComponent**) are *not* discussed in this manual (only those properties added to the parent component are described). For help on inherited properties, refer to Delphi's documentation on the parent component (i.e., **TComponent**).

2.1.4.4   Property

The following is the TXWBRichEdit Component property:

URLDetect Property

2.1.5   TXWBSSOiToken Component

- Properties (All)
- Example

2.1.5.1   Parent Class

TXWBSSOiToken = class(TComponent)

2.1.5.2   Unit

XWBSSOi Unit

2.1.5.3 Description

The **TXWBSSOiToken** component provides Delphi developers with an easy, object-based access to an Identity and Access Management (IAM) Secure Token Service (STS) token. It is compatible with the Delphi object oriented (OO) environment. This component, when placed on a Delphi form, allows applications to authenticate a user with the IAM STS Server and exchange the user's 2-factor authentication (Public Key Infrastructure [PKI] certificate and Personal Identification Number [PIN]) credentials for an STS token.

The **TXWBSSOiToken** component (**XWBSSOi.pas**) does *not* need to be explicitly added to RPC Broker applications for 2-factor authentication (2FA) into VistA but is available should authentication be required into another system that accepts the STS token.

The **TXWBSSOi** component can be found on the **Kernel** tab in the component palette.

> **NOTE:** Properties inherited from the parent component (i.e., **TComponent**) are *not* discussed in this manual (only those properties added to the parent component are described). For help on inherited properties, see Delphi's documentation on the parent component (i.e., **TComponent**).

2.1.5.4 Properties (All)

Table 7 lists all of the properties available with the TXWBSSOiToken Component:

Table 7: TXWBSSOi Component—All Properties (listed alphabetically)

Read-only	RunTime only	Property
	▶	SSOiToken Property (TXWBSSOiToken Component)
	▶	SSOiADUPN Property (TXWBSSOiToken Component)
	▶	SSOiLogonName Property (TXWBSSOiToken Component)
	▶	SSOiSECID Property (TXWBSSOiToken Component)

2.1.5.5 Example

The following example demonstrates how a TXWBSSOiToken Component can be used to:

1. Create (obtain) an IAM STS token.

2. Assign the token and user values to strings.

3. Delete the token (free up memory).

The example in [Figure 3](#) assumes that a [TXWBSSOiToken Component](#) already exists on the form as mySSOiToken:

Figure 2: TXWBSSOiToken Component—Example

```
procedure TForm1.Button1Click(Sender: TObject; myToken: String; myName: String);
begin
   try
      {authenticate to the server}
      mySSOiToken := TXWBSSOiToken.Create(nil);
      //assign token values to strings
      myToken := mySSOiToken.SSOiToken;
      myName := mySSOiToken.SSOiLogonName;
      {release the memory used by the token}
      mySSOiToken.Free;
   except
      //put error handling code here
   end;
end;
```

2.2 Classes

2.2.1 TMult Class

- [Properties](#)
- [Methods](#)
- [Example](#)

2.2.1.1   Unit

[TRPCB Unit](#)

2.2.1.2   Description

The **TMult** class is used whenever a list of multiple values needs to be passed to a remote procedure call (RPC) in a single parameter. The [Mult Property](#) of a parameter is of **TMult** type. The information put in the **TMult** variable is really stored in a **TStringList**, but the access methods (used to read and write) take strings as subscripts and provide the illusion of a string-subscripted array.

It is important to note that items in a **TMult** class may or may not be sorted. If the [Sorted Property](#) is:

- **False (default)**—Items are stored in the order they are added.
- **True**—Items are stored in ascending alphabetical order by subscripts.

If you attempt to reference an element by a nonexistent subscript you get an error in the form of a Delphi exception. Do *not* forget that M syntax dictates that all strings *must* be surrounded by double quotes. So, if your goal is to pass a string subscripted array of strings using **TMult** as a

parameter to an RPC on the VistA M Server, do *not* forget to surround each of the subscripts and their associated values with double quotes ("). Otherwise, M assumes that you are passing a list of variables and attempts to reference them, which is probably *not* what you want.

2.2.1.3   Properties

The following are the **TMult Class** properties:

- Count Property (TMult Class)
- First Property
- Last Property
- **MultArray** property
- Sorted Property

2.2.1.4   Methods

The following are the **TMult Class** methods:

- Assign Procedure (TMult Class)
- Order Function
- Position Function
- Subscript Function

2.2.1.5   Example

The program code in Figure 4 demonstrates how to store and retrieve elements from a **TMult** variable:

Figure 3: TMult Class—Example

```
procedure TForm1.Button1Click(Sender: TObject);
var
    Mult: TMult;
    Subscript: string;
begin
    {Create Mult. Make Form1 its owner}
    Mult := TMult.Create(Form1);
    {Store element pairs one by one}
    Mult['First'] := 'One';
    Mult['Second'] := 'Two';
    {Use double quotes for M strings}
    Mult['"First"'] := '"One"';
    {Label1.Caption gets "One"}
    Label1.Caption := Mult['"First"'];
    {Error! 'Third' subscripted element was never stored}
    Label1.Caption := Mult['Third'];
end;
```

2.2.2   TParamRecord Class

- Properties
- Example

2.2.2.1   Unit

TRPCB Unit

2.2.2.2   Description

The **TParamRecord** class is used to hold all of the information on a single RPC parameter. Depending on the type of the parameter needed, different properties are used. The PType Property is always used to let the Broker on the VistA M Server know how to interpret the parameter. For a single value parameter, the Value Property should be used. In the case of a list or a word-processing text, use the Mult Property.

The **TParamRecord** relationship to the TRPCBroker Component is as follows:

The TRPCBroker Component contains the Param Property (i.e., TParams Class).

The TParams Class contains the **ParamArray** property (array [I:integer]: TParamRecord Class).

The TParamRecord Class contains the Mult Property (i.e., TMult Class).

The TMult Class contains the **MultArray** property (array[S: string]: string).

The MultArray property internally uses a **TStringList** in which each element's object is a **TString**.

CAUTION: Developers should *rarely* need to use TParamRecord by itself in their code. TParamRecord is the type of the elements in the ParamArray, default array property of the TRPCBroker Component Param Property. This means that when you are working with a Param[x] element, you are in reality working with an instance of TParamRecord.

REF: For more information on RPCs, see the "RPC Overview" section.

2.2.2.3   Properties

The following are the **TParamRecord** class properties:

- Mult Property
- PType Property
- Value Property

2.2.2.4 Example

The program code in [Figure 5](#) demonstrates how you can use a **TParamRecord** variable to save a copy of a single parameter of a [TRPCBroker Component](#). This example assumes that prior to calling this procedure, a **TRPCBroker** variable has been created and some parameters have been set up. Pay close attention to how properties are copied one at a time. This is the only way that a separate copy could be created. If you try to simply assign one of the **TRPCBroker** parameters to the **TParamRecord** variable, you simply re-point the **TParamRecord** variable to that parameter:

Figure 4: TParamRecord Class—Example

```
procedure TForm1.Button1Click(Sender: TObject);
var
     ParamRecord: TParamRecord;
begin
   {Create ParamRecord. Make Form1 its owner}
   ParamRecord := TParamRecord.Create(Form1);
   {Store properties one at a time}
   ParamRecord.Value := RPCBroker.Param[0].Value;
   ParamRecord.PType := RPCBroker.Param[0].PType;
   {This is how to copy a Mult}
   ParamRecord.Mult.Assign(RPCBroker.Param[0].Mult);
end;
```

2.2.3 TParams Class

- [Properties](#)
- [Methods](#)
- [Example](#)

2.2.3.1 Unit

[TRPCB Unit](#)

2.2.3.2 Description

The **TParams** class is used to hold parameters (i.e., array of **TParamRecord**) used in a remote procedure call (RPC). You do *not* need to know in advance how many parameters you need or allocate memory for them; a simple reference or an assignment to a parameter creates it.

The **Clear** procedure can be used to remove/clear data from **TParams**.

NOTE: Previously, this procedure was Private, but as of Patch XWB*1.1*13, it was made Public.

2.2.3.3 Properties

The following are the **TParams Class** properties:

- Count Property (TParams Class)
- **ParamArray** property

2.2.3.4 Methods

The following are the **TParams Class** methods:

- Assign Procedure (TParams Class)
- **Clear** procedure

2.2.3.5 Example

The program code in Figure 6 demonstrates how a TParams Class can be used to save off the TRPCBroker Component parameters and restore them later:

Figure 5: TParams Class—Example

```
procedure TForm1.Button1Click(Sender: TObject);
var
    SaveParams: TParams;
    SaveRemoteProcedure: string;
begin
    {create holding variable with Form1 as owner}
    SaveParams := TParams.Create(self);
    {save parameters}
    SaveParams.Assign(brkrRPCBroker1.Param);
    SaveRemoteProcedure := brkrRPCBroker1.RemoteProcedure;
    brkrRPCBroker1.RemoteProcedure := 'SOME OTHER PROCEDURE';
    brkrRPCBroker1.ClearParameters := True;
    brkrRPCBroker1.Call;
    {restore parameters}
    brkrRPCBroker1.Param.Assign(SaveParams);
    brkrRPCBroker1.RemoteProcedure := SaveRemoteProcedure;
    {release memory}
    SaveParams.Free;
end;
```

2.2.4   TVistaLogin Class

Properties

2.2.4.1   Unit

TRPCB Unit

2.2.4.2   Description

The **TVistaLogin** class is used to hold login parameters for Silent Login.

ℹ️   **REF:** For examples of silent logon by passing Access and Verify codes, see the "Silent Login Examples" section.

2.2.4.3   Properties

Table 8 lists all of the properties available with the TVistaLogin Class:

Table 8: TVistaLogin Class—All Properties (listed alphabetically)

Read-only	RunTime only	Property
	▶	AccessCode Property
	▶	Division Property (TVistaLogin Class)
▶	▶	DivList Property (read-only)
	▶	DomainName Property
	▶	DUZ Property (TVistaLogin Class)
	▶	ErrorText Property
	▶	IsProductionAccount Property
	▶	LoginHandle Property
	▶	Mode Property
	▶	MultiDivision Property
	▶	NTToken Property
	▶	OnFailedLogin Property
	▶	PromptDivision Property

Read-only	RunTime only	Property
	▷	[VerifyCode Property](#)

2.2.5   TVistaUser Class

[Properties](#)

2.2.5.1   Unit

[TRPCB Unit](#)

2.2.5.2   Description

The **TVistaUser** class is used to hold parameters related to the current user. These parameters are filled in as part of the login procedure.

NOTE: This class is used as a property by the **TRPCBroker** class. This property, with its associated data, is available to all applications, whether or not they are using a [Silent Login](#).

2.2.5.3   Properties

[Table 9](#) lists all of the properties available with the [TVistaUser Class](#):

Table 9: TVistaUser Class—All Properties (listed alphabetically)

Read-only	RunTime only	Property
	▷	[Division Property (TVistaUser Class)](#)
	▷	[DTime Property](#)
	▷	[DUZ Property (TVistaUser Class)](#)
	▷	[Language Property](#)
	▷	[Name Property](#)
	▷	[ServiceSection Property](#)
	▷	[StandardName Property](#)
	▷	[Title Property](#)
	▷	[VerifyCodeChngd Property](#)

Read-only	RunTime only	Property
	▷	[Vpid Property](#)

2.2.6 TXWBWinsock Class

2.2.6.1 Unit

[TRPCB Unit](#)

2.2.6.2 Description

The code handling connections and transmission was moved into the **TXWBWinsock** class, which is defined in **wsockc.pas**. It facilitates the ability for making and maintaining multiple independent RPC Broker connections. To get around cyclic issues with the **Uses** clause, **XWBWinsock** within **Trpcb.pas** is defined as **TObject** and *must* be cast to **TXWBWinsock** when it is used.

The methods in the **wsockc.pas** unit were originally library methods or methods *not* associated with a class. To ensure that the [TCCOWRPCBroker Component](#) is thread-safe (i.e., thread safe operation of RPC Broker instances created in different threads), it became necessary for each instance of the **TRPCBroker** to have its own instance of these methods, values, etc. Thus, the **TXWBWinsock** class was created to encapsulate the Public members.

Methods within the **TXWBWinsock** class should *not* be referenced directly. Connections to VistA are made by setting the **TVistaLogin** [Connected Property](#) to "**true**" and ended by setting the [Connected Property](#) to "**false**".

2.3 Units

⚠ CAUTION: Not all units found in the source code are documented at this time. Only those documented methods and properties are guaranteed to be made backwards compatible in future versions of the BDK.

The following Units are described in this document (listed alphabetically):

- [CCOWRPCBroker Unit](#)
- [LoginFrm Unit](#)
- [MFunStr Unit](#)
- [RPCConf1 Unit](#)
- [RpcSLogin Unit](#)
- [SplVista Unit](#)
- [TRPCB Unit](#)

- [VCEdit Unit](#)

- [Wsockc Unit](#)

- [XWBHash Unit](#)

- [XWBSSOi Unit](#)

2.3.1   CCOWRPCBroker Unit

The CCOWRPCBroker unit authenticates a user using CCOW user context.

2.3.1.1   Library Method

AuthenticateUser Procedure

REF: To see a listing of items declared in this unit including their declarations, use the **ObjectBrowser**.

2.3.2   LoginFrm Unit

As of Patch XWB*1.1*13, a "**Change VC**" check box was added to the to the login form. The user can use this check box to indicate that she/he wants to change their Verify code. If this box has been checked, after the user has completed logging in to the system, the Change Verify code dialogue is displayed.

REF: To see a listing of items declared in this unit including their declarations, use the **ObjectBrowser**.

2.3.3   MFunStr Unit

The MFunStr unit contains Delphi functions that emulate MUMPS functions.

2.3.3.1   Library Methods

- [Piece Function](#)

- [Translate Function](#)

REF: To see a listing of items declared in this unit including their declarations, use the **ObjectBrowser**.

2.3.4 RPCConf1 Unit

The RPCConf1 unit contains server selection dialogue.

⚠️ **CAUTION: This unit assumes that a single IP address is assigned to a host. That is no longer a reasonable assumption in a modern computing environment. These functions are expected to be deprecated and replaced in future versions of the BDK.**

2.3.4.1 Library Methods

- [GetServerInfo Function](#)
- [GetServerIP Function](#)
- **IsIPAddress** function

ℹ️ **REF:** To see a listing of items declared in this unit including their declarations, use the **ObjectBrowser**.

2.3.5 RpcSLogin Unit

The RpcSLogin unit contains silent login functionality.

2.3.5.1 Library Methods

- [CheckCmdLine Function](#)
- **GetSessionInfo** procedure
- **GetUserInfo** procedure
- **SilentLogIn** Ffunction
- [StartProgSLogin Procedure](#)
- **ValidAppHandle** function
- **ValidAVCodes** function
- **ValidNTToken** function

ℹ️ **REF:** To see a listing of items declared in this unit including their declarations, use the **ObjectBrowser**.

ℹ️ **REF:** For more information on silent login, see the "[Silent Login](#)" section.

2.3.6　SplVista Unit

The SplVista unit displays the VistA splash screen.

2.3.6.1　Library Methods

- **SplashOpen** procedure
- **SplashClose** procedure

REF: For more information on splash screens, see the "VistA Splash Screen Procedures" section.

REF: To see a listing of items declared in this unit including their declarations, use the **ObjectBrowser**.

2.3.7　TRPCB Unit

The TRPCB unit contains the declarations for the various RPC Broker components.

When you add a component declared in this unit to a form, the unit is automatically added to the **uses** clause of that form's unit.

The following items are automatically declared in the **uses** clause:

```
SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms, Dialogs
```

2.3.7.1　Classes

- TParamRecord Class
- TParams Class
- TVistaLogin Class
- TVistaUser Class

2.3.7.2　Component

TRPCBroker Component

2.3.7.3   Library Methods

- **GetAppHandle**

- [TMult Class Methods](#)

- [TParams Class Method](#)

- [TRPCBroker Component](#) [Methods](#)

2.3.7.4   Types

- [EBrokerError](#)

- [TLoginMode Type](#)

- [TParamType](#)

ℹ️   **REF:** To see a listing of items declared in this unit including their declarations, use the **ObjectBrowser**.

2.3.8   VCEdit Unit

The RPC Broker calls the VCEdit unit at logon when users *must* change their Verify code (i.e., Verify code has expired). There is also a check box on the "VistA Sign-on" form that allows uses to change their Verify code at any time.

2.3.8.1   Library Methods

- [ChangeVerify Function](#)

- [SilentChangeVerify Function](#)

ℹ️   **REF:** To see a listing of items declared in this unit including their declarations, use the **ObjectBrowser**.

2.3.9 Wsockc Unit

The Wsockc unit contains the interface to the Microsoft® Windows operating system TCP/IP network interface. It provides the communications between the RPC Broker GUI and the VistA M Server.

When a component declared in this unit is added to a form, the unit is automatically added to the **uses** clause of that form's unit.

The following items are automatically declared in the **uses** clause:

```
AnsiStrings, SysUtils, Classes, Windows, WinTypes, WinProcs, Winsock2, Xwbut1,
Trpcb, RpcbErr, Dialogs, Forms, Controls, StdCtrls, ClipBrd
```

2.3.9.1 Component

TXWBWinsock component

REF: To see a listing of items declared in this unit including their declarations, use the **ObjectBrowser**.

2.3.10 XWBHash Unit

2.3.10.1 Library Methods

- Encrypt Function
- Decrypt Function

REF: For more information on encryption/decryption functions, see the "Encryption Functions" section.

REF: To see a listing of items declared in this unit including their declarations, use the **ObjectBrowser**.

2.3.11 XWBSSOi Unit

The XWBSSOi unit contains the interface to the Identity and Access Management (IAM) Secure Token Service (STS) server.

When a component declared in this unit is added to a form, the unit is automatically added to the **uses** clause of that form's unit.

The following items are automatically declared in the **uses** clause:

```
Messages, Windows, Classes, SysUtils, Variants, Controls, Dialogs, Forms, Graphics,
OleCtrls, MSHTML, SHDocVw, MFunStr, XWBut1
```

2.3.11.1   Component

TXWBSSOiToken Component

> **REF:** To see a listing of items declared in this unit including their declarations, use the **ObjectBrowser**.

2.4   Methods

2.4.1   Assign Method (TMult Class)

2.4.1.1   Applies to

TMult Class

2.4.1.2   Declaration

```
procedure Assign(Source: TPersistent);
```

2.4.1.3   Description

The **Assign** method for a TMult Class takes **Tstrings**, a **TStringList**, or another **TMult**. In the case where the source is a **TMult**, the owner of the **Assign** method gets the exact copy of the source with all string subscripts and values. In the case where the source is a **Tstrings** or a **TStringList**, the items are copied such that the strings property of each item becomes the **Value**, while the index becomes the subscript in the string form.

> **REF:** For information about the size of parameters and results that can be passed to and returned from the TMult Class, see the "RPC Limits" section.

2.4.1.4 Example

2.4.1.4.1 TMult Assign Method—Assigning listbox Items to a TMULT

To assign listbox items to a **TMult**, do the following:

1. Start a new VCL Forms application.

2. Drop one **TListBox**, one **TMemo**, and one **TButton** on the form. Arrange controls as in .

3. Add **Vcl.StdCtrls** and **TRPCB** to the "uses" clause.

4. Copy the code in to the **Button1.OnClick** event:

Figure 6: TMult Assign Method—Code Added to the Button1.OnClick Event

```
procedure TForm1.Button1Click(Sender: TObject);
var
   Mult1: TMult;
   Subscript: string;
begin

   //Create Mult1. Make Form1 its owner
   Mult1 := TMult.Create(Form1);

   //Fill listbox with some strings
   ListBox1.Items.Add('One');
   ListBox1.Items.Add('Two');
   ListBox1.Items.Add('Three');
   ListBox1.Items.Add('Four');
   ListBox1.Items.Add('Five');

   //assign (copy) listbox strings to Mult
   Mult1.Assign(ListBox1.Items);

   //configure memo box for better display
   Memo1.Font.Name := 'Courier';
   Memo1.Lines.Clear;
   Memo1.Lines.Add('Tstrings assigned:');

   //set a starting point
   Subscript := '';
   repeat
     //get next Mult element
     Subscript := Mult1.Order(Subscript, 1);
     //if not the end of list
     if Subscript <> '' then
        //display subscript
        Memo1.Lines.Add(Format('%10s', [Subscript]) + ' - ' + Mult1[Subscript])

   //stop when reached the end
   until Subscript = '';
end;
```

5. Run the project and click on the button.

    The expected output is shown in Figure 8:

Figure 7: TMult Assign Method—Assigning listbox Items to a TMULT: Sample Form Output
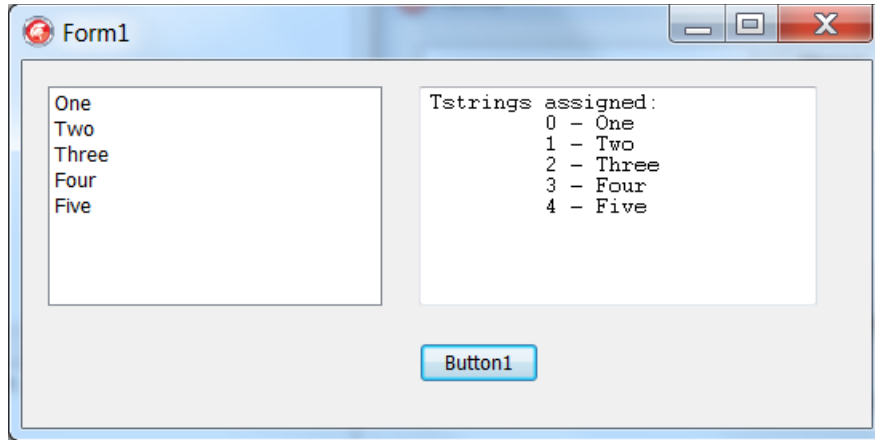
2.4.1.4.2 TMult Assign Method—Assigning One TMULT to Another

The program code in <u>Figure 9</u> demonstrates the use of the **TMult Assign** method to assign one **TMult** to another:

1. Start a new **VCL Forms** application.
2. Drop one **TMemo** and one **TButton** on the form. Arrange controls as in <u>Figure 10</u>.
3. Add **Vcl.StdCtrls** and **TRPCB** to the "uses" clause.
4. Copy the code in <u>Figure 9</u> to the **Button1.OnClick** event:

Figure 8: TMult Assign Method—Code Added to the Button1.OnClick Event

```
procedure TForm1.Button1Click(Sender: TObject);
var
   Mult1, Mult2: TMult;
   Subscript: string;
begin
   //Create Mult1. Make Form1 its owner
   Mult1 := TMult.Create(Form1);
   //Create Mult2. Make Form1 its owner
   Mult2 := TMult.Create(Form1);

   //Fill Mult1 with some strings
   Mult1['First'] := 'One';
   Mult1['Second'] := 'Two';
   Mult1['Third'] := 'Three';
   Mult1['Fourth'] := 'Four';
   Mult1['Fifth'] := 'Five';

   //assign (copy) Mult1 strings to Mult2
   Mult2.Assign(Mult1);

   //configure memo box for better display
   Memo1.Font.Name := 'Courier';
   Memo1.Lines.Clear;
   Memo1.Lines.Add('TMult assigned:');

   //set a starting point
   Subscript := '';
   repeat
     //get next Mult element
     Subscript := Mult2.Order(Subscript, 1);
     //if not the end of list
     if Subscript <> '' then
        //display subscript value
        Memo1.Lines.Add(Format('%10s', [Subscript]) + ' - ' + Mult2[Subscript])
   //stop when reached the end
   until Subscript = '';
end;
```

5. Run the project and click on the button.

The expected output is shown in Figure 10:

Figure 9: TMult Assign Method—Assigning One TMULT to another: Sample Form Output
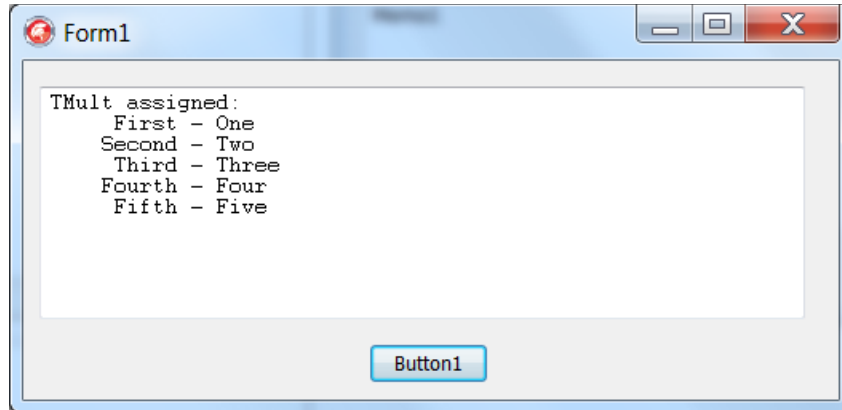
2.4.2   Assign Method (TParams Class)

2.4.2.1   Applies to

TParams Class

2.4.2.2   Declaration

```
procedure Assign(Source: TParams);
```

2.4.2.3   Description

The **Assign** method for a TParams Class takes another TParams Class parameter. The **Assign**
method is useful for copying one TParams Class to another. The entire contents of the passed in
TParams Class are copied into the owner of the **Assign** method. The **Assign** method first deletes
all of the parameters in the receiving class and then copies the parameters from the passed in
class, creating a whole duplicate copy.

REF: For information about the size of parameters and results that can be passed to and
returned from the TParams Class, see the "RPC Limits" section.

2.4.2.4 Example

The program code in Figure 11 demonstrates how a TParams Class **Assign** method can be used to save off the TRPCBroker Component parameters and restore them later:

Figure 10: Assign Method (TParams Class)—Example

```
procedure TForm1.Button1Click(Sender: TObject);
var
    SaveParams: TParams;
    SaveRemoteProcedure: string;
begin
    SaveParams := TParams.Create(self)  {create holding variable with Form1 as owner}
    SaveParams.Assign(brkrRPCBroker1.Param);  {save parameters}
    SaveRemoteProcedure := brkrRPCBroker1.RemoteProcedure;
    brkrRPCBroker1.RemoteProcedure := 'SOME OTHER PROCEDURE';
    brkrRPCBroker1.ClearParameters := True;
    brkrRPCBroker1.Call;
    brkrRPCBroker1.Param.Assign(SaveParams);  {restore parameters}
    brkrRPCBroker1.RemoteProcedure := SaveRemoteProcedure;
    SaveParams.Free;  {release memory}
end;
```

2.4.3 Call Method

2.4.3.1 Declaration

```
procedure Call;
```

2.4.3.2 Description

The **Call** method executes a remote procedure on the VistA M Server and returns the results in the Results Property. **Call** expects the name of the remote procedure and its parameters to be set up in the RemoteProcedure Property and Param Property respectively. If the ClearResults Property is **True**, then the Results Property is cleared before the call. If the ClearParameters Property is **True**, then the Param Property is cleared after the call finishes.

REF: For information about the size of parameters and results that can be passed to and returned from the TRPCBroker Component, see the "RPC Limits" section.

NOTE: Whenever the Broker makes a call to the VistA M Server, if the cursor is **crDefault**, the cursor is automatically changed to the hourglass symbol as seen in other Microsoft-compliant software. If the application has already modified the cursor from **crDefault** to something else, the Broker does *not* change the cursor.

REF: For a demonstration using the **Call** method, run the RPC Broker Example (32-Bit) (i.e., **BrokerExample.exe**); located in the following directory:

BDK32\Samples\BrokerEx

2.4.3.3 Example

The program code in Figure 12 demonstrates the use of the Call Method in a hypothetical example of bringing back demographic information for a patient and then displaying the results in a memo box:

Figure 11: Call Method—Example

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    brkrRPCBroker1.RemoteProcedure := 'GET PATIENT DEMOGRAPHICS';
    brkrRPCBroker1.Param[0].Value := 'DFN';
    brkrRPCBroker1.Param[0].PType := reference;
    brkrRPCBroker1.Call;
    Memo1.Lines := brkrRPCBroker1.Results;
end;
```

REF: For a demonstration using the Call Method, run the RPC Broker Example (32-Bit) (i.e., **BrokerExample.exe**); located in the following directory:

BDK32\Samples\BrokerEx

2.4.4 CreateContext Method

2.4.4.1 Declaration

```
function CreateContext(strContext: string): boolean;
```

Use the **CreateContext** method of the TRPCBroker Component to create a context for your application. To create context, pass an option name in the **strContext** parameter. If the function returns **True**, a context was created, and your application can use all RPCs entered in the option's RPC multiple. If the TRPCBroker Component is *not* connected at the time context is created, a connection is established. If for some reason a context could *not* be created, the **CreateContext** method returns **False**.

Since context is nothing more than a client/server "**B**"-type option in the OPTION (#19) file, standard Kernel Menu Manager (MenuMan) security is applied in establishing a context. Therefore, a context option can be granted to users exactly the same way as regular options are done using MenuMan. Before any RPC can run, it *must* have a context established for it to on the VistA M Server. Thus, all RPCs *must* be registered to one or more "**B**"-type options. This plays a major role in Broker security.

REF: For information about registering RPCs, see the "RPC Security: How to Register an RPC" section.

A context *cannot* be established for the following reasons:

- The user has no access to that option.
- The option is temporarily out of order.

An application can switch from one context to another as often as it needs to. Each time a context is created the previous context is overwritten.

REF: For information about saving off the current context in order to temporarily create a different context and then restore the previous context, see the "CurrentContext Property (read-only)" section.

REF: For information about the size of parameters and results that can be passed to and returned from the TRPCBroker Component, see the "RPC Limits" section.

NOTE: Whenever the Broker makes a call to the VistA M Server, if the cursor is **crDefault**, the cursor is automatically changed to the hourglass symbol as seen in other Microsoft-compliant software. If the application has already modified the cursor from **crDefault** to something else, the Broker does *not* change the cursor.

REF: For a demonstration that creates an application context, run the RPC Broker Example (32-Bit) (i.e., **BrokerExample.exe**); located in the following directory:

BDK32\Samples\BrokerEx

2.4.4.2 Example

The program code in Figure 13 demonstrates the use of the CreateContext Method:

Figure 12: CreateContext Method—Example

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   brkrRPCBroker1.Connected := True;
   if brkrRPCBroker1.CreateContext('MY APPLICATION') then
     Label1.Caption := 'Context MY APPLICATION was successfully created.'
   else
     Label1.Caption := 'Context MY APPLICATION could not be created.';
end;
```

REF: For a demonstration that creates an application context, run the RPC Broker Example (32-Bit) (i.e., **BrokerExample.exe**); located in the following directory:

BDK32\Samples\BrokerEx

2.4.5 GetCCOWtoken Method

2.4.5.1 Declaration

```
function GetCCOWtoken(Contextor: TContextorControl): string;
```

The **GetCCOWtoken** method returns the CCOW token as a string value. This value is passed in as authentication for the current user. The developer should *not* need access to this, since it is handled directly within the code for making the connection.

> **NOTE:** The **TContextorControl** component is the interface for the Sentillion Vergence **ContextorControl** that communicates with the Context Vault. The component is created based on the type library for the DLL.

Since developers may want to use the **TContextorControl** component to initialize their own instances, the **TContextorControl** component is placed on the **Kernel** palette in Delphi; however, it is almost as easy to simply create it at run-time without using a component.

> **REF:** For an example of the **GetCCOWtoken** method, run the RPC Broker Example (32-Bit) (i.e., **BrokerExample.exe**); located in the following directory:
>
> **BDK32\Samples\BrokerEx**

2.4.6 IsUserCleared Method

2.4.6.1 Declaration

```
function IsUserCleared: Boolean;
```

The **IsUserCleared** method returns a value of **True** if the user value in the Context Vault has been cleared. The value is only of interest if the WasUserDefined Method has a **True** value (since unless the user has been defined previously, it would *not* have a value). This method returns:

- **True**—CCOWUser Context is currently cleared.

- **False**—CCOWUser Context is currently *not* cleared

This method is used in response to an **OnPending** event to determine if the pending change is User Context related, and if so, whether the User value in the Context Vault has been cleared. If the value has been cleared, then the application should shut down. Switching User Context is *not* supported, since Office of Cyber and Information Security (OCIS) policy indicates that the

current user *must* sign off the client workstation and the new user *must* sign on the client workstation.

2.4.6.2   Example

In the event handler for the **Commit** event of the **TContextorControl**, developers can check whether or not the user was previously defined and is now undefined or **NULL**. In this case, developers would want to do any necessary processing, and then halt.

Figure 13: IsUserCleared Method—Example

```
Procedure TForm1.CommitHandler(Sender: TObject)
begin
    with CCOWRPCBroker1 do
      if WasUserDefined and IsUserCleared then
      begin
         // do any necessary processing before halting
         halt;
      end;
end;
```

2.4.7   IsUserContextPending Method

2.4.7.1   Declaration

```
function IsUserContextPending(aContextItemCollection: IContextItemCollection):
Boolean;
```

The **IsUserContextPending** method returns a value of **True** if the pending context change is related to User Context; if not, then it may be related to the Patient Context, etc. This method returns:

- **True**—CCOW pending context change is related to User Context.

- **False**—CCOW pending context change is *not* related to User Context (e.g., Patient Context change).

This method is used in response to an **OnPending** event to determine if the pending change is User Context related, and if so, whether the User value in the Context Vault has been cleared. If the value has been cleared, then the application should shut down. Switching User Context is *not* supported, since Office of Cyber and Information Security (OCIS) policy indicates that the current user *must* sign off the client workstation and the new user *must* sign on the client workstation.

REF: For an example of the **IsUserContextPending** method, run the RPC Broker Example (32-Bit) (i.e., **BrokerExample.exe**); located in the following directory:

BDK32\Samples\BrokerEx

2.4.8 lstCall Method

2.4.8.1 Declaration

```
procedure lstCall(OutputBuffer: Tstrings;
```

The **lstCall** method executes a remote procedure on the VistA M Server and returns the results into the passed **Tstrings-** or **TStringList-type** variable, which you create outside of the call. It is important to free the memory later. **lstCall** expects the name of the remote procedure and its parameters to be set up in the RemoteProcedure Property and Param Property respectively. The Results Property is *not* affected by this call. If the ClearParameters Property is **True**, then the Param Property is cleared after the call finishes.

> **REF:** For information about the size of parameters and results that can be passed to and returned from the TRPCBroker Component, see the "RPC Limits" section.

> **NOTE:** Whenever the Broker makes a call to the VistA M Server, if the cursor is **crDefault**, the cursor is automatically changed to the hourglass symbol as seen in other Microsoft-compliant software. If the application has already modified the cursor from **crDefault** to something else, the Broker does not change the cursor.

> **REF:** For a demonstration using the **lstCall** method, run the RPC Broker Example (32-Bit) (i.e., **BrokerExample.exe**); located in the following directory:
>
> **BDK32\Samples\BrokerEx**

2.4.8.2 Example

The program code in Figure 15 demonstrates the use of the lstCall Method in a hypothetical example of bringing back a list of user's keys and automatically filling a list box with data:

Figure 14: lstCall Method—Example

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   brkrRPCBroker1.RemoteProcedure := 'GET MY KEYS';
   brkrRPCBroker1.lstCall(ListBox1.Items);
end;
```

> **REF:** For a demonstration using the lstCall Method, run the RPC Broker Example (32-Bit) (i.e., **BrokerExample.exe**); located in the following directory:
>
> **BDK32\Samples\BrokerEx**

2.4.9 pchCall Method

2.4.9.1 Declaration

```
function pchCall: Pchar;
```

The **pchCall** function is the lowest level call used by the TRPCBroker Component and each of the other **Call** methods (i.e., Call Method, strCall Method, and lstCall Method), which are implemented via **pchCall**. The return value is a **Pchar**, which can contain anything from a **NULL** string, a single text string, or many strings each separated by Return and/or Line Feed characters. For converting multiple lines within the return value into a **Tstrings**, use the **SetText** method of the **Tstrings**.

2.4.10 Order Method

2.4.10.1 Applies to

TMult Class

2.4.10.2 Declaration

```
function Order(const StartSubscript: string; Direction: integer): string;
```

2.4.10.3 Description

The **Order** method works very similar to the **$ORDER** function in M. Using the **Order** method, you can traverse through the list of elements in the Mult Property of an RPC parameter.

The **StartSubscript** parameter is the subscript of the element whose next or previous sibling is returned. If the **Direction** parameter is a positive number, then the subscript of the following element is returned, while if it is **0** or negative, then the predecessor's subscript is returned. If the list is empty, or there are no more elements beyond the **StartSubscript** parameter, then empty string is returned. You can use the empty string as a **StartSubscript** parameter; then, depending on the **Direction** parameter, you get the subscript of the first or the last element in the list.

There are some important differences between this **Order** method and the M **$ORDER** function:

- The **Order** method requires both parameters to be passed in.

- If the **StartSubscript** parameter is *not* an empty string, it *must* be equal to one of the subscripts in the list; otherwise, an empty string is returned.

- It is case-sensitive.

- Unlike arrays in M, elements in **TMult** may or may not be in alphabetical order, depending on the Sorted Property; so, **Order** may not return the next or previous subscript in collating sequence.

REF: For information about the size of parameters and results that can be passed to and returned from the TMult Class, see the "RPC Limits" section.

2.4.10.4 Example

The program code in Figure 16 demonstrates how to get the next and previous elements in a **TMult** list:

Figure 15: Order Method—Sample Code to Get the Next and Previous Elements in a TMult List

```
procedure TForm1.Button1Click(Sender: TObject);
var
    Mult: TMult;
    Subscript: string;
begin
    {Create Mult. Make Form1 its owner}
    Mult := TMult.Create(Form1);
    Mult['First'] := 'One';
    {Store element pairs one by one}
    Mult['Second'] := 'Two';
    Mult['Third'] := 'Three';
    Mult['Fourth'] := 'Four';
    {Subscript is Fourth}
    Subscript := Mult.Order('Third',1);
    {Subscript isnd}
    Subscript := Mult.Order('Third',-1);
    {Subscript is ''. THIRD subscript does not exist}
    Subscript := Mult.Order('THIRD',1);
    {Subscript is First}
    Subscript := Mult.Order('',1);
    {Subscript is Fourth}
    Subscript := Mult.Order('',-1);
end;
```

2.4.11 Position Method

2.4.11.1 Applies to

[TMult Class](#)

2.4.11.2 Declaration

```
function Position(const Subscript: string): longint;
```

2.4.11.3 Description

The **Position** method takes the string subscript of an item in a **TMult** variable and returns its numeric index position, much like a **TStringList**'s **IndexOf** method. Because **TMult** uses a **TStringList** internally, the **IndexOf** method is used to implement the **Position** method. The first position in the **TMult** is **0**. If **TMult** is empty, or the **Subscript** does *not* identify an existing item, **Position** returns **-1**.

The **Position** and **Subscript** methods are the reciprocals of each other.

> **REF**: For information about the size of parameters and results that can be passed to and returned from the [TMult Class](#), see the "[RPC Limits](#)" section.

2.4.11.4 Example

The program code in [Figure 17](#) demonstrates how to get the position of an item in a **TMult** variable:

Figure 16: Position Method—Sample Code that Shows How to Get the Position of an Item in a TMult Variable

```
procedure TForm1.Button1Click(Sender: TObject);
var
   Mult: TMult;
begin
   {Create Mult. Make Form1 its owner}
   Mult := TMult.Create(Form1);
   Label1.Caption := 'The position of the ''Third'' element is ' +
      {is -1 since the list is empty}
      IntToStr(Mult.Postion('Third'));
   Mult['Second'] := 'Two';
   Label1.Caption := 'The position of the ''Third'' element is ' +
      {is -1 since 'Third' item does not exit}
      IntToStr(Mult.Postion('Third'));
   Label1.Caption := 'The position of the ''Second'' element is ' +
      {is 0, TMult positions start with 0}
      IntToStr(Mult.Postion('Second'));
end;
```

2.4.12 strCall Method

```
function strCall: string;
```

The **strCall** method executes a remote procedure on the VistA M Server and returns the results as a value of a function. The **strCall** method expects the name of the remote procedure and its parameters to be set up in the RemoteProcedure Property and Param Property respectively. The Results Property is not affected by this call. If the ClearParameters Property is **True**, then the Param Property is cleared after the call finishes.

REF: For information about the size of parameters and results that can be passed to and returned from the TRPCBroker Component, see the "RPC Limits" section.

NOTE: Whenever the Broker makes a call to the VistA M Server, if the cursor is **crDefault**, the cursor is automatically changed to the hourglass symbol as seen in other Microsoft-compliant software. If the application has already modified the cursor from **crDefault** to something else, the Broker does *not* change the cursor.

REF: For a demonstration using the **strCall** method, run the RPC Broker Example (32-Bit) (i.e., **BrokerExample.exe**); located in the following directory:

BDK32\Samples\BrokerEx

2.4.12.1 Example

The program code in Figure 18 demonstrates the use of the strCall Method in a hypothetical example of bringing back the name of the user currently logged on and automatically displaying it in a label:

Figure 17: strCall Method—Sample Code Showing the Use of the strCall Method

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   brkrRPCBroker1.RemoteProcedure := 'GET CURRENT USER NAME';
   Label1.Caption := brkrRPCBroker1.strCall;
end;
```

REF: For a demonstration using the strCall Method, run the RPC Broker Example (32-Bit) (i.e., **BrokerExample.exe**) located in the following directory:

BDK32\Samples\BrokerEx

2.4.13 Subscript Method

2.4.13.1 Applies to

[TMult Class](#)

2.4.13.2 Declaration

```
function Subscript(const Position: longint): string;
```

2.4.13.3 Description

The **Subscript** method takes the numeric position of an item in a **TMult** variable and returns its string subscript. If **TMult** is empty, or the **Position** is greater than the number of items in the list, an empty string is returned.

The **Subscript** method and [Position Method](#) are the reciprocals of each other.

REF: For information about the size of parameters and results that can be passed to and returned from the [TMult Class](#), see the "[RPC Limits](#)" section.

2.4.13.4 Example

The program code in [Figure 19](#) demonstrates how to get the subscript of an item in a **TMult** variable:

Figure 18: Subscript Method—Example

```
procedure TForm1.Button1Click(Sender: TObject);
var
   Mult: TMult;
begin
   {Create Mult. Make Form1 its owner}
   Mult := TMult.Create(Form1);
   Label1.Caption := 'The subscript of the item at position 1 is ' +
      {is empty since the list is empty}
      Mult.Subscript(1);
   Mult['Second'] := 'Two';
   Label1.Caption := 'The subscript of the item at position 1 is ' +
      {is empty. Only one item in list so far at 0th position}
      Mult.Subscript(1);
   Mult['Third'] := 'Three';
   Label1.Caption := 'The subscript of the item at position 1 is ' +
      {is Third}
      Mult.Subscript(1);
end;
```

2.4.14 WasUserDefined Method

```
function WasUserDefined: Boolean;
```

The **WasUserDefined** method is used to determine whether or not a User Context is currently or was previously defined in the Context Vault. It returns **True** any time after the initial establishment of User Context. This method returns:

- **True**—CCOW User Context established.

- **False**—CCOW User Context *not* established.

This method is used in response to an **OnPending** event to determine if the pending change is User Context related, and if so, whether the User value in the Context Vault has been cleared. If the value has been cleared, then the application should shut down. Switching User Context is *not* supported, since Office of Cyber and Information Security (OCIS) policy indicates that the current user *must* sign off the client workstation and the new user *must* sign on the client workstation.

2.4.14.1 Example

In the event handler for the **Commit** event of the **TContextorControl**, developers can check whether or not the user was previously defined and is now undefined or **NULL**. In this case, developers would want to do any necessary processing, and then halt.

Figure 19: WasUserDefined Method—Example

```
Procedure TForm1.CommitHandler(Sender: TObject);
begin
   with CCOWRPCBroker1 do
     if WasUserDefined and IsUserCleared then
     begin
       // do any necessary processing before halting
       halt;
     end;
end;
```

2.5 Types

2.5.1 TLoginMode Type

The TLoginMode type is used with the [Mode Property](#) as part of the [TVistaLogin Class](#).

2.5.1.1 Unit

[TRPCB Unit](#)

```
type TLoginMode = (lmAVCodes, lmAppHandle);
```

2.5.1.2 Description

The TLoginMode type includes the acceptable values that can be used during [Silent Login](#). If the [KernelLogIn Property](#) is set to **False**, then it is a [Silent Login](#). Thus, one of these mode types has to be set in the [TVistaLogin Class](#) [Mode Property](#). The Broker uses the information to perform a [Silent Login](#).

[Table 10](#) lists the possible values:

Table 10: TLoginMode Type—Silent Login Values

Value	Meaning
lmAVCodes	Used if the application is passing in the user's Access and Verify codes during [Silent Login](#).
lmAppHandle	Used to pass in an application handle rather than a user's Access and Verify codes during [Silent Login](#). It sets the mode to lmAppHandle and the [KernelLogIn Property](#) to **False**. Indicates that an application handle is being passed to the application when it was being started as opposed to Access and Verify codes.

2.5.2   TParamType

2.5.2.1   Unit

TRPCB Unit

2.5.2.2   Declaration

```
TParamType = (literal, reference, list, global, empty, stream, undefined);
```

2.5.2.3   Description

The **TParamType** type defines the possible values of the RPC parameter type (PType Property of TParamRecord Class).

The global, empty, and stream values (added with RPC Broker Patch XU*1.1*40) can only be used if a new-style (i.e., *non*-callback) connection is present.

⚠️   **CAUTION: Use of the underlined TParam Type in applications is *not* supported. It exists for the RPC Broker's *internal use only*.**

2.6   Properties

2.6.1   AccessCode Property

2.6.1.1   Applies to

TVistaLogin Class

2.6.1.2   Declaration

```
property AccessCode: String;
```

2.6.1.3   Description

The **AccessCode** property (**RunTime**) holds the Access code for the lmAVCodes mode of Silent Login. The user's Access code value should be set in as clear text. It is encrypted before it is transmitted to the VistA M Server.

ℹ️   **REF:** For examples of silent logon by passing Access and Verify codes, see the "Silent Login Examples" section.

ℹ️   **REF:** For more information on Access codes, see the "Part 1: Sign-On/Security" section in the *Kernel 8.0 & Kernel Toolkit 7.3 Systems Management Guide*.

2.6.2 BrokerVersion Property (read-only)

2.6.2.1 Applies to

[TRPCBroker Component](#)

2.6.2.2 Declaration

```
property BrokerVersion: String;
```


2.6.2.3 Description

The **BrokerVersion** property (**RunTime**) is a read-only property that indicates the RPC Broker version used in generating the application (currently, it returns the string "XWB*1.1*60").

2.6.3 CCOWLogonIDName Property (read-only)

2.6.3.1 Applies to

[TCCOWRPCBroker Component](#)

2.6.3.2 Declaration

```
property CCOWLogonIDName: String;
```


2.6.3.3 Description

The **CCOWLogonIDName** property (**RunTime**) is a read-only property that is the name used within the CCOW Context Vault to store the **LogonId**.

It permits the user to identify the logon ID name associated with the [CCOWLogonIDValue Property (read-only)](#) logon ID name value used within the Context Vault related to User Context.

2.6.4 CCOWLogonIDValue Property (read-only)

2.6.4.1 Applies to

TCCOWRPCBroker Component

2.6.4.2 Declaration

```
property CCOWLogonIDValue: String;
```

2.6.4.3 Description

The **CCOWLogonIDValue** property (**RunTime**) is a read-only property that gives the value currently associated with the LogonId in the CCOW Context Vault.

It permits the user to identify the logon ID value associated with the CCOWLogonIDName Property (read-only) logon ID name used within the Context Vault related to User Context.

2.6.5 CCOWLogonName Property (read-only)

2.6.5.1 Applies to

TCCOWRPCBroker Component

2.6.5.2 Declaration

```
property CCOWLogonName: String;
```

2.6.5.3 Description

The **CCOWLogonName** property (**RunTime**) is a read-only property that gives the name used to store the **LogonName** of the currently active user.

It permits the user to identify the logon name associated with the CCOWLogonNameValue Property (read-only) logon name value used within the Context Vault related to User Context.

2.6.6 CCOWLogonNameValue Property (read-only)

2.6.6.1 Applies to

TCCOWRPCBroker Component

2.6.6.2 Declaration

```
property CCOWLogonNameValue: String;
```

2.6.6.3 Description

The **CCOWLogonNameValue** property (**RunTime**) is a read-only property that gives the value of the **LogonName** of the currently active user.

It permits the user to identify the logon name value associated with the CCOWLogonName Property (read-only) logon name used within the Context Vault related to User Context.

2.6.7 CCOWLogonVpid Property (read-only)

2.6.7.1 Applies to

TCCOWRPCBroker Component

2.6.7.2 Declaration

```
property CCOWLogonVpid: String;
```

2.6.7.3 Description

The **CCOWLogonVpid** property (**RunTime**) is a read-only property gthat ives the name used to store the **LogonVpid** value in the CCOW Context Vault.

It permits the user to identify the logon VA Person Identification (VPID) name associated with the CCOWLogonVpidValue Property (read-only) logon VPID value used within the Context Vault related to User Context.

2.6.8   CCOWLogonVpidValue Property (read-only)

2.6.8.1   Applies to

[TCCOWRPCBroker Component](#)

2.6.8.2   Declaration

```
property CCOWLogonVpidValue: String;
```

2.6.8.3   Description

The **CCOWLogonVpidValue** property (**RunTime**) is a read-only property that gives the value of the VA Person Identification (VPID) value for the currently logged on user, if the facility has been enumerated; otherwise, the value returned is a **NULL** string.

It permits the user to identify the logon VPID value associated with the [CCOWLogonVpid Property (read-only)](#) logon VPID name used within the Context Vault related to User Context.

2.6.9   ClearParameters Property

2.6.9.1   Applies to

[TRPCBroker Component](#)

2.6.9.2   Declaration

```
property ClearParameters: Boolean;
```

2.6.9.3   Description

The **ClearParameters** property (**DesignTime**) gives the developer the option to clear the [Param Property](#) after every invocation of any of the following methods:

- [Call Method](#)
- [strCall Method](#)
- [lstCall Method](#)

Setting **ClearParameters** to **True** clears the [Param Property](#).

Simple assignment of **True** to this property clears the [Param Property](#) *after* every invocation of the [Call Method](#), [strCall Method](#), and [lstCall Method](#). Thus, the parameters need only be prepared for the *next* call *without* being concerned about what was remaining from the previous call.

By setting **ClearParameters** to **False**, the developer can make multiple calls with the same [Param Property](#). It is also appropriate to set this property to **False** when a majority of the

parameters in the Param Property should remain the same between calls. However, minor changes to the parameters can still be made.

2.6.9.4 Example

The program code in Figure 21 sets the ClearParameters Property to **True**:

Figure 20: ClearParameters Property—Example

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    brkrRPCBroker1.ClearParameters  := True;
end;
```

2.6.10 ClearResults Property

2.6.10.1 Applies to

TRPCBroker Component

2.6.10.2 Declaration

```
property ClearResults: Boolean;
```

2.6.10.3 Description

The **ClearResults** property (**DesignTime**) gives the developer the option to clear the Results Property prior to every invocation of the Call Method. The strCall Method and lstCall Method are unaffected by this property. Setting ClearResults to **True** clears the Results Property.

If this property is **True**, then the Results Property is cleared *before* every invocation of the Call Method; thus, assuring that only the results of the last call are returned. Conversely, a setting of **False** accumulates the results of multiple calls in the Results Property.

2.6.10.4 Example

The program code in Figure 22 sets the ClearResults Property to **True**:

Figure 21: ClearResults Property—Example

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    brkrRPCBroker1.ClearResults := True;
end;
```

2.6.11 Connected Property

2.6.11.1   Applies to

[TRPCBroker Component](#)

```
property Connected: Boolean;
```

2.6.11.2   Description

The **Connected** property (**DesignTime**) connects the application to the VistA M Server:

- Setting this property to **True** connects the application to the server.

- Setting it to **False** disconnects the application from the server.

It is *not* necessary for your application to manually establish a connection to the VistA M Server. RPC Broker 1.1 automatically connects and disconnects from the server. When you invoke an RPC, if a connection has *not* already been established, one is established for you. However, a user is *not* able to run the RPC unless a context has been created with the [CreateContext Method](#).

The **Connected** property is also used to authenticate a user into a VistA M Server. After making the connection, it makes a call to Identity and Access Management (IAM) Secure Token Service (STS) for 2-factor authentication (2FA) of the user. The STS returns a token, which is used to authenticate the user into a VistA M Server. If a token *cannot* be obtained, VistA user authentication fails over to Access and Verify codes.

There are other advantages to establishing a connection manually. You can check if a connection is established, and branch accordingly depending on whether or not a connection was established. One good place to do this is in the application form's **OnCreate** event. For that event, you could include code as shown in [Figure 23](#):

Figure 22: Connected Property—Example (1 of 2)

```
try
   brkrRPCBroker1.Connected:= True;
except
   on EBrokerError do
begin
   ShowMessage('Connection to server could not be established!');
   Application.Terminate;
end;
end;
```

This code sets the [TRPCBroker Component](#)'s **Connected** property to **True** to establish a connection. If a Broker exception (i.e., [EBrokerError](#)) was raised (in which case the connection was *not* established), it provides a message to the user and calls the **Terminate** method to exit.

To verify that your application is connected to the VistA M Server, check the value of the **Connected** property.

If a connected TRPCBroker Component is destroyed (e.g., when the application is closed) that component first disconnects from the VistA M Server. However, for programming clarity, it is advisable to disconnect your application from the server manually by setting the **Connected** property to **False**.

If your application uses more than one Broker component, you should be aware of the component's connect and disconnect behavior.

> **REF:** For more information on connect-disconnect behavior, see the "Component Connect-Disconnect Behavior" section.

2.6.11.3   Example

The program code in Figure 24 sets the Connected Property to **True**:

Figure 23: Connected Property—Example (2 of 2)

```
procedure TForm1.btnConnectClick(Sender: TObject);
begin
   brkrRPCBroker1.Server := edtServer.Text;
   brkrRPCBroker1.ListenerPort := StrToInt(edtPort.Text);
   brkrRPCBroker1.Connected := True;
end;
```

> **NOTE:** The Server Property and ListenerPort Property *must* be set at design or run time before setting the Connected Property to **True**.

2.6.12 Contextor Property

2.6.12.1   Applies to

TCCOWRPCBroker Component

2.6.12.2   Declaration

```
property Contextor: TContextorControl;
                              read Fcontextor write FContextor; //CCOW
```

2.6.12.3   Description

The **Contextor** property (**RunTime**) *must* be set to an active instance of the **TContextorControl** component in order to initiate a login with CCOW User Context. If it is *not* set to an active instance, then the component basically reverts to an instance of a TRPCBroker Component, since none of the features of CCOW User Context is used.

> **NOTE:** The **TContextorControl** component is the interface for the Sentillion Vergence **ContextorControl** that communicates with the Context Vault. The component is created

based on the type library for the DLL.

Since developers may want to use the **TContextorControl** component to initialize their own instances, the **TContextorControl** component is placed on the **Kernel** palette in Delphi; however, it is almost as easy to simply create it at run-time without using a component.

2.6.13 Count Property (TMult Class)

2.6.13.1 Applies to

TMult Class

2.6.13.2 Declaration

```
property Count: Word;
```

2.6.13.3 Description

The **Count** property (**DesignTime**) contains the number of items in a TMult Class. If TMult Class is empty, **Count** is **zero**.

2.6.13.4 Example

The program code in Figure 25 displays the number of items in a **Mult** class in the caption of a label when the user clicks the **CountItems** button:

Figure 24: Count Property (TMult Class)—Example

```
procedure TForm1.CountItemsClick(Sender: TObject);
begin
    Label1.Caption := 'There are ' + IntToStr(Mult.Count) + ' items in the Mult.'
end;
```

2.6.14 Count Property (TParams Class)

2.6.14.1 Applies to

TParams Class

2.6.14.2 Declaration

```
property Count: Word;
```

2.6.14.3 Description

The **Count** property contains the number of parameters in a TParams Class. If the TParams Class is empty, Count is **zero**.

2.6.14.4 Example

The program code in Figure 26 displays the number of parameters in a **TParams** variable within the caption of a label when the user clicks the **CountParameters** button:

Figure 25: Count Property (TParams Class)—Example

```
procedure TForm1.CountParametersClick(Sender: TObject);
begin
    Label1.Caption := 'There are ' + IntToStr(brkrRPCBroker1.Param.Count) + '
parameters.';
end;
```

2.6.15 CurrentContext Property (read-only)

2.6.15.1 Applies to

TRPCBroker Component

2.6.15.2 Declaration

```
property CurrentContext: String;
```

2.6.15.3 Description

The **CurrentContext** property (**RunTime**) is a read-only property that provides the current context. Developers can:

- Save off the current context into a local variable.

- Set a new context.

- Restore the original context from the local variable before finishing.

This permits the application to use the CreateContext Method with an additional context when an initial context has already been established.

2.6.15.4 Example

The program code in Figure 27 demonstrates the use of the CurrentContext Property (read-only) in a hypothetical example of saving and restoring the current context of an application:

Figure 26: CurrentContext Property—Example

```
procedure TForm1.MyFantasticModule;
var
    OldContext: String;
begin
    OldContext := RPCB.CurrentContext;  // save off old context
    try
      RPCB.SetContext('MyContext');
       .
       .
       .
    finally
      RPCB.SetContext(OldContext);  // restore context before leaving
    end;
end;
```

2.6.16 DebugMode Property

2.6.16.1 Applies to

TRPCBroker Component

2.6.16.2 Declaration

```
property DebugMode: Boolean;
```

2.6.16.3 Description

The **DebugMode** property (**DesignTime**) formerly controlled how the VistA M Server process should be started. The default setting is **False**. Setting this property to **True** has no effect on the VistA M Server process. Control of debugging has been moved from the client to the server.

For debugging purposes, it can be very helpful to:

1. Set break points.

2. Run the server process interactively.

3. Step through the execution.

REF: For more information, see the "How to Debug the Application" section.

2.6.17 Division Property (TVistaLogin Class)

2.6.17.1 Applies to

TVistaLogin Class

2.6.17.2 Declaration

```
property Division: String;
```

2.6.17.3 Description

The **Division** property (**RunTime**) can be set to the desired division for a user for Silent Login.

REF: For information about handling multi-divisions during the Silent Login process, see the "Handling Divisions during Silent Login" section.

2.6.18 Division Property (TVistaUser Class)

2.6.18.1 Applies to

TVistaUser Class

```
property Division: String;
```

2.6.18.2 Description

The **Division** property (**RunTime**) is set to the division for a user when they are logged on.

2.6.19 DivList Property (read-only)

2.6.19.1   Applies to

TVistaLogin Class

2.6.19.2   Declaration

```
property DivList: Tstrings;
```

2.6.19.3   Description

The **DivList** property (**RunTime**) is a read-only property that is a list of divisions that are available for selection by the user for the signon division. This list is filled in, if appropriate, during the Silent Login at the same time that the user is determined to have multiple divisions from which to select. The first entry in the list is the number of divisions present, followed by the names of the divisions that are available to the user.

REF: For information about handling multi-divisions during the Silent Login process, see the "Handling Divisions during Silent Login" section.

2.6.20 DomainName Property

2.6.20.1   Applies to

TVistaLogin Class

2.6.20.2   Declaration

```
property DomainName: String;
```

2.6.20.3   Description

The **DomainName** property (**RunTime**) can be used to obtain the domain name of the server to which the RPC Broker is currently connected.

2.6.21 DTime Property

2.6.21.1 Applies to

TVistaUser Class

2.6.21.2 Declaration

```
property DTime: String;
```

2.6.21.3 Description

The **DTime** property (**RunTime**) holds the user's **DTime**. **DTime** sets the time a user has to respond to timed read. It can be set from **1** to **99999** seconds.

2.6.22 DUZ Property (TVistaLogin Class)

2.6.22.1 Applies to

TVistaLogin Class

2.6.22.2 Declaration

```
property DUZ: String;
```

2.6.22.3 Description

The **DUZ** property (**RunTime**) holds the user's Internal Entry Number (IEN) from the NEW PERSON (#200) file for **TVistaLogin**.

2.6.23 DUZ Property (TVistaUser Class)

2.6.23.1 Applies to

TVistaUser Class

2.6.23.2 Declaration

```
property DUZ: String;
```

2.6.23.3 Description

The **DUZ** property (**RunTime**) holds the user's Internal Entry Number (IEN) from the NEW PERSON (#200) file for **TVistaUser**.

2.6.24 ErrorText Property

2.6.24.1   Applies to

TVistaLogin Class

2.6.24.2   Declaration

```
property ErrorText: String;
```

2.6.24.3   Description

The **ErrorText** property (**RunTime**) holds text of any error message returned by the VistA M Server during the attempted Silent Login. It should be checked if the login fails. For example, it could indicate the following:

- Verify code needs to be changed.
- Invalid Access/Verify code pair.
- Invalid Division.

2.6.25 First Property

2.6.25.1   Applies to

TMult Class

2.6.25.2   Declaration

```
property First: String;
```

2.6.25.3   Description

The **First** property (**DesignTime**) contains the subscript of the first item in a TMult Class. The first item is always in the **0th** Position. You can think of the **First** property as a shortcut to executing the **TMult.Order('',1)** method. If a **TMult** variable does *not* contain any items, the First property is empty.

REF: For more information, see the "Order Method" and "Position Method" sections.

2.6.25.4 Example

The program code in [Figure 28](#) displays the subscript and value of the first item in a **Mult** variable in the caption of a label when the user clicks the **GetFirst** button:

Figure 27: First Property—Example

```
procedure TForm1.GetFirstClick(Sender: TObject);
var
    Mult: TMult;
    Subscript: string;
begin
    {Create Mult. Make Form1 its owner}
    Mult := TMult.Create(Form1);
    Mult['Fruit'] := 'Apple';
    {Store element pairs one by one}
    Mult['Vegetable'] := 'Potato';
    Label1.Caption := 'The subscript of the first element: ' + Mult.First + ', and
its value: ' + Mult[Mult.First];
end;
```

2.6.26 IsProductionAccount Property

2.6.26.1 Applies to

[TVistaLogin Class](#)

2.6.26.2 Declaration

```
property IsProductionAccount: Boolean;
```

2.6.26.3 Description

The **IsProductionAccount** property (**RunTime**) can be checked to determine if the current connection is to a Production account:

- **True**—If the account is a Production account.

- **False**—If the account is *not* a Production account.

While it is declared as a read-write property, it should be considered to be read-only, since changing its value does *not* change the nature of the server to which the RPC Broker is connected.

2.6.27 KernelLogIn Property

2.6.27.1   Applies to

[TRPCBroker Component](#)

2.6.27.2   Declaration

```
property KernelLogIn: Boolean;
```

2.6.27.3   Description

The **KernelLogin** property (**DesignTime**) is a Boolean property, which indicates the manner of signon:

- **True**—Presents the regular Kernel login security form.

- **False**—Broker uses the [TVistaLogin Class](#) for signon.

The [TVistaLogin Class](#) is referenced during [Silent Login](#).

> **REF:** For examples of silent logon by passing Access and Verify codes, see the "[Silent Login Examples](#)" section.

2.6.28 Language Property

2.6.28.1   Applies to

[TVistaUser Class](#)

2.6.28.2   Declaration

```
property Language: String;
```

2.6.28.3   Description

The **Language** property (**RunTime**) holds the user's language from the NEW PERSON (#200) file.

2.6.29 Last Property

2.6.29.1   Applies to

TMult Class

2.6.29.2   Declaration

```
property Last: String;
```

2.6.29.3   Description

The **Last** property (**DesignTime**) contains the subscript of the last item in a TMult Class. The last item is always in **count-1** Position. You can think of the **Last** property as a shortcut to executing the **TMult.Order('',-1)** method. If a **TMult** variable does *not* contain any items, the Last property is empty.

REF: For more information, see the "Order Method" and "Position Method" sections.

2.6.29.4   Example

The program code in Figure 29 displays the subscript and value of the last item in a **Mult** variable in the caption of a label when the user clicks the **GetLast** button:

Figure 28: Last Property—Example

```
procedure TForm1.GetLastClick(Sender: TObject);
var
   Mult: TMult;
   Subscript: string;
begin
   {Create Mult. Make Form1 its owner}
   Mult := TMult.Create(Form1);
   Mult['Fruit'] := 'Apple';
   {Store element pairs one by one}
   Mult['Vegetable'] := 'Potato';
   Label1.Caption := 'The subscript of the last element: ' + Mult.Last + ', and its
value: ' + Mult[Mult.Last];
end;
```

2.6.30 ListenerPort Property

2.6.30.1 Applies to

TRPCBroker Component

2.6.30.2 Declaration

```
property ListenerPort: Integer;
```

2.6.30.3 Description

The **ListenerPort** property (**DesignTime**) gives the developer the ability to select the Listener port on the VistA M Server. It *must always* be set *before* connecting to the server.

If one VistA M Server system has two or more environments (UCIs) that support client/server applications (e.g., Test and Production accounts), the Broker Listener processes *must* be listening on unique ports. Thus, you *must* specify which Listener port to use when you start the Listener on the VistA M Server. Consequently, if you have more than one Listener running on the same server, the application needs to specify the correct Listener for its connection request. This is accomplished using the **ListenerPort** property.

After the initial connection, the VistA M Server connection is moved to another port number [i.e., Socket Property (read-only)], which is used for the remainder of the session.

2.6.30.4 Example

The program code in Figure 30 demonstrates using the ListenerPort Property:

Figure 29: ListenerPort Property—Example

```
procedure TForm1.btnConnectClick(Sender: TObject);
begin
   brkrRPCBroker1.ListenerPort := 9001;
   brkrRPCBroker1.Connected := True;
end;
```

2.6.31 LogIn Property

2.6.31.1 Applies to

TRPCBroker Component

2.6.31.2 Declaration

```
property LogIn: TVistaLogin;
```

2.6.31.3 Description

The **LogIn** property (**RunTime**) holds parameters that the application needs to pass for Silent Login. The instance of the **TVistaLogin** used for this property is created automatically during the creation of the **TRPCBroker** object, and is therefore, available for reference as a **TRPCBroker** property *without* any user setup.

REF: For examples of silent logon by passing Access and Verify codes, see the "Silent Login Examples" section.

2.6.32 LoginHandle Property

2.6.32.1 Applies to

TVistaLogin Class

2.6.32.2 Declaration

```
property LoginHandle: String;
```

2.6.32.3 Description

The **LoginHandle** property (**RunTime**) holds the Application Handle for the **lmAppHandle** mode of Silent Login. The Application Handle is obtained from the VistA M Server by a currently running application using the **GetAppHandle** function in the TRPCB Unit. The function returns a String value, which is then passed as a command line argument with an application that is being started. The new application *must* know to look for the handle, and if present, set up the Silent Login. The **StartProgSLogin** (see the "StartProgSLogin Method" section) procedure in the RpcSLogin Unit can be used directly or as an example of how the application would be started with a valid **AppHandle** as a command line argument. The **CheckCmdLine** procedure (see the "CheckCmdLine Function" section) in the RpcSLogin Unit can be used in an application to determine whether an **AppHandle** has been passed and to initiate the Broker connection or used as an example of how this could be done.

NOTE: The two procedures referenced here also pass the current values from the Server Property, ListenerPort Property, and Division Property (TVistaLogin Class) for the user so that the connection would be made to the same VistA M Server as the original application.

The **AppHandle** that is obtained via the **GetAppHandle** function is only valid for approximately **20 seconds**, after which the Silent Login would fail.

2.6.33 Mode Property

2.6.33.1   Applies to

TVistaLogin Class

2.6.33.2   Declaration

```
property Mode: TloginMode;
```

2.6.33.3   Description

The **Mode** property (**RunTime**) indicates the mode of Silent Login. The possible values include:

- **lmAVCodes**
- **lmAppHandle**

REF: For examples of silent logon by passing Access and Verify codes, see the "Silent Login Examples" section.

2.6.34 Mult Property

2.6.34.1   Applies to

TParamRecord Class

2.6.34.2   Declaration

```
property Mult: TMult;
```

2.6.34.3   Description

(**Mult** is a property of the TParamRecord Class used in the Param Property.)

The **Mult** property (**DesignTime**) of a TParamRecord Class, which is the type of each TRPCBroker Component's **Param[x]** element, can be used to pass a string-subscripted array of strings to the VistA M Server. For example, if you need to pass a patient's name and Social Security Number (SSN) to a remote procedure, you could pass them as two separate parameters as **PType** literals, or you could pass them in one parameter using the **Mult** property as a **PType** list. If one is being sent, a Mult *must* be the last element in the **Param** array.

2.6.34.4 Example

The program code in Figure 31 demonstrates how the Mult Property can be used to pass several data elements to the VistA M Server in one parameter:

Figure 30: Mult Property—Example (1 of 2)

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   with brkrRPCBroker1 do begin
   Param[0].PType :=list;
   Param[0].Mult['"NAME"'] := 'XWBBROKER,ONE'
   Param[0].Mult['"SSN"'] := '000456789';
   RemoteProcedure := 'SETUP PATIENT INFO';
   Call;
   end;
end;
```

Assuming variable **P1** is used on the VistA M Server to receive this array, it would look like Figure 32:

Figure 31: Mult Property—Example (2 of 2)

```
P1("NAME")=XWBBROKER,ONE
P1("SSN")=000456789
```

2.6.35 MultiDivision Property

2.6.35.1 Applies to

TVistaLogin Class

2.6.35.2 Declaration

```
property MultiDivision: Boolean;
```

2.6.35.3 Description

The **MultiDivision** property (**RunTime**) indicates whether the user has multi-divisional access. It is set during the Silent Login process, if the user has more than **one** division that can be selected.

> **REF:** For information about handling multi-divisions during the Silent Login process, see the "Handling Divisions during Silent Login" section.

2.6.36 Name Property

2.6.36.1 Applies to

TVistaUser Class

2.6.36.2 Declaration

```
property Name: String;
```

2.6.36.3 Description

The **Name** property (**RunTime**) holds the user's name from the NEW PERSON (#200) file.

2.6.37 OnFailedLogin Property

2.6.37.1 Applies to

TVistaLogin Class

2.6.37.2 Declaration

```
property OnFailedLogin: TOnLoginFailure;
```

2.6.37.3 Description

The **OnFailedLogin** property (**RunTime**) holds a procedure to be invoked on a failed Silent Login that permits an application to handle errors as desired; where **TOnLoginFailure** is defined as:

```
TOnLoginFailure = procedure (VistaLogin: TVistaLogin) of object;
```

For example, an application could define:

```
Procedure HandleLoginError(Sender: TObject);
```

And then set:

```
OnFailedLogin := HandleLoginError;
```

REF: For examples of silent logon by passing Access and Verify codes, see the "Silent Login Examples" section.

2.6.38 OnRPCBFailure Property

2.6.38.1   Applies to

TRPCBroker Component

2.6.38.2   Declaration

```
property OnRPCBFailure: TOnRPCBFailure;
```

2.6.38.3   Description

The **OnRPCBFailure** property (**RunTime**) holds a procedure to be invoked when the Broker generates an exception that permits an application to handle errors as desired, where **TOnRPCBFailure** is defined as:

```
TOnRPCBFailure = procedure (RPCBroker: TRPCBroker) of object;
```

The text associated with the error causing the exception is found in the RPCBError Property (read-only).

NOTE: If the OnFailedLogin Property is also set, it handles any login errors and does *not* pass them up.

2.6.38.4   Example

For example, an application could define:

```
Procedure HandleBrokerError(Sender: TObject);
```

And then set:

```
OnRPCBFailure := HandleBrokerError;
```

NOTE: The initialization of the **OnRPCBFailure** property should be before the first call to the TRPCBroker Component.

[Figure 33](#) shows an instance of an error handler that takes the **Message** property of the exception and stores it with a time date stamp into a file named **Error.Log** in the same directory with the application exe:

Figure 32: Error Handler—Example of Storing a Message with a Time Date Stamp

```
procedure TForm1.HandleBrokerError(Sender: TObject);
var
   ErrorText: String;
   Path: String;
   StrLoc: TStringList;
   NowVal: TDateTime;
begin
   NowVal := Now;
   ErrorText := TRPCBroker(Sender).RPCBError;
   StrLoc := TStringList.Create;
   try
      Path := ExtractFilePath(Application.ExeName);
      Path := Path + 'Error.Log';
      if FileExists(Path) then
         StrLoc.LoadFromFile(Path);
      StrLoc.Add(FormatDateTime('mm/dd/yyyy hh:mm:ss ',NowVal) + ErrorText);
      StrLoc.SaveToFile(Path);
   finally
      StrLoc.Free;
   end;
end;
```

2.6.39 Param Property

2.6.39.1   Applies to

[TRPCBroker Component](#)

2.6.39.2   Declaration

```
property Param: TParams;
```

2.6.39.3   Description

The **Param** property (**RunTime**) holds all of the parameters that the application needs to pass to the remote procedure using any of the following methods:

- [Call Method](#)
- [strCall Method](#)
- [lstCall Method](#)

Param is a **zero**-based array of the TParamRecord Class. You do *not* need to explicitly allocate any memory for the **Param** property. Simple reference to an element or a value assignment (**:=**) dynamically allocates memory as needed. You should start with the **0ᵗʰ** element and proceed in sequence. Do *not* skip elements.

Each element in the **Param** array has the following properties:

- Mult Property
- PType Property
- Value Property

⚠ **CAUTION: Passing multiple parameters of PType list in one remote procedure call (RPC) is *not* supported at this time. Only one list parameter can be passed to an RPC, and it *must* be the last parameter in the actual list.**

The **Param** relationship to the TRPCBroker Component is as follows:

The TRPCBroker Component contains the **Param** property (i.e., TParams Class).

The TParams Class contains the **ParamArray** property (array [I:integer]: TParamRecord Class).

The TParamRecord Class contains the Mult Property (i.e., TMult Class).

The TMult Class contains the **MultArray** property (array[S: string]: string).

The **MultArray** property internally uses a **TStringList** in which each element's object is a **TString**.

If the remote procedure on the VistA M Server does *not* require any incoming parameters, applications can pass an empty **Param** property. The client application merely sets the RemoteProcedure Property and makes the call. If the **Param** property retains a value from a previous call, it can be cleared using the ClearParameters Property. Thus, it is possible to make a call without passing any parameters.

⚠ **CAUTION: The following restrictions apply with the Param property:**

1. **You are *not* allowed to "skip" passing parameters, such as TAG^ROUTINE(1,,3). If there are fewer elements in the Param array than exist as input parameters in the RPC, the subsequent parameters are *not* passed to the RPC.**

2. **Passing multiple array parameters in one remote procedure call is *not* supported at this time. Only one array parameter can be passed to an RPC, and it *must* be the last parameter in the actual list.**

REF: For a demonstration using the Param Property, run the RPC Broker Example (32-Bit) (i.e., **BrokerExample.exe**); located in the following directory:

> **BDK32\Samples\BrokerEx**

2.6.39.4   Example

The program code in Figure 34 demonstrates how the Param Property of a TRPCBroker Component is referenced and filled with two parameters that the remote procedure expects:

Figure 33: Param Property—Example

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    {first parameter is a single string}
    brkrRPCBroker1.Param[0].Value := '02/27/14';
    brkrRPCBroker1.Param[0].PType := literal;
    {second parameter is a list}
    brkrRPCBroker1.Param[1].Mult['"NAME"'] := 'XWBUSER,ONE';
    brkrRPCBroker1.Param[1].Mult['"SSN"'] := '000-45-6789';
    brkrRPCBroker1.Param[1].PType := list;
end;
```

REF: For a demonstration using the Param Property, run the RPC Broker Example (32-Bit) (i.e., **BrokerExample.exe**); located in the following directory:

> **BDK32\Samples\BrokerEx**

2.6.40 PromptDivision Property

2.6.40.1   Applies to

TVistaLogin Class

2.6.40.2   Declaration

```
property PromptDivison: Boolean;
```

2.6.40.3   Description

The **PromptDivision** property (**RunTime**) should be set to:

- **True**—If the user should be prompted for Division during Silent Login. The prompt only occurs if the user has multi-division access.

- **False**—If the prompt should *not* be displayed due to the manner in which the application is running.

However, if set to **False** and multi-division access is a possibility, then the application *must* handle it in another way. For example, the application developer would do the following:

1. Set **Login.PromptDivision** to **False**.

2. Set the Connected Property to **True** to signon.

3. On return, check whether the Connected Property was set to **True** or check whether the Login.ErrorText Property was *non*-**NULL** (and especially "No Division Selected").

4. If the connection was successful, there is no problem; otherwise, proceed to Steps 5 - 8.

5. Check the **Login.**MultiDivision Property and see if it was set to **True,** which is expected.

6. If the **Login.**MultiDivision Property is set to **True**, check the **Login.**DivList Property (read-only) for a list of the available divisions (remember the first entry is the number of entries that follow), and in whatever method was available to the application, have the user select the correct division.

7. Set the **Login.**Division Property (TVistaLogin Class) to the selected division.

8. Set the Connected Property to **True**, so the connection would be attempted to be established again.

ℹ️ **REF:** For examples of silent logon by passing Access and Verify codes, see the "Silent Login Examples" section.

2.6.41 PType Property

2.6.41.1 Applies to

[TParamRecord Class](#)

2.6.41.2 Declaration

```
property PType: TParamType;
```

2.6.41.3 Description

PType is a property of the [TParamRecord Class](#) used in the [Param Property](#).

The **PType** property (**DesignTime**) determines how the parameter is interpreted and handled by the Broker.

Table 11: PType Property—Values

Value	Definition
literal	Delphi string value; passed as a string literal to the VistA M server. The VistA M Server receives the contents of the corresponding [Value Property](#) as one string or one number.
reference	Delphi string value; treated on the VistA M Server as an M variable name and resolved from the symbol table at the time the RPC executes. The VistA M Server receives the contents of the corresponding [Value Property](#) as a name of a variable defined on the server. Using indirection, the Broker on the server resolves this parameter before handing it off to the application. ⚠️ **CAUTION: For enhanced security reasons, the reference parameter type may be deprecated and removed in subsequent updates to the BDK.**
list	A single-dimensional array of strings in the **Mult** subproperty of the [Param Property](#); passed to the VistA M Server where it is placed in an array. String subscripting can be used. This value is used whenever an application wants to send a list of values to the VistA M Server. Data is placed in a local array. In this case, the content of the corresponding [Mult Property](#) is sent, while the [Value Property](#) is ignored.
global	This value is similar to list, but instead of data being placed in a local array, it is placed in a global array. Use of this value removes the potential problem of allocation errors when large quantities of data are transmitted.
empty	This value indicates that no parameter value is to be passed; it simply passes an empty argument.
stream	This value indicates that the data should be passed as a single stream of data.

Value	Definition
undefined	The Broker uses this value internally. It should *not* be used by an application.

For instance, if you need to pass an empty string to the remote procedure call (RPC), the Value Property should be set to '' (i.e., **NULL**) and the **PType** to literal. Using reference, a developer can pass an M variable (e.g., **DUZ**) without even knowing its value. However, if the M variable being referenced is *not* defined on the VistA M Server, a run-time error occurs. When passing a list to an RPC:

1. Set the **PType** to **list**.

2. Populate the Mult Property.

3. Do *not* put anything into the Value Property (in this case, Value is ignored).

REF: For a demonstration using **PType**, run the RPC Broker Example (32-Bit) (i.e., **BrokerExample.exe**); located in the following directory:

BDK32\Samples\BrokerEx

2.6.41.4 Example

The program code in Figure 35 demonstrates a couple of different uses of the PType Property. Remember, that each **Param[x]** element is really a TParamRecord Class.

Figure 34: PType Property—Example

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   with brkrRPCBroker1 do begin
      RemoteProcedure := 'SET NICK NAME';
      Param[0].Value := 'DUZ';
      Param[0].PType := reference;
      Param[1].Value := edtNickName.Text;
      Param[1].PType := literal;
      Call;
   end;
end;
```

REF: For a demonstration using **PType**, run the RPC Broker Example (32-Bit) (i.e., **BrokerExample.exe**); located in the following directory:

BDK32\Samples\BrokerEx

2.6.42 RemoteProcedure Property

2.6.42.1 Applies to

[TRPCBroker Component](#)

2.6.42.2 Declaration

```
property RemoteProcedure: TRemoteProc;
```


2.6.42.3 Description

The **RemoteProcedure** property (**DesignTime**) should be set to the name of the remote procedure call entry in the [REMOTE PROCEDURE (#8994) File](#).

2.6.42.4 Example

The program code in [Figure 36](#) demonstrates using the [RemoteProcedure Property](#):

Figure 35: RemoteProcedure Property—Example

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    brkrRPCBroker1.RemoteProcedure := 'MY APPLICATION REMOTE PROCEDURE';
    brkrRPCBroker1.Call;
end;
```

2.6.43 Results Property

2.6.43.1 Applies to

[TRPCBroker Component](#)

2.6.43.2 Declaration

```
property Results: Tstrings;
```

2.6.43.3 Description

The **Results** property (**DesignTime**) contains the results of a [Call Method](#). In the case where the RPC returns a single value, it is returned in **Results[0]**. If a call returns a list of values, the **Results** property is filled in the order the list collates on the VistA M Server. The **Results** property can only contain values of array elements—subscripts are *not* returned.

For example:

On the VistA M Server, the M routine constructs the list in the sequence shown in [Figure 37](#):

Figure 36: Results Property—Sample Array List Sequence

```
S LIST("CCC")="First"
S LIST(1)="Second"
S LIST("AAA")="Third"
S LIST(2)="Fourth"
```

Before Broker returns the list to the client, M re-sorts it in alphabetical order as shown in [Figure 38](#):

Figure 37: Results Property—Sample Array List Sequence Sorted Alphabetically

```
LIST(1)="Second"
LIST(2)="Fourth"
LIST("AAA")="Third"
LIST("CCC")="First"
```

On the client, the **Results** property content is shown in [Figure 39](#):

Figure 38: Results Property—Example

```
brkrRPCBroker1.Results[0]=Second
brkrRPCBroker1.Results[1]=Fourth
brkrRPCBroker1.Results[2]=Third
brkrRPCBroker1.Results[3]=First
```

2.6.43.4   Example

The program code in Figure 40 demonstrates using the Results Property:

Figure 39: Results Property—Sample Code Using the Results Property

```
procedure TForm1.btnSendClick(Sender: TObject);
begin
   {clears Results between calls}
   brkrRPCBroker1.ClearResults := True;
   {the following code returns a single value}
   brkrRPCBroker1.RemoteProcedure := 'SEND BACK SOME SINGLE VALUE';
   brkrRPCBroker1.Call;
   Label1.Caption := 'Value returned is: ' + brkrRPCBroker1.Results[0];
   {the following code returns several values}
   brkrRPCBroker1.RemoteProcedure := 'SEND BACK LIST OF VALUES';
   brkrRPCBroker1.Call;
   ListBox1.Items := RPCBroker.Results;
end;
```

2.6.44 RPCBError Property (read-only)

2.6.44.1   Applies to

TRPCBroker Component

2.6.44.2   Declaration

```
property RPCBError: String;
```

2.6.44.3   Description

The **RPCBError** property (**RunTime**) is a read-only property that contains the **Message** property associated with the exception or error that was encountered by the instance of the TRPCBroker Component.

2.6.45 RPCTimeLimit Property

2.6.45.1 Applies to

TRPCBroker Component

2.6.45.2 Declaration

```
property RPCTimeLimit: Integer;
```

2.6.45.3 Description

The **RPCTimeLimit** property (**RunTime**) is a public integer property. It specifies the length of time a client waits for a response from an RPC. The default and minimum value of this property is **30 seconds**. If an RPC is expected to take more than **30 seconds** to complete, adjust the **RPCTimeLimit** property accordingly. However, it is *not* advisable to have an **RPCTimeLimit** that is too long; otherwise, the client-end of the application appears to "hang", if the VistA M Server does *not* respond in a timely fashion.

2.6.45.4 Example

The program code in Figure 41 demonstrates using the RPCTimeLimit Property:

Figure 40: RPCTimeLimit Property—Example

```
procedure TForm1.Button1Click(Sender: TObject);
var
    intSaveRPCTimeLimit: integer;
begin
    brkrRPCBroker1.RemoteProcedure := 'GET ALL LAB RESULTS';
    brkrRPCBroker1.Param[0].Value := 'DFN';
    brkrRPCBroker1.Param[0].PType := reference;
    {save off current time limit}
    intSaveRPCTimeLimit := brkrRPCBroker1.RPCTimeLimit;
    {can take up to a minute to complete}
    brkrRPCBroker1.RPCTimeLimit := 60;
    brkrRPCBroker1.Call;
    {restore previous time limit}
    brkrRPCBroker1.RPCTimeLimit := intSaveRPCTimeLimit;
end;
```

2.6.46 RPCVersion Property

2.6.46.1 Applies to

[TRPCBroker Component](#)

2.6.46.2 Declaration

```
property RPCVersion: String;
```


2.6.46.3 Description

The **RPCVersion** property (**DesignTime**) is a published string type property used to pass the version of the RPC. This can be useful for backward compatibility.

As you introduce new functionality into an existing RPC, your RPC can branch into different parts of the code based on the value of the **RPCVersion** property. The Broker sets the **XWBAPVER** variable on the VistA M Server to the contents of the **RPCVersion** property. This property *cannot* be empty and defaults to "**0**" (**zero**).

You can use the application version number in the **RPCVersion** property.

REF: For a suggested method for constructing version numbers, see the "[Application Considerations](#)" section.

2.6.46.4 Example

In the following examples ([Figure 42](#) and [Figure 43](#)), an RPC is first called with **two** parameters that are added together and the sum returned to the client. Again, this same RPC is called with the same parameters; however, this time it uses a different RPC version value. Thus, the **two** numbers are simply concatenated together, and the resulting string is returned:

2.6.46.4.1 On the Client

Figure 41: RPCVersion Property—Example on the Client

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   {make sure the results get cleared}
   brkrRPCBroker1.ClearResults := True;
   {just re-use the same parameters}
   brkrRPCBroker1.ClearParameters := False;
   brkrRPCBroker1.RemoteProcedure := 'MY APPLICATION REMOTE PROCEDURE';
   brkrRPCBroker1.Param[0].Value := '333';
   brkrRPCBroker1.Param[0].PType := literal;
   brkrRPCBroker1.Param[1].Value := '444';
   brkrRPCBroker1.Param[1].PType := literal;
   brkrRPCBroker1.Call;
   {the result is 777}
   Label1.Caption := 'Result of the call: ' + brkrRPCBroker1.Results[0];
   brkrRPCBroker1.RPCVersion := '2';
   brkrRPCBroker1.Call;
   {the result is 333444}
   Label2.Caption := 'Result of the call: ' + brkrRPCBroker1.Results[0];
end;
```

2.6.46.4.2 On the Server

Figure 42: RPCVersion Property—Example on the Server

```
TAG(RESULT,PARAM1,PARAM2)        ;Code for MY APPLICATION REMOTE PROCEDURE
                    IF XWBAPVER<2 SET RESULT=PARAM1+PARAM2
                    ELSE SET RESULT=PARAM1_PARAM2
                    QUIT RESULT
```

2.6.47 SecurityPhrase Property

2.6.47.1 Applies to

[TRPCBroker Component](#)

2.6.47.2 Declaration

```
property SecurityPhrase: String;
```

2.6.47.3 Description

The **SecurityPhrase** property (**RunTime**) holds the unique security phrase for the application to be used with Broker Security Enhancement (BSE) logon. The security phrase identifies the application as an authorized user of BSE visitor access on the VistA M Server.

> **RECOMMENDATION: Since the Security Phrase is the applications identifier, a good security practice is to identify the Security Phrase as a const value in an include file when compiling any RPC Broker Delphi-based program implementing BSE. Add a substitute include file containing a generic Security Phrase (*not* the one used to compile the application) with the release of the source code.**

> **REF:** For more information on the application Security Phrase, see the "[Step-By-Step Procedures to Implement BSE](#)" section.

2.6.48 Server Property

2.6.48.1 Applies to

[TRPCBroker Component](#)

2.6.48.2 Declaration

```
property Server: String;
```

2.6.48.3 Description

The **Server** property (**DesignTime**) contains the name or [Internet Protocol (IP) address](#) of the VistA M Server system. If the name is used instead of the [IP address](#), Microsoft® Windows Winsock should be able to resolve it. Winsock can resolve a name to an [IP address](#) either through the Domain Name Service ([DNS](#)) or by looking it up in the [HOSTS](#) file on the client workstation. In the case where the same name exists in the [DNS](#) and in the HOSTS file, the HOSTS file entry takes precedence. Changing the name of the VistA M Server while the

[TRPCBroker Component](#) is connected disconnects the [TRPCBroker Component](#) from the previous server.

REF: For common Winsock error messages, see the RPC Broker "FAQ: Common Winsock Error/Status Messages" at the RPC Broker VA Intranet website.

2.6.48.4 Example

The program code in [Figure 44](#) demonstrates using the [Server Property](#):

Figure 43: Server Property—Example

```
procedure TForm1.btnConnectClick(Sender: TObject);
begin
    brkrRPCBroker1.ListenerPort := <REDACTED>;
    brkrRPCBroker1.Server := 'DHCPSERVER';
    brkrRPCBroker1.Connected := True;
end;
```

2.6.49 ServiceSection Property

2.6.49.1 Applies to

[TVistaUser Class](#)

2.6.49.2 Declaration

```
property ServiceSection: String;
```

2.6.49.3 Description

The **ServiceSection** property (**RunTime**) holds the user's service section from the NEW PERSON (#200) file.

2.6.50 ShowCertDialog Property

2.6.50.1 Applies to

TRPCBroker Component

2.6.50.2 Declaration

```
property ShowCertDialog
```

2.6.50.3 Description

The **ShowCertDialog** property was added to the **TRPCBroker** component with RPC Broker Patch XWB*1.1*73. It is of type Boolean and defaults to **False**. If set to **True**, either at design time or at run time, the user will be prompted to select a certificate rather than one being auto-selected. This was requested considering the auto-selection process; many applications have various components that require different user attributes to successfully test. The **ShowCertDialog** property affords the software developer with the ability to show the selection dialog to the user who can cancel it and be presented with the Access/Verify code dialog.

2.6.51 ShowErrorMsgs Property

2.6.51.1 Applies to

TRPCBroker Component

2.6.51.2 Declaration

```
property ShowErrorMsgs: TShowErrorMsgs;
```

2.6.51.3 Description

The **ShowErrorMsgs** property (**DesignTime**) gives the developer the ability to determine how an exception is handled, if an error handler has *not* been provided through the **OnRpcbError** property (i.e., a procedure property that is set to the name of a procedure that is called if an error is encountered). If the **OnRpcbError** property is assigned, then exception processing is delegated to that procedure; otherwise, exception handling is based on the value of **ShowErrorMsgs** property.

Table 12 lists the possible values:

Table 12: ShowErrorMsgs Property—Values

Value	Meaning
semRaise (default)	**This is the default value.** The Broker does *not* handle the error directly but passes it off to the application in general to process, which can result in a different message box display or some other type of error indication.
semQuiet	The error is *not* displayed or raised. This requires the application to check the value of the RPCBError Property (read-only) following calls to the Broker to determine whether an error has occurred, and if so, what the error was. This can be desirable, if the application requires that errors *not* result in display boxes, etc., as might be the case with an NT service or Web application.

2.6.52 Socket Property (read-only)

2.6.52.1 Applies to

TRPCBroker Component

2.6.52.2 Declaration

```
property Socket: Integer;
```

2.6.52.3 Description

The **Socket** property (**RunTime**) contains the active port being used for the TCP/IP connection to the VistA M Server. This is the port that is currently in use on the server as opposed to the **ListenerPort** (see ListenerPort Property) that was used to make the initial connection. After the initial connection, the server connection is moved to another port number (i.e., **Socket**), which is used for the remainder of the session.

2.6.52.4 Example

The program code in Figure 45 populates the Socket Property (read-only) with the active port on the VistA M Server:

Figure 44: Socket Property—Example

```
function ExistingSocket(Broker: TRPCBroker): integer;
var
    Index: integer;
begin
   Result := 0;
   if Assigned(BrokerConnections) and
     BrokerConnections.Find(Broker.Server + ':' + IntToStr(Broker.ListenerPort),
Index) then
     Result := TRPCBroker(BrokerConnections.Objects[Index]).Socket;
end;
```

2.6.53 Sorted Property

2.6.53.1 Applies to

TMult Class

2.6.53.2 Declaration

```
property Sorted: Boolean;
```

2.6.53.3 Description

The **Sorted** property (**DesignTime**) value determines the order of the items in a **TMult** variable. If **Sorted** is:

- **True**—Items are sorted in ascending order of their string subscripts.

- **False** (default)—Items are unsorted, and appear in the order they were added.

Keep in mind that changing **Sorted** from **False** to **True** irreversibly sorts the list so that changing **Sorted** back to **False** does *not* put the list back in its original order, unless the original order was already sorted of course.

2.6.53.4 Example

The program code in [Figure 46](#) demonstrates the effect the [Sorted Property](#) has on a **TMult** variable. Notice that by setting the [Sorted Property](#) back to **False**, the list does *not* revert to its unsorted order:

1. Start a new **VCL Forms** application.

2. Drop one **TMemo** and one **TButton** on the form. Arrange controls as in [Figure 47](#).

3. Add **Vcl.StdCtrls** and **TRPCB** to the "uses" clause.

4. Copy the code in [Figure 46](#) to the **Button1.OnClick** event:

Figure 45: Sorted Property—Code Added to the Button1.OnClick Event

```
procedure TForm1.Button1Click(Sender: TObject);
var
    Mult1: TMult;
    Subscript: string;
begin
    //Create Mult1. Make Form1 its owner
    Mult1 := TMult.Create(Form1);
    //Fill Mult1 with some strings
    Mult1['First'] := 'One';
    Mult1['Second'] := 'Two';
    Mult1['Third'] := 'Three';
    Mult1['Fourth'] := 'Four';
    Mult1['Fifth'] := 'Five';
    //configure memo box for better display
    Memo1.Font.Name := 'Courier';
    Memo1.ScrollBars := ssVertical;
    Memo1.Lines.Clear;
    Memo1.Lines.Add('Natural order:');
    //set a starting point
    Subscript := '';
    repeat
      //get next Mult element
      Subscript := Mult1.Order(Subscript, 1);
      //if not the end of list
      if Subscript <> '' then
      //display subscript value
      Memo1.Lines.Add(Format('%10s', [Subscript]) + ' - ' + Mult1[Subscript])
    //stop when reached the end
    until Subscript = '';

    //list is now sorted alphabetically
    Mult1.Sorted := True;
    Memo1.Lines.Add('');
    Memo1.Lines.Add('Sorted order:');
    //set a starting point
    Subscript := '';
    repeat
      //get next Mult element
      Subscript := Mult1.Order(Subscript, 1);
      //if not the end of list
      if Subscript <> '' then
      //display subscript value
      Memo1.Lines.Add(Format('%10s', [Subscript]) + ' = ' + Mult1[Subscript])
      //stop when reached the end
    until Subscript = '';
```

```
   //existing entries remain in sorted order
   Mult1.Sorted := False;
   Memo1.Lines.Add('');
   Memo1.Lines.Add('Unsorted order:');
   //set a starting point
   Subscript := '';
   repeat
     //get next Mult element
     Subscript := Mult1.Order(Subscript, 1);
     //if not the end of list
     if Subscript <> '' then
         //display subscript value
         Memo1.Lines.Add(Format('%10s', [Subscript]) + ' - ' +
Mult1[Subscript])
   //stop when reached the end
   until Subscript = '';
end;
```

5. Run project and select the button.

    The expected output is shown in Figure 47:

Figure 46: Sorted Property—Sample Form Output
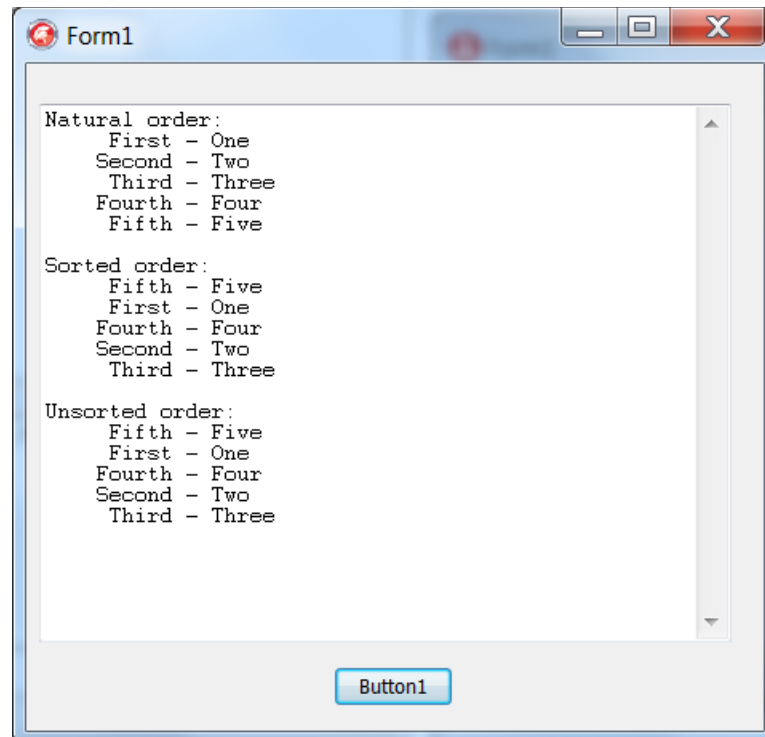
You may have to scroll up and down to see all of the output.

2.6.54 SSHHide Property

2.6.54.1 Applies to

TRPCBroker Component

2.6.54.2 Declaration

```
property SSHHide: Boolean;
```

2.6.54.3 Description

The **SSHhide** property is used when making secured (Secure Shell [SSH]) broker connections. It determines whether the SSH Tunnel application control box is hidden ("**true**") or minimized ("**false**", default) at application run-time.

2.6.55 SSHport Property

2.6.55.1 Applies to

TRPCBroker Component

2.6.55.2 Declaration

```
property SSHport: String;
```

2.6.55.3 Description

The **SSHport** property (**RunTime**) holds a specific port number for SSH Tunneling if the UseSecureConnection Property is set to "**SSH**" or "**PLINK**" (either as a command line option or within the application). If *not* specified, the application uses the RPC Broker listener port for the remote server. This is useful if the server is running separate listeners for SSH and *non*-secured connections.

To set the **SSHport** property as a command line option, include the following:

```
SSHPort=portnumber
```

2.6.56 SSHpw Property

2.6.56.1 Applies to

[TRPCBroker Component](#)

2.6.56.2 Declaration

```
property SSHpw: String;
```

2.6.56.3 Description

The **SSHpw** property (**RunTime**) holds a password for SSH Tunneling if the
[UseSecureConnection Property](#) is set to "**PLINK**" (either as a command line option or within the
application). If *not* specified, the password is set to **NULL**.

To set the **SSHpw** property as a command line option, include the following:

```
SSHpw=password
```

2.6.57 SSHUser Property

2.6.57.1 Applies to

[TRPCBroker Component](#)

2.6.57.2 Declaration

```
property SSHUser: String;
```

2.6.57.3 Description

The **SSHUser** property (**RunTime**) holds a specific username for SSH Tunneling if the
[UseSecureConnection Property](#) is set to "**SSH**" (either as a command line option or within the
application). For VA VistA servers, the username is typically of the form *xxx*vista, where the
xxx is the station's three letter abbreviation.

To set the **SSHUser** property as a command line option, include the following:

```
SSHUser=username
```

2.6.58 SSOiADUPN Property (TRPCBroker Component)

2.6.58.1 Applies to

TRPCBroker Component

2.6.58.2 Declaration

```
property SSOiADUPN: String;
```

2.6.58.3 Description

The **SSOiADUPN** property (**RunTime**) holds the authenticated user's Active Directory (AD) User Principal Name (UPN) for the current connection. The value is obtained from the SSOiADUPN Property (TXWBSSOiToken Component).

2.6.59 SSOiADUPN Property (TXWBSSOiToken Component)

2.6.59.1 Applies to

TXWBSSOiToken Component

2.6.59.2 Declaration

```
property SSOiADUPN: String;
```

2.6.59.3 Description

The **SSOiADUPN** property (**RunTime**) holds the authenticated user's Active Directory (AD) User Principal Name (UPN) for the current token. The value is extracted from the SSOiToken Property (TXWBSSOiToken Component).

2.6.60 SSOiLogonName Property (TRPCBroker Component)

2.6.60.1   Applies to

TRPCBroker Component

2.6.60.2   Declaration

```
property SSOiLogonName: String;
```

2.6.60.3   Description

The **SSOiLogonName** property (**RunTime**) holds the authenticated user's Active Directory (AD) user name for the current connection. The value is obtained from the SSOiLogonName Property (TXWBSSOiToken Component).

2.6.61 SSOiLogonName Property (TXWBSSOiToken Component)

2.6.61.1   Applies to

TXWBSSOiToken Component

2.6.61.2   Declaration

```
property SSOiLogonName: String;
```

2.6.61.3   Description

The **SSOiLogonName** property (**RunTime**) holds the authenticated user's Active Directory (AD) user name for the current token. The value is extracted from the SSOiToken Property (TXWBSSOiToken Component).

2.6.62 SSOiSECID (TRPCBroker Component)

2.6.62.1 Applies to

TRPCBroker Component

2.6.62.2 Declaration

```
property SSOiSECID: String;
```

2.6.62.3 Description

The **SSOiSECID** property (**RunTime**) holds the authenticated user's Identity and Access Management Security ID (SecID) for the current connection. The value is obtained from the SSOiSECID Property (TXWBSSOiToken Component).

2.6.63 SSOiSECID Property (TXWBSSOiToken Component)

2.6.63.1 Applies to

TXWBSSOiToken Component

2.6.63.2 Declaration

```
property SSOiSECID: String;
```

2.6.63.3 Description

The **SSOiSECID** property (**RunTime**) holds the authenticated user's Identity and Access Management Security ID (SecID) for the current token. The value is extracted from the **TXWBSSOiToken** component **SSOiToken** property.

2.6.64 SSOiToken Property (TRPCBroker Component)

2.6.64.1 Applies to

[TRPCBroker Component](#)

2.6.64.2 Declaration

```
property SSOiToken: String;
```

2.6.64.3 Description

The **SSOiToken** property (**RunTime**) holds a digitally-signed XML document (stored as a string) containing user attributes needed to authenticate a user into VistA. The value is obtained from the [SSOiToken Property (TXWBSSOiToken Component)](#).

2.6.65 SSOiToken Property (TXWBSSOiToken Component)

2.6.65.1 Applies to

[TXWBSSOiToken Component](#)

2.6.65.2 Declaration

```
property SSOiToken: String;
```

2.6.65.3 Description

The **SSOiToken** property (**RunTime**) holds a digitally-signed XML document (stored as a string) containing user attributes needed to authenticate a user into VistA for the current connection. The value is obtained by authenticating the user into the Identity and Access Management (IAM) Secure Token Service (STS) server using mutual Transport Layer Security (TLS) 2-factor authentication.

2.6.66 StandardName Property

2.6.66.1 Applies to

TVistaUser Class

2.6.66.2 Declaration

```
property StandardName: String;
```

2.6.66.3 Description

The **StandardName** property (**RunTime**) holds the user's standard name from the NEW PERSON (#200) file.

2.6.67 Title Property

2.6.67.1 Applies to

TVistaUser Class

2.6.67.2 Declaration

```
property Title: String;
```

2.6.67.3 Description

The **Title** property (**RunTime**) holds the user's title from the NEW PERSON (#200) file.

2.6.68 URLDetect Property

2.6.68.1 Applies to

TXWBRichEdit Component

2.6.68.2   Declaration

```
property URLDetect: Boolean;
```

2.6.68.3   Description

The **URLDetect** property (**DesignTime**) is used to create active ("live") links in an application. If this property is set to:

- **True**—URLs (**http:**, **mailto:**, **file:**, etc.) are shown in blue and underlined. If the user clicks on the URL, it opens the URL in the appropriate application.

- **False** (the default)—URLs appear as normal text and are *not* active.

2.6.69 User Property

2.6.69.1   Applies to

TRPCBroker Component

2.6.69.2   Declaration

```
property User: TVistaUser;
```

2.6.69.3   Description

The **User** property (**RunTime**) instance of the TVistaUser Class object is created during the **Create** process for the **TRPCBroker** instance. The object contains data on the current user and is updated as a part of the user authentication process.

REF: For examples of silent logon by passing Access and Verify codes, see the "Silent Login Examples" section.

2.6.70 UseSecureConnection Property

2.6.70.1 Applies to

[TRPCBroker Component](#)

2.6.70.2 Declaration

```
property UseSecureConnection: Constant;
```

2.6.70.3 Description

The **UseSecureConnection** property is used to specify whether SSH Tunneling is to be used when making the connection. It can be specified within an application or at run-time as a command line option.

To set the **UseSecureConnection** property within an application, use one of the following command lines:

- **secureNone**:

```
UseSecureConnection := 0;    //Do not use SSH tunneling (default)
```

- **secureAttachmate**:

```
UseSecureConnection := 1;    //Use Attachmate/Micro Focus Reflection SSH
```

- **securePlink**:

```
UseSecureConnection := 2;    //Use PuTTY Link (Plink) SSH
```

To set the **UseSecureConnection** property as a command line option, include either of the following:

- "**SSH**"—Attachmate®/Micro Focus® Reflection.
- "**PLINK**"—PuTTY Link (Plink).

2.6.71 Value Property

2.6.71.1 Applies to

[TParamRecord Class](#)

2.6.71.2 Declaration

```
property Value: String;
```

2.6.71.3 Description

The **Value** property (**DesignTime**) is used to pass either a single string or a single variable reference to the VistA M Server, depending on the **PType** (see [Table 11](#)).

2.6.71.4 Example

The program code in [Figure 48](#) demonstrates a couple of different uses of the [Value Property](#). Remember that each **Param[x]** element is really a [TParamRecord Class](#).

Figure 47: Value Property—Example

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   with brkrRPCBroker1 do begin
      RemoteProcedure := 'SET NICK NAME';
      {A variable reference}
      Param[0].Value := 'DUZ';
      Param[0].Ptype := reference;
      {A string}
      Param[1].Value := edtNickName.Text;
      Param[1].Ptype := literal;
      Call;
   end;
end;
```

2.6.72 VerifyCode Property

2.6.72.1 Applies to

TVistaLogin Class

2.6.72.2 Declaration

```
property VerifyCode: String;
```

2.6.72.3 Description

The **VerifyCode** property (**RunTime**) holds the Verify code for **lmAVCodes** mode of Silent Login. Like the AccessCode Property, the user's Verify code is also encrypted before it is transmitted to the VistA M Server.

> **REF:** For examples of silent logon by passing Access and Verify codes, see the "Silent Login Examples" section.

> **REF:** For more information on Verify codes, see the "Part 1: Sign-On/Security" section in the *Kernel 8.0 & Kernel Toolkit 7.3 Systems Management Guide*.

2.6.73 VerifyCodeChngd Property

2.6.73.1 Applies to

TVistaUser Class

2.6.73.2 Declaration

```
property VerifyCodeChngd: Boolean;
```

2.6.73.3 Description

The **VerifyCodeChngd** property (**RunTime**) indicates whether or not the user's Verify code has changed.

2.6.74 Vpid Property

2.6.74.1   Applies to

TVistaUser Class

2.6.74.2   Declaration

```
property Vpid: String;
```

2.6.74.3   Description

The **Vpid** property (**RunTime**) returns the Department of Veterans Affairs Person Identifier (VPID) value for the current user from the NEW PERSON (#200) file, if the facility has already been enumerated. If the facility has *not* been enumerated, the value returned is a **NULL** string.

3 Remote Procedure Calls (RPCs)

3.1 RPC Overview

A Remote Procedure Call (RPC) is a defined call to M code that runs on a VistA M Server. A client application, through the RPC Broker, can make a call to the VistA M Server and execute an RPC on the server. This is the mechanism through which a client application can:

- Send data to a VistA M Server.

- Execute code on a VistA M Server.

- Retrieve data from a VistA M Server.

An RPC can take optional parameters to do some task and then return either a single value or an array to the client application. RPCs are stored in the REMOTE PROCEDURE (#8994) File.

The following topics are covered:

- What Makes a Good RPC?

- Using an Existing M API

- Creating RPCs

- M Entry Point for an RPC:

  - Relationship between an M Entry Point and an RPC

  - First Input Parameter (Required)

  - Return Value Types

  - Input Parameters (Optional)

  - Examples

- RPC Entry in the Remote Procedure File:

  - REMOTE PROCEDURE (#8994) File

  - Key Fields for RPC Operation

  - RPC Version

  - Blocking an RPC

  - Cleanup after RPC Execution

  - Documenting RPCs

3.2 What Makes a Good RPC?

The following characteristics help to make a good remote procedure call (RPC):

- Silent calls (no I/O to terminal or screen, no user intervention required).

- Minimal resources required (passes data in brief, controlled increments).

- Discrete calls (requiring as little information as possible from the process environment).

- Generic as possible (different parts of the same package as well as other packages could use the same RPC).

3.3 Using an Existing M API

In some cases, an existing M API provides a useful M entry point for an RPC. As with any M entry point, you need to add the RPC entry that invokes the M entry point, in the [REMOTE PROCEDURE (#8994) File](#).

REF: See also: "[Relationship between an M Entry Point and an RPC](#)" section.

3.4 Creating RPCs

You can create your own custom RPCs to perform actions on the VistA M Server and to retrieve data from the VistA M Server. Then you can call these RPCs from your client application. Creating an RPC requires you to perform the following:

- Write and test the M entry point that is called by the RPC.

- Add the RPC entry that invokes the M entry point, in the [REMOTE PROCEDURE (#8994) File](#).

3.5   M Entry Point for an RPC

3.5.1   Relationship between an M Entry Point and an RPC

An RPC can be thought of as a wrapper placed around an M entry point for use with client applications. Each RPC invokes a single M entry point.

An M entry point has defined input and output values/parameters that are passed via the standard M invoking methods. An RPC, however, needs to do the following:

- Accept input from the Broker (i.e., passing data/parameters from the client application).

- Pass data to the M entry point in a specified manner.

- Receive values back from the M code in a pre-determined format.

- Pass M code output back through the Broker to the client application.

You can use the $$BROKER^XWBLIB function in M code to determine whether the code is being run in an environment where it was invoked by the Broker. This can help you use M code simultaneously for Broker and *non*-Broker applications.

You can use the RPCVersion Property to support multiple versions of an RPC. The RPCVersion Property Example shows you how to do this on the client and server sides.

3.5.2   First Input Parameter (Required)

The RPC Broker always passes a variable by reference in the first input parameter to your M routine. It expects results (one of five Return Value Types) to be returned in this parameter. You *must* always set some return value into that first parameter before your routine returns.

3.5.3   Return Value Types

There are five RETURN VALUE TYPEs for RPCs as shown in Table 13. You should choose a return value type that is appropriate to the type of data your RPC needs to return. For example, to return the **DUZ**, a return value type of SINGLE VALUE would be appropriate.

The RETURN VALUE TYPE you choose determines what values you should set into the return value parameter of your M entry point.

You should always set *some* value into the Return Value parameter of the M entry point, even if your RPC encounters an error condition.

The RPC settings in [Table 13](#), combined with your M entry point, determine how data is returned to your client application:

Table 13: RPC Settings to Determine How Data is Returned

RPC Return Value Type	How M Entry Point Should Set the Return Parameter	RPC Word Wrap On Setting	Values returned in Client Results

Single Value	Set the return parameter to a single value. For example:
	
	`TAG(RESULT) ;`
	`S RESULT="XWBUSER,ONE"`
	`Q`
		No effect	Value of parameter, in Results[0].

Array	Set an array of strings into the return parameter, each subscripted one level descendant. For example:
	
	`TAG(RESULT) ;`
	`S RESULT(1)="ONE"`
	`S RESULT(2)="TWO"`
	`Q`
	
	If your array is large, consider using the GLOBAL ARRAY return value type, to avoid memory allocation errors.
		No effect	Array values, each in a Results item.

Word Processing	Set the return parameter the same as you set it for the ARRAY type. The only difference is that the WORD WRAP ON setting affects the WORD PROCESSING return value type.	True False	Array values, each in a Results item. Array values, all concatenated into Results[0].
Global Array	Set the return parameter to a closed global reference in **^TMP**. The global's data nodes are traversed using **$QUERY**, and all data values on global nodes descendant from the global reference are returned. This type is especially useful for returning data from VA FileMan WORD PROCESSING type fields, where each line is on a **0**-subscripted node. ⚠ **CAUTION: The global reference you pass is killed by the Broker**	True False	Array values, each in a Results item. Array values, all concatenated into Results[0].

RPC Return Value Type	How M Entry Point Should Set the Return Parameter	RPC Word Wrap On Setting	Values returned in Client Results

	at the end of RPC Execution as part of RPC cleanup. Do *not* pass a global reference that is *not* in ^TMP or that should *not* be killed.
	
	This type is useful for returning large amounts of data to the client, where using the ARRAY type can exceed the symbol table limit and crash your RPC.
	
	For example, to return signon introductory text you could do:
	
	`TAG(RESULT);`
	`M ^TMP("A6A",$J)=`
	`^XTV(8989.3,1,"INTRO")`
	`;this node not needed`
	`K ^TMP("A6A",$J,0)`
	`S RESULT=$NA(^TMP("A6A",$J))`
	`Q`		

Global Instance	Set the return parameter to a closed global reference.
	
	For example, the following code returns the whole **0th** node from the NEW PERSON (#200) file for the current user:
	
	`TAG(RESULT) ;`
	`S RESULT=$NA(^VA(200,DUZ,0))`
	`Q`
		No effect	Value of global node, in Results[0].

NOTE: In the M code called by an RPC, you can use the $$RTRNFMT^XWBLIB function to change the RETURN VALUE TYPE of an RPC on-the-fly.

3.5.4   Input Parameters (Optional)

The M entry point for an RPC can optionally have input parameters (i.e., beyond the first parameter, which is always used to return an output value). The client passes data to your M entry point through these parameters.

The client can send data to an RPC (and therefore the entry point) in one of the three Param Property types in Table 14:

Table 14: Param PType Value Types

Param PType	Param Value
literal	Delphi string value; passed as a string literal to the VistA M Server.
reference	Delphi string value; treated on the VistA M Server as an M variable name and resolved from the symbol table at the time the RPC executes. ⚠️ **CAUTION: For enhanced security reasons, the reference parameter type may be deprecated and removed in subsequent updates to the BDK.**
list	A single-dimensional array of strings in the Mult Property of the Param Property; passed to the VistA M Server where it is placed in an array. String subscripting can be used.

The type of the input parameters passed in the Param Property of the TRPCBroker Component determines the format of the data you *must* be prepared to receive in your M entry point.

3.5.5   Examples

The examples in Figure 49 and Figure 50 illustrate sample M code that could be used in simple RPCs:

1. The example in Figure 49 takes two numbers and returns their sum:

Figure 48: RPCs—Sample M Code to Add Two Numbers

```
SUM(RESULT,A,B)  ;add two numbers
     S RESULT=A+B
     Q
```

2. The example in [Figure 50](#) receives an array of numbers and returns them as a sorted array to the client:

Figure 49: RPCs—Sample M Code that Receives an Array of Numbers and Returns them as a Sorted Array to the Client

```
SORT(RESULT,UNSORTED)  ;sort numbers
    N I
    S I=""
    F  S I=$O(UNSORTED(I)) Q:I=""  S RESULT(UNSORTED(I))=UNSORTED(I)
    Q
```

3.6   RPC Entry in the Remote Procedure File

3.6.1   REMOTE PROCEDURE (#8994) File

The RPC Broker consists of a single global that stores the REMOTE PROCEDURE (#8994) file:

Table 15: Remote Procedure File Information

File #	File Name	Global Location
8994	REMOTE PROCEDURE	^XWB(8994,

This is the common file used by all applications to store *all* remote procedure calls accessed via the Broker. All RPCs used by any site-specific client/server application software using the RPC Broker interface *must* be registered and stored in this file.

This file is used as a repository of server-based procedures in the context of the Client/Server architecture. By using the Remote Procedure Call (RPC) Broker, applications running on client workstations can invoke (call) the procedures in this file to be executed by the server and the results are returned to the client application.

NOTE: The RPC (#19.05) subfield of the OPTION (#19) file points to RPC (#.01) field of the REMOTE PROCEDURE (#8994) File.

3.6.2   Key Fields for RPC Operation

After the M code is complete, you need to add the RPC to the REMOTE PROCEDURE (#8994) File. Table 16 lists the fields in the REMOTE PROCEDURE (#8994) file that are key to the correct operation of an RPC:

Table 16: Remote Procedure File—Key Fields for RPC Operation

Field Name	Required?	Description
NAME (#.01)	Yes	The name that identifies the RPC (this entry should be namespaced in the package namespace).
TAG (#.02)	Yes	The tag at which the remote procedure call begins.
ROUTINE (#.03)	Yes	The name of the routine that should be invoked to start the RPC.
WORD WRAP ON (#.08)	No	Affects GLOBAL ARRAY and WORD PROCESSING return value types only. If set to **False**, all data values are returned in a single concatenated string in Results[0]. If set to **True**, each array node on the M side is returned as a distinct array item in Results.
RETURN VALUE TYPE (#.04)	Yes	This can be one of five types: • SINGLE VALUE • ARRAY • WORD PROCESSING • GLOBAL ARRAY • GLOBAL INSTANCE This setting controls how the Broker processes an RPC's return parameter (see "Return Value Types").

3.6.3   RPC Version

The VERSION field of the REMOTE PROCEDURE (#8994) File indicates the version number of an RPC installed at a site. The field can be set either by an application developer and exported by KIDS or by a site manager using VA FileMan.

Applications can use XWB IS RPC AVAILABLE or XWB ARE RPCS AVAILABLE to check the availability of a version of an RPC on a server. This is especially useful for RPCs run remotely, as the remote server may not have the latest RPC installed.

3.6.4   Blocking an RPC

The INACTIVE field of the REMOTE PROCEDURE (#8994) File allows blocking of RPCs. The blocking can apply to local access (users directly logged into the site) or remote access (users logged on to a different site) or both. The field can be set either by an application developer and exported by Kernel Installation & Distribution System (KIDS) or by a site manager using VA FileMan.

REF: For more information on remote access, see the "Running RPCs on a Remote Server" section.

3.6.4.1   Value in INACTIVE field

- 1 = Completely unusable
- 2 = Unusable locally
- 3 = Unusable remotely

3.6.5   Cleanup after RPC Execution

The Broker uses XUTL^XUSCLEAN to clean up globals upon application termination.

In addition, there is an RPC RETURN VALUE TYPE (see "Return Value Types"), GLOBAL ARRAY, where the application RPC returns a closed form global reference, for example:

```
^TMP("EKG",666333551)
```

The Broker kills the data for the global reference for this type of RPC at the end of RPC execution.

3.6.6   Documenting RPCs

Each individual application development team is responsible for identifying and providing documentation for all object components, classes, and remote procedure calls they create. Other developers using these components need to know what RPCs are called, because they need to register them with their applications.

RPCs should be documented in the DESCRIPTION (#1) field in the REMOTE PROCEDURE (#8994) File for those RPCs installed on your system. This gives you the capability of generating a catalogue of RPCs from the REMOTE PROCEDURE (#8994) file.

3.6.6.1 Delphi Component Library and Sample RPCs

In the future, an Enterprise library of services, object components, classes, Application Programming Interfaces (APIs), Remote Procedure Calls (RPCs), etc. that are in use and available to the development community at large may be available. The essential benefit of this type of library is the promotion of object re-use; thereby, enhancing development productivity, application consistency, and quality assurance. Therefore, it could contain a wide variety of services, object components, APIs, classes, RPCs, etc. from many VistA software applications.

The immediate intent is to classify and catalogue all of the object classes in use (including the standard Delphi classes), and to make the catalogue available to all interested parties.

3.7 Executing RPCs from Clients

3.7.1 How to Execute an RPC from a Client

1. If your RPC has any input parameters beyond the mandatory First Input Parameter (Required), set a **Param** node in the TRPCBroker Component's Param Property for each. For each input parameter, set the following sub-properties:

   - Value Property

   - PType Property (**literal**, **list**, or **reference**)

   If the parameter's PType Property is list, instead of specifying a value, set a list of values in the Mult Property.

   Figure 51 is an example of some settings of the Param Property:

Figure 50: RPCs—Param Property—Example Setting a List of Values

```
brkrRPCBroker1.Param[0].Value := '03/31/14';
brkrRPCBroker1.Param[0].PType := literal;
brkrRPCBroker1.Param[1].Mult['"NAME"'] := 'XWBUSER, ONE';
brkrrpcbroker1.param[1].mult['"ssn"'] :="000-45-6789" ;/pre="">
brkrRPCBroker1.Param[1].PType := list;
```

2. Set the TRPCBroker Component's RemoteProcedure Property to the name of the RPC to execute:

```
brkrRPCBroker1.RemoteProcedure:='A6A LIST'
```

3. Invoke the TRPCBroker Component's Call Method to execute the RPC. All calls to the Call Method should be done within an exception handler **try...except** statement, so that all communication errors (which trigger the EBrokerError exception) can be trapped and handled. For example:

Figure 51: Error Handling—Example of a "try...except" Statement

```
try
    brkrRPCBroker1.Call;
except
    On EBrokerError do
        ShowMessage('A problem was encountered communicating with the
server.');
end;
```

4. Any results returned by your RPC are returned in the TRPCBroker Component's Results Property. Depending on how you set up your RPC, results are returned either in a single node of the Results Property (**Results[0]**), or in multiple nodes of the Results Property.

NOTE: You can also use the lstCall Method and strCall Method to execute an RPC. The main difference between these methods and the Call Method is that the lstCall Method and the strCall Method do *not* use the Results Property, instead returning results into a location you specify.

3.7.2 RPC Security: How to Register an RPC

Security for RPCs is handled through the RPC registration process. Each client application *must* create a context for itself, which checks if the application user has access to a "**B**"-type option in the Kernel menu system. Only RPCs assigned to that option can be run by the client application.

To enable your application to create a context for itself:

1. Create a "**B**"-type option in the OPTION (#19) file for your application.

    i   **NOTE:** The OPTION TYPE "**B**" represents a **B**roker client/server type option.

2. In the RPC multiple for this option type, add an entry for each RPC that your application calls. The fields listed in Table 17 can be set up for each RPC in your option:

Table 17: RPC Multiple Fields for "B"-Type Options

Field Name (#)	Entry	Description
RPC (#.01)	Required	This field is used to enter a pointer to the REMOTE PROCEDURE (#8994) File. This field links the remote procedure call in the REMOTE PROCEDURE (#8994) File to the package option.
RPCKEY (#1)	Optional	This field is used to restrict the use of a remote procedure call to a particular package option. The RPCKEY field is a free-text pointer to the SECURITY KEY (#19.1) file.
RULES (#2)	Optional	This field is used to enter M code that is executed when an RPC request is made to verify whether the request should be honored.

3. When you export your package using Kernel Installation and Distribution System (KIDS), export both your RPCs and your package option. KIDS automatically associates the RPCs with the package option.

4. Your application *must* create a context for itself on the VistA M Server, which checks access to RPCs. In the initial code of your client application, make a call to the CreateContext Method of your TRPCBroker Component. Pass your application's "**B**"-type option's name as a parameter. For example:

```
if not brkrRPCBroker1.CreateContext(option_name) then
    Application.Terminate;
```

5. If the CreateContext Method returns **True**, only those RPCs designated in the RPC multiple of your application option is permitted to run.

6. If the CreateContext Method returns **False**, you should terminate your application (if you do not, your application runs but you get errors every time you try to access an RPC).

7. End-users of your application *must* have the "**B**"-type option assigned to them on one of their menus, in order for the [CreateContext Method](#) to return **True**. This allows system managers to control access to client applications.

3.7.3   RPC Limits

The following is a list of various constants, maximum, and minimum parameters associated with the use of the RPC Broker:

- [Maximum Number of Parameters](#) that can be passed to the VistA M Server.

- [Maximum Size of Array](#) that can be passed to the VistA M Server.

- [Maximum Size of Data](#) that can be received in the VistA Graphical User Interface (GUI) application from the VistA M Server.

- [RPC Time Limits](#).

3.7.4   RPC Time Limits

A public **READ/WRITE** property (i.e., [RPCTimeLimit Property](#)) allows the application to change the network operation timeout prior to a call. This can be useful during times when it is known that a certain RPC, by its nature, can take a significant amount of time to execute. The value of this property is an integer that *cannot* be less than **30 seconds** nor greater than **32767** seconds. Care should be taken when altering this value, since the network operation blocks the application until the operation finishes or the timeout is triggered.

There is also a server time limit for how long to stay connected when the client does *not* respond.

3.7.5   Maximum Size of Data

The VistA M Server can transmit very large buffers of data back to the Microsoft® Windows client. The Windows client receives the returned data from an RPC into a 32-bit PASCAL string. RPCs can be written on the VistA M Server so that they store their results in an M GLOBAL structure, which can span RAM and disk storage media. This GLOBAL storage could be quite large depending on the assigned system quotas to the VistA M Server process. The return of the RPC can deliver this quantity to the Windows client. The actual limit depends on the capacity that the Microsoft® Windows operating system allows the client to process. Tests on a 32-megabyte RAM system have allowed buffers of several megabytes of data to be transmitted from the VistA M Server to the Microsoft® Windows client.

3.7.6   Maximum Number of Parameters

The remote procedure calls (RPCs) become M **DO** procedures on the VistA M Server. Since RPCs are communicated to the VistA M Server through a message mechanism, additional information is included with the message.

Parameters are processed as PASCAL short strings with a maximum of **255** characters. Each parameter is encoded with a three-character length plus a type character. Therefore, every parameter occupies length (parameter) + **four**. The maximum transmission at this time is **240** characters, since additional header information is present with every RPC.

A theoretical maximum, where every parameter was length **1** would give number of parameters = **240/5** or **48** parameters. A single parameter (e.g., a long string) could not exceed **240 - 4**, or **236** characters. Future support will be based on the PASCAL 32-bit string, which can, theoretically, reach **2 GB**. Limitations on the VistA M Server still limit this to far less, however.

3.7.7   Maximum Size of Array

Although approximately only 240 characters can be sent to the VistA M Server as call parameters, a single array parameter can be passed in with greater capacity. The RPC can carry both literal and array parameters except that literal parameters are placed first and the single array last in order. Arrays are instantiated at the VistA M Server and are stored in a local array format. The maximum size is dependent on the symbol space available to the VistA M Server process. The index size and the value size are subject to limitations; the index and value each *cannot* exceed **255 - 3**, or **252** characters approximately for each individual array elements.

At the time of this writing, **30** to **40** K arrays have easily been passed to the VistA M Server in a single RPC call.

3.7.8   RPC Broker Example (32-Bit)

The RPC Broker Example sample application provided with the BDK (i.e., **BrokerExample.exe**) demonstrates the basic features of developing RPC Broker applications, including:

- Connecting to a VistA M Server.

- Creating an application context (see CreateContext Method).

- Using the GetServerInfo Function.

- Displaying the VistA splash screen (see "VistA Splash Screen Procedures" section).

- Setting the TRPCBroker Param Property for each Param **Ptype** (**literal**, **reference**, **list**).

- Calling RPCs with the Call Method.

- Calling RPCs with the lstCall Method and strCall Method.

   REF: The **BrokerExample.exe** and client source code files for the **BrokerExample.exe** application are located in the following directory:

      BDK32\Samples\BrokerEx

4 RPC Broker: Developer Tools

In addition to the RPC Broker components, the Broker Development Kit (BDK) provides other development tools, including:

- [Application Programming Interface (API)](#)
- [Functions, Methods, and Procedures](#)
- [Running RPCs on a Remote Server](#)
- [Deferred RPCs](#)

4.1 Application Programming Interface (API)

4.1.1 Overview

RPC Broker uses Kernel Application Programming Interfaces (APIs) and Remote Procedure Calls (RPCs) for most user authentication (Signon) and authorization (Security) actions. However, there are a few APIs and RPCs unique to Broker connections that are used by a variety of interfaces into VistA. For example, VistALink and VistA Services Assembler both use the CHKPRMIT^XWBSEC API to determine if a user has authorization to use a particular Remote Procedure.

The RPC Broker software provides the following APIs on the VistA M Server for use in RPC code:

- [$$BROKER^XWBLIB: Test for Broker Context](#)
- [$$RTRNFMT^XWBLIB(): Change RPC Return](#)
- [CHKPRMIT^XWBSEC(): Check Permissions](#)
- [CRCONTXT^XWBSEC(): Create Context](#)
- [SET^XWBSEC(): Set the State Variable](#)

4.1.2 $$BROKER^XWBLIB: Test for Broker Context

Reference Type:	Supported
Category:	RPC Broker
ICR #:	2198
Description:	Use the $$BROKER^XWBLIB extrinsic function in the M code called by an RPC to determine if the current process is being executed by the RPC Broker.
Format:	`$$BROKER^XWBLIB`
Input Parameters:	none.

Output: return value: Returns:

- **1**—If the current process is being executed by the Broker.

- **0**—If the current process is *not* being executed by the Broker.

4.1.2.1   Example

```
I $$BROKER^XWBLIB D .; broker-specific code
```

4.1.3 $$RTRNFMT^XWBLIB(): Change RPC Return Type

Reference Type: Supported

Category: RPC Broker

ICR #:   2238

Description: Use the $$RTRNFMT^XWBLIB extrinsic function in the M code called by an RPC to change the return value type that the RPC returns on-the-fly.

Format: $$RTRNFMT^XWBLIB(type,wrap)

Input Parameters: type: (required) Set this to the RETURN VALUE TYPE to which you want to change the RPC's setting. Set it to one of the following numeric or free text values:

Table 18: $$RTRNFMT^XWBLIB: The type Input Parameter Values

Numeric	Free Text
1	SINGLE VALUE
2	ARRAY
3	WORD PROCESSING
4	GLOBAL ARRAY
5	GLOBAL INSTANCE

wrap: (required) Set value to:

- **1**—Set RPC's WORD WRAP ON setting to **True**.

- **0**—Set RPC's WORD WRAP ON setting to **False**.

Output: return value: Returns:

- **0**—If the return value type could *not* be changed.
- **numeric code**—Representing the return value type to which the RPC is changed.

4.1.3.1   Example

```
I '$$RTRNFMT^XWBLIB("ARRAY",1) D .; branch to code if cannot change RPC type
```

4.1.4 CHKPRMIT^XWBSEC(): Check Permissions

Reference Type: Controlled Subscription

Category: Signon/Security

ICR #:   4053

Description: The CHKPRMIT^XWBSEC API checks to see if the remote procedure is permitted to run. It checks for:

- User-held security keys
- User context (context option)
- Out-of-order settings
- RPC version

If the user is an Application Proxy, it checks to see if Application Proxy access to the remote procedure is permitted.

Some remote procedures are allowed in any context:

- **XUS BSE TOKEN**
- **XUS CVC**
- **XUS GET USER INFO**
- **XUS GET TOKEN**
- **XUS IAM BIND USER**
- **XUS KAAJEE GET USER INFO**
- **XUS KAAJEE LOGOUT**
- **XUS KEY CHECK**
- **XUS SET VISITOR**
- **XWB CREATE CONTEXT**

- **XWB IM HERE**
- **XWB IS RPC AVAILABLE**
- **XWB RPC LIST**

All Kernel "**XUS**" and RPC Broker "**XWB**" remote procedures are allowed in "XUS SIGNON" context.

Format:　　　　　`CHKPRMIT^XWBSEC(xwbrp)`

Make sure to perform the following steps before calling this API:

1. NEW all *non*-namespaced variables.

2. Set all input variables.

3. Call the API.

Input Parameters:	xwbrp:	(required) This is the name of the remote procedure to look up in the [REMOTE PROCEDURE (#8994) File](#).
Output Variables:	**XWBSEC**:	Returns:

- **""—NULL** in **XWBSEC** environment variable if the remote procedure is permitted to run.

- **Error Message**—If the remote procedure is *not* permitted. If an error is returned, then the **XWBSEC** environment variable is set to return the same error message.

ⓘ NOTE: XWBSEC is an environment variable, so that the information is included in the error trap should a subsequent processing error occur.

4.1.5　CRCONTXT^XWBSEC(): Create Context

Reference Type:	Controlled Subscription
Category:	Signon/Security
ICR #:	4053
Description:	The CRCONTXT^XWBSEC API creates a valid RPC Broker user context.
Format:	`CRCONTXT^XWBSEC(result,option)`

Make sure to perform the following steps before calling this API:

1. NEW all *non*-namespaced variables.
2. Set all input variables.
3. Call the API.

Input Parameters: option: (required) This is the encrypted name of the **B**-type menu option to look up in the OPTION (#19) file. If the option has been assigned to the user as a SECONDARY MENU OPTION, then the context can be set.

> **ℹ** **REF:** For more information on context options, see the "RPC Security: How to Register an RPC" section.

Output: result: Returns:

- **1**—Successful: If the context is successfully created.

- **0 (and error message)**—Unsuccessful: If the context could *not* be created.

Output Variables: **XWBSEC**: **XWBSEC** is an environment variable, so that the information is included in the error trap should a subsequent processing error occur. If an error is returned, then the **XWBSEC** environment variable is set to return the same error message.

4.1.6 SET^XWBSEC(): Set the State Variable

Reference Type: Controlled Subscription

Category: Signon/Security

ICR #: 4053

Description: The SET^XWBSEC API sets the **XWBSTATE** environment variable (array) to contain a passed in value. This is generally used to record the current status of a Broker connection for monitoring or testing.

> **ℹ** **NOTE: XWBSTATE** is an environment variable, so that the information is included in the error trap should a subsequent processing error occur.

Format: `SET^XWBSEC(%,value)`

Make sure to perform the following steps before calling this API:

1. NEW all *non*-namespaced variables.

2. Set all input variables.

3. Call the API.

Input Parameters:	value:	(required) This is FREE TEXT state value to be added to the **XWBSTATE** array.
Output Variables:	**%**:	Returns this variable equal to the **value** input parameter and also sets the **XWBSTATE(%)** environment variable equal to the **value** input parameter.

4.2 Functions, Methods, and Procedures

4.2.1 Overview

Additional functions, methods, and procedures include:

- XWB CREATE CONTEXT

- XWB GET BROKER INFO

- XWB GET VARIABLE VALUE

- XWB IM HERE

- M Emulation Functions

- Encryption Functions

- CheckCmdLine Function

- GetServerInfo Function

- GetServerIP Function

- ChangeVerify Function

- SilentChangeVerify Function

- StartProgSLogin Method

- VistA Splash Screen Procedures

4.2.2 XWB CREATE CONTEXT

The **XWB CREATE CONTEXT** RPC (distributed with the RPC Broker) is used to establish the context on the VistA M Server, which is checked by the Broker Listener before executing any other remote procedure. Since context is nothing more than a client/server "B"-type option in the OPTION (#19) file, standard MenuMan security is applied in establishing a context. Therefore, a context option can be granted to users exactly the same way as regular options are done using MenuMan.

A context *cannot* be established for the following reasons:

- User has no access to that option

- Option is temporarily out of order

An application can switch from one context to another as often as it needs. Each time a context is created the previous context is overwritten.

Pass the encrypted (using the Encrypt function in the **XWBHash** unit) OPTION name in **Param[0]**.Value, and the type (**literal**) in **Param[0].PType**. The **TRPCBroker CreateContext** method sets up these values and calls the RPC for you. Also, the current context of your user *must* give them permission to execute the **XWB GET VARIABLE VALUE** RPC (it *must* be included in the RPC multiple of the "**B**"-type option registered with the CreateContext Method).

> **NOTE: XWB CREATE CONTEXT** is a Private RPC. If an application uses the **TRPCBroker CreateContext** method, a subscription is *not* needed as the RPC is owned by the RPC Broker package.

4.2.3 XWB GET BROKER INFO

The **XWB GET BROKER INFO** RPC (distributed with the RPC Broker) is used to return information regarding setup and parameters of the Broker Listener on the VistA M Server. The RPC currently returns only the timeout period for handler **READ**s.

There are no input parameters.

> **NOTE: XWB GET BROKER INFO** is currently used only within the RPC Broker package and has *not* been made available to other applications.

4.2.4   XWB GET VARIABLE VALUE

You can call the **XWB GET VARIABLE VALUE** RPC (distributed with the RPC Broker) to retrieve the value of any M variable in the VistA M Server environment. Pass the variable name in **Param[0]**.Value, and the type (reference) in **Param[0].PType**. Also, the current context of your user *must* give them permission to execute the **XWB GET VARIABLE VALUE** RPC (it *must* be included in the RPC multiple of the "**B**"-type option registered with the CreateContext Method).

WARNING: The XWB GET VARIABLE VALUE RPC is intended to retrieve the value of an M variable (e.g., DUZ or DTIME). This is the only supported function of this RPC. If the RPC is used in an application in any other way than the way it was intended, results can be unpredictable, and the application could cease to function in future software patches.

NOTE: XWB GET VARIABLE VALUE is available only on a Controlled Subscription basis.

4.2.4.1   Example

Figure 53 is an example of the **XWB GET VARIABLE VALUE** RPC:

Figure 52: XWB GET VARIABLE VALUE RPC—Example

```
brkrRPCBroker1.RemoteProcedure := 'XWB GET VARIABLE VALUE';
brkrRPCBroker1.Param[0].Value :='DUZ';
brkrRPCBroker1.Param[0].PType := reference;
try
   brkrRPCBroker1.Call;
except
   On EBrokerError do
      ShowMessage('Connection to server could not be established!');
end;
ShowMessage('DUZ is '+brkrRPCBroker1.Results[0]);
```

4.2.5   XWB IM HERE

The **XWB IM HERE** RPC (distributed with the RPC Broker) is used to establish continued existence of the client connection to the VistA M Server (keepalive). It resets the server **READ** timeout. The RPC currently returns a meaningless value of "**1**", which is *not* used on the client.

There are no input parameters.

NOTE: XWB IM HERE is a Private RPC. If an application uses the Delphi RPC Broker Development Kit (BDK), a subscription is *not* needed as the RPC is owned by the RPC Broker package.

4.2.6   M Emulation Functions

4.2.6.1   Piece Function

The Piece function is a scaled down Pascal version of M's **$PIECE** function. It is declared in **MFUNSTR.PAS**.

```
function Piece(x: string; del: string; piece: integer) : string;
```

4.2.6.2   Translate Function

The Translate function is a scaled down Pascal version of M's **$TRANSLAT**E function. It is declared in **MFUNSTR.PAS**.

```
function Translate(passedString, identifier, associator: string): string;
```

4.2.6.3   Examples

4.2.6.3.1   Piece Function

```
Piece3Str:=piece('123^456^789','^',3);
```

4.2.6.3.2   Translate Function

```
hiStr:=translate('HI','ABCDEFGHI','abcdefghi');
```

4.2.7   Encryption Functions

Kernel and the RPC Broker provide encryption functions that can be used to encrypt messages sent between the client and the server.

4.2.7.1   In Delphi

Include XWBHash in the "**uses**" clause of the unit in which you are encrypting or decrypting.

Function prototypes are as follows:

```
function Decrypt(EncryptedText: string): string;

function Encrypt(NormalText: string): string;
```

4.2.7.2   On the VistA M Server

4.2.7.2.1   Encrypt Function

To encrypt a string, do the following:

```
>S CIPHER=$$ENCRYP^XUSRB1("Hello world!") W CIPHER
/U'llTG~TVl&f-
```

4.2.7.2.2   Decrypt Function

To decrypt a string, do the following:

```
>S PLAIN=$$DECRYP^XUSRB1(CIPHER) W PLAIN
Hello world!
```

These encryption functions can be used for any communication between the client and the server where encryption is desired.

4.2.8   CheckCmdLine Function

With Patch XWB*1.1*13, the **CheckCmdLine** method was changed from a procedure to a function with a Boolean return value.

```
function CheckCmdLine(SLBroker: TRPCBroker): Boolean;
```

4.2.8.1   Argument

Table 19: CheckCmdLine Function—Argument

Argument	Description
SLBroker	The instance of the Broker with which information on the command line should be used, and to be used for the connection, if a Silent Login is possible.

4.2.8.2   Result

The return value indicates whether the information on the command line was sufficient to connect the RPCBroker instance to the specified Server/ListenerPort (see Server Property and ListenerPort Property).

- **True**—Broker is connected to the VistA M Server.

- **False**—Broker is *not* connected to the VistA M Server.

4.2.9 GetServerInfo Function

The GetServerInfo function retrieves the end-user's selection of server and port to which to connect. Use this function to set a TRPCBroker Component's Server, and ListenerPort (see Server Property and ListenerPort Property) to reflect the end-user's choice before connecting to the VistA M Server.

If there is more than one server/port from which to choose, GetServerInfo displays an application window that allows users to select a service to connect:

Figure 53: GetServerInfo Function—Connect To Dialogue
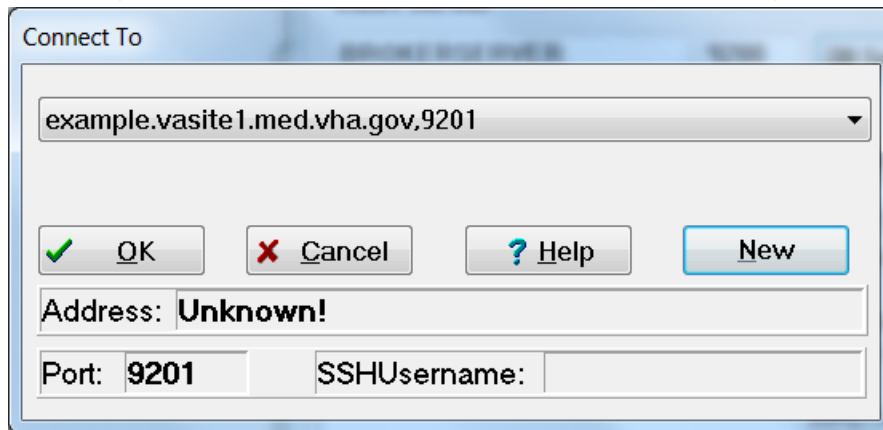
4.2.9.1 Syntax

```
function GetServerInfo(var Server, Port: string): integer;
```

ⓘ   **NOTE:** The unit is the RPCConf1 Unit.

The GetServerInfo function handles the following scenarios:

- If there are no values for server and port in the Microsoft Windows Registry, GetServerInfo does *not* display its dialogue window, and the automatic default values returned are **BROKERSERVER/<REDACTED>**. **GetServerInfo** returns **mrOK**.

- If exactly one server and port entry is defined in the Microsoft Windows Registry, GetServerInfo does *not* display its dialogue window. The values in the single Microsoft Windows Registry entry are returned to the calling application, with no user interaction. GetServerInfo returns **mrOK**.

- If more than one server and port entry exists in the Microsoft Windows Registry, the dialogue window is displayed. The only time that passed in server and port values are returned to the calling application is if the user selects **Cancel**. However, if a user selects an entry and presses **OK**, the server and port parameters are changed and returned to the

calling application. GetServerInfo returns **mrOK** if the user selected **OK**, or **mrCancel** if the user selected **Cancel**.

- A typical Microsoft® Windows Registry entry for a VistA M Server contains the following characteristics:

  o Located in either of the following registries:

    – **HKEY_LOCAL_MACHINE (HKLM)**—Registry for accessibility to *all users* of a computer.

    – **HKEY_CURRENT_USER (HKCU)**—Registry for accessibility to a *single user*.

  o Located in the following registry subdirectory:

    Software\Vista\Broker\Servers

  o String-type registry entry, where Value name contains the IP address or Fully Qualified Domain Name (FQDN) of the server, followed by the port number of the RPC Broker listener on the server. The information should be separated by a comma. For example:

    myserver.<REDACTED>.va.gov,<REDACTED>

  o The optional Value data contains the SSHUsername to be used to establish an encrypted connection to the VistA M Server.

REF: For a demonstration using the Broker and GetServerInfo function, run the RPC Broker Example (32-Bit) (i.e., **BrokerExample.exe**); located in the following directory:

BDK32\Samples\BrokerEx

4.2.9.2 Example

Figure 55 is an example of the GetServerInfo function:

Figure 54: GetServerInfo Function—Example

```
procedure TForm1.btnConnectClick(Sender: TObject);
var
    strServer, strPort, strSSHUsername: string;
begin
    if GetServerInfo(strServer, strPort, strSSHUsername)<> mrCancel then
    begin
    {getsvrinfo begin}
      brkrRPCBroker1.Server := strServer;
      brkrRPCBroker1.ListenerPort := StrToInt(strPort);
      brkrRPCBroker1.SSHUser := strSSHUsername;
      brkrRPCBroker1.Connected := True;
    {getsvrinfo end}
    end;
end;
```

ℹ️ **REF:** For a demonstration using the Broker and GetServerInfo function, run the RPC Broker Example (32-Bit) (i.e., **BrokerExample.exe**); located in the following directory:

BDK32\Samples\BrokerEx

4.2.10 GetServerIP Function

The GetServerIP function provides a means for determining the Internet Protocol (IP) address for a specified VistA M Server address. The value returned is a string containing the IP address, or if it could *not* be resolved, the string "Unknown!"

```
function GetServerIP(ServerName: string): string;
```

4.2.10.1 Example

Figure 56 is an example of the GetServerIP function:

Figure 55: GetServerIP Function—Example

```
   // include the unit RpcConf1 in the Uses clause
   // An edit box on the form is assumed to be named edtIPAddress
   // Another edit box (edtInput) is used to input a desired server name

uses RpcConf1;

procedure Tform1.Button1Click(Sender: TObject);
var
   ServerName: string;
begin
   ServerName := '<REDACTED>.va.gov';
   edtIPAddress.Text := GetServerIP(edtInput.Text);
     // For <REDACTED>.va.gov returns '999.999.9.99'
     // For garbage returns 'Unknown!'
end;
```

⚠️ **CAUTION: The GetServerIP function has limited use in a modern TCP/IP network, as multiple IP addresses can be assigned to a single server. It is expected to be deprecated and replaced in future releases with a function that returns a list of IP addresses.**

4.2.11 ChangeVerify Function

The ChangeVerify function can be used to provide the user with the ability to change his/her Verify code.

```
function ChangeVerify(RPCBroker: TRPCBroker): Boolean;
```

4.2.11.1 Argument

Table 20: ChangeVerify Function—Argument

Argument	Description
RPCBroker	The Broker instance for the account on which the Verify code is to be changed.

4.2.11.2 Result

The return value indicates whether the user changed their Verify code or not.

- **True**—User changed their Verify code.

- **False**—User did *not* change their Verify code.

4.2.12 SilentChangeVerify Function

The SilentChangeVerify function can be used to change the Verify code for a user without any dialogue windows being displayed.

```
function SilentChangeVerify(RPCBroker: TRPCBroker; OldVerify,
 NewVerify1, NewVerify2: String; var Reason: String): Boolean;
```

4.2.12.1 Arguments

Table 21: SilentChangeVerify Function—Arguments

Argument	Description
RPCBroker	The current instance of the Broker for the account for which the Verify code is to be changed.
OldVerify	The string representing the current Verify code for the user.
NewVerify1	A string representing the new Verify code for the user.
NewVerify2	A second independent entry for the string representing the new Verify code for the user.
Reason	A string that on return contains the reason why the Verify code was *not* changed (if the result value is **False**).

4.2.12.2 Result

The return value indicates whether the Verify code was successfully changed or not:

- **True**—Verify code was successfully changed.

- **False**—Verify code was *not* successfully changed. The reason for the failure is in the **Reason** argument.

4.2.13 StartProgSLogin Method

The **StartProgSLogin** method can be used to initiate another program with information sufficient for a Silent Login, or it can be used to launch a standalone program that does *not* use a TRPCBroker Component connection. If the program is being used to launch another executable with information for a Silent Login, it is *recommended* that the CheckCmdLine Function be used

in the program being launched (since this function uses the command line information to make a Silent Login if possible).

```
procedure StartProgSLogin(const ProgLine: String; ConnectedBroker: TRPCBroker);
```

4.2.13.1 Arguments

Table 22: StartProgSLogin Method—Arguments

Argument	Description
ProgLine	This is the command line that should be used as the basis for launching the executable. It contains the executable (and path, if not in the working directory or in the system path) and any command line arguments desired. If the ConnectedBroker argument is *not* nil, then the following are added to the command line and the application launched: • VistA M Server address • ListenerPort • Division • ApplicationToken
ConnectedBroker	This is the instance of the TRPCBroker that should be used to obtain an ApplicationToken for a Silent Login. The VistA M Server address and ListenerPort for this instance are used as command line arguments for launching the application, so that it makes a connection to the same Server/ListenerPort (see Server Property and ListenerPort Property) combination. If the application to be launched is *not* related to the TRPCBroker, then this argument should be set to nil.

4.2.13.2   Example 1

To launch a program, **Sample1.exe**, with command line arguments **xval=MyData** and **yval=YourData**, and connect with a Silent Login (which would be handled in **Sample1.exe** via the CheckCmdLine Function):

Figure 56: SilentChangeVerify Function—Example

```
MyCommand := 'C:\Program Files\VISTA\Test1\Sample1.exe xval=MyData yval=YourData';
StartProgSLogin(MyCommand, RPCBroker1);
```

This results in the command line in Figure 58 being used to launch the application:

Figure 57: SilentChangeVerify Function—Example of Command Line Code to Launch the Application

```
C:\Program Files\VISTA\Test1\Sample1.exe xval=MyData yval=YourData s=ServerName
p=<REDACTED> d=Division h=AppHandleValue
```

4.2.13.3   Example 2

To launch a program unrelated to TRPCBroker and VistA M Server connections (e.g., Microsoft® Notepad), the command line as desired is used as the first argument, and the value nil is used as the second argument:

Figure 58: SilentChangeVerify Function—Example of Command Line Code to Launch Program Unrelated to TRPCBroker and VistA M Server Connections

```
MyCommand := 'Notepad logtable.txt';
StartProgSLogin(MyCommand, nil);
```

4.2.14 VistA Splash Screen Procedures

The BDK provides two procedures in the **SplVista.PAS** unit to display a VistA Splash Screen when an application loads:

- `procedure SplashOpen;`

- `procedure SplashClose(TimeOut: longint);`

It is *recommended* that the VistA Splash Screen be opened and closed in the section of Pascal code in an application's project file (i.e., **.DPR**).

4.2.14.1 Using a Splash Screen in an Application

To use the VistA Splash Screen in an application

1. Open your application's project file (i.e., **.DPR**). In Delphi:

   a. Select **View**.

   b. Select **Project Source**.

2. Include the SplVista in the **uses** clause of the project source.

3. Call SplashOpen *immediately after* the first form of your application is created and call SplashClose *just prior to* invoking the **Application.Run** method.

4. Use the **TimeOut** parameter to ensure a minimum display time. The **TimeOut** parameter is the minimum number of milliseconds the splash screen is displayed to the user.

The VistA Splash Screen is illustrated in <u>Figure 60</u>:

Figure 59: Sample VistA Splash Screen

4.2.14.2 Example

Figure 61 is an example of code to display the VistA Splash Screen in an application:

Figure 60: Sample Code to Display a VistA Splash Screen

```
uses
    Forms, Unit1 in 'Unit1.pas', SplVista;

{$R *.RES}

begin
    Application.Initialize;
    Application.CreateForm(TForm1, Form1);
    SplashOpen;
    SplashClose(2000);
    Application.Run;
end.
```

REF: For a demonstration using the VistA Splash Screen, run the RPC Broker Example (32-Bit) (i.e., **BrokerExample.exe**); located in the following directory:

> **BDK32\Samples\BrokerEx**

4.3 Running RPCs on a Remote Server

4.3.1 Overview

The RPC Broker can be used to facilitate invocation of Remote Procedure Calls on a remote VistA M Server. Applications can use either XWB DIRECT RPC or XWB REMOTE RPC to pass the following:

- Desired remote VistA M Server.

- Desired remote RPC.

- Any parameters for the remote RPC.

The RPC Broker on the local VistA M Server uses VistA Health Level Seven (HL7) as a vehicle to pass the remote RPC name and parameters to the remote VistA M Server. VistA HL7 is used to send any results from the remote server back to the local server. The RPC Broker on the local VistA M Server then passes the results back to the client application.

NOTE: The local VistA M Server is the server the user is logged into. The remote VistA M Server is any server the user is *not* logged into.

4.3.2 Using Direct RPCs

Table 23: Direct RPCs

RPC	Description
XWB DIRECT RPC	This RPC blocks all other Broker calls until the results of the remote RPC are returned. The data is passed, and the user waits for the results to return from the remote system.

4.3.3 Using Remote RPCs

Table 24: Remote RPCs

RPC	Description
XWB REMOTE RPC	This RPC allows other activity while the remote RPC is in process. In response to XWB REMOTE RPC the local VistA M Server returns a HANDLE to the user application. At this point other Broker calls can commence while the server-to-server communication continues in the background.
XWB REMOTE STATUS CHECK	This RPC allows the application to check the local VistA M Server for the presence of results from the remote RPC. This RPC passes the HANDLE to the local server and receives back the status of the remote RPC.
XWB REMOTE GETDATA	This RPC retrieves the results from the remote RPC after the status check indicates that the data has returned to the local VistA M Server. The RPC passes the HANDLE and receives back an array with whatever data has been sent back from the remote site.
XWB REMOTE CLEAR	This RPC *must* be used to clear the data under the HANDLE in the **^XTMP** global.
XWB DEFERRED CLEARALL	Applications using XWB REMOTE RPC should use XWB DEFERRED CLEARALL on application close to clear all known data associated with the job on the VistA M Server.

NOTE: *All* XWB Remote Procedure Calls (RPCs) are available only on a Controlled Subscription basis.

4.3.4 Checking RPC Availability on a Remote Server

Applications can check the availability of RPCs on a remote VistA M Server. Use either of the following:

- XWB DIRECT RPC
- XWB REMOTE RPC

To pass either of the following:

- XWB IS RPC AVAILABLE (example)
- XWB ARE RPCS AVAILABLE (example)

To the remote server.

The **RUN CONTEXT PARAMETER** in XWB IS RPC AVAILABLE or XWB ARE RPCS AVAILABLE should be set to "**R**" or **NULL** to check that the remote VistA M Server allows RPCs to be run by users *not* logged into that remote server.

NOTE: *All* XWB Remote Procedure Calls (RPCs) are available only on a Controlled Subscription basis.

4.3.5 XWB ARE RPCS AVAILABLE

Checking RPC Availability on a Remote Server

Use this RPC to determine if a set of RPCs is available on a VistA M Server. The **RUN CONTEXT PARAMETER** allows you to test availability on a local or remote VistA M Server. The **RPC INPUT PARAMETER** passes the names and (optionally) minimum version number of the RPCs to be checked.

Table 25: XWB ARE RPCS AVAILABLE—Parameters

Parameter	Description
RETURN VALUE	A **0-based** array. The index corresponds to the index of the RPC in the RPC Input Parameter: • **1**—RPC Available. • **0**—RPC Not available.
RUN CONTEXT PARAMETER (Optional)	Pass the run context (local or remote) of the RPC in Param[0].Value, and the type (**literal**) in **Param[0].PType**. Possible values: • **L**—Check if available to be run locally (by a user logged into the VistA M Server). • **R**—Check if available to be run remotely (by a user logged in a different VistA M Server). If this parameter is *not* sent, the RPC is checked for both local and remote, and both run contexts *must* be available for the return to be "**1**" (RPC Available). The check is done against the INACTIVE field in the REMOTE PROCEDURE file (see the "Blocking an RPC" section).
RPC INPUT PARAMETER	Pass a **0**-based array of the names and (optionally) version numbers of RPCs to be tested in **Param[1].Mult[]**, and the type (**List**) in **Param[1].PType**. The format is: `RPCName^RPCVersionNumber`

Parameter	Description
	The RPCVersionNumber is used only if the Run Context parameter = "**R**". If a numeric value is in the second ^-piece and Run Context = "**R**", it is checked against the value in the VERSION field of the REMOTE PROCEDURE file (see the "RPC Version" section). If the version number passed is less than or equal to the number in the VERSION field, the RPC is marked available. ℹ️ **NOTE:** If the VERSION field Is **NULL**, the check fails for a numeric value in this parameter.

Also, the current context of your user *must* give them permission to execute the **XWB ARE RPCS AVAILABLE** (it *must* be included in the RPC multiple of the "**B**"-type option registered with the CreateContext Method).

ℹ️ **NOTE: XWB ARE RPCS AVAILABLE** is available only on a Controlled Subscription basis.

4.3.5.1 Example

Figure 62 is an example of the XWB ARE RPCS AVAILABLE RPC:

Figure 61: XWB ARE RPCS AVAILABLE—Example

```
brkrRPCBroker1.RemoteProcedure := 'XWB ARE RPCS AVAILABLE';
brkrRPCBroker1.Param[0].Ptype:= Literal;
brkrRPCBroker1.Param[0].Value := 'L';
brkrRPCBroker1.Param[1].Ptype := List;
brkrRPCBroker1.Param[1].Mult['0'] = 'MY FIRST RPC';
brkrRPCBroker1.Param[1].Mult['1'] = 'MY OTHER RPC^2';
try
    brkrRPCBroker1.Call;
except
    On EBrokerError do
        ShowMessage('Connection to server could not be established!');
end;
.; branch code to handle availability of RPCs
```

4.3.6 XWB IS RPC AVAILABLE

[Checking RPC Availability on a Remote Server](#)

Use this RPC to determine if a particular RPC is available on a VistA M Server. The **RPC PARAMETER** passes the name of the RPC to be checked. The **RUN CONTEXT PARAMETER** allows you to test availability to a local or a remote user. The **VERSION NUMBER PARAMETER** allows you to check for a minimum version of an RPC on a remote VistA M Server.

Table 26: XWB IS RPC AVAILABLE—Parameters/Output

Parameter/Output	Description
RETURN VALUE	Boolean: • **1—**RPC Available • **0—**RPC Not Available
RPC PARAMETER	Pass the name of the RPC to be tested in **Param[0].Value**, and the type (**literal**) in **Param[0].PType**.
RUN CONTEXT PARAMETER (Optional)	Pass the run context (local or remote) of the RPC in **Param[1].Value**, and the type (**literal**) in **Param[1].PType**. Possible values: • **L—**Check if available to be run locally (by a user logged into the VistA M Server). • **R—**Check if available to be run remotely (by a user logged in a different VistA M Server). If this parameter is *not* sent, the RPC is checked for both local and remote and both run contexts *must* be available for the return to be "**1**" (RPC Available). The check is done against the INACTIVE field in the REMOTE PROCEDURE file (see the "[Blocking an RPC](#)" section).
VERSION NUMBER PARAMETER (Optional)	Pass the minimum acceptable version number of the RPC in **Param[2].Value**, and the type (**literal**) in **Param[2].PType**. This parameter is only used if the RUN CONTEXT parameter = "**R**". If a numeric value is in this parameter, it is checked against the value in the VERSION field of the REMOTE PROCEDURE file (see the "[RPC Version](#)" section). If the version number passed is less than or equal to the number in the VERSION field, the RPC is marked available. ℹ **NOTE:** If the VERSION field is **NULL**, the check fails for a numeric value in this parameter.

Also, the current context of your user *must* give them permission to execute the **XWB IS RPC AVAILABLE** (it *must* be included in the RPC multiple of the "**B**"-type option registered with the CreateContext Method).

i   **NOTE: XWB IS RPC AVAILABLE** is available only on a Controlled Subscription basis.

4.3.6.1   Example

Figure 63 is an example of the XWB IS RPC AVAILABLE RPC:

Figure 62: XWB IS RPC AVAILABLE—Example

```
brkrRPCBroker1.RemoteProcedure := 'XWB IS RPC AVAILABLE';
brkrRPCBroker1.Param[0].Value :='XWB GET VARIABLE VALUE';
brkrRPCBroker1.Param[0].PType := literal;
brkrRPCBroker1.Param[1].Value := 'R';
brkrRPCBroker1.Param[1].PType := literal;
    {no version number passed in this example as XWB GET VARIABLE VALUE has only one
version}
try
    brkrRPCBroker1.Call;
except
    On EBrokerError do
       ShowMessage('Connection to server could not be established!');
end;
.; branch code to handle RPC availability
```

4.3.7 XWB DIRECT RPC

Use this RPC to request that an RPC be run on a remote system. This RPC blocks all other Broker calls until the results of the remote RPC are returned. Use XWB REMOTE RPC to allow other Broker activity while the remote RPC runs.

i   **REF:** For a comparison of the two methods, see the "Running RPCs on a Remote Server" section.

Table 27: XWB DIRECT RPC—Parameters/Output

Parameter/Output	Description
LOCATION PARAMETER	Pass the station number of the remote VistA M Server in **Param[0].Value**, and the type (**literal**) in **Param[0].PType**.
RPC PARAMETER	Pass the name of the RPC to be run in **Param[1].Value**, and the type (**literal**) in **Param[1].PType**.
RPC VERSION PARAMETER (Optional)	Pass minimum version of RPC to be run in **Param[2].Value**, and the type (**literal**) in **Param[2].PType**. It is checked against the value in the VERSION field of the REMOTE PROCEDURE file (see the "RPC Version" section) on the remote VistA M Server.
PARAMETERS TO THE REMOTE RPC	Pass up to seven parameters for the remote RPC in **Param[3]** through **Param[9]**.
RETURN VALUE	An array with whatever data has been sent back from the remote site. In the case of an error condition, the first node of the array is equal to a string with the syntax "**-1^error text**".

i   **NOTE: XWB DIRECT RPC** is available only on a Controlled Subscription basis.

4.3.7.1   Example

Figure 64 is an example of the XWB DIRECT RPC:

Figure 63: XWB DIRECT RPC—Example

```
brkrRPCBroker1.RemoteProcedure := 'XWB DIRECT RPC';
brkrRPCBroker1.Param[0].Ptype:= Literal;
brkrRPCBroker1.Param[0].Value := 'Station Number';
brkrRPCBroker1.Param[1].Ptype:= Literal;
brkrRPCBroker1.Param[1].Value := 'XWB GET VARIABLE VALUE';
{no version numbers for remote RPC so NULL value in Param[2]}
brkrRPCBroker1.Param[2].Ptype:= Literal;
brkrRPCBroker1.Param[2].Value := '';
brkrRPCBroker1.Param[3].Ptype:= Reference;
brkrRPCBroker1.Param[3].Value := 'DUZ';
try
    brkrRPCBroker1.Call;
except
    On EBrokerError do
        ShowMessage('Connection to server could not be established!');
end;
.; code to handle brkrRPCBroker1.Results[]
```

4.3.8   XWB REMOTE RPC

Use this RPC to request that an RPC be run on a remote system. This RPC allows other Broker activity while the remote RPC runs. Use XWB DIRECT RPC to block all other Broker activity while the remote RPC runs.

REF: For a comparison of the two methods, see the "Running RPCs on a Remote Server" section.

XWB REMOTE RPC requests the remote RPC. The return value is a HANDLE that is used to check status and retrieve data. The following RPCs *must* be used to complete the transaction

- XWB REMOTE STATUS CHECK
- XWB REMOTE GETDATA
- XWB REMOTE CLEAR

Table 28: XWB REMOTE RPC—Parameters/Output

Parameter/Output	Description
LOCATION PARAMETER	Pass the station number of the remote VistA M Server in **Param[0].Value**, and the type (**literal**) in **Param[0].PType**.
RPC PARAMETER	Pass the name of the RPC to be run in **Param[1].Value**, and the type (**literal**) in **Param[1].PType**.
RPC VERSION PARAMETER (Optional)	Pass minimum version of RPC to be run in **Param[2].Value**, and the type (**literal**) in **Param[2].PType**. It is checked against the value in the VERSION field of the REMOTE PROCEDURE file (see the "RPC Version" section) on the remote VistA M Server.
PARAMETERS TO THE REMOTE RPC	Pass up to seven parameters for the remote RPC in **Param[3]** through **Param[9]**.
RETURN VALUE	An array. The first node is equal to a string that serves as a HANDLE. This HANDLE should be stored by the application and used to check the status and retrieve the data. In the case of an error condition the first node of the array is equal to a string with the syntax **"-1^error text"**.

ℹ️   **NOTE: XWB REMOTE RPC** is available only on a Controlled Subscription basis.

4.3.8.1   Example

Figure 65 is an example of the **XWB REMOTE RPC**:

Figure 64: XWB REMOTE RPC—Example

```
brkrRPCBroker1.RemoteProcedure := 'XWB REMOTE RPC';
brkrRPCBroker1.Param[0].Ptype:= Literal;
brkrRPCBroker1.Param[0].Value := 'Station Number';
brkrRPCBroker1.Param[1].Ptype:= Literal;
brkrRPCBroker1.Param[1].Value := 'MY RPC';
brkrRPCBroker1.Param[2].Ptype:= Literal;
brkrRPCBroker1.Param[2].Value := '1';
brkrRPCBroker1.Param[3].Ptype:= Reference;
brkrRPCBroker1.Param[3].Value := 'MY RPC PARAMETER';
try
   brkrRPCBroker1.Call;
except
   On EBrokerError do
      ShowMessage('Connection to server could not be established!');
end;
.; code to store HANDLE returned in brkrRPCBroker1.Results[]
```

The application needs to use XWB REMOTE STATUS CHECK, XWB REMOTE GETDATA, *and* XWB REMOTE CLEAR to complete the transaction.

4.3.9 XWB REMOTE STATUS CHECK

Use this RPC to check for results of XWB REMOTE RPC. Periodically call this RPC and pass the HANDLE returned by XWB REMOTE RPC.

Table 29: XWB REMOTE STATUS CHECK—Output

Output	Description
RETURN VALUE	The return value is always an array. The first node of the array is equal to one of the following values: • **"-1^Bad Handle—**An invalid handle has been passed. • **"0^New"—**The request has been sent via VistA HL7. • **"0^Running"—**VistA HL7 indicates that the message is being processed. • **"1^Done"—**RPC has completed, and the data has been returned to the local VistA M Server. The data is *not* returned by this RPC. Use XWB REMOTE GETDATA to retrieve the data.

The second node of the array is the status from the VistA HL7 software.

NOTE: XWB REMOTE STATUS CHECK is available only on a Controlled Subscription basis.

4.3.9.1 Example

Figure 66 is an example of the XWB REMOTE STATUS CHECK RPC:

Figure 65: XWB REMOTE STATUS CHECK—Example

```
brkrRPCBroker1.RemoteProcedure := 'XWB REMOTE STATUS CHECK';
brkrRPCBroker1.Param[0].Value :='MYHANDLE';
brkrRPCBroker1.Param[0].PType := literal;
try
    brkrRPCBroker1.Call;
except
    On EBrokerError do
        ShowMessage('Connection to server could not be established!');
end;
.; code to handle results of check
```

4.3.10 XWB REMOTE GETDATA

Use this RPC to retrieve the results of XWB REMOTE RPC. Before calling this RPC, use XWB REMOTE STATUS CHECK to ensure that the results have been returned to the local VistA M Server. When the results have arrived, call this RPC and pass the HANDLE returned by XWB REMOTE RPC.

After the application is finished with the data on the VistA M Server, it should use XWB REMOTE CLEAR to clear the ^**XTMP** global.

Table 30: XWB REMOTE GETDATA—Output

Output	Description
RETURN VALUE	An array containing the data. In the case of an error condition the first node of the array is equal to a string with the syntax **"-1^error text"**.

ℹ️ **NOTE:** XWB REMOTE GETDATA is available only on a Controlled Subscription basis.

4.3.10.1 Example

Figure 67 is an example of the XWB REMOTE GETDATA RPC:

Figure 66: XWB REMOTE GETDATA—Example

```
brkrRPCBroker1.RemoteProcedure := 'XWB REMOTE GETDATA';
brkrRPCBroker1.Param[0].Value :='MYHANDLE';
brkrRPCBroker1.Param[0].PType := literal;
try
   brkrRPCBroker1.Call;
except
   On EBrokerError do
      ShowMessage('Connection to server could not be established!');
end;
.; code to handle data
```

4.3.11 XWB REMOTE CLEAR

This RPC is used to clear the data created by a remote RPC under the HANDLE in the ^**XTMP** global. Pass the HANDLE returned by XWB REMOTE RPC.

Table 31: XWB REMOTE CLEAR—Output

Output	Description
RETURN VALUE	An array. The first node in the array is equal to **1**.

i   **NOTE:** XWB REMOTE CLEAR is available only on a Controlled Subscription basis.

4.3.11.1   Example

Figure 68 is an example of the XWB REMOTE CLEAR RPC:

Figure 67: XWB REMOTE CLEAR—Example

```
brkrRPCBroker1.RemoteProcedure := 'XWB REMOTE CLEAR';
brkrRPCBroker1.Param[0].Value :='MYHANDLE';
brkrRPCBroker1.Param[0].PType := literal;
try
   brkrRPCBroker1.Call;
except
   On EBrokerError do
      ShowMessage('Connection to server could not be established!');
end;
```

4.4   Deferred RPCs

4.4.1   Overview

Remote Procedure Calls can be run in the background with XWB DEFERRED RPC.

4.4.1.1   Using Deferred RPCs

Table 32: Deferred RPCs

RPC	Description
XWB DEFERRED RPC	Use this RPC to pass the name of the RPC to be run in deferred mode and any parameters associated with the deferred RPC. In response to this RPC the VistA M Server returns a HANDLE to the user application. At this point other Broker calls can commence while the job runs in the background.
XWB DEFERRED STATUS	This RPC allows the application to check the local VistA M Server for the presence of results from the deferred RPC. This RPC passes the HANDLE to the local server and receives back the status of the remote RPC.
XWB DEFERRED GETDATA	This RPC is the vehicle for retrieving the results from the remote RPC after the status check indicates that the data has returned to the local VistA M Server. The RPC passes the HANDLE and receives back an array with whatever data has been returned by the deferred RPC.
XWB DEFERRED CLEAR	This RPC *must* be used to clear the data under the HANDLE in the **^XTMP** global.

RPC	Description
XWB DEFERRED CLEARALL	Applications using XWB DEFERRED RPC should use XWB DEFERRED CLEARALL on application close to clear all known data associated with the job on the VistA M Server.

NOTE: *All* XWB Remote Procedure Calls (RPCs) are available only on a Controlled Subscription basis.

4.4.2 XWB DEFERRED RPC

Use this RPC to request that an RPC be run in deferred mode. The return value is a HANDLE used to check status and retrieve data. The following RPCs *must* be used to complete the transaction:

- XWB DEFERRED STATUS
- XWB DEFERRED GETDATA
- XWB DEFERRED CLEAR

Table 33: XWB DEFERRED RPC—Parameters/Output

Parameter/Output	Description
RPC PARAMETER	Pass the name of the RPC to be run in **Param[0].Value**, and the type (**literal**) in **Param[0].PType**.
RPC VERSION PARAMETER (Optional)	Pass minimum version of RPC to be run in **Param[1].Value**, and the type (**literal**) in **Param[1].PType**. It is checked against the value in the VERSION field of the REMOTE PROCEDURE file (see the "RPC Version" section) on the remote VistA M Server.
PARAMETERS TO THE REMOTE RPC	Pass up to eight parameters for the remote RPC in **Param[2]** through **Param[9]**.
RETURN VALUE	An array. The first node is equal to a string that serves as a HANDLE. This HANDLE should be stored by the application and used to check the status and retrieve the data. In the case of an error condition, the first node of the array is equal to a string with the syntax "**-1^error text**".

4.4.2.1 Example

Figure 69 is an example of the XWB DEFERRED RPC:

Figure 68: XWB DEFERRED RPC—Example

```
brkrRPCBroker1.RemoteProcedure := 'XWB DEFERRED RPC';
brkrRPCBroker1.Param[0].Ptype:= Literal;
brkrRPCBroker1.Param[0].Value := 'MY RPC';
brkrRPCBroker1.Param[1].Ptype:= Literal;
brkrRPCBroker1.Param[1].Value := '1';
brkrRPCBroker1.Param[2].Ptype:= Reference;
brkrRPCBroker1.Param[2].Value := 'MY RPC PARAMETER';
try
    brkrRPCBroker1.Call;
except
    On EBrokerError do
       ShowMessage('Connection to server could not be established!');
end;
.; code to store HANDLE returned in brkrRPCBroker1.Results[0]
```

The application needs to use XWB DEFERRED STATUS, XWB DEFERRED GETDATA, *and* XWB DEFERRED CLEAR to complete the transaction.

4.4.3 XWB DEFERRED STATUS

Use this RPC to check for results of XWB DEFERRED RPC. Periodically, call this RPC and pass the HANDLE returned by XWB REMOTE RPC.

Table 34: XWB DEFERRED STATUS—Output

Output	Description
RETURN VALUE	The return value is always an array. The first node of the array is equal to one of the following values: • **"-1^Bad Handle"—**An invalid handle has been passed. • "**0^New**"—The request has been sent via VistA HL7. • **"0^Running"—**VistA HL7 indicates that the message is being processed. • **"1^Done"—**RPC has completed, and the data has been returned to the local VistA M Server. The data is *not* returned by this RPC. Use XWB REMOTE GETDATA to retrieve the data.

NOTE: XWB DEFERRED STATUS is available only on a Controlled Subscription basis.

4.4.3.1 Example

Figure 70 is an example of the XWB DEFERRED STATUS RPC:

Figure 69: XWB DEFERRED STATUS—Example

```
brkrRPCBroker1.RemoteProcedure := 'XWB DEFERRED STATUS';
brkrRPCBroker1.Param[0].Value :='MYHANDLE';
brkrRPCBroker1.Param[0].PType := literal;
try
   brkrRPCBroker1.Call;
except
   On EBrokerError do
      ShowMessage('Connection to server could not be established!');
end;
 .; code to handle results of check
```

4.4.4 XWB DEFERRED GETDATA

Use this RPC to retrieve the results of XWB DEFERRED RPC. Before calling this RPC, use XWB DEFERRED STATUS to ensure that the job has finished. When the results are available, call this RPC and pass the HANDLE returned by XWB DEFERRED RPC.

After the application is finished with the data on the VistA M Server, it should use XWB DEFERRED CLEAR to clear the ^**XTMP** global.

Table 35: XWB DEFERRED GETDATA—Output

Output	Description
RETURN VALUE	An array containing the data. In the case of an error condition the first node of the array is equal to a string with the syntax "**-1^error text**".

ℹ️ **NOTE: XWB DEFERRED GETDATA** is available only on a Controlled Subscription basis.

4.4.4.1 Example

is an example of the XWB DEFERRED GETDATA RPC:

Figure 70: XWB DEFERRED GETDATA—Example

```
brkrRPCBroker1.RemoteProcedure := 'XWB DEFERRED GETDATA';
brkrRPCBroker1.Param[0].Value :='MYHANDLE';
brkrRPCBroker1.Param[0].PType := literal;
try
    brkrRPCBroker1.Call;
except
    On EBrokerError do
      ShowMessage('Connection to server could not be established!');
end;
.; code to handle data
```

4.4.5 XWB DEFERRED CLEAR

This RPC is used to clear the data created by a deferred RPC under the HANDLE in the **^XTMP** global. Pass the HANDLE returned by XWB DEFERRED RPC.

Table 36: XWB DEFERRED CLEAR—Output

Output	Description
RETURN VALUE	An array. The first node in the array is equal to **1**.

ⓘ **NOTE:** XWB DEFERRED CLEAR is available only on a Controlled Subscription basis.

4.4.5.1 Example

is an example of the XWB DEFERRED CLEAR RPC:

Figure 71: XWB DEFERRED CLEAR—Example

```
brkrRPCBroker1.RemoteProcedure := 'XWB DEFERRED CLEAR';
brkrRPCBroker1.Param[0].Value :='MYHANDLE';
brkrRPCBroker1.Param[0].PType := literal;
try
    brkrRPCBroker1.Call;
except
    On EBrokerError do
      ShowMessage('Connection to server could not be established!');
end;
```

4.4.6 XWB DEFERRED CLEARALL

This RPC is used to CLEAR ALL the data known to a remote RPC or deferred RPC job in the ^**XTMP** global. It makes use of the list in ^**TMP("XWBHDL",$J,handle)**. Applications using XWB REMOTE RPC or the XWB DEFERRED RPC should use this RPC on application close to clear all known data associated with the job on the VistA M Server.

Table 37: XWB DEFERRED CLEARALL—Output

Output	Description
RETURN VALUE	An array. The first node in the array is equal to **1**.

ℹ️ **NOTE: XWB DEFERRED CLEARALL** is available only on a Controlled Subscription basis.

4.4.6.1 Example

Figure 73 is an example of the XWB DEFERRED CLEARALL RPC:

Figure 72: XWB DEFERRED CLEARALL—Example

```
brkrRPCBroker1.RemoteProcedure := 'XWB DEFERRED CLEAR';
try
    brkrRPCBroker1.Call;
except
    On EBrokerError do
        ShowMessage('Connection to server could not be established!');
end;
```

5 Broker Security Enhancement (BSE)

5.1 Overview: Implementing Broker Security Enhancement (BSE)

This section describes how application developers can modify their Veterans Health Information Systems and Technology Architecture (VistA) RPC Broker Delphi-based client/server applications in order to implement the Broker Security Enhancement (BSE). The following topics are discussed:

- Assumptions When Implementing BSE
- Step-By-Step Procedures to Implement BSE

5.2 Assumptions When Implementing BSE

The following assumptions are made regarding application developers and VistA software applications when implementing BSE:

- **Developer Training**—Application developers should already be knowledgeable/trained in creating RPC Broker Delphi-based applications.
- **RPC Broker-based Applications**—RPC Broker Delphi-based application already exists.
- **Login at Startup**—Applications automatically initiate login at application startup (i.e., users are presented with 2-factor authentication or an Access/Verify login dialogue).
- **VistA M Server Patches**—All BSE Project-related VistA M Server patches have been loaded on the appropriate servers.

5.3 Step-By-Step Procedures to Implement BSE

This section describes the procedures that VistA application developers *must* follow in order to implement the Broker Security Enhancement (BSE) in their RPC Broker Delphi-based applications (i.e., COM client applications developed in Embarcadero Delphi), so that the application can make remote user/visitor connections.

1. Create a unique Application Security Phrase *(required)*.

    Use the $$SHAHASH^XUSHSH API to create a Base-64 Encoded SHA256 hashed Security Phrase (case sensitive) that is unique for your application.

    For example, in M, go to Programmer Mode and enter the following command:

    ```
    W $$SHAHASH^XUSHSH(256,"My Application Security Phrase","B")
    ```

    The resulting sample value is:

    ```
    TQu07MtTls83BGuWK/Kyb4naAUWHaVQjTzstCuDJKHw=
    ```

CAUTION: This is a sample value only; do *not* use this as your Application Security Phrase!

This is a one-way hash value for the Security Phrase that is only known to the application that creates it.

RECOMMENDATION: Since the Security Phrase is the application's identifier, VistA Infrastructure (VI) *recommends* developers identify the Security Phrase as a const value in an include file in any RPC Broker Delphi-based program implementing BSE. A substitute include file containing a phrase similar to that used above should then be included with release of the source code.

REF: For more information on the application Security Phrase, see the "Security Phrase" section in the *RPC Broker User Guide*.

2. Create an entry in the REMOTE APPLICATION (#8994.5) file *(required)*.

   An application *must* send out a patch that creates an entry for their RPC Broker Delphi-based application that has implemented BSE in the REMOTE APPLICATION (#8994.5) file. Developers *must* add entries to the following fields in File #8994.5:

   - NAME (#.01)—Enter a descriptive name for your application.

   - CONTEXTOPTION (#.02)—Enter the name of the "**B**"-Type context option that the users will need to run the application. This context option should be within your application's namespace, or your application should have an Integration Control Registration (ICR) agreement in place documenting permission to use a context option owned by another application.

   - APPLICATIONCODE (#.03)—Enter the hashed value of the Security Phrase you created in Step 1.

   - CALLBACKTYPE Multiple (#1):
     - o CALLBACKTYPE (#.01)—Current values for this field are:
       - – **R**—RPC Broker TCP/IP connection (*recommended* for *non*-medical centers).
       - – **M**—M-to-M Broker connection.
       - – **H**—HyperText Transport Protocol (HTTP) connection communication. You must also add an entry in the URLSTRING (#.04) field.
       - – **S**—Station-number callback (*recommended* for medical centers).

- o CALLBACKPORT (#.02)—Enter the Port number to be used for the callback connection. This required field should be set to "**-1**" for Station-number callback type, as the actual port number is passed to the remote VistA M Server as part of the process.

- o CALLBACKSERVER (#.03)—Enter the address of the server to be used for the callback connection. This should be a Domain Name Service (DNS) name-based address rather than an Internet Protocol (IP) address, because IP addresses can change. This field is *not* required for Station-number callback type, as the remote VistA M Server looks up the IP address of the authenticating VistA M Server based on the site number passed to the remote VistA M Server as part of the process.

- o URLSTRING (#.04)—Used only if the CALLBACKTYPE (#.01) field contains H for HyperText Transport Protocol (HTTP). Enter the Uniform Resource Locator (URL) string for the callback to the HTTP server. This field is *not* required for Station-number, RPC Broker, or M-to-M callback types.

NOTE: For more information on the REMOTE APPLICATION (#8994.5) file and specific field entries, see the "REMOTE APPLICATION (#8994.5) file" section in the *RPC Broker User Guide*.

REF: Use the sample code in the **BseSample1.pas** file as a basis for implementing BSE in your application. The **BseSample1.pas** file is located in the following directory:

BDK32\Samples\BSE

Figure 73: BseSample1.pas File—Sample Code Excerpt (#1)

```
procedure TForm1.DoConnection(Key: String);
var
  TokenValue: String;
begin
  RPCB.Server := AuthServer.Text;
  RPCB.ListenerPort := StrToInt(AuthPort.Text);
  RPCB.Connected := True;
  if RPCB.Connected then
  begin
    RPCB.RemoteProcedure := 'XUS SET VISITOR';
    RPCB.Call;
    TokenValue := RPCB.Results[0];
    RPCB.Connected := False;
    ShowMessage('Token: '+TokenValue);
    if not (TokenValue = '') then
    begin
      RPCB.Server := RemoteServer.Text;
      RPCB.ListenerPort := StrToInt(RemotePort.Text);
      RPCB.SecurityPhrase := Key + '^' + TokenValue;
      RPCB.Connected := True;
      if RPCB.Connected then
      begin
        ShowMessage('Signed  on to Remote Server');
        RPCB.CreateContext('XWB BROKER EXAMPLE');
        btnDisconnect.Enabled := True;
        btnEcho.Enabled := True;
        btnM2M.Enabled := False;
        btnTCPIP.Enabled := False;
      end
      else
        ShowMessage('Connection to Remote Server failed!');
    end;
  end
  else
    ShowMessage('Initial Sign-on Failed');
end;
```

3. Get BSE Authentication Token *(required).*

After authenticating the user into the authenticating VistA M Server, the client application calls the XUS SET VISITOR RPC to get the BSE Authentication Token for the user. This token is then passed to the RPC Broker component and used to create an extended Security Pass Phrase (see Step 4). This token is eventually used to obtain the necessary user information for populating a user as a "visitor" entry in the remote site's NEW PERSON (#200) file.

VistA Kernel software on the authenticating VistA M Server creates the BSE Authentication Token. Kernel stores this token in the **^XTMP** temporary global.

Figure 74: BseSample1.pas File—Sample Code Excerpt (#2)

```
RPCB.Server := AuthServer.Text;
RPCB.ListenerPort := StrToInt(AuthPort.Text);
RPCB.Connected := True;
if RPCB.Connected then
begin
  RPCB.RemoteProcedure := 'XUS SET VISITOR';
  RPCB.Call;
  TokenValue := RPCB.Results[0];
  RPCB.Connected := False;
```

4. Create and encode an extended Security Pass Phrase *(required).*

The application creates an extended Security Pass Phrase (string). The Security Pass Phrase consists of the unhashed application Security Phrase (Step 1) concatenated with the BSE Authentication Token delimited by a caret (^). For example:

```
My Application Security Phrase^XWBHDL977-124367_0
```

Station-number callback requires the authenticating VistA M Server's station number as identified in the INSTITUTION (#4) file and the port number of the station's RPC Broker listener, delimited by a caret (^). For example:

```
My Application Security Phrase^XWBHDL977-124367_0^518^19207
```

The Delphi RPC Broker software encodes the Security Pass Phrase, which is passed to the Remote VistA M Server **XUS SIGNON SETUP** RPC for authentication.

For *non*-Delphi applications (those that do *not* use the Broker Development Kit), the **XUS BSE TOKEN** RPC accepts the application Security Phrase on the authenticating server and creates a complete encoded extended Security Pass Phrase, which can be passed to the Remote VistA M Server **XUS SIGNON SETUP** RPC for authentication for Station-number based callback. This RPC does *not* work for Delphi applications, as the encoding is done within the Broker Development Kit.

In the source code excerpt that follows (see Figure 76), the value of **Key** (i.e., constant) was defined earlier by importing an include file that contained the following two lines:

```
const
    Key = ' My Application Security Phrase';
```

ℹ️ **NOTE: Key** is a constant, which is a type of variable that has a fixed value that *cannot* be changed.

Figure 76 shows the code after the RPCBroker login component connection to the Authenticating VistA M Server has been disconnected:

Figure 75: BseSample1.pas File—Sample Code Excerpt (#3)

```
  if not (TokenValue = '') then
  begin
    RPCB.Server := RemoteServer.Text;
    RPCB.ListenerPort := StrToInt(RemotePort.Text);
    RPCB.SecurityPhrase := Key + '^' + TokenValue;
    RPCB.Connected := True;
    if RPCB.Connected then
    begin
      ShowMessage('Signed  on to Remote Server');
      RPCB.CreateContext('XWB BROKER EXAMPLE');
      btnDisconnect.Enabled := True;
      btnEcho.Enabled := True;
      btnM2M.Enabled := False;
      btnTCPIP.Enabled := False;
    end
    else
      ShowMessage('Connection to Remote Server failed!');
  end;
end
```

5. Set RPCBroker login component properties *(required)*.

The developer *must* set the following RPCBroker login component properties when calling the Remote VistA M Server:

- **Server**—Set to the Domain Name Service (DNS) or Internet Protocol (IP) address of the Remote VistA M Server.

- **ListenerPort**—Set to the Listener Port number of the Remote VistA M Server.

- **SecurityPhrase**—Set to the unhashed application's Security Phrase concatenated with the Kernel Authentication Token (See Step 4).

- **Connected**—Set to **True**.

6. Process Remote User/Visitor Login on remote server *(required).*

   After connecting to the Remote VistA M Server, software running on the Remote VistA M Server does the following:

   a. Identify the Security Pass Phrase. Kernel identifies the data passed in as a parameter, which contains the application's Security Phrase and Kernel Authentication Token for the user.

   b. Hash the Security Pass Phrase. Kernel hashes the Security Pass Phrase to parse out the application's Security Phrase and the Kernel Authentication Token.

   c. Get Authenticating VistA M Server Connection Mechanism. Kernel uses the Security Phrase to identify the application's entry in the REMOTE APPLICATION (#8994.5) file.

      Included in that entry is the mechanisms for contacting the Authenticating VistA M Server:

      - Connection type:
        - **R**—RPC Broker TCP/IP connection
        - **M**—M-to-M Broker
        - **H**—HyperText Transport Protocol connection
        - **S**—Station-number callback

      - Port number
      - Address (IP, DNS, or URL)

      NOTE: The mechanisms for contacting the Authenticating VistA M Server allows you to use either the IP address or DNS; however, VistA Infrastructure (VI) *recommends* that you use a DNS Fully Qualified Domain Name (FQDN).

   d. Connect to Authenticating Server using Kernel Authentication Token. The Remote VistA M Server uses the appropriate mechanism to identify and connect to the Authenticating VistA M Server, passing in the BSE Authentication Token that identifies the user.

   e. Obtain user demographics. Kernel uses the **XUS GET VISITOR** RPC to request and obtain the user demographic information from the Authenticating VistA M Server.

      The user demographic information that is returned is a string containing information that can be used to identify the visitor:

      ```
      ssn^name^station name^station number^DUZ^phone^SecID^network username
      ```

      - Social Security Number (SSN)

- Name

- Station Name

- Station Number

- DUZ

- Telephone

- Identity and Access Management (IAM) Security ID (SecID)

- Active Directory Network Username

This user demographic information is used to later establish the user as a remote user/visitor on the Remote VistA M Server.

f. Disconnect from the Authenticating VistA M Server. The Remote VistA M Server disconnects from the Authenticating VistA M Server.

g. Set up user as a visitor entry on the remote VistA M Server. Kernel uses the demographic information obtained from the Authenticating VistA M Server to set up the user as a visitor entry on the Remote VistA M Server.

Kernel creates or matches an entry in the NEW PERSON (#200) file and provides the visitor with the context option specified for the application in the REMOTE APPLICATION (#8994.5) file. The matching process uses the following precedence when matching an existing user:

1. Identity and Access Management (IAM) Security ID (SecID)

2. Social Security Number (SSN)

3. Name (do *not* use if name has a different SSN)

7. Test your application *(recommended)*. Developers should test their RPC Broker Delphi-based applications to ensure they have successfully implemented BSE.

BSE Sample Test Applications

The Broker Development Kit (BDK) includes the BrokerSecurityEnhancement Sample1 application (i.e., **BseSample1.exe**, see Figure 77).

You can use this sample application to help test the sample entries in the REMOTE APPLICATION (#8994.5) file and to test the different connection types (i.e., TCP/IP, M2M, HTTP, and Station-number) to verify that the VistA M Server-side is set up correctly to implement BSE.

⚠️ **CAUTION: In order to implement BSE and use the RPC-Broker callback type, the central Authenticating VistA M server *must* run the RPC Broker as a TCPIP service.**

BrokerSecurityEnhancement Sample1

Figure 77 shows the sample application dialogue provided by the BrokerSecurityEnhancement Sample1 application (i.e., **BseSample1.exe**):

Figure 76: BSE Project—BrokerSecurityEnhancement Sample1 Application (i.e., BseSample1.exe)
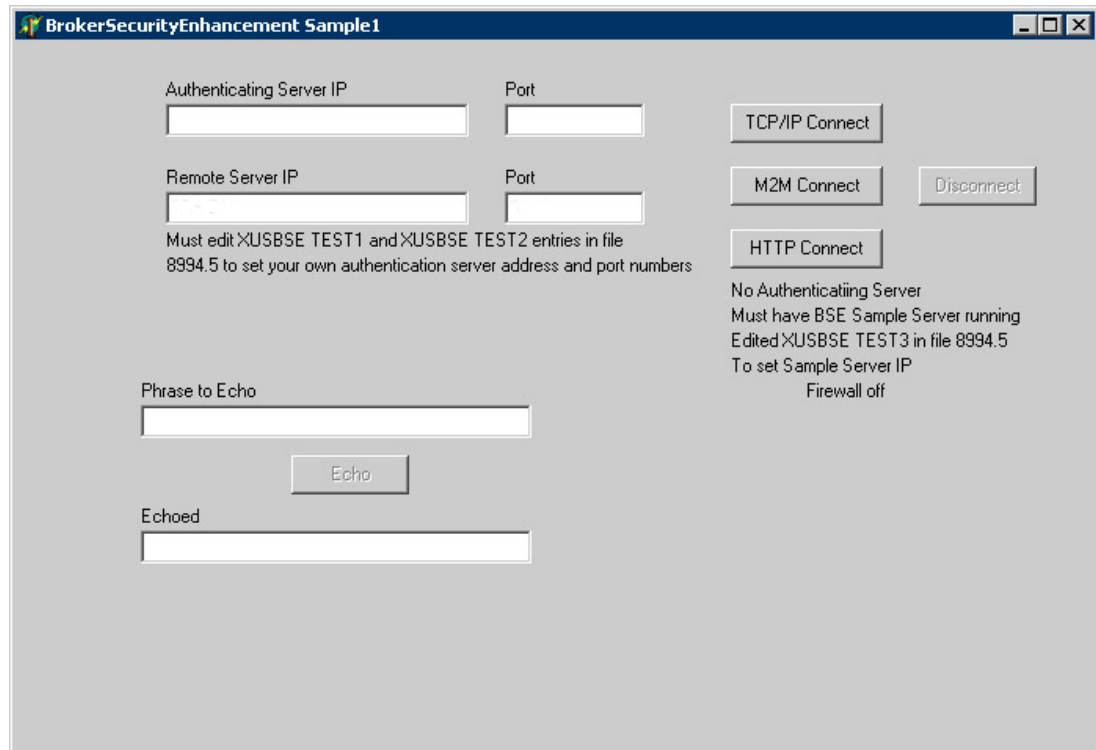
The sample application has the following controls:

- Server Edit Fields:

  - **Authenticating Server IP**—IP address for the Authenticating VistA M Server. This field is empty at initial startup; it is an editable field.

  - (Authenticating Server) **Port**—Port number for the Authenticating VistA M Server. This field is empty at initial startup; it is an editable field.

  - **Remote Server IP**—IP address for the Remote VistA M Server. This field is empty at initial startup; it is an editable field.

  - (Remote Server) **Port**—Port number for the Remote VistA M Server. This field is empty at initial startup; it is an editable field**.**

- Connection Buttons:
  - **TCP/IP Connect**
  - **M2M Connect**
  - **HTTP Connect**
  - **Disconnect**

- Phrase Echo Controls:
  - **Phrase to Echo** Edit Field—Enter an echo phrase.
  - **Echo** Button—Button used to submit the phrase to be echoed back form the Remote VistA M Server.
  - **Echoed** Field—Contains the phrase that gets echoed back once the user/visitor is signed onto the Remote VistA M Server.

To successfully run and test the BrokerSecurityEnhancement Sample1 application (i.e., **BseSample1.exe**), do the following:

a. Edit entries for XUSBSE TEST1 and XUSBSE TEST2 in the following fields in the CALLBACKTYPE Multiple (#1) in the REMOTE APPLICATION (#8994.5) file. These entries are the Authenticating VistA M Servers that are used to authenticate the current user, and to which a callback is made to obtain information to eventually create the visitor entry in the Remote VistA M Server:

- CALLBACKPORT (#.02)
- CALLBACKSERVER (#.03)

The Broker Security Enhancement (BSE)-related code is dependent upon the use of appropriate and valid information for the Authenticating and Remote VistA M Servers. Therefore, running the **BseSample1.exe** program requires that you populate these fields on the Remote VistA M Server.

The Authenticating VistA M Server is the server on which the user already has a valid Kernel Access and Verify code established (i.e., entry in the NEW PERSON [#200] file). Both the Authenticating and Remote VistA M Servers *must* also have RPC Broker Patch XWB*1.1*45 and Kernel Patch XU*8*404 installed.

b. Start the **BseSample1.exe** program.

c. Enter a valid Authenticating VistA M Server IP address and Port number.

> **NOTE:** This is the server against which the user first authenticates.

d. Enter a valid Remote VistA M Server IP address and Port number.

> **i** **NOTE:** This is the server that the user signs onto as a visitor and already contains the updated information for the Authenticating VistA M Server in the REMOTE APPLICATION (#8994.5) file.

e. Press one of the connection buttons (e.g., **TCP/IP Connect** button).

f. Enter Access and Verify codes in the "VistA Sign-on" dialogue box when prompted.

> **i** **NOTE:** This authenticates the user against the Authenticating VistA M Server.

g. (optional) Choose your Division (i.e., Station Number) to log into, if prompted.

h. Press **OK** when presented with the dialogue in Figure 78:

Figure 77: Sample Kernel Authentication Token

> **i** **NOTE:** Figure 78 indicates that the Kernel Authentication Token was created, which means the user is now authenticated on the Authenticating VistA M Server.

i. After a few moments, you get the dialogue shown in Figure 79 confirming the user is now also authenticated on the Remote VistA M Server as a visitor. Press **OK** when presented with the dialogue in Figure 79:

Figure 78: Sample Confirmation Message Indicating the User is Signed onto the Remote VistA M Server as a Visitor
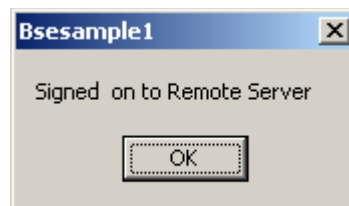
j. You can now enter an echo phrase to the Remote VistA M Server and get the string echoed back.

6 Debugging and Troubleshooting

6.1 Debugging and Troubleshooting Overview

The Broker Development Kit (BDK) provides facilities for debugging and troubleshooting your VistA Graphical User Interface (GUI) applications.

- How to Debug the Application
- RPC Error Trapping
- Broker Error Messages
- EBrokerError
- Testing the RPC Broker Connection
- Client Timeout and Buffer Clearing
- Memory Leaks

REF: For commonly asked questions, see the RPC Broker FAQs on the RPC Broker VA Intranet site.

6.2 How to Debug the Application

Control of debugging has been moved from the client to the server.

To start a debug session, do the following:

1. On the VistA M Server, set initial breakpoints where desired.

2. On the client, follow instructions in the InterSystems Caché documentation on "Debugging with the Caché Debugger." Set initial breakpoints where desired.

3. Start the following VistA M Server process:

```
>D DEBUG^XWBTCPM
```

4. Enter a unique Listener port number (i.e., a port number *not* in general use).

5. Connect the client application to the server using the server's IP address and the port number you entered in Step 4 and select **OK**.

6. You can now step through the code on your client, and simultaneously step through the code on the VistA M Server side for any RPCs that your client calls.

6.3 RPC Error Trapping

M errors on the VistA M Server that occur during RPC execution are trapped by the use of M and Kernel error handling. In addition, the M error message is sent back to the Delphi client. Delphi raises an exception [EBrokerError](#) and a popup dialogue box displaying the error. At this point RPC execution terminates and the channel is closed.

In some instances, an application's RPC could get a memory allocation error on the VistA M Server. Kernel does *not* trap these errors. However, these errors are trapped in the operating system's error trap. For example, if an RPC receives or generates an abundance of data in local memory, the symbol table could be depleted resulting in a memory allocation error. To diagnose this problem, users should check the operating system's error trap.

6.4 Broker Error Messages

list of errors/messages are Broker-specific and are *not* Winsock related:

Table 38: Broker Error Messages

Error/Message	Name	Number	Description
Insufficient Heap	XWB_NO_HEAP	20001	This is a general error condition indicating insufficient memory. It can occur when an application allocates memory for a variable. This error occurs for some of the following reasons: • Too many open applications. • Low physical memory. • Small virtual memory swap file (if dynamic, maybe low disk space). • User selecting too many records. **Resolution:** Common solutions to this error include the following: • Close some or all other applications. • Install more memory. • Increase the swap file size or, if dynamic, leave more free space on disk. • Try working with smaller data sets. • Reboot the workstation.
M Error - Use ^XTER	XWB_M_REJECT	20002	The VistA M Server side of the application errored out. The Kernel error trap has recorded the error. **Resolution:** Examine the Kernel error trap for more information and specific corrective actions.
Signon was not completed	XWB_BadSignOn	20004	This error indicates the user did *not* successfully signon. **Resolution:** Either the Access and Verify codes were incorrect or the user clicked **Cancel** on the "VistA Sign-on" window.

Error/Message	Name	Number	Description
BrokerConnections list could not be created	XWB_BldConnectList	20005	This error is a specific symptom of a low memory condition. **Resolution:** For a detailed explanation and corrective measures, see the "Insufficient Heap" error message.
RpcVersion cannot be empty	XWB_NullRpcVer	20006	This error occurs when an RPC does *not* have an associated version number. Each RPC *must* have a version number. **Resolution:** Contact the developers responsible for the application software to take corrective action.
Server unable to read input data correctly	XWB_BadReads	20008	This error indicates that the format of the RPC input data was incorrect. **Resolution:** Contact the developers responsible for the application software to take corrective action.
System was out of memory, executable file was corrupt, or relocations were invalid	XWB_ExeNoMem	20100	This error may indicate a low memory condition or may have errors in the application executable file. **Resolution:** Contact the developers responsible for the application software to take corrective action.
File was not found	XWB_ExeNoFile	20102	This error indicates that the referenced file could not be found. **Resolution:** Contact the developers responsible for the application software to take corrective action.
Path was not found	XWB_ExeNoPath	20103	This error indicates that the referenced directory could not be found. **Resolution:** Contact the developers responsible for the application software to take corrective action.
Attempt was made to dynamically link to a task or there was a sharing or network-protection error	XWB_ExeShare	20105	This error most likely indicates network problems. **Resolution:** It may resolve itself over time. If not, contact the developers responsible for the application software to take corrective action.

Error/Message	Name	Number	Description
Library required separate data segments for each task	XWB_ExeSepSeg	20106	This error indicates that the format of the RPC data was incorrect. **Resolution:** Contact the developers responsible for the application software to take corrective action.
There was insufficient memory to start the application	XWB_ExeLoMem	20108	This error is a specific symptom of a low memory condition. **Resolution:** For a detailed explanation and corrective measures, see the "Insufficient Heap" error message.
Windows version was incorrect	XWB_ExeWinVer	20110	This error indicates that the application was developed for a specific version of Windows and is not compatible with this system. **Resolution:** Contact the developers responsible for the application software to take corrective action.
Executable file was invalid. Either it was not a Windows application or there was an error in the EXE	XWB_ExeBadExe	20111	This error indicates a problem with the Windows executable application. **Resolution:** Contact the developers responsible for the application software to take corrective action.
Application was designed for a different operating system	XWB_ExeDifOS	20112	This error indicates that the application is not compatible with this operating system. **Resolution:** Contact the developers responsible for the application software to take corrective action.
Remote procedure not registered to application	XWB_RpcNotReg	20201	This error indicates the application attempted to execute an RPC that was *not* entered into the RPC Multiple field in the REMOTE PROCEDURE (#8994) File for this application. **Resolution:** The developers responsible for the application should be contacted. As a "last resort" corrective measure, you can try to re-index the cross-reference on the RPC (#.01) field in the REMOTE PROCEDURE (#8994) File with the RPC (#320) field of the

Error/Message	Name	Number	Description
			OPTION (#19) file. Ideally, this should only be attempted during off or low system usage.

6.5 EBrokerError

6.5.1 Unit

[TRPCB Unit](#)

6.5.2 Description

The EBrokerError is an exception raised by the [TRPCBroker Component](#). This exception is raised when an error is encountered when communicating with the VistA M Server. You should use a **try...except** block around all server calls to handle any EbrokerError exceptions that may occur.

For example:

Figure 79: Error Handling—EBrokerError Exception

```
try
   brkrRPCBroker1.Connected:= True;
except
   on EBrokerError do
   begin
     ShowMessage('Connection to server could not be established!');
     Application.Terminate;
   end;
end;
```

REF: For descriptions/resolutions to specific error messages that can be displayed by EBrokerError, see the "[Broker Error Messages](#)" section.

6.6   Testing the RPC Broker Connection

To test the RPC Broker connection from your workstation to the VistA M Server, use the RPC Broker Diagnostic Program (i.e., **RPCTEST.exe**, distributed with patch XWB*1.1*47).

REF: For a complete description of the RPC Broker Diagnostic program, see Section 4, "Troubleshooting," in the *RPC Broker Systems Management Guide*.

REF: For a demonstration/test using the Broker to connect to a VistA M Server, run the RPC Broker Example (32-Bit) (i.e., **BrokerExample.exe**); located in the following directory:

BDK32\Samples\BrokerEx

6.7   Client Timeout and Buffer Clearing

If a remote procedure call (RPC) fails to successfully complete due to a timeout on the client, the buffer on the VistA M Server contains data from the uncompleted call. Without special handling, this buffer on the server is returned whenever the next RPC is executed.

The solution to this problem is:

1. The RPCTimeLimit Property on the TRPCBroker Component on the client helps avoid the problem in the first place.

2. In the event of a cancellation of a Network **I/O** operation, the Broker state on the client changes from **NO FLUSH** to **FLUSH**. When this state change occurs, the next RPC executed undergoes a **READ** operation prior to execution where any leftover incoming buffer is discarded. At the end of this operation, the Broker state on the client returns to **NO FLUSH** and the RPC executes normally. While the **FLUSH** state exists, users can experience a delay while the corrupted RPC data is discarded. The delay is proportional to the amount of data in the buffer.

6.8   Memory Leaks

A good indication of a memory leak is when a running program is steadily decreasing the free pool of memory. As it runs or every time the program is started and stopped, free memory is steadily decreased.

Specifically, a program requests some bytes of memory from the Microsoft® Windows operating system (OS). When the OS provides it, it marks those bytes as taken. The free pool of memory (i.e., unmarked bytes) is decreased. When the program is finished with the memory, it should return the memory back to the OS by calling the FREE or DISPOSE functions. This allows the OS to clear the "**taken**" status of that memory; thereby, replenishing its free pool. When a developer forgets to free the memory after use or the program fails before it has a chance to execute the code that frees the memory, the memory is *not* reclaimed.

At all times, the program should keep track of which memory it is using. It does this by storing "Handles" (i.e., memory addresses of the beginning byte of each memory block). Later, when

freeing memory, the Handle is used to indicate which memory address to free. If the variable that holds such a Handle is overwritten, there is no way to determine the Handle.

Nine out of ten times, memory leakage can be traced back to the application code that requests memory and then forgets to return it or *cannot* clean up after a crash.

As common with other professional-level languages (e.g., C/C++), Delphi has constructs that applications can use to:

1. Request memory.
2. Type cast it.
3. Return it.

This requires developers to use their best judgement on how to best work with the system memory.

Avoiding memory leaks (and the often-subtle coding errors that lead to them) is a challenge for Delphi developers, especially for those whose main experience is working with M.

The insidious effect of these leaks (e.g., gobbling up **1K** of memory each time that a certain event occurs) makes them difficult to detect with normal program testing. "Normal testing" means exercising all the possible paths through the code once, a difficult enough process in a Microsoft® Windows environment. Often, these leaks result in a symptom only under peculiar conditions (e.g., several other applications are running, reducing system resources), or only after extended use of the application (e.g., do you notice that Microsoft® Windows problems crop up in the afternoon, even though you were doing the same thing that morning?).

The most common symptom described is the following:

"The computer was working fine until the user installed the XYZ VistA software application on their PC. Now, it freezes up (gives an error message, says it is out of memory, etc.) all the time, even when the user is *not* using the XYZ package. No, the user *cannot* duplicate it, it just happens!"

One of the reasons that there is an extensive market for automated testing tools for Microsoft® Windows and client/server applications is that thorough testing is very difficult to do manually.

Fortunately, there are diagnostic products available for detecting code that cause memory leaks. It helps developers and code reviewers to find these leaks. Its use by people just starting out in Delphi development helps them identify the situations that cause memory leaks. This can serve as a good learning experience for new Delphi developers.

No application is immune from memory leaks, careful analysis of previous Broker code revealed some places where, under certain conditions, memory was *not* being released after it was used (i.e., memory leaks). These areas have been identified and corrected with RPC Broker 1.1.

7 Tutorial

7.1 Tutorial: Introduction

The major functions of a [TRPCBroker Component](#) in a Delphi-based application are to:

- Connect to an RPC Broker VistA M Server system from a client.

- Execute remote procedure calls (RPCs) on that system.

- Return data results from RPC to the client.

This tutorial guides users through using a [TRPCBroker Component](#) to perform each of these tasks by having you create a Delphi-based application, step-by-step. This application retrieves a list of terminal types from the VistA M Server and displays information about each terminal type.

After you have completed this tutorial, you should be able to:

- Include a [TRPCBroker Component](#) in a Delphi-based application.

- Retrieve the end-user client workstation's designated VistA M Server and port to connect.

- Establish a connection through the RPC Broker component to an RPC Broker VistA M Server.

- Create M routines that return data in the formats necessary to be called from RPCs.

- Create RPCs.

- Call RPCs from a Delphi-based application to retrieve data from VistA M database.

- Pass parameters from the Delphi-based application to RPCs.

7.1.1 Tutorial Procedures

- [Tutorial: Advanced Preparation](#)

- [Tutorial—Step 1: RPC Broker Component](#)

- [Tutorial—Step 2: Get Server/Port](#)

- [Tutorial—Step 3: Establish Broker Connection](#)

- [Tutorial—Step 4: Routine to List Terminal Types](#)

- [Tutorial—Step 5: RPC to List Terminal Types](#)

- [Tutorial—Step 6: Call ZxxxTT LIST RPC](#)

- [Tutorial—Step 7: Associating IENs](#)

- [Tutorial—Step 8: Routine to Retrieve Terminal Types](#)

- [Tutorial—Step 9: RPC to Retrieve Terminal Types](#)

- [Tutorial—Step 10: Call ZxxxTT RETRIEVE RPC](#)

- [Tutorial—Step 11: Register RPCs](#)

- [Tutorial—Using VA FileMan Delphi Components (FMDC)](#)

- [Tutorial—Source Code (Sample)](#)

7.2 Tutorial: Advanced Preparation

7.2.1 Namespacing of Routines and RPCs

Each tutorial user should choose a unique namespace beginning with **Z**, concatenated with two or three other letters, for example ZYXU. Use this namespace as the beginning of the names for all routines and RPCs created during this tutorial. Using the unique namespace protects the system you are using from having existing routines and RPCs overwritten. This namespace is referred to as Z*xxx* during the tutorial.

7.2.2 Tutorial Prerequisites

To use this tutorial:

- User should already have M programming skills, and some familiarity with Delphi and Object Pascal.

- User *must* have Delphi and the Broker Development Kit (BDK) installed on the workstation.

- The client workstation *must* have network access to an M account that is running a RPC Broker server process.

- Users *must* have programmer access in this M account, and it should be a Test account (*not* Production). Also, users need the [XUPROGMODE](#) security key assigned to their user account.

7.3 Tutorial—Step 1: RPC Broker Component

The first step of this tutorial is to create a Delphi-based application that includes a [TRPCBroker Component](#).

To create a Delphi-based application that includes a [TRPCBroker Component](#), do the following:

1. In Delphi, create a new application. Delphi creates a blank form, named **Form1**.

2. Set Form1's **Caption** property to **Terminal Type Display**.

3. From the **Kernel** component palette tab, add a [TRPCBroker Component](#) to your form. The instance of the component is automatically named **RPCBroker1**. It should be renamed to **brkrRPCBroker1**.

NOTE: In general the name of the component can be any meaningful name that begins with "**brkr**" to indicate a [TRPCBroker Component](#).

4. Leave the default values for Server and ListenerPort (see Server Property and ListenerPort Property) as is (they are retrieved from your workstation's Registry). In Section 7.4 you will add code to retrieve these values at run-time from the workstation's Registry.

5. Set the ClearParameters Property and ClearResults Property to **True** if they are not set to **True** already. This ensures that each time a call to an RPC is made, the Results Property is cleared beforehand, and the Param Property is cleared afterwards.

6. Your form should look like Figure 81:

Figure 80: Tutorial—Step 1: RPC Broker Component: Sample Form Output
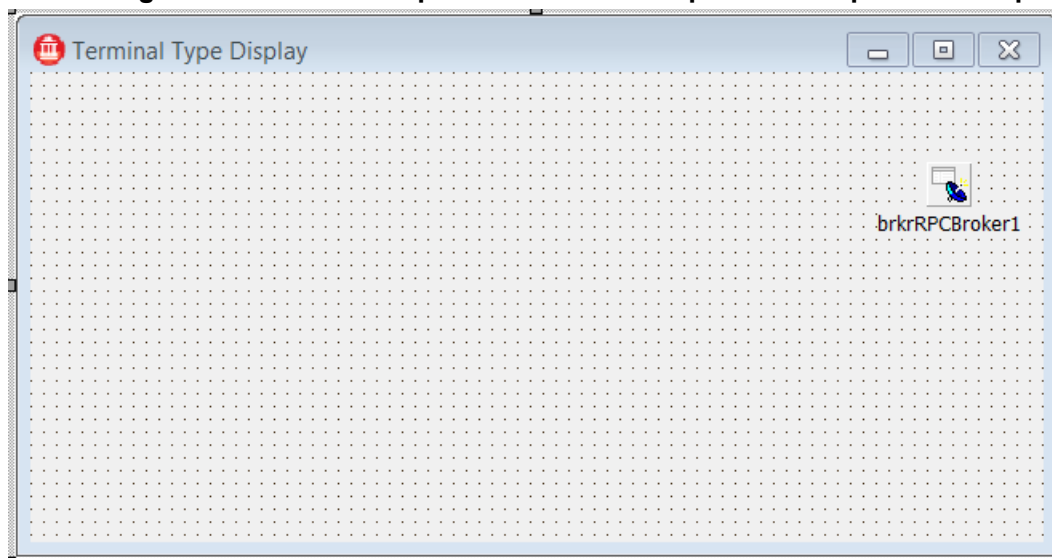
The next tasks are to use the TRPCBroker Component to retrieve the client workstation's RPC Broker server and port information (Tutorial—Step 2: Get Server/Port), and then to establish a connection through the TRPCBroker Component to the VistA M Server (Tutorial—Step 3: Establish Broker Connection).

7.4   Tutorial—Step 2: Get Server/Port

The TRPCBroker Component added to your form is hard-coded to access the Broker server and listener port that it picks up from the (developer) workstation (by default, **BROKERSERVER** and **<REDACTED>**). Naturally, you do *not* want this to be the only server and port to which your application can connect. To retrieve the end-user workstation's designated Broker server and port to connect, as stored in their Registry, you can use the GetServerInfo Function.

To retrieve the end-user workstation's designated server and port, do the following:

1. Include the **RPCConf1 Unit** in the Pascal file's **uses** clause. This is the unit of which GetServerInfo Function is a part.

2. Double-click on a blank region of the form. This creates an event handler procedure, **TForm1.FormCreate**, in the Pascal source code.

3. Add code to the **FormCreate** event handler that retrieves the correct server and port to connect, using the GetServerInfo Function. If **mrCancel** is returned, the code should quit. Otherwise, the code should then set **brkrRPCBroker1**'s Server Property and ListenerPort Property to the returned values.

   The code should look like Figure 82:

Figure 81: Tutorial—Step 2: Get Server/Port: Example

```
procedure TForm1.FormCreate(Sender: TObject);
var
    ServerStr: String;
    PortStr: String;
begin
    // Get the correct port and server from the Registry.
    if GetServerInfo(ServerStr,PortStr)<> mrCancel then
    begin
      brkrRPCBroker1.Server:=ServerStr;
      brkrRPCBroker1.ListenerPort:=StrToInt(PortStr);
    {connectOK}
    end
    else
       Application.Terminate;
end;
```

4. Now that you have code to retrieve the appropriate RPC Broker server and listener port, the next step of the tutorial (Tutorial—Step 3: Establish Broker Connection) is for the application to use the TRPCBroker Component to establish a connection to the VistA M Server.

7.5 Tutorial—Step 3: Establish Broker Connection

Now that the application can determine the appropriate RPC Broker server and port to connect (Tutorial—Step 2: Get Server/Port), add code to establish a connection to the designated RPC Broker server from the application. The act of establishing a connection leads the user through signon. If signon succeeds, a connection is established.

To establish a connection from the application to a RPC Broker server, do the following:

1. Add code to Form1's **OnCreate** event handler. The code should:

   a. Set **brkrRPCBroker1**'s Connected Property to **True** (inside of an exception handler **try...except** block). This causes an attempt to connect to the RPC Broker server.

   b. Check if an **EBrokerError** exception is raised. If this happens, connection failed, and the code should inform the user of this and terminate the application.

   The **OnCreate** event handler should now look like Figure 83:

Figure 82: Tutorial—Step 3: Establish Broker Connection: Example

```
procedure TForm1.FormCreate(Sender: TObject);
var
    ServerStr: String;
    PortStr: String;
begin
    // Get the correct port and server from the Registry.
    if GetServerInfo(ServerStr,PortStr)<> mrCancel then
    {connectOK begin}
    begin
       brkrRPCBroker1.Server:=ServerStr;
       brkrRPCBroker1.ListenerPort:=StrToInt(PortStr);
       // Establish a connection to the RPC Broker server.
       try
          brkrRPCBroker1.Connected:=True;
       except
          On EBrokerError do
          {error begin}
          begin
             ShowMessage('Connection to server could not be established!');
             Application.Terminate;
          {error end}
          end;
       {try end}
       end;
    {connectOK end}
    end
    else
       Application.Terminate;
end;
```

NOTE: Every call that invokes an RPC Broker server connection should be done in an "exception handler" **try...except** block, so that EBrokerError exceptions can be trapped.

2. Save, compile and run the application. It should connect to the VistA M Server returned by the GetServerInfo Function. You may be prompted to sign on with 2-factor authentication (2FA) or Access and Verify codes. If you can connect successfully, the application runs (at this point, it is just a blank form). Otherwise, troubleshoot the RPC Broker connection until the application connects.

3. If the server system defined in the Registry is *not* the development system (the one on which RPCs are created for this application), update the Registry using the **ServerList.exe** program so that the application connects to the proper VistA M Server.

4. Now that the application can establish a connection to the end-user's server system, you can retrieve data from the VistA M Server.

   The next steps of the tutorial create a custom RPC that retrieves a list of all of the terminal types on the VistA M Server and calls that RPC from the application.

7.6 Tutorial—Step 4: Routine to List Terminal Types

Now that the application uses an RPC Broker component to connect correctly to an RPC Broker server (Tutorial—Step 3: Establish Broker Connection), you are ready to create custom RPCs that the application can call. For the tutorial, you will create an RPC that retrieves the list of all terminal types from the RPC Broker server.

The first step in creating an RPC is to create the routine that the RPC executes. You *must* create its input and output in a defined format that is compatible with being executed as an RPC.

To create the routine that the RPC executes, do the following:

1. Choose the data format that the RPC should return. The type of data needed to return to the client application determines the format of the routine that the RPC calls. There are five return value types for RPCs:

   - **SINGLE VALUE**

   - **ARRAY**

   - **WORD PROCESSING**

   - **GLOBAL ARRAY**

   - **GLOBAL INSTANCE**

Since the type of data the tutorial application would like returned is a list of terminal types, and that list could be quite long, use a return value type **GLOBAL ARRAY** for the RPC. For the routine called by the RPC, this means that:

- The routine should return a list of terminal types in a global. Each terminal type should be on an individual data node, subscripted numerically.

- The return value of the routine (always returned in the routine's first parameter) should be the global reference of the data global, in closed root form. The data nodes should be one level descendant from the global reference.

2. In the M account that the TRPCBroker Component connects to, create a routine that outputs a list of terminal types in the format determined above. The format for each data node that is returned for a terminal type could be anything; for the sake of this application, set each data node to "**ien^.01 field**" for the terminal type in question. Store each node in **^TMP($J,"ZxxxTT",#)**, as shown in Figure 84.

Figure 83: Tutorial—Step 4: Routine to List Terminal Types: Example

```
ZxxxTT ;ISC-SF/KC TUTORIAL RTN, BRK 1.1; 7/22/97
 ;;1.0;;
TERMLIST(GLOBREF) ; retrieve list of term types
 ; return list in ^TMP($J,"ZxxxTT")
 ; format of returned results: ien^.01 field
 N %                                      ;   scratch variable
 K ^TMP($J,"ZxxxTT")              ;   clear data return area
 D LIST^DIC(3.2)                          ;   retrieve list of termtype entries
 ; now set termtype entries into data global
 I '$D(DIERR) D
 .S %=0 F  S %=$O(^TMP("DILIST",$J,2,%)) Q:%=""  D
 ..S
^TMP($J,"ZxxxTT",%)=$G(^TMP("DILIST",$J,2,%))_"^"_$G(^TMP("DILIST",$J,1,%))
 K ^TMP("DILIST",$J)                        ; clean up
 S GLOBREF=$NA(^TMP($J,"ZxxxTT"))  ; set return value
 Q
```

3. Test the routine. Call it like the Broker would:

```
> D TERMLIST^ZxxxTT(.RESULT)
```

a. Confirm that the return value is the correct global reference:

```
> W RESULT
^TMP(566363396,"ZxxxTT")
```

b. Confirm that the data set into the global is in the format shown in Figure 85:

Figure 84: Tutorial—Step 4: Routine to List Terminal Types: Example confirming global data format

```
^TMP(566347920,"ZxxxTT",1) = 1^C-3101
^TMP(566347920,"ZxxxTT",2) = 2^C-ADDS
^TMP(566347920,"ZxxxTT",3) = 3^C-ADM3
^TMP(566347920,"ZxxxTT",4) = 38^C-DATAMEDIA
^TMP(566347920,"ZxxxTT",5) = 106^C-DATATREE
^TMP(566347920,"ZxxxTT",6) = 4^C-DEC
^TMP(566347920,"ZxxxTT",7) = 5^C-DEC132
^TMP(566347920,"ZxxxTT",8) = 93^C-FALCO
^TMP(566347920,"ZxxxTT",9) = 6^C-H1500
^TMP(566347920,"ZxxxTT",10) = 103^C-HINQLINK
^TMP(566347920,"ZxxxTT",11) = 132^C-HINQLINK
^TMP(566347920,"ZxxxTT",12) = 63^C-HP110
^TMP(566347920,"ZxxxTT",13) = 34^C-HP2621
```

4. Once you have tested the routine, and confirmed that it returns data correctly, the next step (Tutorial—Step 5: RPC to List Terminal Types) is to create the RPC that calls this routine.

7.7 Tutorial—Step 5: RPC to List Terminal Types

Now that you have created an RPC-compatible routine to list terminal types (Tutorial—Step 4: Routine to List Terminal Types), you can go ahead and create the RPC itself [the entry in the REMOTE PROCEDURE (#8994) File] that calls the routine.

To create an RPC that uses the **TERMLIST^ZxxxTT** routine, do the following:

1. Using VA FileMan, create a new RPC entry in the REMOTE PROCEDURE (#8994) File. Set up the RPC as shown in Figure 86:

<p align="center">Figure 85: Tutorial—Step 5: RPC to List Terminal Types: Example</p>

```
NAME: ZxxxTT LIST
TAG: TERMLIST
ROUTINE: ZxxxTT
RETURN VALUE TYPE: GLOBAL ARRAY
WORD WRAP ON: TRUE
DESCRIPTION: Used in RPC Broker developer tutorial.
```

2. The RPC's **RETURN VALUE TYPE** is set to **GLOBAL ARRAY**. This means that the RPC expects a return value that is a global reference (with data stored at that global reference).

3. Also, the RPC's **WORD WRAP ON** is set to **TRUE**. This means each data node from the VistA M Server is returned as a single node in the Results Property of the TRPCBroker Component in Delphi. Otherwise, the data would be returned concatenated into a single node in the Results Property.

4. The next step of the tutorial (Tutorial—Step 6: Call ZxxxTT LIST RPC) is to call this RPC from the tutorial application, through its TRPCBroker Component.

7.8 Tutorial—Step 6: Call ZxxxTT LIST RPC

Once you have created and tested the **ZxxxTT LIST** RPC on the VistA M Server, use the Delphi-based application's TRPCBroker Component to call that RPC.

To call the **ZxxxTT LIST** RPC from the Delphi-based application to populate a list box, do the following:

1. Place a **TListBox** component on the form. It should be automatically named **ListBox1**.

   Size it so that it uses the full width of the form, and half of the form's height.

2. Place a button beneath **ListBox1**:

   - Set its caption to "Retrieve Terminal Types".
   - Size the button so that it is larger than its caption.

3. Double-click on the button. This creates an event handler procedure, **TForm1.Button1Click**, in the Pascal source code.

4. In the **TForm1.Button1Click** event handler, add code to call the **ZxxxTT LIST** RPC and populate the list box with the retrieved list of terminal type entries. This code should:

   a. Set **RCPBroker1**'s RemoteProcedure Property to **ZxxxTT LIST**.

   b. Call **brkrRPCBroker1**'s Call Method (in a **try...except** exception handler block) to invoke that RPC.

   c. Retrieve results from **brkrRPCBroker1**'s Results Property, setting them one-by-one into the list box's **Items** property.

   This code should look like the code in Figure 87:

Figure 86: Tutorial—Step 6: Call ZxxxTT LIST RPC: Example

```
Procedure TForm1.Button1Click(Sender: TObject);
var
    i: integer;
begin
    brkrRPCBroker1.RemoteProcedure:='ZxxxTT LIST';
    try
      {call begin}
      begin
         brkrRPCBroker1.Call;
         ListBox1.Clear;
         for i:=0 to (brkrRPCBroker1.Results.Count-1) do
            ListBox1.Items.Add(piece(brkrRPCBroker1.Results[i],'^',2));
      {call end}
      end;
    except
      On EBrokerError do
         ShowMessage('A problem was encountered communicating with the
server.');
    {try end}
    end;
end;
```

5. Include the **mfunstr** unit in the **Uses** clause of the project's Pascal source file. This enables the application to use the piece function included in **mfunstr** (see the "XWB IM HERE" section).

6. The user account *must* have XUPROGMODE security key assigned. This allows the application to execute any RPC, without the RPC being registered. Later in the tutorial you will register your RPCs.

7. Run the application and click **Retrieve Terminal Types**. It should retrieve and display terminal type entries, and appear as shown in Figure 88:

Figure 87: Tutorial—Step 6: Call ZxxxTT LIST RPC: Sample Output Form

8. Now that you can retrieve a list of terminal type entries, the next logical task is to retrieve a particular entry when a user selects that entry in the list box.

7.9   Tutorial—Step 7: Associating IENs

When a user selects a terminal type entry in the list box, a typical action is to retrieve the corresponding record and display its fields. The key to retrieving any VA FileMan record is knowing the IEN of the record. Thus, when a user selects an entry in the list box, you need to know the IEN of the corresponding VA FileMan entry. However, the list box items themselves only contain the name of each entry, not the IEN.

The subscripting of items in the list box still matches the original subscripting of items returned in **brkrRPCBroker1**'s Results Property, as performed by the following code in **Button1Click** event handler:

```
for i:=0 to (brkrRPCBroker1.Results.Count-1) do
    ListBox1.Items.Add(piece(brkrRPCBroker1.Results[i],'^',2));
```

If no further calls to **brkrRPCBroker1** were made, you could simply refer back to **brkrRPCBroker1**'s **Results[x]** item to obtain the matching IEN of a list boxes' **Items[x]** item. But, since **brkrRPCBroker1** is used again, the Results Property is cleared. So, the results *must* be saved off in another location, if you want to be able to refer to them after other Broker calls are made.

To save off the **Results** to another location, do the following:

1. Create a variable named **TermTypeList**, of type **TStrings**. This is where **brkrRPCBroker1.Results** is saved. Create the variable in the section of code where **TForm1** is defined as a class:

Figure 88: Tutorial—Step 7: Associating IENs: Example of Creating a Variable to Save Results

```
type
   TForm1 = class(TForm)
      brkrRPCBroker1: TRPCBroker;
      ListBox1: TListBox;
      Button1: TButton;
      procedure FormCreate(Sender: TObject);
      procedure Button1Click(Sender: TObject);
   private
      {Private declarations}
   public
      {Public declarations}
// Added declaration of TermTypeList.
      TermTypeList: TStringList;
end;
```

2. In **Form1**'s **OnCreate** event handler, call the **Create** method to initialize the **TermTypeList**. Do this in the first line of code of the event handler:

```
TermTypeList:=TStringList.Create;
```

3. Create an event handler for **Form1**'s **OnDestroy** event (select **Form1**, go to the **Events** tab of the Object Inspector, and double-click on the right-hand column for the **OnDestroy** event). In that event handler, add one line of code to call the **Free** method for **TermTypeList**. This frees the memory used by the list:

Figure 89: Tutorial—Step 7: Associating IENs: Example of Creating an Event Handler to Free Memory

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
   TermTypeList.Free;
end;
```

4. In **Button1**'s **OnClick** event handler, add a line of code to populate **TermTypeList** with the list of terminal types returned in **brkrRPCBroker1**'s <u>Results Property</u>. This code uses the **Add** method of **TStrings** sequentially so that the subscripting of **TermTypeList** matches the subscripting of **Results**. The code for that event handler should then look like <u>Figure 91</u>:

Figure 90: Tutorial—Step 7: Associating IENs: Example of Creating an Event Handler to Populate a List of Terminal Types

```
procedure TForm1.Button1Click(Sender: TObject);
var
     i: integer;
begin
   brkrRPCBroker1.RemoteProcedure:='Zxxx LIST';
   try
   {call begin}
     begin
        brkrRPCBroker1.Call;
        for i:=0 (brkrRPCBroker1.Results.Count-1) do begin {copy begin}
        ListBox1.Items.Add(piece(brkrRPCBroker1.Results[i],'^',2));
           // Added line.
           TermTypeList.Add(brkrRPCBroker1.Results[i]);
        {copy end}
        end;
     {call end}
     end;
   except
     On EBrokerError do
        ShowMessage('A problem was encountered communicating with the
server.');
   {try end}
   end;
end;
```

5. Determine (and display) the IEN of the corresponding terminal type when a user selects an item in the list box:

   a. Create an **OnClick** event handler for **ListBox1** by double-clicking on the list box.

   b. Add code to the new event handler that checks if an item is selected. If an item is selected in the list box, display the first piece of the corresponding item saved off in the **TermTypeList** array (the index subscripts of **TermTypeList** and of the list box match each other). This is the IEN of the corresponding VA FileMan entry.

Figure 91: Tutorial—Step 7: Associating IENs: Example of Creating an Event Handler to Check if an Item is Selected

```
procedure TForm1.ListBox1Click(Sender: TObject);
var
    ien: String;
begin
    if (ListBox1.ItemIndex <> -1) then
    {displayitem begin}
    begin
       ien:=piece(TermTypeList[ListBox1.ItemIndex],'^',1);
       ShowMessage(ien);
    {displayitem end}
    end;
end;
```

6. Compile and run the application. When you click on an item in the list box, the IEN corresponding to that item should be displayed in a popup message window.

7. Now that you can determine the IEN of any entry the user selects in the list box, you can retrieve and display the corresponding VA FileMan record for any selected list box entry.

7.10 Tutorial—Step 8: Routine to Retrieve Terminal Types

Now that you have associated an IEN for each record displayed in the list box (Tutorial—Step 7: Associating IENs), you can use that IEN to display fields from any terminal type entry in the list box that a user selects. To retrieve the fields for any selected terminal type entry, create a second custom RPC. This RPC needs to take an input parameter, the record IEN, to know which record to retrieve.

To create an RPC routine to retrieve individual terminal type records, do the following:

1. Choose the data format that the RPC should return. The type of data needed to return to the client application determines the format of the routine that the RPC calls. In this case, the RPC should, given an IEN, return fields **.01**, **1**, **2**, **3**, **4**, **6**, and **7** (Name, Right Margin, Form Feed, Page Length, Back Space, Open Execute, and Close Execute).

   Since this information is constrained and small, a return type of ARRAY would be suitable for this RPC. The return format of the array is arbitrary; for the sake of this application, the routine should return fields **.01**, **1**, **2**, **3**, and **4** in node **0**; field **6** (a **245-character** field) in node **1**; and field **7** (also a **245-character** field) in node **2**. This array *must* be returned in the first parameter to the routine.

2. The routine for this RPC also needs to be able to take an IEN as an input parameter. Any additional input parameters, such as one for the IEN, *must follow* the required input parameter in which results are returned.

3. Add a second subroutine to the **ZxxxTT** routine for the second RPC, similar to Figure 93. This subroutine uses an IEN to retrieve fields for a particular terminal type. It then sets three result nodes, each containing a specified set of fields for the record corresponding to the IEN parameter.

Figure 92: Tutorial—Step 8: Routine to Retrieve Terminal Types: Example of a Subroutine to Retrieve Fields for a Particular Terminal Type and Set Result Nodes

```
TERMENT(RESULT,IEN) ; retrieve a string of fields for a termtype
 ; format of results (by field number):
 ; RESULT(0)=.01^1^2^3^4
 ; RESULT(1)=6
 ; RESULT(2)=7
 ;
 N I,ARRAY S IEN=IEN_",",RESULT(1)=""
 D GETS^DIQ(3.2,IEN,".01;1;2;3;4;6;7","","ARRAY")
 S RESULT(0)=""
I '$D(DIERR) D
 .F I=.01,1,2,3,4 D
 ..S RESULT(0)=RESULT(0)_ARRAY(3.2,IEN,I)_"^"
 .S RESULT(1)=ARRAY(3.2,IEN,6)
 .S RESULT(2)=ARRAY(3.2,IEN,7)
 Q
```

4. Test the routine. Call it like the Broker would:

```
>D TERMENT^ZxxxTT(.ARRAY,103)
```

5. Confirm that the return array contains the specified fields in the nodes shown in Figure 94:

Figure 93: Tutorial—Step 8: Routine to Retrieve Terminal Types: Example Confirming Returned Array Contains the Specified Fields

```
ARRAY(0)=C-HINQLINK^80^#,$C(27,91,50,74,27,91,72)^24^$C(8)^
ARRAY(1)=U $I:(0:255::255:512)
ARRAY(2)=U $I:(:::::512) C $I
```

6. Once you have tested the routine, and confirmed that it returns data correctly, the next step (Tutorial—Step 9: RPC to Retrieve Terminal Types) is to create the RPC that calls this routine.

7.11 Tutorial—Step 9: RPC to Retrieve Terminal Types

Now that you have created an RPC-compatible routine to retrieve fields from a terminal type record (Tutorial—Step 8: Routine to Retrieve Terminal Types), create the RPC itself.

To create an RPC that uses the **TERMENT^ZxxxTT** routine, do the following:

1. Using VA FileMan, create a new RPC entry in the REMOTE PROCEDURE (#8994) File. Set up the RPC as shown in Figure 95:

Figure 94: Tutorial—Step 9: RPC to Retrieve Terminal Types: Example of an RPC Setup

```
NAME: ZxxxTT RETRIEVE
TAG: TERMENT
ROUTINE: ZxxxTT
RETURN VALUE TYPE: ARRAY
DESCRIPTION: Used in RPC Broker tutorial.
INPUT PARAMETER: IEN PARAMETER TYPE: LITERAL
```

2. The RPC's **RETURN VALUE TYPE** is set to **ARRAY**. This means that the RPC expects a return value that contains results nodes, each subscripted only at the first subscript level.

3. The **WORD WRAP ON** setting does *not* affect RPCs whose **RETURN VALUE TYPE** is **ARRAY**.

4. The additional input parameter needed to pass in a record IEN is documented in the INPUT PARAMETER Multiple. Its parameter type is **LITERAL**, which is appropriate when being passed the numeric value of an IEN.

5. This RPC can now be called from a Delphi-based application, through the TRPCBroker Component.

7.12 Tutorial—Step 10: Call ZxxxTT RETRIEVE RPC

When a user selects a terminal type entry in the list box, the **OnClick** event is triggered. The **ZxxxTT RETRIEVE** RPC can be called from that **OnClick** event, as a replacement for the code there that simply displays the IEN of any selected record.

To use the **ZxxxTT RETRIEVE** RPC to display fields from a selected terminal type, do the following:

1. Create labels and edit boxes for each of the fields the RPC returns from the Terminal type file:

Table 39: Tutorial—Step 10: Call ZxxxTT RETRIEVE RPC: Sample RPC Fields Returned and Label Information

Terminal Type Field:	Add a TEdit component named:	Add a Label with the Caption:
.01	Name	Name
1	RightMargin	Right Margin:
2	FormFeed	Form Feed:
3	PageLength	Page Length:
4	BackSpace	Back Space:
6	OpenExecute	Open Execute:
7	CloseExecute	Close Execute:

2. Update **ListBox1**'s **OnClick** event handler. Add code so that when the user clicks on an entry in the list box, the application calls the **ZxxxTT RETRIEVE** RPC to retrieve fields for the corresponding terminal type and displays those fields in the set of **TEdit** controls on the form. This code should:

   a. Set **RCPBroker1**'s RemoteProcedure Property to **ZxxxTT RETRIEVE**.

   b. Pass the IEN of the selected terminal type to the RPC, using the TRPCBroker Component's **RunTime** Param Property. Pass the IEN in the Value Property (i.e., **brkrRPCBroker1.Param[0].Value**).

   c. Pass the **PType** for the IEN parameter in the **brkrRPCBroker1.Param[0].PType**. Possible types are literal, reference, and list. In this case, to pass in an IEN, the appropriate **PType** is literal.

   d. Call **brkrRPCBroker1**'s Call Method (in a **try...except** exception handler block) to invoke the **ZxxxTT RETRIEVE** RPC.

   e. Set the appropriate pieces from each of the three **Results** nodes into each of the **TEdit** boxes corresponding to each returned field.

The code for the **OnClick** event handler should look like Figure 96:

Figure 95: Tutorial—Step 10: Call ZxxxTT RETRIEVE RPC: Sample of an OnClick Event Handler

```
procedure TForm1.ListBox1Click(Sender: TObject);
var
    ien: String;
begin
    if (ListBox1.ItemIndex <> -1) then
    {displayitem begin}
    begin
       ien:=piece(TermTypeList[ListBox1.ItemIndex],'^',1);
       brkrRPCBroker1.RemoteProcedure:='ZxxxTT RETRIEVE';
       brkrRPCBroker1.Param[0].Value := ien;
       brkrRPCBroker1.Param[0].PType := literal;
       try
          {call code begin}
          begin
          brkrRPCBroker1.Call;
             Name.Text:=piece(brkrRPCBroker1.Results[0],'^',1);
             RightMargin.Text:=piece(brkrRPCBroker1.Results[0],'^',2);
             FormFeed.Text:=piece(brkrRPCBroker1.Results[0],'^',3);
             PageLength.Text:=piece(brkrRPCBroker1.Results[0],'^',4);
             BackSpace.Text:=piece(brkrRPCBroker1.Results[0],'^',5);
             OpenExecute.Text:=brkrRPCBroker1.Results[1];
             CloseExecute.Text:=brkrRPCBroker1.Results[2];
          {call code end}
          end;
       except
       On EBrokerError do
          ShowMessage('A problem was encountered communicating with the
server.');
       {try end}
       end;
    {displayitem end}
    end;
end;
```

3. Compile and run the application. When you click on an entry in the list box now, the corresponding fields should be retrieved and displayed in the set of edit boxes on your form, as shown in Figure 97:

Figure 96: Tutorial—Step 10: Call ZxxxTT RETRIEVE RPC: Testing the Application
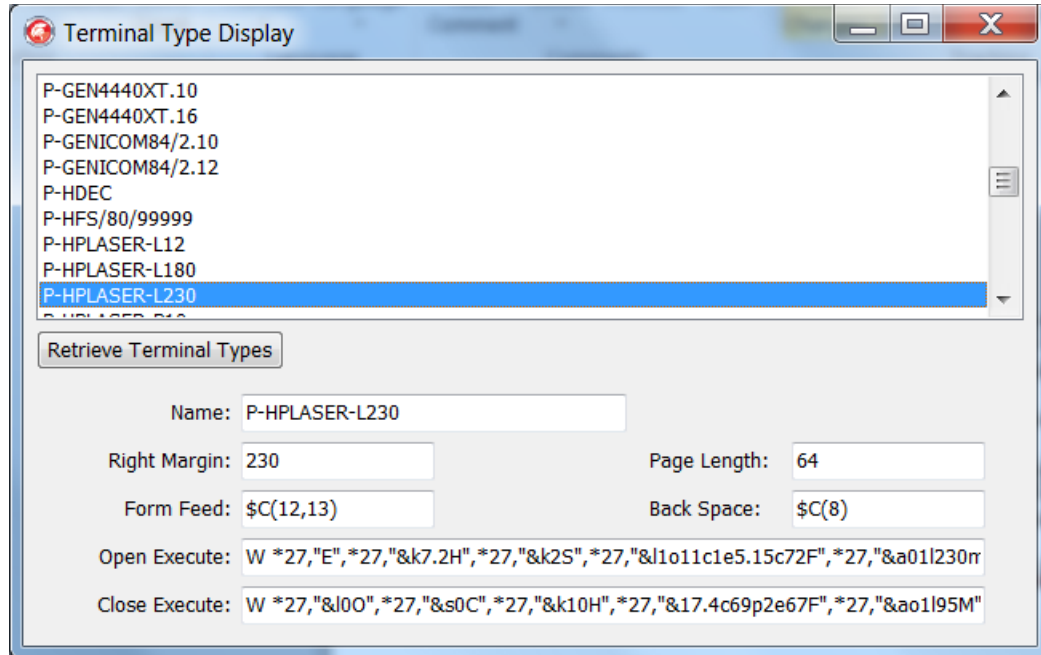
7.13 Tutorial—Step 11: Register RPCs

Up until now, it has been assumed that the only user of the application is you, and that you have programmer access and the XUPROGMODE security key in the account where the RPCs are accessed.

Under any other circumstance, any RPCs that the application uses *must* be registered for use by the application on the host system. Registration authorizes the RPCs for use by the client based on user privileges.

To register the RPCs used by the tutorial application, do the following:

1. Create an option of type "**B**" (Broker). For example, create an option called **ZxxxTT TERMTYPE** for the tutorial application.

2. In the "**B**"-type option's RPC multiple, make one entry for each RPC the application calls. In the case of this tutorial, there should be two entries:

   • **ZxxxTT LIST**

   • **ZxxxTT RETRIEVE**

3. Follow the steps in the "RPC Security: How to Register an RPC" section to create an application context, using the **ZxxxTT TERMTYPE** option.

   Essentially, add a line of code that calls the CreateContext Method, and terminates the application if **False** is returned. The code for **Form1**'s **OnCreate** event should now look like Figure 98:

Figure 97: Tutorial—Step 11: Register RPCs: Example

```
procedure TForm1.FormCreate(Sender: TObject);
var
    ServerStr: String;
    PortStr: String;
begin
    TermTypeList:=TStringList.Create;
    // Get the correct port and server from Registry.
    if GetServerInfo(ServerStr,PortStr)<> mrCancel then
    {connectOK begin}
    begin
      brkrRPCBroker1.Server:=ServerStr;
      brkrRPCBroker1.ListenerPort:=StrToInt(PortStr);
      // Establish a connection to the RPC Broker server.
      try
          brkrRPCBroker1.Connected:=True;
          // Check security.
          if not brkrRPCBroker1.CreateContext('ZxxxTT TERMTYPE') then
            Application.Terminate;
      except
          On EBrokerError do
          {error begin}
          begin
            ShowMessage('Connection to server could not be established!');
            Application.Terminate;
          {error end}
          end;
      {try end}
      end;
    {connectOK end}
    end
    else
      Application.Terminate;
end;
```

4. Compile and run the application. Try running it both with and without the XUPROGMODE security key assigned to you. Without the XUPROGMODE security key, you are *not* able to run the application unless the **ZxxxTT TERMTYPE** option is assigned to your menu tree.

7.14 Tutorial—Using VA FileMan Delphi Components (FMDC)

Congratulations! You have created a sample application that performs entry lookup and retrieves fields from any record selected by the end-user. You are now ready to create Delphi-based applications using the RPC Broker.

If the application needs to perform database tasks with VA FileMan on a VistA M Server, consider using the FileMan Delphi Components (FMDC). These components automate the major tasks of working with database records through Delphi. Among the functions they provide are:

- Automated entry retrieval into a set of controls.

- Automated online help for database fields.

- Automated validation of user data entry.

- Automated filing of changed data.

- IEN tracking in all controls.

- Automated DBS error tracking on the Delphi client.

- Generic lookup dialogue.

- Record locking.

- Record deletion.

If you need to do more than the most simple database tasks in your Delphi-based applications, the FileMan Delphi Components (FMDC) encapsulate most of the coding needed to retrieve, validate, and file VA FileMan data.

REF: For more information on the VA FileMan Delphi Components (FMDC), see the FMDC VA Intranet website.

7.15 Tutorial—Source Code (Sample)

Figure 98: Tutorial Source Code

```
unit tut1;

interface

Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, Trpcb,
RPCConf1, StdCtrls, MFunStr;

type
    TForm1 = class(TForm)
      brkrRPCBroker1: TRPCBroker;
      ListBox1: TListBox;
      Button1: TButton;
      Name: TEdit;
      RightMargin: TEdit;
      FormFeed: TEdit;
      OpenExecute: TEdit;
      CloseExecute: TEdit;
      PageLength: TEdit;
      BackSpace: TEdit;
      Label1: TLabel;
      Label2: TLabel;
      Label3: TLabel;
      Label4: TLabel;
      Label5: TLabel;
      Label6: TLabel;
      Label7: TLabel;
      procedure FormCreate(Sender: TObject);
      procedure Button1Click(Sender: TObject);
      procedure ListBox1Click(Sender: TObject);
      procedure FormDestroy(Sender: TObject);
    private
      {Private declarations}
    public
      {Public declarations}
      // Added declaration of TermTypeList.
      TermTypeList: TStrings;
    end;

var
    Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
var
    ServerStr: String;
    PortStr: String;
begin
    TermTypeList:=TStringList.Create;
    // Get the correct port and server from the Registry.
    if GetServerInfo(ServerStr,PortStr)<> mrCancel then
    {connectOK begin}
    begin
      brkrRPCBroker1.Server:=ServerStr;
      brkrRPCBroker1.ListenerPort:=StrToInt(PortStr);
```

```
      // Establish a connection to the RPC Broker server.
      try
          brkrRPCBroker1.Connected:=True;
          if not brkrRPCBroker1.CreateContext('ZxxxTT TERMTYPE') then
             Application.Terminate;
      except
          On EBrokerError do
          {error begin}
          begin
             ShowMessage('Connection to server could not be established!');
             Application.Terminate;
          {error end}
          end;
      {try end}
      end;
   {connectOK end}
   end
   else
      Application.Terminate;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
    i: integer
    brkrRPCBroker1.RemoteProcedure:='ZxxxTT LIST';
    try
       {call begin}
       begin
          brkrRPCBroker1.Call;
       for i:=0 to (brkrRPCBroker1.Results.Count-1) do begin  {copy begin}
          ListBox1.Items.Add(piece(brkrRPCBroker1.Results[i],'^',2));
          // Added line.
          TermTypeList.Add(brkrRPCBroker1.Results[i]);
        {copy end}
        end;
          {call end}
          end;
    except
       On EBrokerError do
          ShowMessage('A problem was encountered communicating with the server.');
    {try end}
begin
    end;
end;

procedure TForm1.ListBox1Click(Sender: TObject);
var
    ien: String;
begin
    if (ListBox1.ItemIndex <> -1) then
    {displayitem begin}
    begin
       ien:=piece(TermTypeList[ListBox1.ItemIndex],'^',1);
       brkrRPCBroker1.RemoteProcedure:='ZxxxTT RETRIEVE';
       brkrRPCBroker1.Param[0].Value := ien;
       brkrRPCBroker1.Param[0].PType := literal;
       try
          {call code begin}
          begin
             brkrRPCBroker1.Call;
             Name.Text:=piece(brkrRPCBroker1.Results[0],'^',1);
             RightMargin.Text:=piece(brkrRPCBroker1.Results[0],'^',2);
             FormFeed.Text:=piece(brkrRPCBroker1.Results[0],'^',3);
```

```
            PageLength.Text:=piece(brkrRPCBroker1.Results[0],'^',4);
            BackSpace.Text:=piece(brkrRPCBroker1.Results[0],'^',5);
            CloseExecute.Text:=brkrRPCBroker1.Results[2];
        {call code end}
        end;
    except
    On EBrokerError do
        ShowMessage('A problem was encountered communicating with the server.');
    {try end}
    end;
    {displayitem end}
    end;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
    TermTypeList.Free;
end;

end.
```

7.16 Silent Login

The RPC Broker provides "Silent Login" capability. Silent Login is a way to log in a user with known login information. Silent Login skips the step of asking the user for login information. It provides functionality associated with the ability to make logins to a VistA M Server without the RPC Broker asking for Access and Verify code information.

ℹ️ **REF:** For some examples, see the "Silent Login Examples" section.

There are three types of Silent Login provided with the RPC Broker 1.1 BDK:

- **Access/Verify Code**—Type of Silent Login that uses Access and Verify codes provided by the application. This type of Silent Login may be necessary for an application that runs as a background task and repeatedly signs on for short periods. Another case would be for applications that are interactive with the user but are running under conditions where they *cannot* provide a standard dialogue window, such as that used by the Broker to request Access and Verify codes. Examples might be applications running on handheld devices or within a browser window.

- **CCOW Token**—Type of Silent Login that uses a CCOW "User Context" token that is passed for authentication.

- **Auto Sign-On (ASO) Token**—Type of Silent Login that uses an ASO token obtained by one application that is passed along with other information as a command line argument to a second application that it is starting. The token is obtained from the VistA server and remains valid for about **twenty** (**20**) seconds. When the newly started application sends this token during login the server identifies the same user and completes the login.

Due to the various conditions under which Silent Logins might be used, it was also necessary to provide options to the applications on error handling and processing. Applications that run as

system services crash if they attempt to show a dialogue box. Similarly, applications running within Web browsers are *not* permitted to show a dialogue box or to accept Windows messages. Properties have been provided to permit the application to handle errors in a number of ways.

As a part of the Silent Login functionality, the TVistaUser Class providing basic user information was added. This class is used as a property by the **TRPCBroker** class and is filled with data following completion of the login process. This property and its associated data are available to all applications, whether or not they are using a Silent Login.

> **REF:** For more information on handling divisions during Silent Login, see the "Handling Divisions during Silent Login" section.

7.16.1 Handling Divisions during Silent Login

A login may be successful, but if the user has multiple divisions from which to choose and fails to select one, the connection is terminated, and a failed login message is generated. This becomes a potential problem in that a Silent Login can have problems if the user has multiple divisions from which to choose and the PromptDivision Property is *not* set to **True**.

If the application wishes to handle the user specification of the division, it can attempt to set the TRPCBroker Component Connected Property to **True**. If upon return, the Connected Property is still **False**, it can check the Login. MultiDivision Property. If the MultiDivision Property is **True**, the user has multiple divisions from which to choose. The application finds the possible values for selection in the Login. DivList Property (read-only) (i.e., **Tstrings**). The values that are present in the DivList Property (read-only) are similar to Figure 100:

Figure 99: DivList Property—Sample List of Divisions

```
3
1^SAN FRANCISCO^66235
2^NEW YORK^630
3^SAN DIEGO^664^1
```

The first (index = **0**) entry is the total number of divisions that can be selected (e.g., **3** in this example). This is followed by the different divisions comprised of the following pieces:

- The second ^-piece of each entry is the division name.
- The third ^-piece of each entry is the division number.
- The fourth ^-piece with the value of 1, if present in one of the entries, is the user's default division.

The safest value to set as the Login.Division property might be the third ^-piece of the selected division.

If the desired division is known ahead of time, it can be set into the Login.Division property for the TRPCBroker Component prior to attempting the connection.

7.16.2 Silent Login Examples

7.16.2.1 Example 1: lmAVCodes

Figure 101 is an example of how to use Silent Login by passing the Access and Verify codes to the TVistaLogin Class.

Figure 100: Silent Login—Example of Passing the Access and Verify Codes

```
brkrRPCBroker1.KernelLogIn := False;
brkrRPCBroker1.LogIn.Mode := lmAVCodes;
brkrRPCBroker1.LogIn.AccessCode := ********;
brkrRPCBroker1.LogIn.VerifyCodeCode := ********;
brkrRPCBroker1.LogIn.PromptDivison := True;
brkrRPCBroker1.LogIn.OnFailedLogin := myevent;
Try
    brkrRPCBroker1.Connected := True;
except
    exit
end;
```

If brkrRPCBroker1.Connected is **True**, then Silent Login has worked.

7.16.2.2 Example 2: lmAppHandle

Figure 102 is an example of how to use Silent Login by passing an Application Handle to the TVistaLogin Class.

The **lmAppHandle** mode of the Silent Login is used when an application starts up a second application. If the second application tests for arguments on the command line, it is possible for this application to be started and make a connection to the VistA M Server *without* user interaction.

An example of a procedure for starting a second application with data on the command line to permit a Silent Login using the **LoginHandle** provided by the first application is shown in Figure 102. This is followed by a procedure that can be called in the processing related to **FormCreate** to use this command line data to initialize the TRPCBroker Component for Silent Login (Figure 103).

> ⚠️ **CAUTION: The procedures shown here are included within the RpcSLogin unit and can be used directly from there.**

If the value for **ConnectedBroker** is **NIL**, the application specified in **ProgLine** is started and any command line included in **ProgLine** is passed to the application.

In the second application, a call to the Broker should be made shortly after starting, since the **LoginHandle** passed in has a finite lifetime (approximately **20 seconds**) during which it is valid for the Silent Login.

Figure 101: Silent Login—Example of Passing in an Application Handle

```
procedure StartProgSLogin(const ProgLine: String ; ConnectedBroker: TRPCBroker);
var
    StartupInfo: TStartupInfo;
    ProcessInfo: TProcessInformation;
    AppHandle: String;
    CmndLine: String;
begin
    FillChar(StartupInfo, SizeOf(TStartupInfo), 0);
    with StartupInfo do
    begin
       cb := SizeOf(TStartupInfo);
       dwFlags := STARTF_USESHOWWINDOW;
       wShowWindow := SW_SHOWNORMAL;
    end;
    CmndLine := ProgLine;
    if ConnectedBroker <> nil then
    begin
       AppHandle := GetAppHandle(ConnectedBroker);
       CmndLine := CmndLine + ' s='+ConnectedBroker.Server + ' p='
                                   + IntToStr(ConnectedBroker.ListenerPort) + ' h='
                                   + AppHandle + ' d=' +
ConnectedBroker.User.Division;
    end;
    CreateProcess(nil, Pchar(CmndLine), nil, nil, False,
          NORMAL_PRIORITY_CLASS, nil, nil, StartupInfo, ProcessInfo);
end;

{btnStart is clicked to start the second application Test2.exe}
procedure TForm1.btnStartClick(Sender: TObject);
var
    CurDir: string;
begin
    {Use Test2.exe and expecting it to be in the startup directory for the current
application}
    CurDir := ExtractFilePath(ParamStr(0)) + 'Test2.exe';
    {Now start application with Silent Login}
    StartProgSLogin(CurDir, brkrRPCB1);
end;
```

The following procedure (**CheckCmdLine**) would be called in the **FormCreate** code of the application being started to check for command line input, and if relevant to the Broker connection, to set it up.

This code assumes that the following commands are used in conjunction with the values shown:

- **s**= Server
- **p**= ListenerPort
- **d**= User.Division
- **h**= LoginHandle

The command line might look like:

```
ProgramName.exe s=DHCPSERVER p=<REDACTED> d=692 h=~1XM34XYYZZQQ_X
```

The [TRPCB Unit](#) and [RpcSLogin Unit](#) would need to be included in the **USES** clause.

Figure 102: Silent Login—Calling the CheckCmdLine Procedure

```
procedure CheckCmdLine(brkrRPCB: TRPCBroker);
var
   j: integer;
begin
   // Iterate through possible command line arguments
   for j := 0 to 15 do
   begin
     if ParamStr(j) <> '' then
        Form1.Memo1.Lines.Add(IntToStr(j) + ' ' + ParamStr(j));
     if Pos('p=',ParamStr(j)) > 0 then
        brkrRPCB.ListenerPort := StrToInt(Copy(ParamStr(j),
           (Pos('=',ParamStr(j))+1),length(ParamStr(j))));
     if Pos('s=',ParamStr(j)) > 0 then
        brkrRPCB.Server := Copy(ParamStr(j),
           (Pos('=',ParamStr(j))+1),length(ParamStr(j)));
     if Pos('h=',ParamStr(j)) > 0 then
     begin
        brkrRPCB.Login.LoginHandle := Copy(ParamStr(j),
           (Pos('=',ParamStr(j))+1),length(ParamStr(j)));
        if brkrRPCB.Login.LoginHandle <> '' then
        begin
          brkrRPCB.KernelLogIn := False;
          brkrRPCB.Login.Mode := lmAppHandle;
        end;
     end;
     if Pos('d=',ParamStr(j)) > 0 then
        brkrRPCB.Login.Division := Copy(ParamStr(j),
           (Pos('=',ParamStr(j))+1),length(ParamStr(j)));
   // for end
   end;
end;
```

7.17 Microsoft Windows Registry

Applications built with RPC Broker 1.1 use the Microsoft® Windows Registry to store the available servers and ports accessed via the Broker.

The Windows Registry replaces the **[RPCBroker_Servers]** section of the **VISTA.INI** file. The **VISTA.INI** file is no longer used by applications built with Broker 1.1. However, this file continues to be used by applications built using RPC Broker 1.0. During the installation of the Broker, relevant data from the **VISTA.INI** file is moved to the Windows Registry. Subsequent reads and writes are done via the Registry.

⚠ CAUTION: The VISTA.INI file created with RPC Broker 1.0 *must not* be removed from the Windows directory on the client workstation. It is still required for 16-bit Broker-based applications created using RPC Broker 1.0.

8   DLL Interfaces (C, C++, Visual Basic)

8.1   DLL Interface Introduction

RPC Broker 1.1 provides Dynamic Link Library (DLL) functions that allow applications written in *any* Microsoft® Windows-based development environment (e.g., Embarcadero's® Delphi, Embarcadero C++, Microsoft® Visual Basic, and other COTS products), to take advantage of all the features offered by the RPC Broker component. This reflects VistA's continued movement toward open systems that support multiple GUI and client front-ends.

The Dynamic Link Library (DLL) functions act like a "shell" around the Delphi **TRPCBroker** component and provide developers with an easy function-based access to the Broker component. These functions allow GUI and client front-end applications written in Embarcadero's Delphi and other COTS products to take advantage of all the features that the Broker offers. All of the communication to the server is handled by the **TRPCBroker** component accessed via the DLL interface.

The functionality of the TRPCBroker Component for Delphi is provided in a 32-bit Dynamic Link Library (DLL) interface, in **BAPI32.DLL**. This enables the use of any development product that can access Windows 32-bit DLLs to create applications that communicate with VistA M Servers through the RPC Broker.

> **i**   **NOTE:** The **BAPI32.DLL** contains all of the 32-bit Broker DLL functions. It provides an interface to the Broker component.

In Delphi, you have direct access to the TRPCBroker Component itself, and its properties and methods. In other development environments, you can only access the properties and methods of the TRPCBroker Component through DLL functions. To understand the DLL interface, you should understand how the TRPCBroker Component is used in its native environment (Delphi).

The following special issues should be considered when accessing RPC Broker functionality through its DLL:

- RPC Results from DLL Calls
- GetServerInfo Function and the DLL

> **i**   **REF:** For a list of DLL Exported Functions, see the "DLL Exported Functions" section.

8.1.1   Header Files

Header files for using the DLL are provided for C (**BAPI32.H**), C++ (**BAPI32.HPP**), and Visual Basic (**BAPI32.BAS**).

- [C: Guidelines Overview](#)
- [C++: Guidelines Overview](#)
- [Visual Basic: Guidelines Overview](#)

8.1.2   Sample DLL Application

The **VB5EGCHO** sample application, distributed with an earlier Broker Development Kit (BDK), demonstrates use of the RPC Broker 32-bit DLL from Microsoft Visual Basic. The source code was located in the following directory:

BDK32\Samples\Vb5Egcho

8.2   DLL Exported Functions

[Table 40](#) lists the [TRPCBroker Component](#) functions that are exported in **BAPI32.DLL**:

Table 40: DLL Exported Functions

Function	Description
[MySsoToken Function](#)	Get a Secure Token Service (STS) token from Identity and Access Management (IAM) using 2-factor authentication (2FA).
[RPCBCall Function](#)	Execute an RPC.
[RPCBCreate Function](#)	Create a [TRPCBroker Component](#).
[RPCBCreateContext Function](#)	Create context.
[RPCBFree Function](#)	Destroy a [TRPCBroker Component](#).
[RPCBMultItemGet Function](#)	Get value of a Mult item in a Param.
[RPCBMultPropGet Function](#)	Get value of a [Mult Property](#) in a Param.
[RPCBMultSet Function](#)	Set a Mult item in a Param to a value.
[RPCBMultSortedSet Function](#)	Sorts a Mult [Param Property](#).
[RPCBParamGet Function](#)	Get the value of a Param.
[RPCBParamSet Function](#)	Set the value of a Param.
[RPCBPropGet Function](#)	Get the value of a [TRPCBroker Component](#) property.
[RPCBPropSet Function](#)	Set the value of a [TRPCBroker Component](#) property.

8.3 DLL Special Issues

8.3.1 RPC Results from DLL Calls

When executing an RPC on a VistA M Server, results from the RPC are returned as a text stream. This text stream may or may not have embedded **<CR><LF>** character combinations.

In Delphi, when you call an RPC using the TRPCBroker Component directly, the text stream returned from an RPC is automatically parsed and returned in the TRPCBroker Component's Results Property, either in **Results[0]** or in multiple **Results** nodes. If there are no embedded **<CR><LF>** character combinations in the text stream, only **Results[0]** is used. If there are embedded **<CR><LF>** character combinations, results are placed into separate **Results** nodes based on the **<CR><LF>** delimiters.

When using the DLL interface, the return value is a text stream, but no processing of the text stream is performed for you. It is up to you to parse out what would have been individual **Results** nodes in Delphi, based on the presence of any **<CR><LF>** character combinations in the text stream.

> **NOTE:** You *must* create a character buffer large enough to receive the entire return value of an RPC.

8.3.2 GetServerInfo Function and the DLL

When you use the TRPCBroker Component for Delphi, you are able to call the GetServerInfo Function to retrieve the end-user workstation's server and port settings.

The functionality provided by GetServerInfo Function is *not* currently available through the RPC Broker 32-bit DLL interface. To work around this limitation when using the RPC Broker 32-bit DLL, you should prompt the user directly for the server and port to connect.

8.4 C DLL Interface

8.4.1 C: Guidelines Overview

The **BAPI32.H** header file defines the function prototypes for all functions exported in the RPC Broker 32-bit DLL.

REF: For a list of DLL Exported Functions, see the "DLL Exported Functions" section.

To use the DLL Broker functions, using C, exported in **BAPI32.DLL**, do the following:

1. C: Initialize—LoadLibrary and GetProcAddress
2. C: Create Broker Components
3. C: Connect to the Server
4. C: Execute RPCs
5. C: Destroy Broker Components

8.4.2 C: Initialize—LoadLibrary and GetProcAddress

The first step to using the RPC Broker 32-bit DLL in a C program is to load the DLL and get the process addresses for the exported functions.

To initialize access to the Broker DLL functions, do the following:

1. Use the Windows API LoadLibrary function to load the DLL.

Figure 103: C: Initialize—LoadLibrary and GetProcAddress: Using the Windows API LoadLibrary Function to Load the DLL

```
HINSTANCE hLib = LoadLibrary("bapi32.dll");
if((unsigned)hLib<=HINSTANCE_ERROR)
{
    /* Add your error handler for case where library fails to load. */
    return 1;
}
```

2. If you successfully load the DLL, map function pointers to the addresses of the functions in the DLL that you need for your application:

Figure 104: C: Initialize—LoadLibrary and GetProcAddress: Mapping Function Pointers to the Addresses of the Functions in the DLL

```
MySsoToken = (voic *(__stdcall*)()) GetProcAddress(hLib, "MySsoToken");
RPCBCreate = (void *(__stdcall*)()) GetProcAddress(hLib, "RPCBCreate");
RPCBFree = (void (__stdcall*)(void *)) GetProcAddress(hLib, "RPCBFree");
RPCBCall = (char *(__stdcall*)(void *, char *)) GetProcAddress(hLib,
"RPCBCall");
RPCBCreateContext = (bool (__stdcall*)(void *, char *)) GetProcAddress(hLib,
"RPCBCreateContext");
RPCBMultSet = (void (__stdcall*)(void *, int, char *, char *))
GetProcAddress(hLib, "RPCBMultSet");
RPCBParamGet = (void (__stdcall*)(void *, int, int, char *))
GetProcAddress(hLib, "RPCBParamGet");
RPCBParamSet = (void (__stdcall*)(void *, int, int, char *))
GetProcAddress(hLib, "RPCBParamSet");
RPCBPropGet = (void (__stdcall*)(void *, char *, char *)) GetProcAddress(hLib,
"RPCBPropGet");RPCBPropGet = (void (__stdcall*)(void *, char *, char *))
GetProcAddress(hLib, "RPCBPropGet");
RPCBPropSet =(void (__stdcall*)(void *, char *, char *)) GetProcAddress(hLib,
"RPCBPropSet");
//
// GetProcAddress, returns null on failure.
//
if( RPCBCreate == NULL || RPCBFree == NULL || RPCBCall == NULL ||
RPCBCreateContext == NULL
    || RPCBMultSet == NULL || RPCBParamGet == NULL || RPCBParamSet == NULL ||
RPCBPropGet == NULL
    || RPCBPropSet == NULL)
{
/* Add your error handler for cases where functions are not found. */
return 1;
}
```

Now you can use functions exported in the DLL.

8.4.3   C: Create Broker Components

To create TRPCBroker Components in your C program, do the following:

1. Create a pointer for the TRPCBroker Component:

```
// Generic pointer for the TRPCBroker component instance.
void * RPCBroker;
```

2. Call the RPCBCreate Function to create a TRPCBroker Component and return its address into the pointer you created:

```
// Create the TRPCBroker component instance.
RPCBroker = RPCBCreate();
```

Now you can use the pointer to the created Broker component to call its methods.

8.4.4   C: Connect to the Server

To connect to the VistA M Server from the C program, do the following:

1. Set the server and port to connect:

```
// Set the Server and Port properties to determine where to connect.
RPCBPropSet(RPCBroker,"Server", "BROKERSERVER");
RPCBPropSet(RPCBroker, "ListenerPort", "<REDACTED>");
```

2. Set the Connected Property to **true**; this attempts a connection to the VistA M Server:

```
// Set the Connected property to True, to connect.
RPCBPropSet(RPCBroker, "Connected", "1");
```

3. Check if you are still connected. If so, continue because the connection was made. If not, quit or branch accordingly:

```
// If still connected, can continue.
RPCBPropGet(RPCBroker, "Connected", Value);
if (atoi(Value) != 1) return false;
```

4. Attempt to create context for your application's "**B**"-type option. If you cannot create context, you should quit or branch accordingly. If RPCBCreateContext Function returns **True**, then you are ready to call all RPCs registered to your application's "**B**"-type option:

```
// Create Context for your application's option (in this case, XWB EGCHO).
result = RPCBCreateContext(RPCBroker, "XWB EGCHO");
return result;
```

8.4.5   C: Execute RPCs

If you can make a successful connection to the RPC Broker VistA M Server, and create an application context, you can execute any RPCs registered to your context.

To execute RPCs from your C program, do the following:

1. Create a character buffer large enough to hold your RPC's return value:

```
static char Value [1024];
```

2. Set the RemoteProcedure Property of the TRPCBroker Component to the RPC to execute:

```
RPCBPropSet(RPCBroker, "RemoteProcedure","XWB GET VARIABLE VALUE");
```

3. Set the **Param** values for any parameters needed by the RPC. In the following example, one **TRPCBroker Param** node is set (the equivalent of **Param[0]**):

   a. A value of **0** for parameter **2** denotes the integer index of the **Param** node being set (**Param[0]**).

   b. A value of *reference* for parameter **3** denotes the setting for the equivalent of **Param[0].PType**. This uses the enumerated values for **PType** (see Table 11) declared in the header file.

   c. A value of "**DUZ**" for parameter **4** denotes that the equivalent of **Param[0].**Value is "**DUZ**":

   ```
   RPCBParamSet(RPCBroker, 0, reference, "DUZ");
   ```

4. Use the RPCBCall Function to execute the RPC:

   ```
   RPCBCall(RPCBroker, Value);
   ```

The return value from the RPC is returned in the second parameter (in this case, the Value character buffer).

8.4.6 C: Destroy Broker Components

When you are done using any TRPCBroker Component, you should call its destroy method to free it from memory.

To destroy TRPCBroker Components from your C program, do the following:

1. Make sure the TRPCBroker Component is *not* connected:

   ```
   RPCBPropSet(RPCBroker, "Connected", "0");
   ```

2. Call the **RPCBFree** method to destroy the object:

   ```
   // Destroy the RPCBroker component instance.
   RPCBFree(RPCBroker);
   ```

3. When you have destroyed all TRPCBroker Components, but before your application terminates, you should call the Windows API FreeLibrary function to unload the DLL:

   ```
   FreeLibrary(hLib);
   ```

8.5 C++ DLL Interface

8.5.1   C++: Guidelines Overview

The **BAPI32.HPP** header file defines a class "wrapper" around the RPC Broker 32-bit DLL, defining a TRPCBroker C++ class. Objects of this class include all functions exported in the DLL, as methods of the TRPCBroker C++ class.

> **REF:** For a list of C++ class methods, see the "C++: TRPCBroker C++ Class Methods" section.

One feature of wrapping a class around the RPC Broker 32-bit DLL is that all the ordinary details of working with a DLL (loading the DLL, getting the addresses of the functions in the DLL, and freeing the DLL) are done within the class definition. When you initialize the class, all of the details of loading and unloading the detail (**LoadLibrary**, **GetProcAddress**, and **FreeLibrary**) are done for you.

To use objects of the class, simply initialize the class, and then create and destroy objects of the class.

To use the TRPCBroker C++ class that encapsulates **BAPI32.DLL**, do the following:

1. C++: Initialize the Class
2. C++: Create Broker Instances
3. C++: Connect to the Server
4. C++: Execute RPCs
5. C++: Destroy Broker Instances

8.5.2   C++: Initialize the Class

The first step to using the RPC Broker 32-bit DLL in a C++ program is to load the DLL and get the process addresses for the exported functions.

To initialize access to the Broker DLL functions, do the following:

1. Include **bapi32.hpp** in the program:

```
#include bapi32.hpp
```

This includes the TRPCBroker C++ class definition in the program.

2. Later, when you create a TRPCBroker C++ class object in the program, the class definition takes care of the following:

   - Loading the DLL if *not* already loaded.
   - Mapping the DLL functions if *not* already mapped.

- Creating the instance of the TRPCBroker Component.

8.5.3 C++: Create Broker Instances

To create instances of TRPCBroker C++ class objects in a C++ program, do the following:

1. Create a variable of type **TRPCBroker**. This does the following:

   - Initializes the **TRPCBroker** class.
   - Creates a TRPCBroker C++ class object instance.
   - Creates a TRPCBroker Component.

```
// Initialize the TRPCBroker class.
TRPCBroker RPCInst;
```

2. Access the properties and methods of the created TRPCBroker Component through the TRPCBroker C++ class object.

8.5.4 C++: Connect to the Server

To connect to the VistA M Server from the C++ program, do the following:

1. Set the server and port to connect:

```
// Set the Server and Port properties to determine where to connect.
RPCInst.RPCBPropSet("Server", server);
RPCInst.RPCBPropSet("ListenerPort", "<REDACTED>");
```

2. Set the Connected Property to **True**; this attempts a connection to the VistA M Server:

```
// Set the Connected property to True, to connect.
RPCInst.RPCBPropSet("Connected", "1");
```

3. Check if you are still connected. If so, continue because the connection was made. If not, quit or branch accordingly:

```
// If still connected, can continue.
RPCInst.RPCBPropGet("Connected", Value);
if (atoi(Value) != 1) return false;
```

4. Attempt to create context for the application's "**B**"-type option. If you *cannot* create context, quit or branch accordingly. If RPCBCreateContext Function returns **Tru**e, then you are ready to call all RPCs registered to the application's "**B**"-type option:

```
// Create Context for your application's option (in this case, XWB EGCHO).
result = RPCInst.RPCBCreateContext("XWB EGCHO");
return result;
```

8.5.5   C++: Execute RPCs

If you can make a successful connection to the RPC Broker VistA M Server, and create an application context, you can execute any RPCs registered to your context.

To execute RPCs from a C++ program, do the following:

1. Create a character buffer large enough to hold your RPC's return value:

```
char Value [1024];
```

2. Set the RemoteProcedure Property of the TRPCBroker Component to the RPC to execute:

```
RPCInst.RPCBPropSet("RemoteProcedure","XWB GET VARIABLE VALUE");
```

3. Set the **Param** values for any parameters needed by the RPC. In the following example, one **TRPCBroker Param** node is set (the equivalent of **Param[0]**):

   a. A value of **0** for parameter **1** denotes the integer index of the **Param** node being set (**Param[0]**).

   b. A value of *reference* for parameter **2** denotes the setting for the equivalent of **Param[0].PType**. This uses the enumerated values for **PType** (see Table 11) declared in the header file.

   c. A value of "**DUZ**" for parameter **3** denotes that the equivalent of **Param[0]**.Value is "**DUZ**":

```
RPCInst.RPCBParamSet(0, reference, "DUZ");
```

4. Use the RPCBCall Function to execute the RPC:

```
RPCInst.RPCBCall(Value);
```

The return value from the RPC is returned in the first parameter (in this case, the **Value** character buffer).

8.5.6 C++: Destroy Broker Instances

You do *not* need to do anything special to free up memory used by the [TRPCBroker Component](#) instances and their companion TRPCBroker C++ class objects. They are automatically destroyed when your program terminates, just as normal variables are automatically destroyed.

Also, when your program terminates, the **FreeLibrary** Windows API call is automatically executed to unload the RPC Broker 32-bit DLL, so there is no need to do this manually.

8.5.7 C++: TRPCBroker C++ Class Methods

The functions in the RPC Broker 32-bit DLL are encapsulated in the TRPCBroker C++ class methods shown in [Table 41](#):

Table 41: C++: TRPCBroker C++ Class Methods

DLL Function	TRPCBroker C++ Class Method
[MySsoToken Function](#)	char * MySSOToken();
[RPCBCall Function](#)	char * RPCBCall(char * s);
[RPCBCreateContext Function](#)	bool RPCBCreateContext (char * s);
[RPCBMultItemGet Function](#)	void RPCBMultItemGet (int i, char * s, char * t);
[RPCBMultPropGet Function](#)	void RPCBMultPropGet (void * ptr, int i , char * s, char * t);
[RPCBMultSet Function](#)	void RPCBMultSet (int i, char * s, char * t);
[RPCBMultSortedSet Function](#)	void RPCBMultSortedSet (void * ptr, int i, bool v);
[RPCBParamGet Function](#)	void RPCBParamGet (int i, int j, char * s);
[RPCBParamSet Function](#)	void RPCBParamSet (int i, int j, char * s);
[RPCBPropGet Function](#)	void RPCBPropGet (char * s, char * t);
[RPCBPropSet Function](#)	void RPCBPropSet (char * s, char * t);

8.6 Visual Basic DLL Interface

8.6.1 Visual Basic: Guidelines Overview

The **BAPI32.BAS** header file defines the function prototypes for all functions exported in the RPC Broker 32-bit DLL.

REF: For a list of DLL exported functions, see the "DLL Exported Functions" section.

To use the DLL Broker functions, using Visual Basic, exported in **BAPI32.DLL**, do the following:

1. Visual Basic: Initialize
2. Visual Basic: Create Broker Components
3. Visual Basic: Connect to the Server
4. Visual Basic: Execute RPCs
5. Visual Basic: Destroy Broker Components

8.6.1.1 Sample DLL Application

The **VB5EGCHO** sample application, distributed with an earlier Broker Development Kit (BDK), demonstrates use of the RPC Broker 32-bit DLL from Microsoft Visual Basic. The source code was located in the following directory:

BDK32\Samples\Vb5Egcho

8.6.2 Visual Basic: Initialize

The first step to using the RPC Broker 32-bit DLL in a Visual Basic program is to load the DLL and get the process addresses for the exported functions.

To initialize access to the Broker DLL functions, do the following:

1. Include **BAPI32.BAS** as a module in your Visual Basic program.
2. Visual Basic takes care of loading the DLL and mapping its functions.

8.6.3   Visual Basic: Create Broker Components

To create TRPCBroker Components in your Visual Basic program, do the following:

1. Create a variable to be a handle for the TRPCBroker Component:

```
Public intRPCBHandle As Long
```

2. Call the RPCBCreate Function to create a TRPCBroker Component and return its address into the variable you created:

```
intRPCBHandle = RPCBCreate()
```

Now, you can use the handle to the created Broker component to call its methods.

8.6.4   Visual Basic: Connect to the Server

To connect to the VistA M Server from the Visual Basic program, do the following:

1. Set the server and port to connect:

```
Call RPCBPropSet(intRPCBHandle, "Server", "BROKERSERVER")
Call RPCBPropSet(intRPCBHandle, "ListenerPort", "<REDACTED>")
```

2. Set the Connected Property to **true**; this attempts a connection to the VistA M Server:

```
Call RPCBPropSet(intRPCBHandle, "Connected", "1")
```

3. Check if you are still connected. If so, continue because the connection was made. If not, quit or branch accordingly:

```
RPCBPropGet(intRPCBHandle, "Connected", strResult)
```

4. Attempt to create context for your application's "**B**"-type option. If you *cannot* create context, quit or branch accordingly. If RPCBCreateContext Function returns **True**, then you are ready to call all RPCs registered to the application's "**B**"-type option:

```
intResult = RPCBCreateContext(intRPCBHandle, "MY APPLICATION")
```

8.6.5 Visual Basic: Execute RPCs

If you can make a successful connection to the RPC Broker VistA M Server, and create an application context, you can execute any RPCs registered to your context.

To execute RPCs from your Visual Basic program, do the following:

1. Create a character buffer large enough to hold your RPC's return value:

```
Public strBuffer As String * 40000
```

2. Set the RemoteProcedure Property of the TRPCBroker Component to the RPC to execute:

```
Call RPCBPropSet(intRPCBHandle, "RemoteProcedure", "XWB GET VARIABLE VALUE")
```

3. Set the **Param** values for any parameters needed by the RPC. In the following example, one **TRPCBroker Param** node is set (the equivalent of **Param[0]**):

   a. A value of **0** for parameter **2** denotes the integer index of the Param node being set (**Param[0]**).

   b. A value of *reference* for parameter **3** denotes the setting for the equivalent of **Param[0].PType**. This uses the enumerated values for **PType** (see Table 11) declared in the header file.

   c. A value of "**DUZ**" for parameter **4** denotes that the equivalent of **Param[0]**.Value is "**DUZ**":

```
Call RPCBParamSet(intRPCBHandle, 0, reference, "DUZ");
```

4. Use the RPCBCall Function to execute the RPC:

```
Call RPCBCall(intRPCBHandle, strBuffer
```

The return value from the RPC is returned in the second parameter (in this case, the Value character buffer).

8.6.6   Visual Basic: Destroy Broker Components

When you are done using any TRPCBroker Component, you should call its destroy method to free it from memory (see the "Memory Leaks").

To destroy TRPCBroker Components from your Visual Basic program, do the following:

1. Make sure the TRPCBroker Component is *not* connected:

```
Call RPCBPropSet(intRPCBHandle, "Connected", "0")
```

2. Call the RPCBFree Function to destroy the object:

```
RPCBFree(intRPCBHandle)
```

Visual Basic takes care of the details of unloading the DLL.

8.7   MySsoToken Function

Get a Secure Token Service (STS) token from Identity and Access Management (IAM) using 2-factor authentication. This encapsulates the following Broker Development Kit methods:

- **TXWBSSOiToken Create**
- **SSOiToken**
- **Free**

8.7.1   Declarations

Table 42: MySsoToken Function—Declarations

Software	Declaration
Delphi	`function MySsoToken: PChar; stdcall;`
C	`void *(__stdcall *MySsoToken) (void);`
C++	`void * MySsoToken(void);`
VB	`Function MySsoToken () As String`

8.7.2   Return Value

- **String**—Digitally signed XML (SAML) token containing authenticated user identity.
- **Null string**—If authentication failed or token could not be obtained.

8.7.3 Examples

8.7.3.1 C

```
result = MySsoToken();
```

8.7.3.2 C++

```
Result = MySsoToken();
```

8.7.3.3 Visual Basic

```
StrResult = MySsoToken()
```

8.8 RPCBCall Function

Executes a remote procedure call and fills the passed buffer with the data resulting from the call. This is equivalent to the TRPCBroker Component's Call Method.

8.8.1 Declarations

Table 43: RPCBCall Function—Declarations

Software	Declaration
Delphi	**procedure** RPCBCall(**const** RPCBroker: TRPCBroker; CallResult: PChar);
C	char *(__stdcall *RPCBCall) (void *, char *);
C++	char * RPCBCall(char * s);
VB	Sub RPCBCall (ByVal intRPCBHandle As Long, ByVal strCallResult As String)

8.8.2 Parameter Description

Table 44: RPCBCall Function—Parameters

Parameter	Description
RPCBroker	Handle of the Broker component that contains the name of the remote procedure and all of the required parameters.
CallResult	An empty character buffer that the calling application must create (allocate memory for) before making this call. This buffer is filled with the result of the call.

REF: For information about the size of parameters and results that can be passed to and returned from the TRPCBroker Component, see the "RPC Limits" section.

8.8.3 Examples

8.8.3.1 C

```
RPCBCall(RPCBroker, Value);
```

8.8.3.2 C++

```
// MyInstance is defined as an instance of the TRPCBroker.
MyInstance.RPCBCall( strbuffer);
```

8.8.3.3 Visual Basic

```
Call RPCBCall(intRPCBHandle, strBuffer)
```

8.9 RPCBCreate Function

The RPCBCreate Function creates a new RPC Broker component for the application, which can then be used to connect to the VistA M Server and call remote procedures.

8.9.1 Declarations

Table 45: RPCBCreate Function—Declarations

Software	Declarations
Delphi	`function RPCBCreate: TRPCBroker;`
C	`void * (__stdcall *RPCBCreate)(void);`
C++	`N/A (encapsulated in TRPCBroker class definition)`
VB	`Function RPCBCreate () As Long`

8.9.2 Return Value

A handle for the TRPCBroker Component created.

8.9.3 Examples

8.9.3.1 C

```
// Create the TRPCBroker component instance.
RPCBroker = RPCBCreate();
```

8.9.3.2 Visual Basic

```
intRPCBHandle = RPCBCreate()
```

8.10 RPCBCreateContext Function

The **RPCBCreateContext** function calls the TRPCBroker Component's CreateContext Method to set up the environment on the VistA M Server for subsequent RPCs.

8.10.1 Declarations

Table 46: RPCBCreateContext Function—Declarations

Software	Declarations
Delphi	**function** RPCBCreateContext(**const** RPCBroker: TRPCBroker; **const** Context: PChar): **boolean**;
C	bool (__stdcall *RPCBCreateContext) (void *, char *);
C++	bool RPCBCreateContext (char * s);
VB	Function RPCBCreateContext (ByVal intRPCBHandle As Long, ByVal strContext As String) As Integer

8.10.2 Return Value

- **True/1**—If context could be created.
- **False/0**—If context could *not* be created.

8.10.3 Parameter Description

Table 47: RPCBCreateContext Function—Parameters

Parameter	Description
RPCBroker	Handle of the TRPCBroker Component.
Context	**Null-terminated** string identifying the option on the VistA M Server for which subsequent RPCs are registered.

8.10.4 Examples

8.10.4.1 C

```
// XWB EGCHO is a "B" (Broker) type option in the OPTION file.
result = RPCBCreateContext(RPCBroker, "XWB EGCHO");
```

8.10.4.2 C++

```
// XWB EGCHO is a "B" (Broker) type option in the OPTION file.
MyInstance.RPCBCreateContext("XWB EGCHO")
```

8.10.4.3 Visual Basic

```
intResult = RPCBCreateContext(intRPCBHandle, "MY APPLICATION")
'where MY APPLICATION is a "B" (Broker) type option in the Option file.
```

8.11 RPCBFree Function

The **RPCBFree** function destroys the RPC Broker component and releases associated memory (see "Memory Leaks" section).

8.11.1 Declarations

Table 48: RPCBFree Function—Declarations

Software	Declaration
Delphi	`procedure RPCBFree(RPCBroker: TRPCBroker);`
C	`void (__stdcall *RPCBFree)(void *);`
C++	`N/A (encapsulated in TRPCBroker class definition)`
VB	`Sub RPCBFree (ByVal intRPCBHandle As Long)`

8.11.2 Parameter Description

Table 49: RPCBFree Function—Parameter

Parameter	Description
RPCBroker	Handle of the TRPCBroker Component to destroy.

8.11.3 Examples

8.11.3.1 C

```
// Destroy the TRPCBroker component instance.
RPCBFree(RPCBroker);
```

8.11.3.2 Visual Basic

```
RPCBFree (intRPCBHandle)
```

8.12 RPCBMultItemGet Function

The **RPCBMultItemGet** function returns a requested item in a <u>TRPCBroker Component</u> <u>Param Property</u>'s <u>Mult Property</u>.

8.12.1 Declarations

Table 50: RPCBMultItemGet Function—Declarations

Software	Declaration
Delphi	**procedure** RPCBMultItemGet (**const** RPCBroker: TRPCBroker; ParamIndex: **integer**; Subscript, Value: PChar);
C	void (__stdcall *RPCBMultItemGet) (void *, int, char *, char *);
C++	void RPCBMultItemGet (int i, char * s, char * t);
VB	Sub RPCBMultItemGet (ByVal intRPCBHandle As Long, ByVal intParIdx As Integer, ByVal strSubscript As String, ByVal strValue As String)

8.12.2 Parameter Description

Table 51: RPCBMultItemGet Function—Parameters

Parameter	Description
RPCBroker	Handle of the <u>TRPCBroker Component</u>.
ParamIndex	Integer index of the parameter that contains the <u>Mult Property</u>.
Subscript	**Null-terminated** string identifying the Mult element to get.
Value	An empty buffer that the calling application *must* create (allocate memory for) before making this call. This buffer is filled with the value of the <u>Mult Property</u> item.

REF: For information about the size of parameters and results that can be passed to and returned from the TRPCBroker Component, see the "RPC Limits" section.

8.12.3 Examples

8.12.3.1   C

```
// The following corresponds to getting the value of PARAM[0].Mult["0"]
RPCBMultItemGet(RPCBroker, 0 , "0", Value);
```

8.12.3.2   C++

```
MyInstance.RPCBMultItemGet(0 , "0", Value);
```

8.12.3.3   Visual Basic

```
Call RPCBMultItemGet(intRPCBHandle, 0, "0", strResult)
```

8.13 RPCBMultPropGet Function

The **RPCBMultPropGet** function returns a selected property value of a TRPCBroker Component Param Property's Mult Property.

8.13.1 Declarations

Table 52: RPCBMultPropGet—Declarations

Software	Declaration
Delphi	**procedure** RPCBMultPropGet(**const** RPCBroker: TRPCBroker; ParamIndex: **integer**; Prop,Value: PChar);
C	void (__stdcall *RPCBMultPropGet) (void *, int, char *, char *);
C++	void RPCBMultPropGet (int i , char * s, char * t);
VB	Sub RPCBMultPropGet (ByVal intRPCBHandle As Long, ByVal intParIdx As Integer, ByVal strProp As String, ByRef strValue As String)

8.13.2 Parameter Description

Table 53: RPCBMultPropGet—Parameters

Parameter	Description
RPCBroker	Handle of the TRPCBroker Component.
ParamIndex	Integer index of the parameter that contains the Mult Property.
Prop	**Null-terminated** string identifying the name of the **TMult** property to get.
Value	An empty buffer that the calling application *must* create (allocate memory for) before making this call. This buffer is filled with value of the Mult Property that is in the **Prop**.

REF: For information about the size of parameters and results that can be passed to and returned from the TRPCBroker Component, see the "RPC Limits" section.

8.13.3 Examples

8.13.3.1 C

```
RPCBMultPropGet(RPCBroker, 0, "Count", Value);
```

8.13.3.2 C++

```
MyInstance.RPCBMultPropGet(0, "Count", Value);
```

8.13.3.3 Visual Basic

```
Call RPCBMultPropGet(intRPCBHandle, 0, "Count", strResult)
```

8.14 RPCBMultSet Function

The **RPCBMultSet** function sets an item in a TRPCBroker Component Param Property's Mult Property to a value.

8.14.1 Declarations

Table 54: RPCBMultSet Function—Declarations

Software	Declaration
Delphi	`procedure RPCBMultSet(const RPCBroker: TRPCBroker; ParamIndex: integer; Subscript, Value: PChar);`
C	`void (__stdcall *RPCBMultSet) (void *, int, char *, char *);`
C++	`void RPCBMultSet ( int i, char * s, char * t);`
VB	`Sub RPCBMultSet (ByVal intRPCBHandle As Long, ByVal intParIdx As Integer, ByVal strSubscript As String, ByVal strValue As String)`

8.14.2 Parameter Description

Table 55: RPCBMultSet Function—Parameters

Parameter	Description
RPCBroker	Handle of the TRPCBroker Component.
ParamIndex	Integer index of the parameter that contains the Mult Property.
Subscript	**Null-terminated** string of the Mult item to set.
Value	**Null-terminated** string containing the value that the **Mult** item should be set to.

REF: For information about the size of parameters and results that can be passed to and returned from the TRPCBroker Component, see the "RPC Limits" section.

8.14.3 Examples

8.14.3.1 C

```
RPCBMultSet(RPCBroker, 0, "1", "12/19/97");
```

8.14.3.2 C++

```
MyInstance.RPCBMultSet(0, "1", "12/19/97");
```

8.14.3.3 Visual Basic

```
Call RPCBMultSet(intRPCBHandle, 0, "1", "12/19/97")
```

8.15 RPCBMultSortedSet Function

The **RPCBMultSortedSet** function sets the Sorted Property of a Mult Property. In essence, sorts and keeps the Mult Property sorted or just leaves it in the order it is populated.

8.15.1 Declarations

Table 56: RPCBMultSortedSet Function—Declarations

Software	Declaration
Delphi	**procedure** RPCBMultSortedSet(**const** RPCBroker: TRPCBroker; ParamIndex: integer; Value: **boolean**);
C	void (__stdcall *RPCBMultSortedSet) (void *, int, bool);
C++	void RPCBMultSortedSet (int i, bool v);
VB	Sub RPCBMultSortedSet (ByVal intRPCBHandle As Long, ByVal intParIdx As Integer, ByVal intValue As Integer)

8.15.2 Parameter Description

Table 57: RPCBMultSortedSet Function—Parameters

Parameter	Description
RPCBroker	Handle of the <u>TRPCBroker Component</u>.
ParamIndex	Integer index of the parameter that contains the <u>Mult Property</u>.
Value	Can be either a Boolean or, if the calling application language does *no*t support Boolean type, can be an integer: • **True** or **1—**Sorts the Mult and keeps it sorted thereafter when other elements are added. • **False** or **0—**Does *not* sort the Mult and the elements are stored in the order they are added.

8.15.3 Examples

8.15.3.1   C

```
RPCBMultSortedSet(RPCBroker, 0, 1);
```

8.15.3.2   C++

```
MyInstance.RPCBMultSortedSet(0, 1);
```

8.15.3.3   Visual Basic

```
Call RPCBMultPropGet(intRPCBHandle, 0, 1)
```

8.16 RPCBParamGet Function

The **RPCBParamGet** function returns two values in two parameters: the value and the parameter type of a **Param** item.

You can compare the returned parameter type to the following enumerated values:

- **literal**
- **reference**
- **list**

8.16.1 Declarations

Table 58: RPCBParamGet Function—Declarations

Software	Declaration
Delphi	**procedure** RPCBParamGet(**const** RPCBroker: TRPCBroker; ParamIndex: integer; var ParamType: TParamType; var ParamValue: PChar);
C	void (__stdcall *RPCBParamGet) (void *, int, int, char *);
C++	void RPCBParamGet (int i, int j, char * s);
VB	Sub RPCBParamGet (ByVal intRPCBHandle As Long, ByVal intParIdx As Integer, ByVal intParTyp As Integer, ByVal intParVal As String)

8.16.2 Parameter Description

Table 59: RPCBParamGet Function—Parameters

Parameter	Description
RPCBroker	Handle of the TRPCBroker Component.
ParamIndex	Integer index of the parameter to get the value.
ParamType	This variable, after making the RPCBParamGet call, is filled with PType Property of a Param[ParamIndex].
ParamValue	An empty buffer that you *must* create (allocate memory for) before making this call. This buffer, after making the **RPCBParamGet** call, is filled with **Value** of a **Param[ParamIndex]**.

ⓘ   **REF:** For information about the size of parameters and results that can be passed to and returned from the TRPCBroker Component, see the "RPC Limits" section.

8.16.3 Examples

8.16.3.1   C

```
// PType and Value are variables to retrieve values into.
RPCBParamGet(RPCBroker, 0, PType, Value);
```

8.16.3.2   C++

```
// PType and Value are variables to retrieve values into.
MyInstance.RPCBParamGet(0, PType, Value);
```

8.16.3.3   Visual Basic

```
Call RPCBParamGet(intRPCBHandle, 0, PType, strResult)
' where PType and strResult are variables to retrieve values into
```

8.17 RPCBParamSet Function

The **RPCBParamSet** function sets one **Param** item's Value Property and PType Property, in a single call.

8.17.1 Declarations

Table 60: RPCBParamSet Function—Declarations

Software	Declaration
Delphi	**procedure** RPCBParamSet(**const** RPCBroker: TRPCBroker; const ParamIndex: integer; **const** ParamType: TParamType; const ParamValue: PChar);
C	void (__stdcall *RPCBParamSet) (void *, int, int, char *);
C++	void RPCBParamSet (int i, int j, char * s);
VB	Sub RPCBParamSet (ByVal intRPCBHandle As Long, ByVal intParIdx As Integer, ByVal intParTyp As Integer, ByVal intParVal As String)

8.17.2 Parameter Description

Table 61: RPCBParamSet Function—Parameters

Parameter	Description
RPCBroker	Handle of the <u>TRPCBroker Component</u>.
ParamIndex	Integer index of the parameter.
ParamType	Set to the parameter type for the parameter you are setting. The value should be one of the integer values that are valid as a **PType**: • **0** (literal) • **1** (reference) • **2** (list) You can use the enumerated values literal, reference and list, as declared in the C, C++, or Visual Basic header file. ⚠️ **CAUTION: For enhanced security reasons, the reference parameter type may be deprecated and removed in subsequent updates to the BDK.**
ParamValue	**Null-terminated** string containing the Value to set.

ℹ️ **REF:** For information about the size of parameters and results that can be passed to and returned from the <u>TRPCBroker Component</u>, see the "<u>RPC Limits</u>" section.

8.17.3 Examples

8.17.3.1   C

```
RPCBParamSet(RPCBroker, 0, reference, "DUZ");
```

8.17.3.2   C++

```
MyInstance.RPCBParamSet(0, reference, "DUZ");
```

8.17.3.3   Visual Basic

```
Call RPCBParamSet(intRPCBHandle, 0, literal, Text3.Text)
```

8.18 RPCBPropGet Function

The **RPCBPropGet** function returns a requested property of a [TRPCBroker Component](#).

8.18.1 Declarations

Table 62: RPCBPropGet Function—Declarations

Software	Declaration
Delphi	`procedure RPCBPropGet(const RPCBroker: TRPCBroker; const Prop: PChar; {var} Value: PChar);`
C	`void (__stdcall *RPCBPropGet) (void *, char *, char *);`
C++	`void RPCBPropGet ( char * s, char * t);`
VB	`Sub RPCBPropGet (ByVal intRPCBHandle As Long, ByVal strProp As String, ByVal strValue As String)`

Table 63: RPCBPropGet Function—Parameters

Parameter	Description
RPCBroker	Handle of the [TRPCBroker Component](#).
Prop	**Null-terminated** string of the property to get. *Not* case-sensitive. Valid properties to get are: • [ClearParameters Property](#) • [ClearResults Property](#) • [Connected Property](#) • [DebugMode Property](#) • [ListenerPort Property](#) • [RemoteProcedure Property](#) • [RPCTimeLimit Property](#) • [RPCVersion Property](#) • [Server Property](#)
Value	An empty buffer that you *must* create (allocate memory for) before making this call. After this call, the buffer is filled with value of the property that is in the **Prop**. This procedure takes care of all the necessary type conversions. Boolean property values are returned as either of the following: • **1** (True) • **0** (False)

8.18.2 Examples

8.18.2.1 C

```
RPCBPropGet(RPCBroker, "Connected", Value);
```

8.18.2.2 C++

```
MyInstance.RPCBPropGet("Connected", Value);
```

8.18.2.3 Visual Basic

```
Call RPCBPropGet(intRPCBHandle, "Server", strResult)
```

8.19 RPCBPropSet Function

The **RPCBPropSet** function sets a [TRPCBroker Component](#) property to some value.

8.19.1 Declarations

Table 64: RPCBPropSet Function—Declarations

Software	Declaration
Delphi	**procedure** RPCBPropSet(**const** RPCBroker: TRPCBroker; Prop,Value: PChar);
C	void (__stdcall *RPCBPropSet) (void *, char *, char *);
C++	void RPCBPropSet (char * s, char * t);
VB	Sub RPCBPropSet (ByVal intRPCBHandle As Long, ByVal strProp As String, ByVal strValue As String)

Table 65: RPCBPropSet Function—Parameters

Parameter	Description
RPCBroker	Handle of the <u>TRPCBroker Component</u>.
Prop	**Null-terminated** string of the property to set; *not* case-sensitive. Valid properties to set are: • <u>ClearParameters Property</u> • <u>ClearResults Property</u> • <u>Connected Property</u> • <u>DebugMode Property</u> • <u>ListenerPort Property</u> • <u>RemoteProcedure Property</u> • <u>RPCTimeLimit Property</u> • <u>RPCVersion Property</u> • <u>Server Property</u>
Value	**Null-terminated** string of the value to which the **Prop** property should be set. This procedure takes care of converting the passed in value to whatever type the property expects. For Boolean properties, pass in either of the following: • **1** (True) • **0** (False)

8.19.2 Examples

8.19.2.1 C

```
RPCBPropSet(RPCBroker, "ListenerPort", "<REDACTED>");
```

8.19.2.2 C++

```
MyInstance.RPCBPropSet("ListenerPort", "<REDACTED>");
```

8.19.2.3 Visual Basic

```
Call RPCBPropSet(intRPCBHandle, "Server", cboServer.Text)
```

Glossary

Table 66: Glossary of Terms and Acronyms

Term	Description
Client	A single term used interchangeably to refer to the user, the workstation, and the portion of the program that runs on the workstation. In an object-oriented environment, a client is a member of a group that uses the services of an unrelated group. If the client is on a local area network (LAN), it can share resources with another computer (server).
Component	An object-oriented term used to describe the building blocks of GUI applications. A software object that contains data and code. A component may or may not be visible. These components interact with other components on a form to create the GUI user application interface.
DHCP	**D**ynamic **H**ost **C**onfiguration **P**rotocol.
DLL	**D**ynamic **L**ink **L**ibrary. A DLL allows executable routines to be stored separately as files with a DLL extension. These routines are only loaded when a program calls for them. DLLs provide several advantages: 1. DLLs help save on computer memory, since memory is only consumed when a DLL is loaded. They also save disk space. With static libraries, your application absorbs all the library code into your application, so the size of your application is greater. Other applications using the same library also carry this code around. With the DLL, you do *not* carry the code itself; you have a pointer to the common library. All applications using it then share one image. 2. DLLs ease maintenance tasks. Because the DLL is a separate file, any modifications made to the DLL do *not* affect the operation of the calling program or any other DLL. 3. DLLs help avoid redundant routines. They provide generic functions that can be used by a variety of programs.
DNS	The Domain Name Service (DNS) is a distributed database that maps names to their Internet Protocol (IP) addresses or IP addresses to their names. A query to this database is used to resolve names and IP addresses.
GUI	**G**raphical **U**ser Interface. A type of display format that enables users to choose commands, initiate programs, and other options by selecting pictorial representations (icons) via a mouse or a keyboard.

Term	Description
HANDLE	A HANDLE is a string returned by [XWB REMOTE RPC](#) or [XWB DEFERRED RPC](#). The application should store the HANDLE and use it to: 1. Check for the return of the data. 2. Retrieve the data. 3. Clear the data from the VistA M Server.
HOSTS File	The HOSTS file is an ASCII text file that contains a list of the servers and their Internet Protocol (IP) addresses. It is recommended that you put in a "**DHCPSERVER**" entry that points to the main server you intend using with the Broker the majority of the time. In your applications, you are able to specify any server you wish; however, if the **Server** property = " (i.e., **NULL**), you get an error.
IAM	Identity and Access Management.
Icon	A picture or symbol that graphically represents an object or a concept.
IP Address	The Internet Protocol (IP) address is the network interface address for a client workstation, server, or device.
$JOB	Contains your operating system job number on the VistA M Server.
$ORDER	M code: `$ORDER(variable name{,integer code})` Returns the subscript of the previous or next sibling in collating sequence of a specified array node. To obtain the first subscript of a level, specify a **NULL** subscript in the argument.
Remote Procedure Call	A remote procedure call (RPC) is essentially M code that may take optional parameters to do some work and then return either a single value or an array back to the client application.
SAML	Security Assertion Markup Language. An XML-based industry standard for communicating identities over the Internet.
Server	The computer where the data and the Business Rules reside. It makes resources available to client workstations on the network. In VistA, it is an entry in the OPTION (#19) file. An automated mail protocol that is activated by sending a message to a server at another location with the "**S.server**" syntax. A server's activity is specified in the OPTION (#19) file and can be the running of a routine or the placement of data into a file.

Term	Description
User Access	This term is used to refer to a limited level of access to a computer system that is sufficient for using/operating software, but does not allow programming, modification to data dictionaries, or other operations that require programmer access. Any of VistA's options can be locked with a security key (e.g., XUPROGMODE, which means that invoking that option requires programmer access). The user's access level determines the degree of computer use and the types of computer programs available. The Systems Manager assigns the user an access level.
User Interface	The way the software is presented to the user, such as Graphical User Interfaces that display option prompts, help messages, and menu choices. A standard user interface can be achieved by using Embarcadero's Delphi Graphical User Interface to display the various menu option choices, commands, etc.
Window	An object on the screen (dialogue) that presents information such as a document or message.
XML	eXtensible Markup Language.
XUPROGMODE	A security key distributed by Kernel as part of its Menu Manager (MenuMan). This security key enables access to a number of developer-oriented options in Kernel.

REF: For a list of commonly used terms and definitions, see the OIT Master Glossary VA Intranet Website.

For a list of commonly used acronyms, see the VA Acronym Lookup Intranet Website.