

[Home](#) > Contents

RPC Broker 1.1 Developer's Guide

Contents

[Overview](#)

Components

(listed alphabetically)

- [TCCOWRPCBroker](#)
- [TContextorControl](#)
- [TRPCBroker](#)
- [TSharedBroker](#)
- [TSharedRPCBroker](#)
- [TXWBRichEdit](#)

Classes

(listed alphabetically)

- [TMult](#)
- [TParamRecord](#)
- [TParams](#)
- [TVistaLogin](#)
- [TVistaUser](#)
- [TXWBWinsock](#)

Units

(listed alphabetically)

- [Hash](#)
- [LoginFrm](#)
- [MFunStr](#)
- [RPCConf1](#)
- [RpcSLogin](#)
- [SpVista](#)

- [TRPCB](#)
- [TVCEdit](#)

[Remote Procedure Calls \(RPCs\)](#)

[Other RPC Broker APIs](#)

[Debugging and Troubleshooting](#)

[Tutorial](#)

[DLL Interface](#)

[Home](#) > [Overview](#) > Introduction

Introduction: RPC Broker 1.1

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.


The online help 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 online help 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*.

The online help is intended for the VistA development community and Information Resource Management (IRM) staff. A wider audience of technical personnel engaged in operating and maintaining the Department of Veterans Affairs (VA) software might also find it useful as a reference.

- [Contents](#)
- [Broker Overview](#)



- [Definitions](#)
- [About this Version of the RPC Broker](#)
- [What's New in the BDK](#)
- [Developer Considerations](#)
- [Application Considerations](#)
- [Online Help](#)

 **REF:** For the latest RPC Broker product information, see the RPC Broker Intranet Website: [redacted](#)

[Home](#) > [Overview](#) > Broker Overview

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.

Client Side of the RPC Broker	Server Side of RPC the Broker
<ul style="list-style-type: none">• Manages the connection to the client workstation. <p> REF: For details, see the <i>RPC Broker Systems Management Guide</i>.</p> <ul style="list-style-type: none">• The RPC Broker components allow Delphi-based applications to make RPCs to the server.	<ul style="list-style-type: none">• Manages the connection to the client. <p> REF: For details, see the <i>RPC Broker Systems Management Guide</i>.</p> <ul style="list-style-type: none">• Authenticates client workstation.• Authenticates user.• Manages RPCs from the client, executes the

- The Broker Dynamic Link Library (DLL) provides support for Commercial-Off-The-Shelf (COTS)/HOST client/server software.

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.

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 machine. When a client machine 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](#).

2. Authentication of user. After the server connects back to the client machine, the user is asked for an Access and Verify code.

GUI Developer Issues:

- Creating user context—Applications *must* create a context for the user. 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](#) or [TSharedRPCBroker](#) components 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.


 **REF:** For more information on the VA FileMan Delphi Components (FMDC), see the FMDC VA Intranet Website: [redacted](#)

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.

 **REF:** For details, see the "[RPCs](#)" topic.

Issues include:

- Determining the best RPC—The BDK provides some [RPC BROKER APIs](#).
- [Creating RPCs from scratch](#)— In many cases, [an existing M API](#) can be wrapped into an RPC.
- 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](#)" topic.

5. RPC returns information to the client.

GUI Developer Issues:

Handling the return values, including any error messages.

[Home](#) > [Overview](#) > Definitions

Definitions

The RPC Broker SDK 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}

Units

A Unit is a Pascal source-code file (e.g., winsockc.pas in the BDK) containing all of the other elements. It is sometimes called a "program;" however, that can be misleading as a working program can contain or reference many units. For example, 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.

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.



NOTE: This Help file documents some of the units provided, and details what parts of the BDK are declared in each unit.

Classes

A class is a data type that wraps up code and data all in one bundle.

Objects

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

Components

In Embarcadero Delphi, the term "component" is used to describe elements of a unit, function, procedure, etc.




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

The RPC Broker and associated documentation uses a more common definition for a "component." A "component" is something bigger than a unit; basically a small program that can be embedded into a larger program. In this case, a component is an object with additional properties, methods, and events that makes it suitable for specialized purposes. A component may or may not be visible.

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 topics in this Help file describing types, the unit and declaration for each type, as well as a description of the type is also provided.

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."

Routines, Functions, and Procedures

Routines can either be functions or procedures. A function returns a value, and a procedure does not.

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.




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

[Home](#) > [Overview](#) > About this Version of the RPC Broker

About this Version of the RPC Broker

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


- [TCCOWRPCBroker](#)
- [TContextorControl](#)
- [TRPCBroker](#) (original component)
- [TSharedBroker](#)
- [TSharedRPCBroker](#)
- [TXWBRichEdit](#)

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

To develop VistA applications in a 32-bit environment you *must* have Delphi XE2 or greater. This version of the RPC Broker does *not*

allow you to develop new applications in Delphi 1.0 (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.

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 may 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.

What's New in the BDK

Here is What's New Through March 2014

This topic highlights some of the major changes made to the RPC Broker 1.1, since its original release (patch references are included where applicable):

- [Classes Added](#)
- [Components Added](#)
- [Design-time and Run-time Packages](#)
- [Functionality Added](#)
- [Library Methods](#)
 - [Added](#)
 - [Modified](#)
- [Properties Added](#)
- [Source Code Availability](#)

- [Types Added/Modified](#)

Classes Added

As of RPC Broker Patches XWB*1.1*13, the following Classes were added:

- [TVistaLogin](#)
- [TVistaUser](#)

As of RPC Broker Patches XWB*1.1*40, the following Class was added:

- [TXWBWinsock](#)

Components Added or Modified:

As of RPC Broker Patch XWB*1.1*50, the following RPC Broker components were added or modified:

- [TRPCBroker](#)

Modified the TRPCBroker component in RPC Broker 1.1. 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.

As of RPC Broker Patch XWB*1.1*40, the following RPC Broker components were added or modified:

- [TCCOWRPCBroker](#)

Added the TCCOWRPCBroker component to RPC Broker 1.1. This component allows applications to be CCOW-enabled and

Single Sign-On/User Context (SSO/UC)-aware.

- [TContextorControl](#)

Added the TContextorControl component to RPC Broker 1.1. The TContextorControl Delphi component communicates with the Vergence Locator service.

As of RPC Broker Patch XWB*1.1*26, the following RPC Broker components were added or modified:

- [TSharedBroker](#)

Added the TSharedBroker component to RPC Broker 1.1. This component allows applications to share a single Broker connection.

- [TSharedRPCBroker](#)

Added the TSharedRPCBroker component to RPC Broker 1.1. This component allows applications to share a single Broker connection.

As of RPC Broker Patch XWB*1.1*13, the following RPC Broker components were added or modified:

- [TXWBRichEdit](#)

Added the TXWBRichEdit component to RPC Broker 1.1. This component replaced the Introductory Text Memo component on the Login Form. It permits URLs to be identified and launched.

Design-time and Run-time Packages

As of RPC Broker Patch XWB*1.1*14, the BDK contains separate run-time and design-time packages.



REF: For details and compiling instructions, see the "[Developer Considerations](#)" topic.

Functionality Added


As of RPC Broker Patch XWB*1.1*50, the following RPC Broker 1.1 functionality was added or modified:

- **Support for Later Delphi Versions**—BDK supports Delphi XE5, XE4, XE3, and XE2.
- **[Supports Secure Shell \(SSH\)](#)**—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[®] Reflections 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.
- **[Supports Broker Security Enhancement \(BSE\)](#)**—The TRPCBroker component enabled visitor access to remote sites using authentication established at a home site.

As of RPC Broker Patch XWB*1.1*40, the following RPC Broker 1.1 functionality was 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).

As of RPC Broker Patch XWB*1.1*35, the following RPC Broker 1.1 functionality was 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. This functionality is controlled via the [TRPCBroker](#) component [IsBackwardCompatibleConnection](#) property.

As of RPC Broker Patch XWB*1.1*13, the following RPC Broker 1.1 functionality was added or modified:

- **Supports Silent Login**—The RPC Broker provides "[Silent Login](#)" capability. 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.
- **Documented Deferred RPCs and Capability to Run RPCs on a Remote Server:**

- [Running RPCs on a Remote Server](#)
- [Deferred RPCs](#)
- **Multi-instances of the RPC Broker**—The RPC Broker code was modified to permit an application to open two separate Broker instances with the same Server/ListenerPort 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 yet guaranteed to be thread safe*.

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

Library Methods

Added

As of RPC Broker Patch XWB*1.1*40, the following library methods were added to the [TCCOWRPCBroker](#) component:

- [GetCCOWtoken](#)

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

- [IsUserCleared](#)

```
function IsUserCleared: Boolean;
```

- [IsUserContextPending](#)

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

- [WasUserDefined](#)

```
function WasUserDefined: Boolean;
```

As of RPC Broker Patch XWB*1.1*13, the following library methods were added to the [TVCEdit Unit](#):

- [ChangeVerify](#)

```
function ChangeVerify(RPCBroker: TRPCBroker):  
Boolean;
```

- [SilentChangeVerify](#)

```
function SilentChangeVerify(RPCBroker: TRPCBroker;  
OldVerify, NewVerify1, NewVerify2: String; var  
Reason: String): Boolean;
```

- [StartProgSLogin](#)

```
procedure StartProgSLogin(const ProgLine: String;  
ConnectedBroker: TRPCBroker);
```

Modified

As of RPC Broker Patch XWB*1.1*13, the following library methods were modified:

- [CheckCmdLine](#)

```
function CheckCmdLine(SLBroker: TRPCBroker):  
Boolean;
```

Changed from procedure to function with a Boolean return value.

- [GetServerInfo](#)

The [GetServerInfo](#) function in the [RpcConf1](#) unit, which can be used to select the desired Server name and ListenerPort, was modified to add a **new** button. This button can be used 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](#)

The procedure Clear was moved from Private to Public.

- [TRPCB Unit](#)

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

Changed from Object: TObject, since this is what should be expected by the procedure if it is called.

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

Changed from Object: TObject, since this is what should be expected by the procedure if it is called.

Properties Added

As of RPC Broker Patch XWB*1.1*40, the following Properties were added to or modified (listed by component/class):

- **TCCOWRPCBroker Properties**

The following [TCCOWRPCBroker](#) component properties were added:

- [CCOWLogonIDName](#) (Public)
- [CCOWLogonIDValue](#) (Public)
- [CCOWLogonName](#) (Public)
- [CCOWLogonNameValue](#) (Public)
- [CCOWLogonVpid](#) (Public)
- [CCOWLogonVpidValue](#) (Public)
- [Contextor](#) (Public)

- **TVistaLogin Properties**

The following [TVistaLogin](#) class properties were added:

- [DomainName](#) (Public)
- [IsProductionAccount](#) (Public)

- **TVistaUser Property**

The following [TVistaUser](#) class properties were added:

- [Vpid](#) (Public)

As of RPC Broker Patches XWB*1.1*13 and 35, the following Properties were added to or modified (listed by component/class):

- **TRPCBroker Properties**

The following [TRPCBroker](#) component properties were added:

- [BrokerVersion](#) (Public)
- [CurrentContext](#) (Public)

- [IsBackwardCompatibleConnection](#) (Published)
- [IsNewStyleConnection](#) (Public)
- [KernelLogIn](#) (Published)
- [LogIn](#) (Public)
- [OldConnectionOnly](#) (Published)
- [OnRPCBFailure](#) (Public)
- [RPCBError](#) (Public)
- [ShowErrorMsgs](#) (Published)
- [User](#) (Public)

As of RPC Broker Patch XWB*1.1*23, the following Properties were added to or modified (listed by component/class):

- **TSharedBroker and TSharedRPCBroker Properties**

The following [TSharedBroker](#) and [TSharedRPCBroker](#) component properties were added:

- [AllowShared](#) (Public)
- [OnConnectionDropped](#) (Public)
- [OnLogout](#) (Published)

Source Code Availability

As of RPC Broker Patch XWB*1.1*14, the BDK contains the Broker source code. The source code is located in the ..\BDK32\Source directory.

 **CAUTION:** Modified BDK source code should *not* be used to create Vista GUI applications. For more details, see the "[Developer Considerations](#)" topic.

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.

Types Added/Modified

As of RPC Broker Patch XWB*1.1*13 and XWB*1.1*40, the following Types were added or modified:

- [TLoginMode](#)
- [TShowErrorMsgs](#)
- [TOnLoginFailure](#)
- [TOnRPCBFailure](#)
- [TParamType](#)

[Home](#) > [Overview](#) > Developer Considerations

Developer Considerations

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 Remedy Request Action support system for review and possible inclusion in a future patch.

The source code is located in the ..\BDk32\Source directory.

Design-time and Run-time Packages

As of RPC Broker Patch XWB*1.1*14, the BDK has separate run-time and design-time packages. There is no longer a Vista Broker package. The new packages are **XWB_DXEn** and **XWB_RXEn**, where "**D**" means Design-time and "**R**" means Run-time and where "**XEn**" is the Delphi version with which it should be used (e.g., XWB_DXE5 is the design-time package for Delphi XE5). The run-time package should *not* be used to create executables that depend on a separate XWB_RXEn.bpl installed on client workstations. There is no procedure in place at this time to reliably install the correct version of the run-time bpl on client workstations.

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

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: [redacted](#)

- [RPC BROKER APIs](#)
- [Existing M APIs](#)

Component Connect-Disconnect Behavior

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.

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.

[Home](#) > [Overview](#) > Application Considerations

Application Considerations

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
```

Deferred RPCs

In order to increase efficiency, applications can run [RPCs in the background](#).

Remote RPCs

In order to work with patient data across sites, applications can run [RPCs on a remote server](#).

Blocking RPCs

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

Silent Login

In special cases, applications can use one of two types of [Silent Login](#) to log in users *without* the RPC Broker prompting for login information.


[Home](#) > [Overview](#) > Online Help

Online Help

Distribution of the BDK includes online help, which provides an overview of development with the RPC Broker (e.g., components, properties, methods, etc.).

The help is distributed in two zip files:

- **Broker_1_1.zip** (i.e., **Broker_1_1.chm**)—This zip file contains the standalone online HTML help file. Unzip the contents and double-click on the **Broker_1_1.chm** file to open the help.
- **Broker_1_1-HTML_Files.zip**—This zip file contains the associated HTML help files. Unzip the contents in the same directory and double-click on the **index.htm** file to open the help.

 **NOTE:** You may want to make an entry for **Broker_1_1.chm** in Delphi's Tools Menu, to make it easily accessible from within Delphi. To do this, use Delphi's Tools | Configure Tools option and create a new menu entry.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Components and Examples](#) > TCCOWRPCBroker
Component

TCCOWRPCBroker Component

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

Parent Class

```
TRPCBroker = class(TComponent)
```


Unit


CCOWRPCBroker.pas

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 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" topic 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 Help file (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](#)).

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Components and Examples](#) > TCCOWRPCBroker
Example

TCCOWRPCBroker Example

For examples, see the Samples directory on the use of the [TCCOWRPCBroker](#) component: located in the `..\BDK32\Samples\CCOWRPCBroker` directory.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Components and Examples](#) > TContextorControl
Component

TContextorControl Component

As of RPC Broker Patch XWB*1.1*40, the TContextorControl component was added to RPC Broker 1.1. The TContextorControl Delphi component communicates with the Vergence Locator service.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Components and Examples](#) > TRPCBroker Component



TRPCBroker Component

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

Parent Class

```
TRPCBroker = class(TComponent)
```

Unit

[TRPCB](#)


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](#) and [RemoteProcedure](#) properties and invoking any of the following methods:

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

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 Help file (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).

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® Reflections 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 Attachmate® Reflection, "SSH" is set as a command line option or as a property within the application (set to Attachmate® Reflection). SSH is set to **True** 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, "PLINK" is set as a command line option or as a property within the application (set to Plink). SSH

is set to **True** if the following command line parameter is set:


SSHPort=portnumber

Support for Broker Security Enhancement (BSE)

As of RPC Broker Patch XWB*1.1*50, 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.

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.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Components and Examples](#) > TRPCBroker Example

TRPCBroker 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.

This example assumes that a [TRPCBroker](#) component already exists on the form as brkrRPCBroker1:

```
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 ..\BDK32\Samples\RPCBroker directory.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Components and Examples](#) > TSharedBroker
Component

TSharedBroker Component

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

Parent Class

```
TSharedBroker = class(TComponent)
```

Unit

[TRPCB](#)

Description

The TSharedBroker component is derived from the existing [TRPCBroker](#) component. The TSharedBroker component provides applications or plugins to applications easy access to an RPCBroker without the need for a separate M partition. Each component has its own security (i.e., option) as well. The default value of the [AllowShared](#) property is **True**. If an application has RPCs that require extensive time, it would be best to *not* share a Broker instance and the [AllowShared](#) property should then be set to **False**.

For its functionality, the TSharedBroker component uses the RPCSharedBrokerSessionMgr.EXE, which is an out of process Common Object Module (COM) component. This executable handles the actual Broker connections and communication, permitting multiple applications to use a single connection and partition to the VistA M Server. However, it also handles connections of different applications to different Server/Port combinations and can handle multiple connections to a specific Server/Port combination, if an application sets [AllowShared](#) property to **False**.

Like the [TRPCBroker](#), the TSharedBroker component and RPCSharedBrokerSessionMgr executable provide VistA application developers with all of the client/server-related functionality in one integrated component. Using the TSharedBroker 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](#) and [RemoteProcedure](#) properties and invoking any of the following methods:

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

Using the TSharedBroker Component

To use the TSharedBroker component in place of the [TRPCBroker](#) component with an existing application, do the following:

1. Open the application.
2. Notate the current name assigned to the [TRPCBroker](#) component.
3. Remove the [TRPCBroker](#) component.
4. Add the TSharedBroker component and give it the same name that was used for the [TRPCBroker](#) component (see Step #2).
5. If you do not have any other components (e.g., FileMan Delphi Components) that reference the original [TRPCBroker](#) component (see Step #2), simply recompile and run the application. Otherwise, proceed to Step #6.
6. If you have components (e.g., FMLister, FMGets, etc.) that reference the original [TRPCBroker](#) component, do the following:
 - a. Click on the components.
 - b. Select the new TSharedBroker component at the TRPCBroker reference for this component in the object


inspector. The assignment is *not* by name but to the actual component instance or location in memory at the time, and this has to be reset.

c. Repeat Steps #6a-6b for each additional component.

d. Recompile and run the application.

 **NOTE:** Application developers *must remember* to include the RPCSharedBrokerSessionMGR.EXE when building their applications.

When the first application connects, you see an instance of the RPCSharedBrokerSessionMgr appear in the taskbar. All Broker connections via the TSharedBroker component are routed through this executable.

 **REF:** For help on inherited properties, see the parent component (i.e., [TRPCBroker](#)).

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Components and Examples](#) > TSharedBroker Example


TSharedBroker Example

The following example demonstrates how a [TSharedBroker](#) 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.

This example assumes that a [TSharedBroker](#) component already exists on the form:

```
procedure TForm1.Button1Click(Sender: TObject);  
var  
    i: Integer;  
begin  
    try  
        if not SharedBroker1.Connected then  
            SharedBroker1.Connected := True;    {connect to the  
server}  
            //assign RPC name  
            SharedBroker1.RemoteProcedure := 'SOME APPLICATION  
RPC';  
            SharedBroker1.Call;    {make the call}  
            for i=0 to Pred(SharedBroker1.Results.Count) do  
                ListBox1.Items.Add(SharedBroker1.Results[ i ] );  
            {display results}  
        except  
            //put error handling code here  
        end;  
end;
```

 **REF:** For more examples, see the Samples directory on the use of the [TSharedBroker](#) component located in the ..\BDK32\Samples\SharedRPCBroker directory.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Components and Examples](#) > TSharedRPCBroker
Component

TSharedRPCBroker Component

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

Parent Class

```
TSharedRPCBroker = class(TComponent)
```

Unit

[TRPCB](#)

Description

The TSharedRPCBroker component is derived from the existing [TRPCBroker](#) component. The TSharedRPCBroker component provides applications or plugins to applications easy access to an RPCBroker without the need for a separate M partition. Each component has its own security (i.e., option) as well. The default value of the [AllowShared](#) property is **True**. If an application has RPCs that require extensive time, it would be best to *not* share a Broker instance and the [AllowShared](#) property should then be set to **False**.

For its functionality, the TSharedRPCBroker component uses the RPCSharedBrokerSessionMgr.EXE, which is an out of process Common Object Module (COM) component. This executable handles the actual Broker connections and communication, permitting multiple applications to use a single connection and partition to the VistA M Server. However, it also handles connections of different applications to different Server/Port combinations and can handle multiple connections to a specific Server/Port combination, if an application sets [AllowShared](#) property to **False**.

Like the [TRPCBroker](#), the TSharedRPCBroker component and RPCSharedBrokerSessionMgr executable provide VistA application developers with all of the client/server-related functionality in one integrated component. Using the TSharedRPCBroker 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](#) and [RemoteProcedure](#) properties and invoking any of the following methods:

- [Call](#)
- [strCall](#)
- [IstCall](#)

Using the TSharedRPCBroker Component


To use the TSharedRPCBroker component in place of the [TRPCBroker](#) component with an existing application, do the following:

1. Open the application.
2. Notate the current name assigned to the [TRPCBroker](#) component.
3. Remove the [TRPCBroker](#) component.
4. Add the TSharedRPCBroker component and give it the same name that was used for the [TRPCBroker](#) component (see Step #2).
5. If you do not have any other components (e.g., FileMan Delphi Components) that reference the original [TRPCBroker](#) component (see Step #2), simply recompile and run the application. Otherwise, proceed to Step #6.
6. If you have components (e.g., FMLister, FMGets, etc.) that reference the original [TRPCBroker](#) component, do the following:

- a. Click on the components.
- b. Select the new TSharedRPCBroker component at the [TRPCBroker](#) reference for this component in the object inspector. The assignment is *not* by name but to the actual component instance or location in memory at the time, and this has to be reset.
- c. Repeat Steps #6a-6b for each additional component.
- d. Recompile and run the application.

 **NOTE:** Application developers *must remember* to include the RPCSharedBrokerSessionMGR.EXE when building their applications.

When the first application connects, you see an instance of the RPCSharedBrokerSessionMgr appear in the taskbar. All Broker connections via the TSharedRPCBroker component are routed through this executable.

 **REF:** For help on inherited properties, see the parent component (i.e., [TRPCBroker](#)).

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Components and Examples](#) > TSharedRPCBroker
Example

TSharedRPCBroker Example

The following example demonstrates how a [TSharedRPCBroker](#) 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.

This example assumes that a [TSharedRPCBroker](#) component already exists on the form:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
begin
  try
    if not SharedRPCBroker1.Connected then
      SharedRPCBroker1.Connected := True;   {connect to the
server}
      //assign RPC name
      SharedRPCBroker1.RemoteProcedure := 'SOME APPLICATION
RPC';
      SharedRPCBroker1.Call;   {make the call}
      for i=0 to Pred(SharedRPCBroker1.Results.Count) do
        ListBox1.Items.Add(SharedRPCBroker1.Results[ i ] );   {d
isplay results}
      except
        //put error handling code here
      end;
  end;
```

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Components and Examples](#) > TXWBRichEdit
Component

TXWBRichEdit Component

[Property.](#)

Parent Class


```
TXWBRichEdit = class(TComponent)
```

Unit

XwbRich20

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.

 **NOTE:** Properties inherited from the parent component (i.e., TComponent) are *not* discussed in this Help file (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).

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Classes](#) > TMulti Class

TMult Class

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

- [Example](#)

Unit

[TRPCB](#)

Description

The TMulti class is used whenever a list of multiple values needs to be passed to a remote procedure call ([RPC](#)) in a single parameter. The [Multi](#) property of a parameter is of TMulti type. The information put in the TMulti 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 TMulti 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 TMulti 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.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Classes](#) > TParamRecord Class

TParamRecord Class

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

Unit

[TRPCB](#)

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.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Classes](#) > TParams Class

TParams Class

- [Property](#)
- [Method](#)

- [Example](#)


Unit

[TRPCB](#)

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.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Classes](#) > TVistaLogin Class

TVistaLogin Class

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

Unit

[TRPCB](#)

Description

The TVistaLogin class is used to hold login parameters for [Silent Login](#).

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Classes](#) > TVistaUser Class

TVistaUser Class


[Properties](#)

Unit

[TRPCB](#)

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](#).

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Classes](#) > TXWBWinsock Class

TXWBWinsock Class

Unit

[TRPCB](#)

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 Using 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.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Units](#) > Hash Unit

Hash Unit

Library Methods

[Encrypt](#)

[Decrypt](#)



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

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Units](#) > LoginFrm Unit

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.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Units](#) > MFunStr Unit

MFunStr Unit

Library Methods

- [piece](#)
- [translate](#)


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

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Units](#) > RPCConf1 Unit

RPCConf1 Unit

Library Methods

- [GetServerInfo](#)
- [GetServerIP](#)


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

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Units](#) > RpcSLogin Unit

RpcSLogin Unit

Library Methods

- [CheckCmdLine](#)
- [StartProgSLogin](#)


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

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Units](#) > SpIVista Unit

SplVista Unit

Library Methods

- [SplashOpen](#)
- [SplashClose](#)

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

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Units](#) > TRPCB Unit

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`

Classes

- [TMult](#)
- [TParamRecord](#)

- [TParams](#)

- [TVistaLogin](#)

- [TVistaUser](#)

Component

[TRPCBroker](#)

Library Methods

- GetAppHandle
- [TMult Class Methods](#)
- [TParams Class Method](#)
- [TRPCBroker Component Methods](#)

Types

- [EBrokerError](#)
- [TloginMode](#)
- [TParamType](#)



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

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Units](#) > TVCEdit Unit

TVCEdit Unit

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

Library Methods

- [ChangeVerify](#).
- [SilentChangeVerify](#).



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

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Methods and Examples](#) > Assign Method (TMult Class)

Assign Method (TMulti Class)

[Example](#)

Applies to

[TMult](#) class

Declaration

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

Description

The Assign method for a TMulti class takes either a Tstrings, a TStringList, or another TMulti. In the case where the source is a TMulti, 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 [TMulti](#) class, see the "[RPC Limits](#)" topic.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Methods and Examples](#) > Assign Example (TMult Class)

Assign Example (TMult Class)

TMult Assign Method—Assigning listbox items to a TMULT

1. Start a new application.
2. Drop one listbox, one memo and one button on the form. Arrange controls as in the figure below.
3. Copy the following code to the **Button1.OnClick** event:

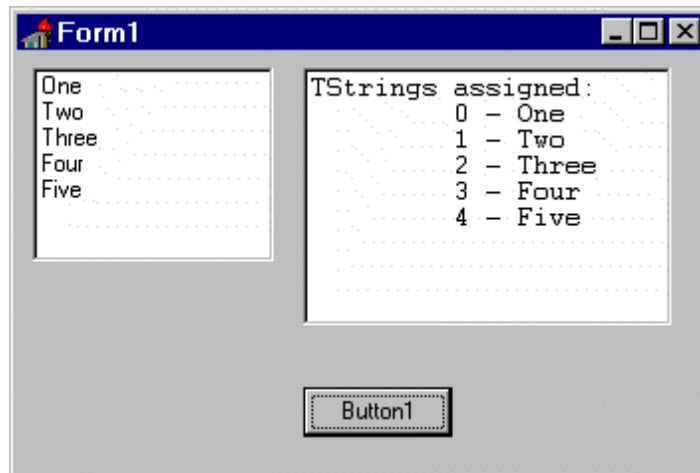
```
procedure TForm1.Button1Click(Sender: TObject);  
var  
    Mult1: TMult;  
    Subscript: string;  
begin  
  
    //Create Mult1. Make Form1 its owner  
    Mult1 := TMult.Create(Form1);  
  
    //Fill listbox with some strings  
    ListBox1.Items.Add('One');  
    ListBox1.Items.Add('Two');  
    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
    Mem1.Lines.Add(Format('%10s', [ Subscript] ) + ' - ' + Mult1[ Subscript] )

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

4. Run the project and click on the button.

Expected output:



TMult Assign Method—Assigning One TMULT to Another

The following program code demonstrates the use of the TMult assign method to assign one TMult to another:

1. Start a new application.
2. Drop one memo and one button on the form. Arrange controls as in the figure below.
3. Copy the following code 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  
    Mem1.Font.Name := 'Courier';
```

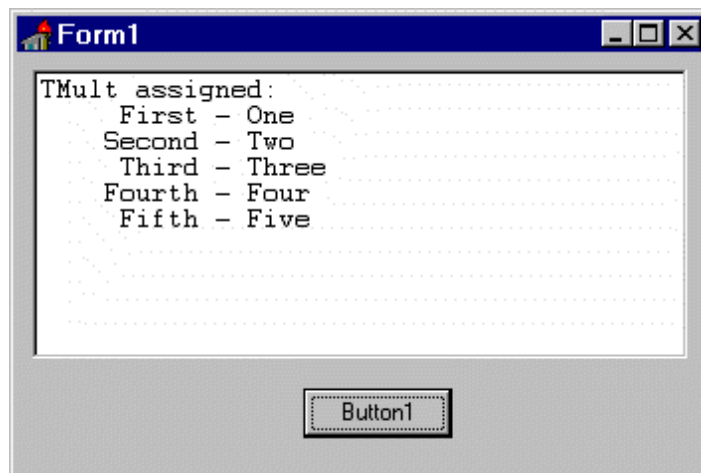


```
Mem1.Lines.Clear;
Mem1.Lines.Add('TMult assigned:');

//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
        Mem1.Lines.Add(Format('%10s', [ Subscript] ) + '
- ' + Mult2[ Subscript] )
        //stop when reached the end
    until Subscript = '';
end;
```

4. Run the project and click on the button.

Expected output:



[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Methods and Examples](#) > Assign Method (TParams Class)

Assign Method (TParams Class)

[Example](#)

Applies to


[TParams](#) class

Declaration

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

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](#)" topic.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Methods and Examples](#) > Assign Example (TParams Class)

Assign Example (TParams Class)

The following program code demonstrates how a TParams assign method can be used to save off the [TRPCBroker](#) component parameters and restore them later:

```
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;
```


[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Methods and Examples](#) > Call Method

Call Method

[Example](#)

```
procedure Call;
```

Description

This 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](#) and [Param](#) properties respectively. If [ClearResults](#) is **True**, then the Results property is cleared before the call. If [ClearParameters](#) is **True**, then the Param property is cleared after the call finishes.

i **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](#) topic.

i **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.


i **REF:** For a demonstration using the Call method, run the RPC Broker Example (i.e., [BrokerExample.EXE](#)) located in the `..\BDK32\Samples\BrokerEx` directory.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Methods and Examples](#) > Call Example

Call Example

The following program code 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:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    brkrRPCBroker1.RemoteProcedure := 'GET PATIENT  
DEMOGRAPHICS';  
    brkrRPCBroker1.Param[ 0 ].Value := 'DFN';  
    brkrRPCBroker1.Param[ 0 ].PType := reference;  
    brkrRPCBroker1.Call;  
    Mem1.Lines := brkrRPCBroker1.Results;  
end;
```

 **REF:** For a demonstration using the Call method, run the RPC Broker Example (i.e., [BrokerExample.EXE](#)) located in the ..\BDK32\Samples\BrokerEx directory.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Methods and Examples](#) > CreateContext Method

CreateContext Method

[Example](#)

```
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 file (#19), 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.

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.

i **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.

i **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](#)" topic.

i **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.


i **REF:** For a demonstration that creates an application context, run the RPC Broker Example (i.e., [BrokerExample.EXE](#)) located in the `..\BDK32\Samples\BrokerEx` directory.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Methods and Examples](#) > CreateContext Example

CreateContext Example

The following program code demonstrates the use of the [CreateContext](#) method:

```
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 (i.e., [BrokerExample.EXE](#)) located in the ..\BDK32\Samples\BrokerEx directory.


[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Methods and Examples](#) > GetCCOWtoken Method

GetCOWtoken Method


Declaration

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

This 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 runtime without using a component.

 **REF:** For an example of the GetCCOWtoken method, run the RPC Broker Example (i.e., [BrokerExample.EXE](#)) located in the ..\BDK32\Samples\BrokerEx directory.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Methods and Examples](#) > IsUserCleared Method

IsUserCleared Method

[Example](#)

```
function IsUserCleared: Boolean;
```

This method returns a value of **True** if the user value in the Context Vault has been cleared. The value is only of interest if [WasUserDefined](#) 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.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Methods and Examples](#) > IsUserCleared Example

IsUserCleared 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, then halt.

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

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Methods and Examples](#) > IsUserContextPending
Method


IsUserContextPending Method

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

This 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 (i.e., [BrokerExample.EXE](#)) located in the ..\BDK32\Samples\BrokerEx directory.


[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Methods and Examples](#) > IstCall Method


IstCall Method

[Example](#)

```
procedure IstCall(OutputBuffer: Tstrings;
```

This 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. IstCall expects the name of the remote procedure and its parameters to be set up in the [RemoteProcedure](#) and [Param](#) properties respectively. The [Results](#) property is *not* affected by this call. If [ClearParameters](#) 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](#) topic.

 **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 `IstCall` method, run the RPC Broker Example (i.e., [BrokerExample.EXE](#)) located in the `..\BDK32\Samples\BrokerEx` directory.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Methods and Examples](#) > IstCall Example

IstCall Example

The following program code demonstrates the use of the [IstCall](#) method in a hypothetical example of bringing back a list of user's keys and automatically filling a list box with data:

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

 **REF:** For a demonstration using the IstCall method, run the RPC Broker Example (i.e., [BrokerExample.EXE](#)) located in the ..\BDK32\Samples\BrokerEx directory.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Methods and Examples](#) > pchCall Method

pchCall Method

```
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](#), [strCall](#), and [lstCall](#)), 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.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Methods and Examples](#) > Order Method

Order Method

[Example](#)

Applies to

[TMult](#) class

Declaration

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

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](#) topic.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Methods and Examples](#) > Order Example

Order Example

The following program code demonstrates how 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;
```

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Methods and Examples](#) > Position Method

Position Method

[Example](#)

Applies to

[TMult](#) class

Declaration

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

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](#) topic.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Methods and Examples](#) > Position Example

Position Example

The following program code demonstrates 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);  
    Labell.Caption := 'The position of the ''Third'' element  
is ' +  
        {is -1 since the list is empty}  
        IntToStr(Mult.Postion('Third'));  
    Mult[ 'Second'] := 'Two';  
    Labell.Caption := 'The position of the ''Third'' element  
is ' +  
        {is -1 since 'Third' item does not exit}  
        IntToStr(Mult.Postion('Third'));  
    Labell.Caption := 'The position of the ''Second'' element  
is ' +  
        {is 0, TMult positions start with 0}  
        IntToStr(Mult.Postion('Second'));  
end;
```

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Methods and Examples](#) > strCall Method

strCall Method

[Example](#)

```
function strCall: string;
```

This 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](#) and [Param](#) properties respectively. The [Results](#) property is not affected by this call. If [ClearParameters](#) is **True**, then the [Param](#) property is cleared after the call finishes.

i **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](#)" topic.

i **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.


i **REF:** For a demonstration using the strCall method, run the RPC Broker Example (i.e., [BrokerExample.EXE](#)) located in the ..\BDK32\Samples\BrokerEx directory.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Methods and Examples](#) > strCall Example

strCall Example

The following program code 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:

```
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 (i.e., [BrokerExample.EXE](#)) located in the ..\BDK32\Samples\BrokerEx directory.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Methods and Examples](#) > Subscript Method

Subscript Method

[Example](#)

Applies to

[TMult](#) class

Declaration

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

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 and [Position](#) 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](#)" topic.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Methods and Examples](#) > Subscript Example

Subscript Example

The following program code demonstrates how to get the subscript 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 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;
```

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Methods and Examples](#) > TCCOWRPCBroker Methods

TCCOWRPCBroker Methods

- [TCCOWRPCBroker](#)
- [GetCCOWtoken](#)

- [IsUserCleared](#)

- [IsUserContextPending](#)

- [WasUserDefined](#)

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Methods and Examples](#) > TRPCBroker Methods

TRPCBroker Methods

- [TRPCBroker](#)
- [Call](#)
- [CreateContext](#)
- [IstCall](#)
- [pchCall](#)
- [strCall](#)

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Methods and Examples](#) > TMulti Methods

TMult Methods

- [TMult](#)
- [Assign](#)
- [Order](#)
- [Position](#)
- [Subscript](#)

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Methods and Examples](#) > TParams Method

TParams Method

- [TParams](#)
- [Assign](#)

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Methods and Examples](#) > WasUserDefined Example

WasUserDefined 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, then halt.

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

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Methods and Examples](#) > WasUserDefined Method

WasUserDefined Method

[Example](#)

```
function WasUserDefined: Boolean;
```

This 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.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > AccessCode Property

AccessCode Property

[Example](#)

Applies to


[TVistaLogin](#) class


Declaration

```
property AccessCode: String;
```

Description

The AccessCode property is available at run-time only. It holds the Access code for the ImAVCodes 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.

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

 **REF:** For a demonstration using the ImAVCodes, run the ImAVCodes_Demo.EXE located in the ..\BDK32\Samples\SilentSignOn directory.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > AllowShared Property

AllowShared Property

Applies to


[TSharedRPCBroker](#) component

Declaration

`property AllowShared: Boolean;`

Description

The AllowShared property determines whether or not the connection through the RPCBroker to the VistA M Server can be shared with other applications. If it is not set, the value is **False** and the application has its own dedicated partition on the server. If it is set to **True**, the partition can be shared with other applications.

 **CAUTION:** If an application depends on whether local variables from previous calls are present in the partition ****DO NOT**** permit the partition to be Shared. If the partition is shared, local variables are cleared out between RPC calls.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > BrokerVersion Property

BrokerVersion Property (read-only)

Applies to

[TRPCBroker](#) component

Declaration

```
property BrokerVersion: String;
```

Description

The BrokerVersion property is available at run-time only. This read-only property indicates the RPC Broker version used in generating the application (currently, it returns the string "XWB*1.1*50").

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > CCOWLogonIDName Property

CCOWLogonIDName Property **(read-only)**

Applies to

[TCCOWRPCBroker](#) component

Declaration

```
property CCOWLogonIDName: String;
```

Description

The CCOWLogonIDName property is available at run-time only. This read-only property 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 logon ID name value used within the Context Vault related to User Context.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > CCOWLogonIDValue Property

CCOWLogonIDValue Property **(read-only)**

Applies to

[TCCOWRPCBroker](#) component

Declaration

```
property CCOWLogonIDValue: String;
```


Description

The CCOWLogonIDValue property is available at run-time only. This read-only property 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 logon ID name used within the Context Vault related to User Context.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > CCOWLogonName
Property

CCOWLogonName Property **(read-only)**

Applies to

[TCCOWRPCBroker](#) component

Declaration

```
property CCOWLogonName: String;
```

Description

The CCOWLogonName property is available at run-time only. This read-only property 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 logon name value used within the Context Vault related to User Context.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > CCOWLogonNameValue
Property

**CCOWLogonNameValue
Property (read-only)**

Applies to

[TCCOWRPCBroker](#) component

Declaration

```
property CCOWLogonNameValue: String;
```

Description

The CCOWLogonNameValue property is available at run-time only. This read-only property 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 logon name used within the Context Vault related to User Context.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > CCOWLogonVpid Property

CCOWLogonVpid Property (read-only)

Applies to

[TCCOWRPCBroker](#) component

Declaration

```
property CCOWLogonVpid: String;
```

Description

The CCOWLogonVpid property is available at run-time only. This read-only property gives the name used to store the LogonVpid value in the CCOW Context Vault.

It permits the user to identify the logon VPID name associated with the [CCOWLogonVpidValue](#) property logon VPID value used within the Context Vault related to User Context.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > CCOWLogonVpidValue
Property

CCOWLogonVpidValue
Property (read-only)

Applies to

[TCCOWRPCBroker](#) component

Declaration

```
property CCOWLogonVpidValue: String;
```

Description

The CCOWLogonVpidValue property is available at run-time only. This read-only property 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 logon VPID name used within the Context Vault related to User Context.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > ClearParameters Property

ClearParameters Property

[Example](#)

Applies to

[TRPCBroker](#) component

Declaration

```
property ClearParameters: Boolean;
```

Description

The ClearParameters design-time property gives the developer the option to clear the [Param](#) property following every invocation of any of the following methods:

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

- [lstCall](#)

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](#), [strCall](#), and [lstCall](#) methods. 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.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > ClearParameters Example

ClearParameters Example

The following program code sets the [ClearParameters](#) property to **True**:

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

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > ClearResults Property

ClearResults Property

[Example](#)

Applies to

[TRPCBroker](#) component

Declaration

```
property ClearResults: Boolean;
```


Description

The ClearResults design-time property gives the developer the option to clear the [Results](#) property prior to every invocation of the [Call](#) method. The [strCall](#) and [lstCall](#) methods 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.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > ClearResults Example

ClearResults Example

The following program code sets the [ClearResults](#) property to **True**:

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

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > Connected Property

Connected Property

[Example](#)

Applies to

[TRPCBroker](#) component

property Connected: **Boolean;**

Description

The Connected design-time property connects your 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 your RPC unless a context has been created with the [CreateContext](#) method.

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 your application form's OnCreate event. For that event, you could include code like the following:

```
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 (when the application is closed, for example), 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](#).

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > Connected Example

Connected Example

The following program code sets the [Connected](#) property to **True**:

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

 **NOTE:** The [Server](#) and [ListenerPort](#) properties *must* be set at design or run-time before setting the Connected property to **True**.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > Contextor Property

Contextor Property

Applies to


[TCCOWRPCBroker](#) component

Declaration

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

Description

The Contextor property is available at run-time only. It *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 [TRPCBroker](#), 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 runtime without using a component.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > Count Property (TMult Class)

Count Property (TMulti Class)

[Example](#)

Applies to

[TMult](#) class

Declaration

`property` Count: Word;

Description

The Count design-time property contains the number of items in a [TMult](#) class. If TMult class is empty, Count is zero.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > Count Example (TMult Class)

Count Example (TMulti Class)

The following program code displays the number of items in a [Multi](#) class in the caption of a label when the user clicks the **CountItems** button:

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

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > Count Property (TParams Class)

Count Property (TParams Class)

[Example](#)

Applies to

[TParams](#) class

Declaration

`property` Count: Word;

Description

The Count property contains the number of parameters in a [TParams](#) class. If the TParams class is empty, Count is zero.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > Count Example (TParams Class)

Count Example (TParams Class)

The following program code displays the number of parameters in a [TParams](#) variable within the caption of a label when the user clicks the **CountParameters** button:

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

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > CurrentContext Property

CurrentContext Property (read-only)

[Example](#)

Applies to

[TRPCBroker](#) component

Declaration

```
property CurrentContext: String;
```

Description

The CurrentContext property is available at run-time only. This read-only property provides the current context. Developers can save off the current context into a local variable, set a new context, and then 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.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > CurrentContext Example

CurrentContext Example

The following program code demonstrates the use of the [CurrentContext](#) property in a hypothetical example of saving and restoring the current context of an application:

```
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;
```

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > DebugMode Property

DebugMode Property

Applies to

[TRPCBroker](#) component

Declaration

```
property DebugMode: Boolean;
```


Description

The DebugMode design-time property controls how the VistA M Server process should be started. The default setting is **False**.

A setting of **False** starts the VistA M Server in the usual manner, as a background process.

For debugging purposes, it can be very helpful to:

1. Set break points.
2. Run the server process interactively.
3. Step through the execution.

For those situations, set this property to **True**. When the [TRPCBroker](#) component connects with this property set to **True**, you get a dialogue window indicating your workstation [Internet Protocol \(IP\) address](#) and the port number.

At this point, you should:

1. Switch over to the server.
2. Enter break points.

3. Issue the debug command (e.g., [ZDEBUG](#) in DSM).

4. Start the following server process:

```
>D EN^XWBTCP
```

You are prompted for the workstation [Internet Protocol \(IP\) address](#) and the port number. After entering the information, switch over to the client application and click **OK**.



REF: For more information, see the "[How to Debug Your Application](#)" topic.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > Division Property
(TVistaLogin Class)

Division Property ([TVistaLogin](#) Class)

Applies to

[TVistaLogin](#) class

Declaration

```
property Division: String;
```

Description

The Division property is available at run-time only. It 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](#)" topic.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > Division Property
(TVistaUser Class)

**Division Property ([TVistaUser](#)
class)**

Applies to

[TVistaUser](#) class

```
property Division: String;
```

Description

The Division property is available at run-time only. It is set to the division for a user when they are logged on.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > DivList Property

DivList Property (read-only)

Applies to


[TVistaLogin](#) class

Declaration

```
property DivList: Tstrings;
```

Description

The DivList property is available at run-time only. This read-only property 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](#)" topic.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > DomainName Property

DomainName Property

Applies to

[TVistaLogin](#) class

Declaration

```
property DomainName: String;
```

Description

The `DomainName` property is available at run-time only. It can be used to obtain the domain name of the server to which the RPC Broker is currently connected.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > DTime Property

DTime Property

Applies to

[TVistaUser](#) class

Declaration

```
property DTime: String;
```

Description

The DTime property is available at run-time only. It 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.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > DUZ Property (TVistaLogin Class)

DUZ Property ([TVistaLogin](#) Class)

Applies to

[TVistaLogin](#) class

Declaration

```
property DUZ: String;
```

Description

The DUZ property is available at run-time only. It holds the user's Internal Entry Number (IEN) from the NEW PERSON file (#200) for TVistaLogin.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > DUZ Property (TVistaUser Class)

DUZ Property ([TVistaUser](#) Class)

Applies to

[TVistaUser](#) class

Declaration

```
property DUZ: String;
```

Description

The DUZ property is available at run-time only. It holds the user's Internal Entry Number (IEN) from the NEW PERSON file (#200) for TVistaUser.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > ErrorText Property

ErrorText Property

Applies to

[TVistaLogin](#) class

Declaration

```
property ErrorText: String;
```


Description

The ErrorText property is available at run-time only. It 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:

- The Verify code needs to be changed.
- An invalid Access/Verify code pair.
- An invalid Division.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > First Property

First Property

[Example](#)

Applies to

[TMult](#) class

Declaration

```
property First: String;
```

Description

The First design-time property 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.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > First Example

First Example

The following program code 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:

```
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;
```


[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > IsBackwardCompatibleConnection Property

IsBackwardCompatibleConnection Property

Applies to

[TRPCBroker](#) component

Declaration

```
property IsBackwardCompatibleConnection: Boolean;
```

Description

The `IsBackwardCompatibleConnection` property is used to determine whether the connection to be made should be an old-style (i.e., callback) or a new-style (i.e., UCX or *non-callback*) RPC Broker connection. When set to:

- **True (default)**—The RPC Broker makes a regular callback connection.
- **False**—The RPC Broker makes a UCX or *non-callback* connection, so that it can be used from behind firewalls, routers, etc.

The value should be set to **False** if any of the [ParamType](#) values are used in RPCs.

 **NOTE:** This property was introduced with RPC Broker Patch XWB*1.1*35.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > IsNewStyleConnection
Property

IsNewStyleConnection Property **(read-only)**

Applies to

[TRPCBroker](#) component

Declaration

```
property IsNewStyleConnection: Boolean;
```

Description

The `IsNewStyleConnection` property is available at run-time only. This read-only property indicates whether or not the current connection is a new-style (i.e., *non-callback*) connection.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > IsProductionAccount
Property

IsProductionAccount Property

Applies to

[TVistaLogin](#) class

Declaration

```
property IsProductionAccount: Boolean;
```

Description

The `IsProductionAccount` property is available at run-time only. It 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.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > KernelLogIn Property

KernelLogIn Property

[Example](#)

Applies to

[TRPCBroker](#) component

Declaration

```
property KernelLogIn: Boolean;
```

Description

The KernelLogin design-time property 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](#).

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > Language Property

Language Property

Applies to

[TVistaUser](#) class

Declaration

```
property Language: String;
```


Description

The Language property is available at run-time only. It holds the user's language from the NEW PERSON file (#200).

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > Last Property

Last Property

[Example](#)

Applies to

[TMult](#) class

Declaration

```
property Last: String;
```

Description

The Last design-time property 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.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > Last Example

Last Example

The following program code 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:

```
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;
```


[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > ListenerPort Property

ListenerPort Property

[Example](#)

Applies to

[TRPCBroker](#) component

Declaration

```
property ListenerPort: Integer;
```

Description

The ListenerPort design-time property 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](#)), which is used for the remainder of the session.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > ListenerPort Example

ListenerPort Example

The following program code demonstrates using the [ListenerPort](#) property:

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

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > LogIn Property

LogIn Property

[Example](#)

Applies to

[TRPCBroker](#) component

Declaration

`property` LogIn: [TVistaLogin](#);

Description

The LogIn property is available at run-time only. It 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.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > LoginHandle Property

LoginHandle Property

Applies to


[TVistaLogin](#) class

Declaration

```
property LoginHandle: String;
```


Description

The LoginHandle property is available at run-time only. It holds the Application Handle for the ImAppHandle 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](#) 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 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 [Server](#), [ListenerPort](#), and [Division](#) 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.



REF: For a demonstration using the ImAppHandle, run the ImAppHandle_Demo.EXE located in the ..\BDK32\Samples\SilentSignOn directory.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > Mode Property

Mode Property

[Example](#)

Applies to


[TVistaLogin](#) class


Declaration

`property Mode: TloginMode;`

Description

The Mode property is available at run-time only. It indicates the mode of [Silent Login](#). The possible values include: ImAVCodes and ImAppHandle.

 **REF:** For a demonstration using the ImAVCodes, run the ImAVCodes_Demo.EXE located in the ..\BDK32\Samples\SilentSignOn directory.

 **REF:** For a demonstration using the ImAppHandle, run the ImAppHandle_Demo.EXE located in the ..\BDK32\Samples\SilentSignOn directory.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > TLoginMode Type

TLoginMode Type

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

Unit


TRPCB

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

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.

The following table lists the possible values:

Value	Meaning
ImAVCodes	<p>Used if the application is passing in the user's Access and Verify codes during Silent Login.</p> <p> REF: For a demonstration using the ImAVCodes, run the ImAVCodes_Demo.EXE located in the ..\BDK32\Samples\SilentSignOn directory.</p>
ImAppHandle	<p>Used to pass in an application handle rather than a user's Access and Verify codes during Silent Login. It sets the mode to ImAppHandle and the KernelLogIn property to False. Indicates that an application handle is being passed to the application when it</p>

was being started as opposed to Access and Verify codes.



REF: For a demonstration using the ImAppHandle, run the ImAppHandle_Demo.EXE located in the ..\BDK32\Samples\SilentSignOn directory.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > Mult Property

Mult Property

[Example](#)

Applies to

[TParamRecord](#) class

Declaration

```
property Mult: TMult;
```


Description

(Mult is a property of the [TParamRecord](#) used in the [Param](#) property.)

The Mult design-time property 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 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.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > Mult Example

Mult Example

The following program code demonstrates how the [Mult](#) property can be used to pass several data elements to the VistA M Server in one parameter:

```
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 the following:

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

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > MultiDivision Property

MultiDivision Property

Applies to

[TVistaLogin](#) class

Declaration

```
property MultiDivision: Boolean;
```

Description

The MultiDivision property is available at run-time only. It 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](#)" topic.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > Name Property

Name Property

Applies to

[TVistaUser](#) class

Declaration

```
property Name: String;
```

Description

The Name property is available at run-time only. It holds the user's name from the NEW PERSON file (#200).

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > OldConnectionOnly
Property

OldConnectionOnly Property

Applies to

[TRPCBroker](#) component

Declaration

```
property OldConnectionOnly: Boolean;
```

Description

The OldConnectionOnly property can be set to **True** if only an old-style (i.e., callback) connection is desired. This is primarily for developers during debugging, if an initial connection is *not* a new-style (i.e., *non-callback*) RPC Broker connection, an error message is displayed. Clicking **OK** and **F9** (run) causes the old-style (or callback) connection to be made. However, setting this property to **True** does away with the initial attempt to make a new-style connection, and thus, the error message during debugging. The error message is *not* seen outside of the debugger within Delphi.



CAUTION: Applications should *not* be released with this property set to True!

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > OnConnectionDropped
Property

OnConnectionDropped Property

Applies to

[TSharedRPCBroker](#) component

Declaration

```
property OnConnectionDropped: TOnConnectionDropped;
```

Description

The `OnConnectionDropped` property is used to provide a capability for handling a connection loss. You can recognize a loss of connection to the VistA M Server when:

- An RPC call is made.
- The `RPCBroker` sends its background messages to the VistA M Server at intervals of 45 to 60 seconds to let the server know that the application is still connected.

Since with the `SharedRPCBroker`, this loss would be recognized in the `RPCSharedBrokerSessionMgr` and *not* in the individual applications, they would not normally be aware of this problem until another RPC call is made.

To promptly notify the application, the `RPCSharedBrokerSessionMgr` sends a message back to the applications that have implemented a procedure for the `OnConnectionDropped` property. This procedure accepts as arguments:

- An integer (the connection index).
- A `WideString` containing any error text for the disconnection; as indicated by the `TOnConnectionDropped` declaration.

A default procedure is supplied that displays a dialogue box indicating the loss of connection and any error text supplied. If further processing is desired, a custom procedure should be created and the OnConnectionDropped property set to it.

```
TOnConnectionDropped = procedure (ConnectionIndex: Integer;  
ErrorText: WideString) of object;
```

This indicates the format of the procedure and arguments required for a method to be used as the OnConnectionDropped property.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > OnFailedLogin Property

OnFailedLogin Property

[Example](#)

Applies to

[TVistaLogin](#) class

Declaration

```
property OnFailedLogin: TOnLoginFailure;
```

Description

The OnFailedLogin property is available at run-time only. It 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;
```

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > OnLogout Property

OnLogout Property

Applies to

[TSharedRPCBroker](#) component

Declaration

```
property OnLogout: TNotifyEvent;
```

Description

The OnLogout property is used to provide a capability in the future for the RPCSharedBrokerSessionMgr to receive a message requesting all applications to log out. This message would then be propagated to all applications that are connected with the RPCSharedBrokerSessionMgr. If a procedure is specified as a value for the OnLogout property, it is called when this message is received. It can do any processing necessary prior to logging out of the system. There is a default method that passes the message along to the main window for the application, requesting it to close.

```
TLogout = procedure () of object;
```

This class defines the structure of the procedure necessary to be used as an OnLogout value. It would simply be a procedure with no arguments.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > OnRPCBFailure Property

OnRPCBFailure Property

[Example](#)

Applies to

[TRPCBroker](#) component

Declaration


```
property OnRPCBFailure: TOnRPCBFailure;
```


Description

The OnRPCBFailure property is available at run-time only. It 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.

 **NOTE:** If [OnFailedLogin](#) is also set, it handles any login errors and not pass them up.

 **REF:** To illustrate the effects of TRPCBroker properties related to Error Handling, run the "Error Handling Demo" application (i.e., XWBOOnFail.EXE) located in the ..\BDK\Samples\SilentSignOn p [13] directory.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > OnRPCBFailure Example


OnRPCBFailure 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.

The following instance of an error handler 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:

```
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;
```

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > Param Property

Param Property

[Example](#)

Applies to

[TRPCBroker](#) component

Declaration

`property` Param: [TParams](#);

Description


The Param property is available at run-time only. It holds all of the parameters that the application needs to pass to the remote procedure using any of the following methods:

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

Param is a zero-based array of [TParamRecord](#). 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](#)
- [PType](#)
- [Value](#)

 **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), where you can skip passing the second parameter in DSM code. If there are fewer elements in the Param array than exist as input parameters in the [RPC](#), the subsequent parameters is *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 (i.e., [BrokerExample.EXE](#)) located in the ..\BDK32\Samples\BrokerEx directory.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > Param Example

Param Example

The following program code demonstrates how the [Param](#) property of a [TRPCBroker](#) component is referenced and filled with two parameters that the remote procedure expects:

```
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 (i.e., [BrokerExample.EXE](#)) located in the ..\BDK32\Samples\BrokerEx directory.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > PromptDivision Property

PromptDivision Property

[Example](#)

Applies to

[TVistaLogin](#) class

Declaration

`property PromptDivison: Boolean;`

Description

The PromptDivision property is available at run-time only. It 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 what would be expected.

6. If the Login.[MultiDivision](#) property is set to **True**, then check the Login.[DivList](#) property 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 to the selected Division.

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

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > PType Property

PType Property

[Example](#)

Applies to

[TParamRecord](#) class

Declaration

```
property PType: TParamType;
```

Description

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

The PType design-time property determines how the parameter is interpreted and handled by the Broker.


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.
list	A single-dimensional array of strings in the Mult subproperty of the Param property, passed to the VistA

	<p>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 contents of the corresponding Mult property is sent, while the Value property is ignored.</p>
global	<p>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. This value was made available as of RPC Broker Patch XWB*1.1*40.</p>
empty	<p>This value indicates that no parameter value is to be passed; it simply passes an empty argument. This value was made available as of RPC Broker Patch XWB*1.1*40.</p>
stream	<p>This value indicates that the data should be passed as a single stream of data. This value was made available as of RPC Broker Patch XWB*1.1*40.</p>
undefined	<p>The Broker uses this value internally. It should <i>not</i> be used by an application.</p>

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 (i.e., [BrokerExample.EXE](#)) located in the ..\BDK32\Samples\BrokerEx directory.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > PType Example

PType Example

The following program code demonstrates a couple of different uses of the [PType](#) property. Remember, that each Param[x] element is really a [TParamRecord](#)-type class.

```
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 (i.e., [BrokerExample.EXE](#)) located in the ..\BDK32\Samples\BrokerEx directory.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > RemoteProcedure Property

RemoteProcedure Property

[Example](#)

Applies to

[TRPCBroker](#) component

Declaration

```
property RemoteProcedure: TRemoteProc;
```

Description

The RemoteProcedure design-time property should be set to the name of the remote procedure call entry in the [REMOTE PROCEDURE file](#) (#8994).

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > RemoteProcedure Example

RemoteProcedure Example

The following program code demonstrates using the [RemoteProcedure](#) property:

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

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > Results Property

Results Property

[Example](#)

Applies to

[TRPCBroker](#) component

Declaration

`property Results: Tstrings;`

Description

The Results design-time property 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 following 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:

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

On the client, the Results property contains the following:

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

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > Results Example

Results Example

The following program code demonstrates 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;
```

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > RPCBError Property

RPCBError Property (read-only)

Applies to

[TRPCBroker](#) component

Declaration

```
property RPCBError: String;
```

Description

The RPCBError property is available at run-time only. This read-only property contains the Message property associated with the exception or error that was encountered by the instance of the [TRPCBroker](#) component.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > RPCTimeLimit Property

RPCTimeLimit Property

[Example](#)

Applies to

[TRPCBroker](#) component

Declaration

```
property RPCTimeLimit: Integer;
```

Description

The RPCTimeLimit property is a public integer property that is available at run-time only. 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.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > RPCTimeLimit Example

RPCTimeLimit Example

The following program code demonstrates using the [RPCTimeLimit](#) property:

```
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;
```

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > RPCVersion Property

RPCVersion Property

[Example](#)

Applies to

[TRPCBroker](#) component

Declaration

```
property RPCVersion: String;
```

Description

The RPCVersion design-time property 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 Issues](#) topic.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > RPCVersion Example

RPCVersion Example

In the following example, 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:

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;
```

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
```

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > Server Property

Server Property

[Example](#)

Applies to

[TRPCBroker](#) component

Declaration

```
property Server: String;
```

Description

The Server design-time property 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 following Intranet website:
[redacted](#)

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > Server Example

Server Example

The following program code demonstrates using the [Server](#) property:

```
procedure TForm1.btnConnectClick(Sender: TObject);  
begin  
    brkrRPCBroker1.ListenerPort := 9999;  
    brkrRPCBroker1.Server := 'DHCPSEVER';  
    brkrRPCBroker1.Connected := True;  
end;
```

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > TParamType

TParamType

Unit

[TRPCB](#)

Declaration

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

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 (and if these are to be used, it is recommended that the [TRPCBroker](#) [IsBackwardCompatibleConnection](#) be set to **False** to insure that only a new-style connection can be made).

 **CAUTION:** Use of the [undefined](#) TParamType in applications is *not* supported. It exists for the RPC Broker's *internal use only*.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > ServiceSection Property

ServiceSection Property

Applies to

[TVistaUser](#) class

Declaration

```
property ServiceSection: String;
```

Description

The ServiceSection property is available at run-time only. It holds the user's service section from the NEW PERSON file (#200).

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > ShowErrorMsgs Property

ShowErrorMsgs Property

Applies to

[TRPCBroker](#) component

Declaration

```
property ShowErrorMsgs: TShowErrorMsgs;
```


Description

The ShowErrorMsgs design-time property 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.

The following table lists the possible values:

Value	Meaning
semRaise (default)	This is the default value. The Broker does <i>not</i> 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 <i>not</i> displayed or raised. This requires the application to check the value of the RpcbError property 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 <i>not</i> result in display boxes, etc., as

might be the case with an NT service or Web application.

 **NOTE:** To illustrate the effects of [TRPCBroker](#) properties related to error handling, run the "Error Handling Demo" application (i.e., XWBOOnFail.EXE) located in the ..\BDK\Samples\SilentSignOn p [13] directory.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > Socket Property

Socket Property

[Example](#)

Applies to

[TRPCBroker](#) component

Declaration

```
property Socket: Integer;
```

Description

The Socket property is available at run-time only. It 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](#) 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.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > Socket Example

Socket Example

The following program code populates the [Socket](#) property with the active port on the Vista M Server:

```
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;
```

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > Sorted Property

Sorted Property

[Example](#)

Applies to

[TMult](#) class

Declaration

`property Sorted: Boolean;`

Description

The Sorted design-time property value determines the order of the items in a [TMult](#) variable. If Sorted is **True**, the items are sorted in ascending order of their string subscripts. If Sorted is **False** (default), the items are unsorted, and appears 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.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > Sorted Example

Sorted Example

The following program code 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 application.
2. Drop one memo and one button on the form. Arrange controls as in the figure below (Step #4).
3. Copy the following code 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  
    Mem1.Font.Name := 'Courier';  
    Mem1.ScrollBars := ssVertical;  
    Mem1.Lines.Clear;  
    Mem1.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 = '';

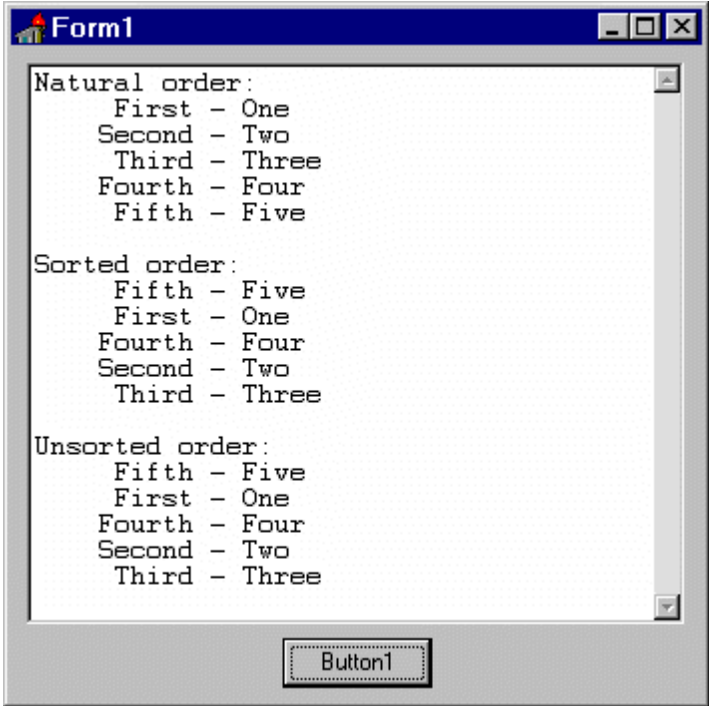
        //list is now sorted alphabetically
        Mult1.Sorted := True;
        Memo1.Lines.Add('');
        Memo1.Lines.Add('Sorted order:');
        //set a starting point
        Subscript := '';
        repeat
            //get next Mult element
            Subscript := Mult1.Order(Subscript, 1);
            //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;

```

4. Run project and click on the button.

Expected output:



You may have to scroll up and down to see all of the output.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > StandardName Property

StandardName Property

Applies to

[TVistaUser](#) class

Declaration

```
property StandardName: String;
```

Description

The StandardName property is available at run-time only. It holds the user's standard name from the NEW PERSON file (#200).

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > TCCOWRPCBroker
Properties (Alpha)

TCCOWRPCBroker Properties (All)

- [TCCOWRPCBroker](#) Component
- [Unique Properties](#) (available with the TCCOWRPCBroker Component)

The following table lists all of the properties available with the [TCCOWRPCBroker](#) component (includes those properties inherited from the [TRPCBroker](#) parent component):

Read-only	Run-time only	Property
▶	▶	BrokerVersion
▶	▶	CCOWLogonIDName
▶	▶	CCOWLogonIDValue
▶	▶	CCOWLogonName
▶	▶	

		<u>CCOWLogonNameValue</u>
▶	▶	<u>CCOWLogonVpid</u>
▶	▶	<u>CCOWLogonVpidValue</u>
		<u>ClearParameters</u>
		<u>ClearResults</u>
		<u>Connected</u>
▶		<u>Contextor</u>
▶	▶	<u>CurrentContext</u>
		<u>DebugMode</u>
		<u>IsBackwardCompatibleConnection</u>
▶	▶	<u>IsNewStyleConnection</u>
		<u>KernelLogIn</u>

		<u>ListenerPort</u>
	▶	<u>LogIn</u>
		<u>OldConnectionOnly.</u>
	▶	<u>OnRPCBFailure</u>
	▶	<u>Param</u>
		<u>RemoteProcedure</u>
		<u>Results</u>
▶	▶	<u>RPCBError</u>
	▶	<u>RPCTimeLimit</u>
		<u>RPCVersion</u>
		<u>Server</u>
		<u>ShowErrorMsgs</u>

	▶	Socket
	▶	User

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > TCCOWRPCBroker
Properties

TCCOWRPCBroker Properties (Unique)

- [TCCOWRPCBroker](#) Component
- [All Properties](#) (available with the TCCOWRPCBroker Component)

The following table lists the unique properties available with the [TCCOWRPCBroker](#) component:

Read-only	Run-time only	Property
▶	▶	CCOWLogonIDName
▶	▶	CCOWLogonIDValue
▶	▶	CCOWLogonName
▶	▶	CCOWLogonNameValue
▶	▶	CCOWLogonVpid

		<u>CCOWLogonVpidValue</u>
		<u>Contextor</u>

i **NOTE:** Since the [TCCOWRPCBroker](#) component is a class derived from the [TRPCBroker](#) component, it contains all of the [properties](#), [methods](#), etc., of its parent.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > Title Property

Title Property

Applies to

[TVistaUser](#) class

Declaration

```
property Title: String;
```

Description

The Title property is available at run-time only. It holds the user's title from the NEW PERSON file (#200).

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > TMulti Properties

TMult Properties

- [TMult](#)
- [Count](#)

- [First](#)

- [Last](#)

- [Sorted](#)

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > TMulti Example

TMult Example

The following program code demonstrates how to store and retrieve elements from a TMult variable:

```
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;
```

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > TParamRecord Properties

TParamRecord Properties

- [TParamRecord](#)
- [Mult](#)
- [PType](#)
- [Value](#)

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > TParamRecord Example

TParamRecord Example

The following program code 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:

```
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;
```

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > TParams Property

TParams Properties

- [TParams](#)
- [Count](#)

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > TParams Example

TParams Example

The following program code demonstrates how a TParams class can be used to save off the [TRPCBroker](#) component parameters and restore them later:

```
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;
```

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > TRPCBroker Properties

TRPCBroker Properties (All)

- [TRPCBroker](#) Component

The following table lists all of the properties available with the [TRPCBroker](#) component:

Read-only	Run-time only	Property
▶	▶	BrokerVersion
		ClearParameters
		ClearResults
		Connected
▶	▶	CurrentContext
		DebugMode
		IsBackwardCompatibleConnection

▶	▶	<u>IsNewStyleConnection</u>
		<u>KernelLogIn</u>
		<u>ListenerPort</u>
	▶	<u>LogIn</u>
		<u>OldConnectionOnly.</u>
	▶	<u>OnRPCBFailure</u>
	▶	<u>Param</u>
		<u>RemoteProcedure</u>
		<u>Results</u>
▶	▶	<u>RPCBError</u>
	▶	<u>RPCTimeLimit</u>
		<u>RPCVersion</u>

		<u>Server</u>
		<u>ShowErrorMsgs</u>
	▶	<u>Socket</u>
	▶	<u>User</u>

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > TSharedBroker Properties (Alpha)

TSharedBroker Properties (All)

- [TSharedBroker](#) Component
- [Unique Properties](#) (available with the TSharedBroker Component)

The following table lists all of the properties available with the [TSharedBroker](#) component (includes those properties inherited from the [TRPCBroker](#) parent component):

Read-only	Run-time only	Property
		AllowShared
▶	▶	BrokerVersion
		ClearParameters
		ClearResults
		Connected

▶	▶	<u>CurrentContext</u>
		<u>DebugMode</u>
		<u>IsBackwardCompatibleConnection</u>
▶	▶	<u>IsNewStyleConnection</u>
		<u>KernelLogIn</u>
		<u>ListenerPort</u>
	▶	<u>LogIn</u>
		<u>OldConnectionOnly.</u>
		<u>OnConnectionDropped</u>
		<u>OnLogout</u>
	▶	<u>OnRPCBFailure</u>
	▶	<u>Param</u>

		<u>RemoteProcedure</u>
		<u>Results</u>
▶	▶	<u>RPCBError</u>
	▶	<u>RPCTimeLimit</u>
		<u>RPCVersion</u>
		<u>Server</u>
		<u>ShowErrorMsgs</u>
	▶	<u>Socket</u>
	▶	<u>User</u>


[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > TSharedBroker Properties

TSharedBroker Properties (Unique)

- [TSharedBroker](#) Component
- [All Properties](#) (available with the TSharedBroker Component)

The following table lists the unique properties available with the [TSharedBroker](#) component:

Read-only	Run-time only	Property
		AllowShared
		OnConnectionDropped
		OnLogout

 **NOTE:** Since [TSharedBroker](#) is a class derived from the [TRPCBroker](#) component, it contains all of the [properties](#), [methods](#), etc., of its parent.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > TSharedRPCBroker Properties (Alpha)

TSharedRPCBroker Properties (All)

- [TSharedRPCBroker](#) Component
- [Unique Properties](#) (available with the TSharedRPCBroker Component)

The following table lists all of the properties available with the [TSharedRPCBroker](#) component (includes those properties inherited from the [TRPCBroker](#) parent component):

Read-only	Run-time only	Property
		AllowShared
▶	▶	BrokerVersion
		ClearParameters
		ClearResults
		Connected

▶	▶	<u>CurrentContext</u>
		<u>DebugMode</u>
		<u>IsBackwardCompatibleConnection</u>
▶	▶	<u>IsNewStyleConnection</u>
		<u>KernelLogIn</u>
		<u>ListenerPort</u>
	▶	<u>LogIn</u>
		<u>OldConnectionOnly.</u>
		<u>OnConnectionDropped</u>
		<u>OnLogout</u>
	▶	<u>OnRPCBFailure</u>
	▶	<u>Param</u>

		<u>RemoteProcedure</u>
		<u>Results</u>
▶	▶	<u>RPCBError</u>
	▶	<u>RPCTimeLimit</u>
		<u>RPCVersion</u>
		<u>Server</u>
		<u>ShowErrorMsgs</u>
	▶	<u>Socket</u>
	▶	<u>User</u>


[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > TSharedRPCBroker
Properties

TSharedRPCBroker Properties (Unique)

- [TSharedRPCBroker](#) Component
- [All Properties](#) (available with the TSharedRPCBroker Component)

The following table lists the unique properties available with the [TSharedRPCBroker](#) component:

Read-only	Run-time only	Property
		AllowShared
		OnConnectionDropped
		OnLogout

 **NOTE:** Since the [TSharedRPCBroker](#) component is a class derived from the [TRPCBroker](#) component, it contains all of the [properties](#), [methods](#), etc., of its parent.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > TVistaLogin Properties

TVistaLogin Properties

- [TVistaLogin Class](#)

The following table lists all of the properties available with the [TVistaLogin Class](#):

Read-only	Run-time only	Property
	▶	AccessCode
	▶	Division
▶	▶	DivList
	▶	DomainName
	▶	DUZ
	▶	ErrorText
	▶	IsProductionAccount

	▶	LoginHandle
	▶	Mode
	▶	MultiDivision
	▶	OnFailedLogin
	▶	PromptDivision
	▶	VerifyCode

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > TVistaUser Properties

TVistaUser Properties

- [TVistaUser Class](#)

The following table lists all of the properties available with the [TVistaUser Class](#):

Read-only	Run-time only	Property
	▶	Division
	▶	DTime
	▶	DUZ
	▶	Language
	▶	Name
	▶	ServiceSection
	▶	StandardName

	▶	Title
	▶	VerifyCodeChngd
	▶	Vpid

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > TXWBRichEdit Property

TXWBRichEdit Properties

- [TXWBRichEdit](#)
- [URLDetect](#)

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > URLLDetect Property

URLDetect Property

Applies to

[TXWBRichEdit](#) component

Declaration

```
property URLDetect: Boolean;
```


Description

The URLDetect design-time property 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. If the property is **False** (the default), URLs appear as normal text and are *not* active.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > User Property

User Property

[Example](#)

Applies to

[TRPCBroker](#) component

Declaration

`property` User: [TVistaUser](#);

Description

The User property is available at run-time only. This instance of the [TVistaUser](#) 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.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > Value Property

Value Property

[Example](#)

Applies to

[TParamRecord](#) class

Declaration

```
property Value: String;
```

Description

The Value design-time property is used to pass either a single string or a single variable reference to the VistA M Server, depending on the [PType](#).

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > Value Example

Value Example

The following program code demonstrates a couple of different uses of the [Value](#) property. Remember that each Param[x] element is really a [TParamRecord](#)-type class.

```
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;
```

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > VerifyCode Property

VerifyCode Property

[Example](#)

Applies to

[TVistaLogin](#) class

Declaration

```
property VerifyCode: String;
```

Description

The VerifyCode property is available at run-time only. It holds the Verify code for ImAVCodes 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 more information on Verify codes, see the "Part 1: Sign-On/Security" section in the *Kernel Systems Management Guide*.



REF: For a demonstration using the ImAVCodes, run the ImAVCodes_Demo.EXE located in the ..\BDK32\Samples\SilentSignOn directory.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > VerifyCodeChngd Property

VerifyCodeChngd Property

Applies to

[TVistaUser](#) class

Declaration

```
property VerifyCodeChngd: Boolean;
```

Description

The VerifyCodeChngd property is available at run-time only. It indicates whether or not the user's Verify code has changed.

[Home](#) > [RPC Broker Classes, Components, Units, Methods, and Properties](#) > [Properties and Examples](#) > Vpid Property

Vpid Property

Applies to

[TVistaUser](#) class

Declaration

```
property Vpid: String;
```

Description

The Vpid property is available at run-time only. It returns the Department of Veterans Affairs Personal Identification (VPID) value for the current user from the NEW PERSON file (#200), if the facility has already been enumerated. If the facility has *not* been enumerated, the value returned is a null string.

[Home](#) > [Remote Procedure Calls \(RPCs\)](#) > RPC Overview

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](#) file (#8994).

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 Remote Procedure File:
 - [REMOTE PROCEDURE FILE](#)
 - [RPC Entry in the Remote Procedure File](#)
 - [RPC Version in the Remote Procedure File](#)
 - [Blocking an RPC in the Remote Procedure File](#)
 - [Cleanup after RPC Execution](#)
 - [Documenting RPCs](#)
- Executing RPCs from Clients:

- [How to Execute an RPC from a Client](#)
- [RPC Security: How to Register an RPC](#)
- [RPC Limits](#)
- [RPC Time Limits](#)
- [Maximum Size of Data](#)
- [Maximum Size of Array.](#)
- [RPC Broker Example \(32-Bit\)](#)

[Home](#) > [Remote Procedure Calls \(RPCs\)](#) > What makes a good
RPC?

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).

[Home](#) > [Remote Procedure Calls \(RPCs\)](#) > Using an Existing M API

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 file \(#8994\)](#).



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

[Home](#) > [Remote Procedure Calls \(RPCs\)](#) > Creating RPCs

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 are called by the RPC.
- Add the RPC entry that invokes the M entry point, in the [REMOTE PROCEDURE file \(#8994\)](#).

[Home](#) > [Remote Procedure Calls \(RPCs\)](#) > [M Entry Point for an RPC](#) > Relationship Between an M Entry Point and an RPC

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 Example](#) shows you how to do this on the client and server sides.

[Home](#) > [Remote Procedure Calls \(RPCs\)](#) > [M Entry Point for an RPC](#) > First Input Parameter (Required)

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.

[Home](#) > [Remote Procedure Calls \(RPCs\)](#) > [M Entry Point for an RPC](#)
> Return Value Types

Return Value Types

There are five RETURN VALUE TYPEs for RPCs as shown in the table below. 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 following RPC settings, combined with your M entry point, determine how data is returned to your client application:

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:	No effect	Value of parameter, in

	<pre>TAG (RESULT) ; S RESULT="XWBUSER, ONE" Q</pre>		Results[0] .
Array	<p>Set an array of strings into the return parameter, each subscripted one level descendant. For example:</p> <pre>TAG (RESULT) ; S RESULT (1) =" ONE" S RESULT (2) =" TWO" Q</pre> <p>If your array is large, consider using the GLOBAL ARRAY return value type, to avoid memory allocation errors.</p>	No effect	Array values, each in a Results item.
Word Processing	<p>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.</p>	True False	<p>Array values, each in a Results item.</p> <p>Array values, all concatenated into Results[0].</p>
Global Array	<p>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.</p>	True False	<p>Array values, each in a Results item.</p> <p>Array values, all concatenated</p>

into
[Results\[0\]](#).

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 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 (8 98 9 . 3 , 1 , " INTRO" )
; this node not needed
K ^TMP (" A6A" , $J , 0)
S
RESULT=$NA (^TMP (" A6A" , $J) )
Q
```

Global Instance	<p>Set the return parameter to a closed global reference.</p> <p>For example the following code returns the whole 0th node from the NEW PERSON file (#200) for the current user:</p> <pre> TAG (RESULT) ; S RESULT=\$NA (^VA (200, DUZ, 0)) Q </pre>	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.

[Home](#) > [Remote Procedure Calls \(RPCs\)](#) > [M Entry Point for an RPC](#)
> Input Parameters (Optional)

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 your entry point) in one of the following three [Param](#) 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.
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.

[Home](#) > [Remote Procedure Calls \(RPCs\)](#) > [M Entry Point for an RPC](#) > Examples

Examples

The following two examples illustrate sample M code that could be used in simple RPCs:

1. This example takes two numbers and returns their sum:

```
SUM(RESULT,A,B) ;add two numbers
  S RESULT=A+B
  Q
```

2. This example 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
```

[Home](#) > [Remote Procedure Calls \(RPCs\)](#) > [RPC Entry in the Remote Procedure File](#) > REMOTE PROCEDURE File


REMOTE PROCEDURE File

The RPC Broker consists of a single global that stores the REMOTE PROCEDURE file:

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 subfield (#19.05) of the OPTION File (#19) points to RPC field (#.01) of the REMOTE PROCEDURE file (#8994).

[Home](#) > [Remote Procedure Calls \(RPCs\)](#) > [RPC Entry in the Remote Procedure File](#) > RPC Entry in the Remote Procedure File

RPC Entry in the Remote Procedure File

After the M code is complete, you need to add the RPC to the [REMOTE PROCEDURE file \(#8994\)](#). The following fields in the REMOTE PROCEDURE file (#8994) are key to the correct operation of an RPC:

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	<p>This can be one of five types:</p> <ul style="list-style-type: none">• SINGLE VALUE• ARRAY• WORD PROCESSING• GLOBAL ARRAY• GLOBAL INSTANCE <p>This setting controls how the Broker processes an RPC's return parameter.</p>

[Home](#) > [Remote Procedure Calls \(RPCs\)](#) > [RPC Entry in the Remote Procedure File](#) > RPC Version in the Remote Procedure File

RPC Version in the Remote Procedure File

The VERSION field of the [REMOTE PROCEDURE file \(#8994\)](#) 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.

[Home](#) > [Remote Procedure Calls \(RPCs\)](#) > [RPC Entry in the Remote Procedure File](#) > Blocking an RPC in the Remote Procedure File

Blocking an RPC in the Remote Procedure File

The INACTIVE field of the [REMOTE PROCEDURE file \(#8994\)](#) 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 a package developer and exported by KIDS or by a site manager using VA FileMan.

Value in INACTIVE field

- 1 = Completely unusable
- 2 = Unusable locally
- 3 = Unusable remotely

[Home](#) > [Remote Procedure Calls \(RPCs\)](#) > [RPC Entry in the Remote Procedure File](#) > Cleanup after RPC Execution

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](#), GLOBAL ARRAY, where the application RPC returns a closed form global reference, for example:

```
^TMP (" EKG" , 220333551)
```

The Broker kills the data for the global reference for this type of [RPC](#) at the end of RPC execution.

[Home](#) > [Remote Procedure Calls \(RPCs\)](#) > [RPC Entry in the Remote Procedure File](#) > Documenting RPCs

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 field (#1) in the [REMOTE PROCEDURE file \(#8994\)](#) for those RPCs installed on your system. This gives you the capability of generating a catalogue of RPCs from File #8994.

Delphi Component Library and Sample RPCs

In the future, an Enterprise library of object components, classes, and remote procedure calls 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 components, classes, and RPCs 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.

[Home](#) > [Remote Procedure Calls \(RPCs\)](#) > [Executing RPCs from Clients](#) > How to Execute an RPC from a Client Application

How to Execute an RPC from a Client

1. If your RPC has any input parameters beyond the mandatory first parameter, set a [Param](#) node in the [TRPCBroker](#)'s [Param](#) property for each. For each input parameter, set the following sub-properties:

- [Value](#)
- [PType](#) (literal, list, or reference).

If the parameter's PType is list, however, instead of specifying a value, instead set a list of values in the [Mult](#) property.

Here is an example of some settings of the [Param](#) property:

```
brkrRPCBroker1.Param[ 0 ].Value := '03/31/14';
brkrRPCBroker1.Param[ 0 ].PType := literal;
brkrRPCBroker1.Param[ 1 ].Mult[ '"NAME"' ] := 'XWBUSER,
ONE';
brkrRPCBroker1.param[ 1 ].mult[ '"ssn"' ] := "000-45-678 9"
;/pre="">
brkrRPCBroker1.Param[ 1 ].PType := list;
```

2. Set the [TRPCBroker](#)'s RemoteProcedure property to the name of the RPC to execute:

```
brkrRPCBroker1.RemoteProcedure:='A6A LIST'
```

3. Invoke the [Call](#) method of the [TRPCBroker](#) component 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:

```
try
    brkrRPCBroker1.Call;
except
    On EBrokerError do
        ShowMessage('A problem was encountered
communicating with the server.');
```

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.

i **NOTE:** You can also use the [lstCall](#) and [strCall](#) methods to execute an RPC. The main difference between these methods and the [Call](#) method is that [lstCall](#) and [strCall](#) do *not* use the [Results](#) property, instead returning results into a location you specify.

[Home](#) > [Remote Procedure Calls \(RPCs\)](#) > [Executing RPCs from Clients](#) > RPC Security: How to Register an RPC

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 file (#19) for your application.



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 following fields can be set up for each RPC in your option:

Field Name (#)	Entry	Description
----------------	-------	-------------

RPC (#.01)	Required	This field is used to enter a pointer to the REMOTE PROCEDURE file (#8994) . This field links the remote procedure call in the REMOTE PROCEDURE file (#8994) 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 file (#19.1).
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.

- 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.
- 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;

```

If the [CreateContext](#) method returns **True**, only those RPCs designated in the RPC multiple of your application option is permitted to run.

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).

5. End-users of your application must have the "**B**"-type option assigned to them on one of their menus, in order for [CreateContext](#) to return **True**. This allows system managers to control access to client applications.

[Home](#) > [Remote Procedure Calls \(RPCs\)](#) > [Executing RPCs from Clients](#) > RPC Limits

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 limit](#).

[Home](#) > [Remote Procedure Calls \(RPCs\)](#) > [Executing RPCs from Clients](#) > RPC Time Limits

RPC Time Limits

A public READ/WRITE property (i.e., [RPCTimeLimit](#)) 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.

[Home](#) > [Remote Procedure Calls \(RPCs\)](#) > [Executing RPCs from Clients](#) > Maximum Size of Data

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.

[Home](#) > [Remote Procedure Calls \(RPCs\)](#) > [Executing RPCs from Clients](#) > Maximum Number of Parameters

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.

[Home](#) > [Remote Procedure Calls \(RPCs\)](#) > [Executing RPCs from Clients](#) > Maximum Size of Array

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.


[Home](#) > [Remote Procedure Calls \(RPCs\)](#) > [Executing RPCs from Clients](#) > RPC Broker Example (32-Bit)

RPC Broker Example (32-Bit)

The RPC Broker Example sample application provided with the BDK (i.e., BrokerExample.EXE, located in the ..\BDK32\Samples\BrokerEx directory) demonstrates the basic features of developing RPC Broker applications, including:

- Connecting to a Vista M Server.
- Creating an application [context](#).
- Using the [GetServerInfo](#) function.
- Displaying the Vista [splash screen](#).
- Setting the TRPCBroker.[Param](#) property for each [Param PType](#) (literal, reference, list).
- Calling RPCs with the [Call](#) method.
- Calling RPCs with the [IstCall](#) and [strCall](#) methods.

The client source code files for the BrokerExample application are located in the ..\BDK32\Samples\BrokerEx directory.

 **NOTE:** Initially, use Delphi to compile the BrokerExample.DPR into an executable.

[Home](#) > [Other RPC Broker APIs](#) > Overview: Other APIs

Overview: Other RPC Broker APIs

The Broker Development Kit (BDK) provides the following development APIs in addition to the RPC Broker components:

- [Encryption Functions](#)
- [M Emulation Functions](#)
- [XWB ARE RPCS AVAILABLE](#)
- [XWB IS RPC AVAILABLE](#)
- [GetServerInfo Function](#)
- [Vista Splash Screen Procedures](#)
- [XWB GET VARIABLE VALUE RPC](#)
- [Running RPCs on a Remote Server](#)
- [Deferred RPCs](#)

The RPC Broker software provides the following application program interfaces (APIs) on the Vista M Server for use in RPC code:

- [\\$\\$BROKER^XWBLIB](#) (Determine if running from a Broker call)
- [\\$\\$RTRNFMT^XWBLIB](#) (Change return format of RPC)

[Home](#) > [Other RPC Broker APIs](#) > HANDLE

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.

[Home](#) > [Other RPC Broker APIs](#) > [Functions, Methods, and Procedures](#)
> \$\$BROKER^XWBLIB

\$\$BROKER^XWBLIB

Use this 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

None

Output

Return Value	<p>Results:</p> <ul style="list-style-type: none">• 1—If the current process is being executed by the Broker.• 0—If the current process is <i>not</i> being executed by the Broker.
--------------	--

Example

```
I $$BROKER^XWBLIB D  
. ; broker-specific code
```

[Home](#) > [Other RPC Broker APIs](#) > [Functions, Methods, and Procedures](#)
> \$\$RTRNFMT^XWBLIB

\$\$RTRNFMT^XWBLIB

Use this 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

type	<p>Set this to the RETURN VALUE TYPE to change the RPC's setting to. Set it to one of the following numeric or free text values:</p> <table><thead><tr><th data-bbox="464 722 613 751">numeric</th><th data-bbox="683 722 833 751">free text</th></tr></thead><tbody><tr><td data-bbox="464 856 483 886">1</td><td data-bbox="630 856 769 932">SINGLE VALUE</td></tr><tr><td data-bbox="464 1037 483 1066">2</td><td data-bbox="630 1037 756 1066">ARRAY</td></tr><tr><td data-bbox="464 1171 483 1201">3</td><td data-bbox="630 1171 878 1247">WORD PROCESSING</td></tr><tr><td data-bbox="464 1352 483 1381">4</td><td data-bbox="630 1352 781 1428">GLOBAL ARRAY</td></tr><tr><td data-bbox="464 1533 483 1562">5</td><td data-bbox="630 1533 870 1608">FOR GLOBAL INSTANCE</td></tr></tbody></table>	numeric	free text	1	SINGLE VALUE	2	ARRAY	3	WORD PROCESSING	4	GLOBAL ARRAY	5	FOR GLOBAL INSTANCE
numeric	free text												
1	SINGLE VALUE												
2	ARRAY												
3	WORD PROCESSING												
4	GLOBAL ARRAY												
5	FOR GLOBAL INSTANCE												
wrap	<ul style="list-style-type: none">• Set to 1 to set the RPC's WORD WRAP ON setting to												

True.

- Set to **0** to set the RPC's WORD WRAP ON setting to **False**.

Output

Return Value	<p>Returns:</p> <ul style="list-style-type: none">• 0—If the return value type could <i>not</i> be changed.• numeric code—Representing the return value type to which the RPC is changed.
--------------	--

Example

```
I '$$RTRNFMT^XWBLIB("ARRAY",1) D
```

.; branch to code if cannot change RPC type

[Home](#) > [Other RPC Broker APIs](#) > [Functions, Methods, and Procedures](#) > XWB GET VARIABLE VALUE

XWB GET VARIABLE VALUE

[Example](#)

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](#) function).

[Home](#) > [Other RPC Broker APIs](#) > [Functions, Methods, and Procedures](#) > XWB GET VARIABLE VALUE Example

XWB GET VARIABLE VALUE

Example

The following is example of the XWB GET VARIABLE VALUE RPC:

```
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 ] );
```

[Home](#) > [Other RPC Broker APIs](#) > [Functions, Methods, and Procedures](#) > M Emulation Functions

M Emulation Functions

[Examples](#)

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;
```

Translate Function

The Translate function is a scaled down Pascal version of M's \$TRANSLATE function. It is declared in MFUNSTR.PAS.

```
function Translate(passedString, identifier, associator:  
string): string;
```

[Home](#) > [Other RPC Broker APIs](#) > [Functions, Methods, and Procedures](#) > M Emulation Examples

M Emulation Examples

Piece Function

```
Piece3Str:=piece('123^456^789','^',3);
```

Translate Function

```
hiStr:=translate('HI','ABCDEFGH','abcdefghi');
```

[Home](#) > [Other RPC Broker APIs](#) > [Functions, Methods, and Procedures](#) > Encryption Functions

Encryption Functions

Kernel and the RPC Broker provide encryption functions that can be used to encrypt messages sent between the client and the server.

In Delphi

Include HASH 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;
```

On the VistA M Server

To encrypt:

```
KRN,KDE>S CIPHER=$$ENCRYP^XUSRB1("Hello world!") W CIPHER  
/U'11TG~TV1&f-
```

To decrypt:

```
KRN,KDE>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.

[Home](#) > [Other RPC Broker APIs](#) > [Functions, Methods, and Procedures](#) > CheckCmdLine Function

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;
```

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.

Result

The return value indicates whether the information on the command line was sufficient to connect the RPCBroker instance to the specified [Server/ListenerPort](#).

- **True**—Broker is connected to the VistA M Server.
- **False**—Broker is *not* connected to the VistA M Server.

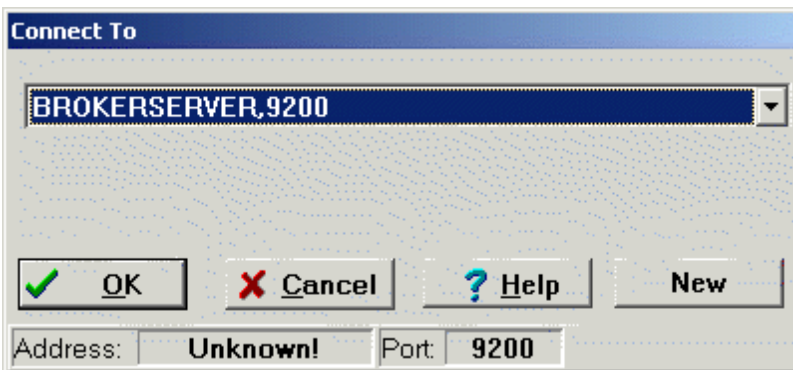
[Home](#) > [Other RPC Broker APIs](#) > [Functions, Methods, and Procedures](#) > GetServerInfo Function

GetServerInfo Function

[Example](#)

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](#) properties 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:



Syntax:


```
function GetServerInfo(var Server, Port: string): integer;
```



NOTE: The unit is RpcConf1.

The GetServerInfo function handles the following scenarios:

- If there are no values for server and port in the [Windows Registry](#), GetServerInfo does *not* display its dialogue window, and the automatic default values returned are BROKERSERVER/9999. GetServerInfo returns **mrOK**.
- If exactly one server and port entry is defined in the [Windows Registry](#), GetServerInfo does *not* display its dialogue window. The values in the single 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 [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 clicks **Cancel**. However, if a user selects an entry and clicks **OK**, the server and port parameters are changed and returned to the calling application. GetServerInfo returns **mrOK** if the user clicked **OK**, or **mrCancel** if the user clicked **Cancel**.


 **REF:** For a demonstration using the Broker and GetServerInfo function, run the RPC Broker Example (i.e., [BrokerExample.EXE](#)) located in the ..\BDK32\Samples\BrokerEx directory.

[Home](#) > [Other RPC Broker APIs](#) > [Functions, Methods, and Procedures](#) > GetServerInfo Example

GetServerInfo Example

The following is example of the GetServerInfo function:

```
procedure TForm1.btnConnectClick(Sender: TObject);
var
    strServer, strPort: string;
begin
    if GetServerInfo(strServer, strPort) <> mrCancel then
begin {getsvrinfo begin}
    brkrRPCBroker1.Server := strServer;
    brkrRPCBroker1.ListenerPort := StrToInt(strPort);
    brkrRPCBroker1.Connected := True;
    {getsvrinfo end}
end;
end;
```

 **REF:** For a demonstration using the Broker and GetServerInfo function, run the RPC Broker Example (i.e., [BrokerExample.EXE](#)) located in the `..\BDK32\Samples\BrokerEx` directory.

[Home](#) > [Other RPC Broker APIs](#) > [Functions, Methods, and Procedures](#) > GetServerIP Function

GetServerIP Function

[Example](#)

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;
```

[Home](#) > [Other RPC Broker APIs](#) > [Functions, Methods, and Procedures](#) > GetServerIP Example

GetServerIP Example

The following is example of the GetServerIP function:

```
// 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 := 'xxxxx.xxx.xxx.xxx';
    edtIPAddress.Text := GetServerIP(edtInput.Text);
    // For xxxxx.xx.xxx returns '999.999.999'
    garbage returns 'Unknown!'
end;
```

[Home](#) > [Other RPC Broker APIs](#) > [Functions, Methods, and Procedures](#) > ChangeVerify Function

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;
```

Argument

Argument	Description
RPCBroker	The Broker instance for the account on which the Verify code is to be changed.

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.

[Home](#) > [Other RPC Broker APIs](#) > [Functions, Methods, and Procedures](#) > SilentChangeVerify Function

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;
```

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 <i>not</i> changed (if the result value is False).

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.

[Home](#) > [Other RPC Broker APIs](#) > [Functions, Methods, and Procedures](#) > StartProgSLogin Method

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](#) 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);
```

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 <i>not</i> nil, then the VistA M Server address, ListenerPort, Division, and ApplicationToken is added to the command line and the application launched.
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 combination. If the application to be launched is <i>not</i> related to the TRPCBroker, then this argument should be set to nil.

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):

```
MyCommand := 'C:\Program Files\VISTA\Test1\Sample1.exe  
xval=MyData yval=YourData';  
StartProgSLogin(MyCommand, RPCBroker1);
```

This results in the following command line being used to launch the application:

```
C:\Program Files\VISTA\Test1\Sample1.exe xval=MyData  
yval=YourData s=ServerName p=9999 d=Division  
h=AppHandleValue
```

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:

```
MyCommand := 'Notepad logtable.txt';  
StartProgSLogin(MyCommand, nil);
```

[Home](#) > [Other RPC Broker APIs](#) > [Functions, Methods, and Procedures](#) > VistA Splash Screen Procedures

VistA Splash Screen Procedures

[Example](#)

Two procedures in SpIVista.PAS unit are provided 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).

Using a Splash Screen in an Application

To use the VistA Splash Screen in an application:

1. Open the application's project (.DPR) file. 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 below:



i **REF:** For a demonstration using the VistA Splash Screen, run the RPC Broker Example (i.e., [BrokerExample.EXE](#)) located in the ..\BDK32\Samples\BrokerEx directory.

[Home](#) > [Other RPC Broker APIs](#) > [Functions, Methods, and Procedures](#) > VistA Splash Screen Example


VistA Splash Screen Example

The following is example of displaying the VistA Splash Screen in an application:

```
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 (i.e., [BrokerExample.EXE](#)) located in the ..\BDK32\Samples\BrokerEx directory.


[Home](#) > [Other RPC Broker APIs](#) > [Running RPCs on a Remote Server](#) > Overview: Running RPCs on a Remote Server

Overview: Running RPCs on a Remote Server

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 desired remote VistA M Server.
- The desired remote RPC.
- Any parameters for the remote RPC.

The RPC Broker on the local VistA M Server uses VistA 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.

Using Direct RPCs


RPC	Description
<u>XWB DIRECT RPC</u>	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.

Using 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 may 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 <i>must</i> be used to clear the data under the HANDLE in the ^XTMP Global.

XWB
DEFERRED
CLEARALL

Applications using XWB REMOTE XWB should use XWB DEFERRED CLEARALL on application close to clear all known data associated with the job on the VistA M Server.

 **NOTE:** XWB DIRECT RPC and XWB REMOTE RPC are available only on a controlled subscription basis.

[Home](#) > [Other RPC Broker APIs](#) > [Running RPCs on a Remote Server](#) > Checking RPC Availability on a Remote Server

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.

[Home](#) > [Other RPC Broker APIs](#) > [Running RPCs on a Remote Server](#) > XWB ARE RPCS AVAILABLE

XWB ARE RPCS AVAILABLE

[Example](#)

[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.

Output	Description
RETURN VALUE	A 0-based array. The index corresponds to the index of the RPC in the RPC Input Parameter: <ul style="list-style-type: none">• 1—RPC Available.• 0—RPC Not available.
RUN CONTEXT	Pass the run context (local or remote) of the RPC in Param [0].Value, and the type (literal) in Param [0].PType. Possible values:

PARAMETER
(Optional)

- **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](#).

**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`

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](#). If the version number passed is less than or equal to the number in the VERSION field, the RPC is marked available.



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](#) function).

[Home](#) > [Other RPC Broker APIs](#) > [Running RPCs on a Remote Server](#) > XWB ARE RPCS AVAILABLE Example

XWB ARE RPCS AVAILABLE

Example

The following is example of the XWB ARE RPCS AVAILABLE RPC:

```
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
```

[Home](#) > [Other RPC Broker APIs](#) > [Running RPCs on a Remote Server](#) > XWB IS RPC AVAILABLE


XWB IS RPC AVAILABLE

[Example](#)

[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.

Parameter/Output	Description
RETURN VALUE	Boolean: <ul style="list-style-type: none">• 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.

<p>RUN CONTEXT PARAMETER (Optional)</p>	<p>Pass the run context (local or remote) of the RPC in Param[1].Value, and the type (literal) in Param[1].PType. Possible values:</p> <ul style="list-style-type: none"> • 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) <p>If this parameter is not sent, the RPC is checked for both local and remote and both run contexts <i>must</i> be available for the return to be "1" (RPC Available). The check is done against the INACTIVE field in the REMOTE PROCEDURE file.</p>
<p>VERSION NUMBER PARAMETER (Optional)</p>	<p>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. If the version number passed is less than or equal to the number in the VERSION field, the RPC is marked available.</p> <p> NOTE: If the VERSION field is null, the check fails for a numeric value in this</p>

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](#) function).

[Home](#) > [Other RPC Broker APIs](#) > [Running RPCs on a Remote Server](#) > XWB IS RPC AVAILABLE Example

XWB IS RPC AVAILABLE

Example

The following is example of the XWB IS RPC AVAILABLE RPC:

```
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
```

[Home](#) > [Other RPC Broker APIs](#) > [Running RPCs on a Remote Server](#) > XWB DIRECT RPC

XWB DIRECT RPC

[Example](#)

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.



REF: For a comparison of the two methods, see the "[Options For Running RPCs on a Remote Server](#)" topic.

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	Pass minimum version of RPC to be run in

PARAMETER (Optional)	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 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".



NOTE: XWB DIRECT RPC is available only on a controlled subscription basis.

[Home](#) > [Other RPC Broker APIs](#) > [Running RPCs on a Remote Server](#) > XWB DIRECT RPC Example

XWB DIRECT RPC Example

The following is example of the XWB DIRECT RPC:

```
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[ ]
```

[Home](#) > [Other RPC Broker APIs](#) > [Running RPCs on a Remote Server](#) > XWB REMOTE RPC

XWB REMOTE RPC

[Example](#)

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 "[Options For Running RPCs on a Remote Server](#)" topic.

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](#)

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 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".



NOTE: XWB REMOTE RPC is available only on a controlled subscription basis.

[Home](#) > [Other RPC Broker APIs](#) > [Running RPCs on a Remote Server](#) > XWB REMOTE RPC Example

XWB REMOTE RPC Example

The following is example of the XWB REMOTE RPC:

```
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.

[Home](#) > [Other RPC Broker APIs](#) > [Running RPCs on a Remote Server](#) > XWB REMOTE STATUS CHECK

XWB REMOTE STATUS CHECK

[Example](#)

Use this RPC to check for results of [XWB REMOTE RPC](#). Periodically call this RPC and pass the [HANDLE](#) returned by XWB REMOTE RPC.

Output	Description
RETURN VALUE	<p>The return value is always an array. The first node of the array is equal to one of the following values:</p> <ul style="list-style-type: none"><li data-bbox="586 1234 1409 1318">• "-1^Bad Handle"—An invalid handle has been passed.<li data-bbox="586 1413 1341 1497">• "0^New"—The request has been sent via VistA HL7.<li data-bbox="586 1591 1362 1675">• "0^Running"—VistA HL7 indicates that the message is being processed.<li data-bbox="586 1770 1406 1890">• "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 package.

[Home](#) > [Other RPC Broker APIs](#) > [Running RPCs on a Remote Server](#) > XWB REMOTE STATUS CHECK Example

XWB REMOTE STATUS CHECK

Example

The following is example of the XWB REMOTE STATUS CHECK RPC:

```
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
```

[Home](#) > [Other RPC Broker APIs](#) > [Running RPCs on a Remote Server](#) > XWB REMOTE GETDATA

XWB REMOTE GETDATA

[Example](#)

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.

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".

[Home](#) > [Other RPC Broker APIs](#) > [Running RPCs on a Remote Server](#) > XWB REMOTE GETDATA Example

XWB REMOTE GETDATA

Example

The following is example of the XWB REMOTE GETDATA RPC:

```
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
```

[Home](#) > [Other RPC Broker APIs](#) > [Running RPCs on a Remote Server](#) > XWB REMOTE CLEAR

XWB REMOTE CLEAR

[Example](#)

This RPC is used to clear the data created by a remote RPC under the [HANDLE](#) in the ^XTMP. Pass the HANDLE returned by [XWB REMOTE RPC](#).

Output	Description
RETURN VALUE	An array. The first node in the array is equal to 1.

[Home](#) > [Other RPC Broker APIs](#) > [Running RPCs on a Remote Server](#) > XWB REMOTE CLEAR Example

XWB REMOTE CLEAR Example

The following is example of the XWB REMOTE CLEAR RPC:

```
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;
```

[Home](#) > [Other RPC Broker APIs](#) > [Deferred RPCs](#) > Overview:
Deferred RPCs

Overview: Deferred RPCs

Remote Procedure Calls can now be run in the background with [XWB DEFERRED RPC](#).

Using 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 <i>must</i> be used to clear the data under the HANDLE in the ^XTMP Global.

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.

[Home](#) > [Other RPC Broker APIs](#) > [Deferred RPCs](#) > XWB
DEFERRED RPC

XWB DEFERRED RPC

Example

Use this RPC to request that an RPC be run in deferred mode. 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 DEFERRED STATUS](#)
- [XWB DEFERRED GETDATA](#)
- [XWB DEFERRED CLEAR](#)

Parameter/Output	Description
RPC PARAMETER	Pass the name of the RPC to be run in <u>Param</u> [0].Value, and the type (literal) in <u>Param</u> [0].PType.
RPC VERSION PARAMETER (Optional)	Pass minimum version of RPC to be run in <u>Param</u> [1].Value, and the type (literal) in <u>Param</u> [1].PType. It is checked against the value in the <u>VERSION field of the REMOTE PROCEDURE file</u> 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".

[Home](#) > [Other RPC Broker APIs](#) > [Deferred RPCs](#) > XWB
DEFERRED RPC Example

XWB DEFERRED RPC Example

The following is example of the XWB DEFERRED RPC:

```
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.

[Home](#) > [Other RPC Broker APIs](#) > [Deferred RPCs](#) > XWB
DEFERRED STATUS

XWB DEFERRED STATUS

[Example](#)

Use this RPC to check for results of [XWB DEFERRED RPC](#). Periodically, call this RPC and pass the [HANDLE](#) returned by XWB REMOTE RPC.

Output	Description
RETURN VALUE	<p>The return value is always an array. The first node of the array is equal to one of the following values:</p> <ul style="list-style-type: none"><li data-bbox="586 1234 1422 1325">• "-1^Bad Handle"—An invalid handle has been passed.<li data-bbox="586 1413 1422 1503">• "0^New"—The request has been sent via VistA HL7.<li data-bbox="586 1591 1422 1682">• "0^Running"—VistA HL7 indicates that the message is being processed.<li data-bbox="586 1770 1422 1892">• "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.

[Home](#) > [Other RPC Broker APIs](#) > [Deferred RPCs](#) > XWB
DEFERRED STATUS Example

XWB DEFERRED STATUS

Example

The following is example of the XWB DEFERRED STATUS RPC:

```
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
```

[Home](#) > [Other RPC Broker APIs](#) > [Deferred RPCs](#) > XWB
DEFERRED GETDATA

XWB DEFERRED GETDATA

[Example](#)

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.

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".

[Home](#) > [Other RPC Broker APIs](#) > [Deferred RPCs](#) > XWB
DEFERRED GETDATA Example

XWB DEFERRED GETDATA Example

The following is example of the XWB DEFERRED GETDATA RPC:

```
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
```


[Home](#) > [Other RPC Broker APIs](#) > [Deferred RPCs](#) > XWB
DEFERRED CLEAR

XWB DEFERRED CLEAR

[Example](#)

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](#).

Output	Description
RETURN VALUE	An array. The first node in the array is equal to 1 .

[Home](#) > [Other RPC Broker APIs](#) > [Deferred RPCs](#) > XWB
DEFERRED CLEAR Example

XWB DEFERRED CLEAR

Example

The following is example of the XWB DEFERRED CLEAR RPC:

```
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;
```

[Home](#) > [Other RPC Broker APIs](#) > [Deferred RPCs](#) > XWB
DEFERRED CLEARALL

XWB DEFERRED CLEARALL

[Example](#)

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.

Output	Description
RETURN VALUE	An array. The first node in the array is equal to 1 .

[Home](#) > [Other RPC Broker APIs](#) > [Deferred RPCs](#) > XWB
DEFERRED CLEARALL Example

XWB DEFERRED CLEARALL Example

The following is example of the XWB DEFERRED CLEARALL RPC:

```
brkrRPCBroker1.RemoteProcedure := 'XWB DEFERRED CLEAR';  
try  
    brkrRPCBroker1.Call;  
except  
    On EBrokerError do  
        ShowMessage('Connection to server could not be  
established! ');  
end;
```


[Home](#) > [Debugging and Troubleshooting](#) > Overview: Debugging and Troubleshooting

Overview: Debugging and Troubleshooting

The Broker Development Kit (BDK) provides facilities for debugging and troubleshooting your Vista Graphical User Interface (GUI) applications.

- [How to Debug Your Application](#)
- [Identifying the Listener Process](#)
- [Identifying the Handler Process](#)
- [RPC Error Trapping](#)
- [Testing Your RPCBroker Connection](#)
- [Memory Leaks](#)
- [Client Timeout and Buffer Clearing](#)
- [Broker Errors](#)



REF: For commonly asked questions see the RPC Broker FAQs at the following Intranet website:
redacted

[Home](#) > [Debugging and Troubleshooting](#) > How to Debug the Application

How to Debug the Application

Beside the normal debugging facilities provided by Delphi, you can also invoke a debug mode so that you can step through your code on the client side and your RPC code on the VistA M Server side simultaneously.

To invoke a debug mode, do the following:

1. On the client side, set the [DebugMode](#) property on the [TRPCBroker](#) component to **True**. When the TRPCBroker component connects with this property set to **True**, you get a dialogue window indicating your workstation [IP address](#) and the port number.
2. At this point, switch over to the VistA M Server, and set any break points in the routines being called in order to help isolate the problem. Then issue the M debug command (e.g., [ZDEBUG](#) in DSM).
3. Start the following VistA M Server process:

```
>D EN^XWBTCP
```

You are prompted for the workstation [IP address](#) and the port number. After entering the information, switch over to the client application and click **OK**.

4. 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.

[Home](#) > [Debugging and Troubleshooting](#) > RPC Error Trapping

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 (in DSM an "allocation failure"). 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.

[Home](#) > [Debugging and Troubleshooting](#) > Broker Error Messages

Broker Error Messages

The following table lists the errors/messages that are Broker-specific and are *not* Winsock related:

Error/Message	Name	Number	Description
Insufficient Heap	XWB_NO_HEAP	20001	<p>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:</p> <ul style="list-style-type: none"><li data-bbox="1166 1514 1414 1640">• Too many open applications.<li data-bbox="1166 1724 1349 1850">• 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

			<p>data sets.</p> <ul style="list-style-type: none"> • Reboot the workstation.
M Error - Use ^XTER	XWB_M_REJECT	20002	<p>The Vista M Server side of the application errored out. The Kernel error trap has recorded the error.</p> <p>Resolution: Examine the Kernel error trap for more information and specific corrective actions.</p>
Signon was not completed	XWB_BadSignOn	20004	<p>This error indicates the user did <i>not</i> successfully signon.</p> <p>Resolution: Either the Access and Verify codes were incorrect or the user clicked Cancel on the</p>

			Vista Sign-on window.
BrokerConnections list could not be created	&XWB_BldConnectList	20005	<p>This error is a specific symptom of a low memory condition.</p> <p>Resolution: For a detailed explanation and corrective measures, see the "Insufficient Heap" error message.</p>
RpcVersion cannot be empty	XWB_NullRpcVer	20006	<p>This error occurs when an RPC does <i>not</i> have an associated version number. Each RPC <i>must</i> have a version number.</p> <p>Resolution: Contact the developers responsible for the application software to take corrective action.</p>
Remote procedure	XWB_RpcNotReg	20201	This error

not registered to application

indicates the application attempted to execute an RPC that was *not* entered into the RPC Multiple field in the [REMOTE PROCEDURE file \(#8994\)](#) 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 field (#.01) in the [REMOTE PROCEDURE file \(#8994\)](#) with the RPC field (#320) of the OPTION file (#19). Ideally, this should only be attempted during off or low system usage.



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

[redacted](#)

[Home](#) > [Debugging and Troubleshooting](#) > EBrokerError

EBrokerError

Unit


[TRPCB](#)

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:


```
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](#)" topic.

[Home](#) > [Debugging and Troubleshooting](#) > Testing the RPC Broker Connection

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, located in the ..\Broker directory that was installed with the client workstation software).

 **REF:** For a complete description of the RPC Broker Diagnostic program, see Chapter 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 (i.e., [BrokerExample.EXE](#)) located in the ..\BDK32\Samples\BrokerEx directory.

[Home](#) > [Debugging and Troubleshooting](#) > Identifying the Listener
Process on the Server

Identifying the Listener Process on the Server

On DSM systems, where the Broker Listener is running, the Listener process name is `RPCB_Port:NNNN`, where *NNNN* is the port number being listened to. This should help quickly locate Listener processes when troubleshooting any connection problems.

Example

RPCB_Port:9999

[Home](#) > [Debugging and Troubleshooting](#) > Identifying the Handler
Process on the Server

Identifying the Handler Process on the Server

On DSM systems the name of a Handler process is `ipXXX.XXX:NNNN`, where `XXX.XXX` are the last two octets of the client [IP address](#) and `NNNN` is the port number.

Example:

ip1.999:9999

[Home](#) > [Debugging and Troubleshooting](#) > Client Timeout and Buffer Clearing

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.

[Home](#) > [Debugging and Troubleshooting](#) > Memory Leaks

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 help 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.

[Home](#) > [Debugging and Troubleshooting](#) > ZDEBUG

ZDEBUG

A command used to enable or disable debug mode in Digital Standard M (DSM):

- Turn debugging on:

>**ZDEBUG ON**

- Turn debugging off"

>**ZDEBUG OFF**

[Home](#) > [Tutorial](#) > Introduction

Tutorial: Introduction

[Contents](#)

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.

Tutorial Contents

- [Advanced Preparation](#)
- [Step 1: RPC Broker Component](#)
- [Step 2: Get Server and Port](#)
- [Step 3: Establish Broker Connection](#)
- [Step 4: Routine to List Terminal Types](#)
- [Step 5: RPC to List Terminal Types](#)
- [Step 6: Call ZxxxTT LIST RPC](#)
- [Step 7: Associating IENs](#)
- [Step 8: Routines to Retrieve Terminal Types](#)
- [Step 9: RPC to Retrieve Terminal Types](#)
- [Step 10: Call ZxxxTT RETRIEVE RPC](#)
- [Step 11: Register RPCs](#)
- [Final: Using VA FileMan Delphi Components \(FMDCs\)](#)
- [Tutorial Source Code \(Sample\)](#)

[Home](#) > [Tutorial](#) > Advanced Preparation

Tutorial: Advanced Preparation

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 Zxxx during the tutorial.

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/developer 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.


[Home](#) > [Tutorial](#) > 1-RPC Broker Component

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:

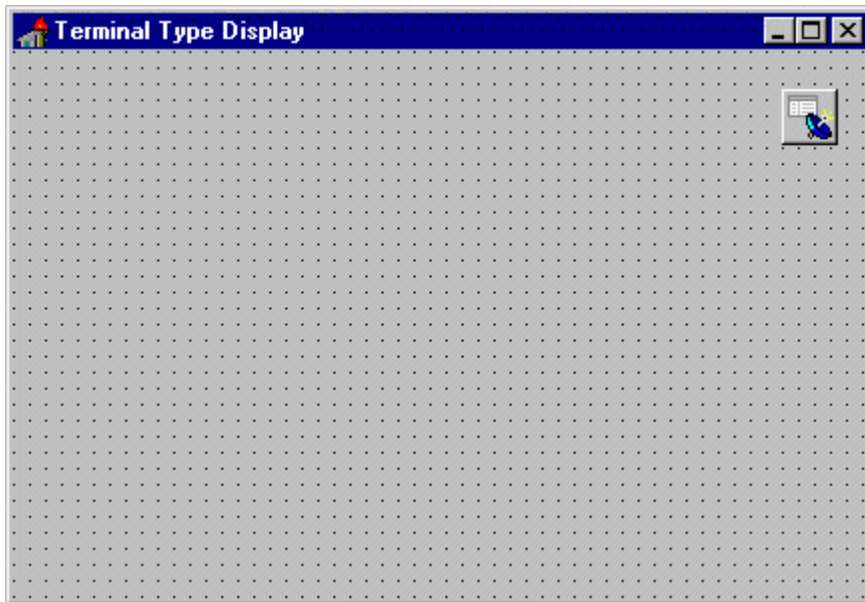
1. In Delphi, create a new application. Delphi creates a blank form, named **Form1**.
2. Set Form1's Caption property to **Terminal Type Display**.
3. From the **Kernel** component palette tab, add a [TRPCBroker](#) component to your form. The instance of the component is automatically named **RPCBroker1**. 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](#) as is (they are retrieved from your workstation's Registry). In the next section of the tutorial, you will add code to retrieve these values

at run-time from the workstation's Registry.

5. Set the [ClearParameters](#) and [ClearResults](#) properties 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 the following:



The next tasks are to use the [TRPCBroker](#) component to retrieve the client workstation's RPC Broker server and port information ([Step 2](#)), and then to establish a connection through the [TRPCBroker](#) component to the VistA M Server ([Step 3](#)).

[Home](#) > [Tutorial](#) > 2-Get Server and Port

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 9200). Naturally, you do *not* want this to be the only server and port that 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](#) 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](#) and [ListenerPort](#) properties to the returned values.

The code should look like the following:

```
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 ([Step 3](#)) is for the application to use the [TRPCBroker](#) component to establish a connection to the VistA M Server.

[Home](#) > [Tutorial](#) > 3-Establish Broker Connection

Tutorial: Step 3—Establish Broker Connection

Now that the application can determine the appropriate RPC Broker server and port to connect ([Step 2](#)), 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:

```
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;

```

i **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 Access and Verify codes (unless Auto Signon is enabled, and you are already signed on). 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.

[Home](#) > [Tutorial](#) > 4-Routine to List Terminal Types

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 ([Step 3](#)), 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. Create a routine, in the M account that the [TRPCBroker](#) component connects to, 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",#).

```
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 termtypes entries
; now set termtypes 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(^T
MP("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 following 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 ([Step 5](#)) is to create the [RPC](#) that calls this routine.

[Home](#) > [Tutorial](#) > 5-RPC to List Terminal Types

Tutorial: Step 5—RPC to List Terminal Types

Now that you have created an [RPC](#)-compatible routine to list terminal types ([Step 4](#)), you can go ahead and create the RPC itself (the entry in the REMOTE PROCEDURE 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 file. Set up the RPC as follows:

```
NAME: ZxxxTT LIST
TAG: TERMLIST
ROUTINE: ZxxxTT
RETURN VALUE TYPE: GLOBAL ARRAY
WORD WRAP ON: TRUE
DESCRIPTION: Used in RPC Broker developer
tutorial.
```

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).

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.

2. The next step of the tutorial ([Step 6](#)) is to call this [RPC](#) from the tutorial application, through its [TRPCBroker](#) component.

[Home](#) > [Tutorial](#) > 6-Call ZxxxTT LIST RPC

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.

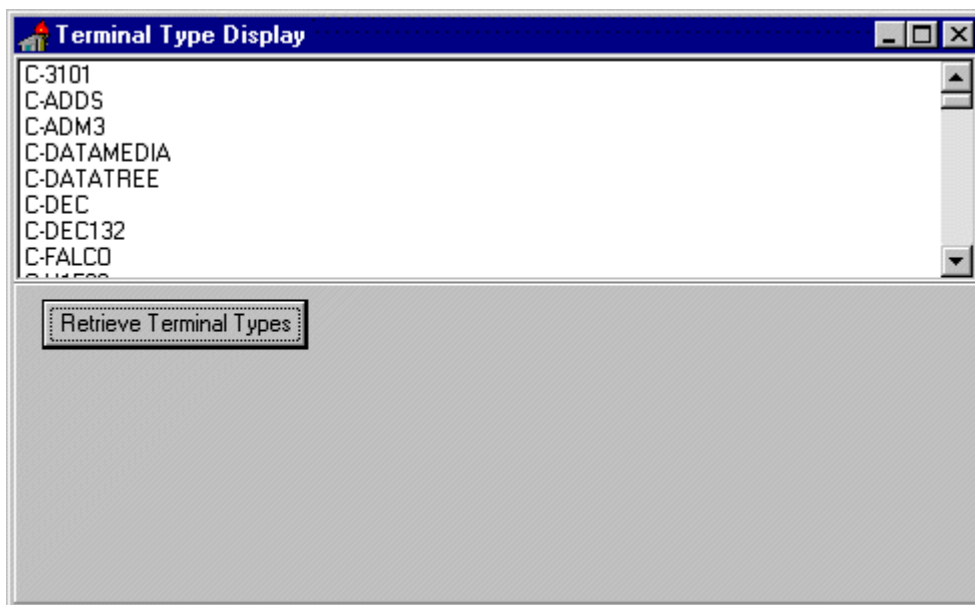
4. In the TForm1.Button1Click event handler, add code to call the ZxxxTT LIST [RPC](#) and populate the list box with the retrieved list of terminal type entries. This code should:

- 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 as follows:

```
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.');
```

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**.
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 follows:



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.

[Home](#) > [Tutorial](#) > 7-Associating IENs

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:

```
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:

```
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 Results property. 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:

```
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.');
```

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.

```
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.

[Home](#) > [Tutorial](#) > 8-Routine to Retrieve Terminal Types

Tutorial: Step 8—Routine to Retrieve Terminal Types

Now that you have associated an IEN for each record displayed in the list box ([Step 7](#)), 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.

- 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.
- Add a second subroutine to the ZxxxTT routine for the second [RPC](#), similar to the following. 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.

```

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

```

- Test the routine. Call it like the Broker would:

```
>D TERMENT^ZxxxTT (. ARRAY, 103)
```

Confirm that the return array contains the specified fields in the following nodes:

```

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

```

- Once you have tested the routine, and confirmed that it returns data correctly, the next step ([Step 9](#)) is to create the [RPC](#) that

calls this routine.

[Home](#) > [Tutorial](#) > 9-RPC to Retrieve Terminal Types

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 ([Step 8](#)), 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 file. Set up the RPC as follows:

```
NAME: ZxxxTT RETRIEVE
TAG: TERMENT
ROUTINE: ZxxxTT
RETURN VALUE TYPE: ARRAY
DESCRIPTION: Used in RPC Broker tutorial.
INPUT PARAMETER: IEN PARAMETER TYPE: LITERAL
```

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.

The WORD WRAP ON setting does *not* affect RPCs whose RETURN VALUE TYPE is ARRAY.

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.

2. This [RPC](#) can now be called from a Delphi-based application, through the [TRPCBroker](#) component.

[Home](#) > [Tutorial](#) > 10-Call ZxxxTT RETRIEVE RPC

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:

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](#)'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 the following:

```

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.Res
                    ults[ 0] , '^',3);
                    PageLength.Text:=piece(brkrRPCBroker1.R
                    esults[ 0] , '^',4);

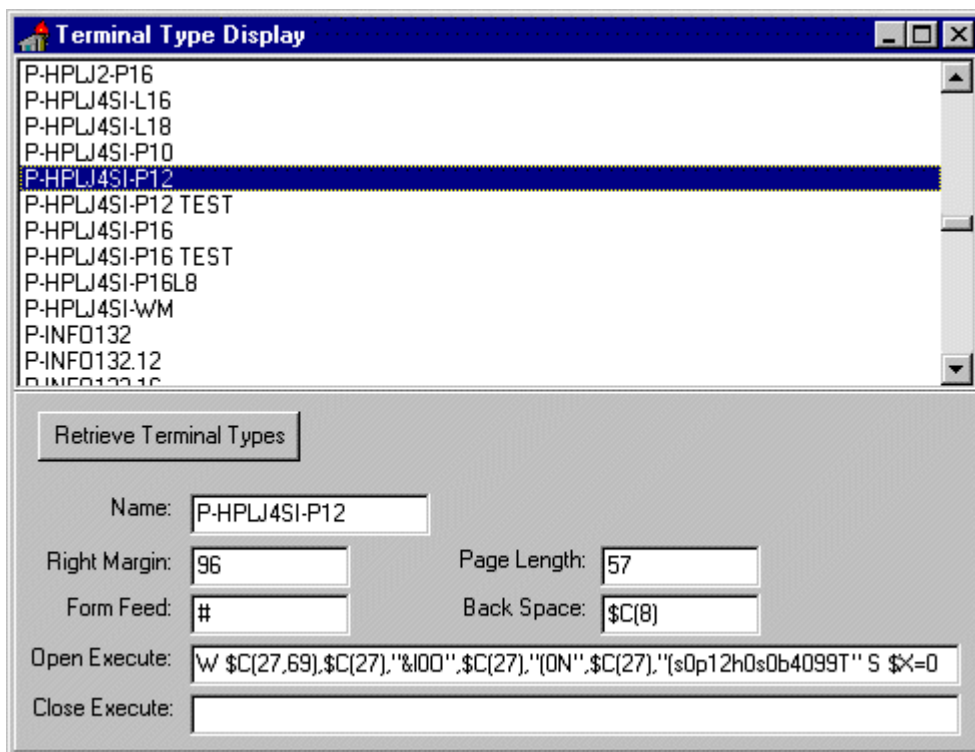
```

```

BackSpace.Text:=piece(brkrRPCBroker1.Re
sults[ 0] , '^' , 5);
OpenExecute.Text:=brkrRPCBroker1.Result
s[ 1] ;
CloseExecute.Text:=brkrRPCBroker1.Result
s[ 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.



[Home](#) > [Tutorial](#) > 11-Register RPCs

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/developer 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 "[How to Register an RPC](#)" topic to create an application context, using the ZxxxTT TERMTYPE option.

Essentially, add a line of code that calls the [CreateContext](#) function, and terminates the application if **False** is returned. The code for Form1's OnCreate event should now look like:

```
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.

[Home](#) > [Tutorial](#) > FileMan Delphi Components (FMDC)

Tutorial: 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: [redacted](#)

[Home](#) > [Tutorial](#) > Source Code

Tutorial Source Code

```
unit tut1;
```

```
interface
```

```
Windows, Messages, SysUtils, Classes, Graphics, Controls,  
Forms, Dialogs, Trpcb, RPCConfl, 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.
```

[Home](#) > [Tutorial](#) > DNS

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.

[Home](#) > [Tutorial](#) > HOSTS File

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 "DHCPSEVER" 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.

[Home](#) > [Tutorial](#) > IP Address

IP Address

The Internet Protocol (IP) address is the network interface address for a client workstation, server, or device.

[Home](#) > [Tutorial](#) > JOB

\$JOB

Contains your operating system job number on the VistA M Server:

- On DSM for OpenVMS systems—This is the process identification (PID) number of your process.
- On DSM for DEC OSF/1 systems—The job number is the PID of the child process created when you enter the DSM command.

[Home](#) > [Tutorial](#) > ORDER

\$ORDER

`$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.

[Home](#) > [Tutorial](#) > Silent Login

Silent Login

[Example](#)

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. It is similar to Auto Signon in some ways, but there are important differences.

There are two types of Silent Login are provided with the RPC Broker 1.1 BDK:

- **Access/Verify Code-based**—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.
- **Token-based**—Type of Silent Login that uses a 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 is available to all applications, whether or not they are using a Silent Login.



REF: For more information on handling divisions during Silent Login, see "[Handling Divisions During Silent Login.](#)"

Silent Login Compared to Auto Signon

In Auto Signon, the Client Agent manages the login process on the client. On a primary login (i.e., no existing connections), the user is prompted for Access and Verify codes. On secondary logins, the Client Agent handles the login with the information from the primary login. Developers do *not* have access to the Auto Signon process.




REF: For more information on Auto Signon, see the *RPC Broker Systems Management Guide*.

Silent Login offers developers an opportunity to skip the login process for the user if they have access to login information from some other source. It is up to the developer to deliver the appropriate login information to the application and enable the Silent Login process.

Interaction between Silent Login and Auto Signon

On primary login, Silent Login happens if it is enabled (the [KernelLogin](#) property is set to **False** and the [AccessCode](#), [VerifyCode](#), and [Mode](#) properties of the [Login](#) property are set or the [AppHandle](#) and [Mode](#) properties are set. On secondary logins, the Client Agent, if enabled, handles the login and the Silent Login is bypassed. In other words, if there already is a connection and the Client Agent is enabled, the Silent Login information is *not* used.

 **NOTE:** The "XUS SIGNON SETUP" RPC is called before a normal login or a Silent Login, and if it identifies the user via Client Agent at the IP address, that identification is used to log in the user.

[Home](#) > [Tutorial](#) > Silent Login Examples

Silent Login Examples


[Silent Login](#)

Example 1: ImAVCodes

The following is an example of how to use Silent Login by passing the Access and Verify codes to the [TVistaLogin](#) class.

```
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.

 **REF:** For a demonstration using the ImAVCodes, run the ImAVCodes_Demo.EXE located in the ..\BDK32\Samples\SilentSignOn directory.

Example 2: ImAppHandle

The following is an example of how to use Silent Login by passing an Application Handle to the [TVistaLogin](#) class.

The ImAppHandle 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 below. 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.



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.

```

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 s=, p=, d=, and h= are used in conjunction with the values for Server, ListenerPort, User.Division, and LoginHandle, respectively.

The command line might look like:

```
ProgramName.exe s=DHCPSEVER p=9200 d=692
h=~1XM34XYZZZQQ_X
```

The [TRPCB](#) and [RpcSLogin](#) units would need to be included in the **USES** clause.

```
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;

```

i **REF:** For a demonstration using the lmAppHandle, run the lmAppHandle_Demo.EXE located in the ..\BDK32\Samples\SilentSignOn directory.

[Home](#) > [Tutorial](#) > Handling Divisions During Silent Login

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 (i.e., Tstrings). The values that are present in the [DivList](#) property are similar to the following example:

```
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.

[Home](#) > [Tutorial](#) > Windows Registry

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.

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > DLL Interface
Introduction

DLL Interface Introduction

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.

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. So to understand the DLL interface, you should understand how the TRPCBroker component is used in its native environment (Delphi).

DLL Exported Functions

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](#)

Header Files

Header files for using the DLL are provided for C (BAPI32.H), C++ (BAPI32.HPP), and Visual Basic (BAPI32.BAS).

- [Guidelines for C Overview](#)
- [Guidelines for C++ Overview](#)
- [Guidelines for Visual Basic Overview](#)

Sample DLL Application

The VB5EGCHO sample application, distributed with the Broker Development Kit (BDK), demonstrates use of the RPC Broker 32-bit DLL from Microsoft Visual Basic. The source code is located in the `..\BDK32\Samples\Vb5Egcho` directory.

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > DLL Function Reference

DLL Exported Functions

The following functions for [TRPCBroker](#) components are exported in BAPI32.DLL:

Function	Description
RPCBCall	Execute an RPC.
RPCBCreate	Create a TRPCBroker component.
RPCBCreateContext	Create context.
RPCBFree	Destroy a TRPCBroker component.
RPCBMultItemGet	Get value of a Mult item in a Param .

RPCBMultPropGet	Get value of a Mult property in a Param .
RPCBMultSet	Set a Mult item in a Param to a value.
RPCBMultSortedSet	Sorts a Mult Param property.
RPCBParamGet	Get the value of a Param .
RPCBParamSet	Set the value of a Param .
RPCBPropGet	Get the value of a TRPCBroker component property.

[RPCBPropSet](#)

Set the value of a [TRPCBroker](#) component property.


[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [DLL Special Issues](#) > RPC Results through the DLL

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](#).

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [DLL Special Issues](#) > GetServerInfo Function and the DLL

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](#) is *not* currently available through the RPC Broker 32-bit DLL interface. A future version of the RPC Broker will provide access to the [GetServerInfo](#) functionality for DLL users.

To work around this for now, when using the RPC Broker 32-bit DLL, you should prompt the user directly for the server and port to connect.

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [DLL Exported Functions](#) > RPCBCall Function

RPCBCall Function

[Examples](#)


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.

Declarations

Software	Declaration
Delphi	<pre>procedure RPCBCall(const RPCBroker: TRPCBroker; CallResult: PChar);</pre>
C	<pre>char *(__stdcall *RPCBCall) (void *, char *);</pre>
C++	<pre>char * RPCBCall(char * s);</pre>
VB	<pre>Sub RPCBCall (ByVal intrPCBHandle As Long, ByVal strCallResult As String)</pre>

Parameter Description

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](#)" topic.

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [DLL Exported Functions](#) > RPCBCall Examples

RPCBCall Examples

C

```
RPCBCall(RPCBroker, Value);
```

C++

```
// MyInstance is defined as an instance of the TRPCBroker.  
MyInstance.RPCBCall( strbuffer);
```

Visual Basic

Call `RPCBCall(intRPCBHandle, strBuffer)`

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [DLL Exported Functions](#) > RPCBCreate Function

RPCBCreate Function

[Examples](#)

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.

Declarations

Software	Declaration
Delphi	<code>function RPCBCreate: TRPCBroker;</code>
C	<code>void * (__stdcall *RPCBCreate)(void);</code>
C++	N/A (encapsulated in TRPCBroker class definition)
VB	Function RPCBCreate () As Long

Return Value

A handle for the [TRPCBroker](#) component created.

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [DLL Exported Functions](#) > RPCBCreate Examples

RPCBCreate Examples

C

```
// Create the TRPCBroker component instance.  
RPCBroker = RPCBCreate();
```

Visual Basic

```
intRPCBHandle = RPCBCreate()
```

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [DLL Exported Functions](#) > RPCBCreateContext Function

RPCBCreateContext Function

[Examples](#)

This procedure calls the [TRPCBroker](#) component's [CreateContext](#) method to set up the environment on the Vista M Server for subsequent [RPCs](#).

Declarations

Software	Declaration
Delphi	<pre>function RPCBCreateContext(const RPCBroker: TRPCBroker; const Context: PChar): boolean;</pre>
C	<pre>bool (__stdcall *RPCBCreateContext) (void *, char *);</pre>
C++	<pre>bool RPCBCreateContext (char * s);</pre>
VB	<pre>Function RPCBCreateContext (ByVal intRPCBHandle As Long, ByVal strContext As String) As Integer</pre>

Return Value

- **True/1**—If context could be created.
- **False/0**—If context could *not* be created.

Parameter Description

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.

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [DLL Exported Functions](#) > RPCBCreateContext Examples

RPCBCreateContext Examples

C

```
// XWB EGCHO is a "B" (Broker) type option in the OPTION  
file.  
result = RPCBCreateContext(RPCBroker, "XWB EGCHO");
```

C++

```
// XWB EGCHO is a "B" (Broker) type option in the OPTION  
file.  
MyInstance.RPCBCreateContext("XWB EGCHO")
```

Visual Basic

```
intResult = RPCBCreateContext(intRPCBHandle, "MY  
APPLICATION")  
'where MY APPLICATION is a "B" (Broker) type option in the  
Option file.
```

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [DLL Exported Functions](#) > RPCBFree Function

RPCBFree Function

[Examples](#)

Destroys the RPC Broker component and releases associated [memory](#).

Declarations

Software	Declaration
Delphi	<pre>procedure RPCBFree(RPCBroker: TRPCBroker);</pre>
C	<pre>void (__stdcall *RPCBFree)(void *);</pre>
C++	N/A (encapsulated in TRPCBroker class definition)
VB	<pre>Sub RPCBFree (ByVal intrPCBHandle As Long)</pre>

Parameter Description

Parameter	Description
RPCBroker	Handle of the TRPCBroker component to destroy.

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [DLL Exported Functions](#) > RPCBFree Examples

RPCBFree Examples

C

```
// Destroy the TRPCBroker component instance.  
RPCBFree(RPCBroker);
```

Visual Basic

```
RPCBFree (intrPCBHandle)
```

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [DLL Exported Functions](#) > RPCBMultiItemGet Function

RPCBMultItemGet Function

[Examples](#)


Returns a requested item in a TRPCBroker [Param](#) property's [Mult](#) property.

Declarations

Software	Declaration
Delphi	<pre>procedure RPCBMultiItemGet (const RPCBroker: TRPCBroker; ParamIndex: integer; Subscript, Value: PChar);</pre>
C	<pre>void (__stdcall *RPCBMultiItemGet) (void *, int, char *, char *);</pre>
C++	<pre>void RPCBMultiItemGet (int i, char * s, char * t);</pre>
VB	<pre>Sub RPCBMultiItemGet (ByVal intRPCBHandle As Long, ByVal intParIdx As Integer, ByVal strSubscript As String, ByVal strValue As String)</pre>

Parameter Description

Parameter	Description
RPCBroker	Handle of the TRPCBroker component.
ParamIndex	Integer index of the parameter that contains the Mult property.
Subscript	Null-terminated string identifying the Mult element to get.
Value	An empty buffer that the calling application <i>must</i> create (allocate memory for) before making this call. This buffer is filled with the value of the Mult property 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](#)" topic.

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [DLL Exported Functions](#) > RPCBMultItemGet Examples

RPCBMultiItemGet Examples

C

```
// The following corresponds to getting the value of  
PARAM[0].Mult["0"]  
RPCBMultiItemGet(RPCBroker, 0 , "0", Value);
```

C++

```
MyInstance.RPCBMultiItemGet(0 , "0" , Value);
```

Visual Basic

```
Call RPCBMultiItemGet(intRPCBHandle, 0, "0", strResult)
```


[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [DLL Exported Functions](#) > RPCBMultPropGet Function

RPCBMultiPropGet Function

[Examples](#)


Returns a selected property value of a TRPCBroker [Param](#) property's [Mult](#) property.

Declarations

Software	Declaration
Delphi	<pre>procedure RPCBMultiPropGet (const RPCBroker: TRPCBroker; ParamIndex: integer; Prop, Value: PChar);</pre>
C	<pre>void (__stdcall *RPCBMultiPropGet) (void *, int, char *, char *);</pre>
C++	<pre>void RPCBMultiPropGet (int i , char * s, char * t);</pre>
VB	<pre>Sub RPCBMultiPropGet (ByVal intRPCBHandle As Long, ByVal intParIdx As Integer, ByVal strProp As String, ByRef strValue As String)</pre>

Parameter Description

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 <i>must</i> 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](#)" topic.

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [DLL Exported Functions](#) > RPCBMultiPropGet Examples

RPCBMultiPropGet Examples

C

```
RPCBMultiPropGet(RPCBroker, 0, "Count", Value);
```

C++

```
MyInstance.RPCBMultiPropGet(0, "Count", Value);
```


Visual Basic

```
Call RPCBMultiPropGet(intRPCBHandle, 0, "Count", strResult)
```

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [DLL Exported Functions](#) > RPCBMultiSet Function

RPCBMultiSet Function

[Examples](#)


Sets an item in a TRPCBroker [Param](#) property's [Mult](#) property to a value.

Declarations

Software	Declaration
Delphi	<pre>procedure RPCBMultiSet(const RPCBroker: TRPCBroker; ParamIndex: integer; Subscript, Value: PChar);</pre>
C	<pre>void (__stdcall *RPCBMultiSet) (void *, int, char *, char *);</pre>
C++	<pre>void RPCBMultiSet (int i, char * s, char * t);</pre>
VB	<pre>Sub RPCBMultiSet (ByVal intRPCBHandle As Long, ByVal intParIdx As Integer, ByVal strSubscript As String, ByVal strValue As String)</pre>

Parameter Description

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](#)" topic.

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [DLL Exported Functions](#) > RPCBMultiSet Examples

RPCBMultiSet Examples

C

```
RPCBMultiSet(RPCBroker, 0, "1", "12/19/97");
```


C++

```
MyInstance.RPCBMultiSet(0, "1", "12/19/97");
```

Visual Basic

```
Call RPCBMultiSet(intRPCBHandle, 0, "1", "12/19/97")
```

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [DLL Exported Functions](#) > RPCBMSortedSet Function

RPCBMSortedSet Function

[Examples](#)

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.

Declarations

Software	Declaration
Delphi	<pre>procedure RPCBMultSortedSet(const RPCBroker: TRPCBroker; ParamIndex: integer; Value: boolean);</pre>
C	<pre>void (__stdcall *RPCBMultSortedSet) (void *, int, bool);</pre>
C++	<pre>void RPCBMultSortedSet (int i, bool v);</pre>
VB	<pre>Sub RPCBMultSortedSet (ByVal intRPCBHandle As Long, ByVal intParIdx As Integer, ByVal intValue As Integer)</pre>

Parameter Description

Parameter	Description
RPCBroker	Handle of the TRPCBroker component.
ParamIndex	Integer index of the parameter that contains the Mult property.
Value	<p>Can be either a Boolean or, if the calling application language does <i>not</i> support Boolean type, can be an integer:</p> <ul style="list-style-type: none">• True or 1—Sorts the Mult and keeps it sorted thereafter when other elements are added.• False or 0—Does <i>not</i> sort the Mult and the elements are stored in the order they are added.

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [DLL Exported Functions](#) > RPCBMSortedSet Examples

RPCBMSortedSet Examples

C

```
RPCBMultiSortedSet(RPCBroker, 0, 1);
```

C++

```
MyInstance.RPCBMultiSortedSet(0, 1);
```

Visual Basic

Call `RPCBMultiPropGet(intRPCBHandle, 0, 1)`

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [DLL Exported Functions](#) > RPCBParamGet Function

RPCBParamGet Function

[Examples](#)

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 and list.

Declarations

Software	Declaration
Delphi	<pre>procedure RPCBParamGet(const RPCBroker: TRPCBroker; ParamIndex: integer; var ParamType: TParamType; var ParamValue: PChar);</pre>
C	<pre>void (__stdcall *RPCBParamGet) (void *, int, int, char *);</pre>
C++	<pre>void RPCBParamGet (int i, int j, char * s);</pre>
VB	<pre>Sub RPCBParamGet (ByVal intRPCBHandle As Long, ByVal intParIdx As Integer, ByVal intParTyp As Integer, ByVal intParVal As String)</pre>

Parameter Description

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](#)" topic.

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [DLL Exported Functions](#) > RPCBParamGet Examples

RPCBParamGet Examples

C

```
// PType and Value are variables to retrieve values into.  
RPCBParamGet(RPCBroker, 0, PType, Value);
```

C++

```
// PType and Value are variables to retrieve values into.  
MyInstance.RPCBParamGet(0, PType, Value);
```

Visual Basic

Call `RPCBParamGet(intRPCBHandle, 0, PType, strResult)`
' where `PType` and `strResult` are variables to retrieve
values into

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [DLL Exported Functions](#) > RPCBParamSet Function

RPCBParamSet Function

[Examples](#)

Sets one [Param](#) item's [Value](#) and [PType](#) properties, in a single call.

Declarations

Software	Declaration
Delphi	<pre>procedure RPCBParamSet(const RPCBroker: TRPCBroker; const ParamIndex: integer; const ParamType: TParamType; const ParamValue: PChar);</pre>
C	<pre>void (__stdcall *RPCBParamSet) (void *, int, int, char *);</pre>
C++	<pre>void RPCBParamSet (int i, int j, char * s);</pre>
VB	<pre>Sub RPCBParamSet (ByVal intRPCBHandle As Long, ByVal intParIdx As Integer, ByVal intParTyp As Integer, ByVal intParVal As String)</pre>

Parameter Description

Parameter	Description
RPCBroker	Handle of the TRPCBroker component.
ParamIndex	Integer index of the parameter.
ParamType	<p>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:</p> <ul style="list-style-type: none">• 0 (literal)• 1 (reference)• 2 (list) <p>You can use the enumerated values literal, reference and list, as declared in the C, C++ , or Visual Basic header file.</p>
ParamValue	Null-terminated string containing the Value to set.

i **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](#)" topic.

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [DLL Exported Functions](#) > RPCBParamSet Examples

RPCBParamSet Examples

C

```
RPCBParamSet(RPCBroker, 0, reference, "DUZ");
```

C++

```
MyInstance.RPCBParamSet(0, reference, "DUZ");
```

Visual Basic

```
Call RPCBParamSet(intRPCBHandle, 0, literal, Text3.Text)
```

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [DLL Exported Functions](#) > RPCBPropGet Function

RPCBPropGet Function

[Examples](#)

Returns a requested property of a [TRPCBroker](#) component.

Declarations

Software	Declaration
Delphi	<pre>procedure RPCBPropGet(const RPCBroker: TRPCBroker; const Prop: PChar; {var} Value: PChar);</pre>
C	<pre>void (__stdcall *RPCBPropGet) (void *, char *, char *);</pre>
C++	<pre>void RPCBPropGet (char * s, char * t);</pre>
VB	<pre>Sub RPCBPropGet (ByVal intRPCBHandle As Long, ByVal strProp As String, ByVal strValue As String)</pre>

Parameter Description

Parameter	Description
RPCBroker	Handle of the TRPCBroker component.
Prop	<p>Null-terminated string of the property to get. Not case-sensitive. Valid properties to get are:</p> <ul style="list-style-type: none">• ClearParameters• ClearResults• Connected• DebugMode• ListenerPort• RemoteProcedure• RPCTimeLimit• RPCVersion

	<ul style="list-style-type: none">• Server
Value	<p>An empty buffer that you <i>must</i> 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:</p> <ul style="list-style-type: none">• 1 (True)• 0 (False)

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [DLL Exported Functions](#) > RPCBPropGet Examples

RPCBPropGet Examples

C

```
RPCBPropGet (RPCBroker, "Connected", Value);
```

C++

```
MyInstance.RPCBPropGet ("Connected", Value);
```

Visual Basic

```
Call RPCBPropGet(intRPCBHandle, "Server", strResult)
```


[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [DLL Exported Functions](#) > RPCBPropSet Function

RPCBPropSet Function

[Examples](#)

Sets a [TRPCBroker](#) property to some value.

Declarations

Software	Declaration
Delphi	<pre>procedure RPCBPropSet(const RPCBroker: TRPCBroker; Prop, Value: PChar);</pre>
C	<pre>void (__stdcall *RPCBPropSet) (void *, char *, char *);</pre>
C++	<pre>void RPCBPropSet (char * s, char * t);</pre>
VB	<pre>Sub RPCBPropSet (ByVal intRPCBHandle As Long, ByVal strProp As String, ByVal strValue As String)</pre>

Parameter Description

Parameter	Description
RPCBroker	Handle of the TRPCBroker component.
Prop	Null-terminated string of the property to set; not case-sensitive. Valid properties to set are: <ul style="list-style-type: none">• ClearParameters• ClearResults• Connected• DebugMode• ListenerPort• RemoteProcedure• RPCTimeLimit• RPCVersion

	<ul style="list-style-type: none">• Server
Value	<p>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:</p> <ul style="list-style-type: none">• 1 (True)• 0 (False)

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [DLL Exported Functions](#) > RPCBPropSet Examples

RPCBPropSet Examples

C

```
RPCBPropSet (RPCBroker, "ListenerPort", "9999");
```


C++

```
MyInstance.RPCBPropSet("ListenerPort", "9999");
```

Visual Basic

```
Call RPCBPropSet(intRPCBHandle, "Server", cboServer.Text)
```

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [C++ Guidelines](#) >
Overview: C++ Guideline

Overview: Guidelines for C++

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.

TRPCBroker C++ class methods

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. [Initialize the Class](#)
2. [Create Broker Instances](#)
3. [Connect to the Server](#)
4. [Execute RPCs](#)
5. [Destroy Broker Instances](#)

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [C++ Guidelines](#) >
C++ Class Methods

C++: TRPCBroker C++ Class Methods

The functions in the RPC Broker 32-bit DLL are encapsulated in the following TRPCBroker C++ class methods:

DLL Function	TRPCBroker C++ Class Method
RPCBCall	<code>char * RPCBCall(char * s);</code>
RPCBCreateContext	<code>bool RPCBCreateContext (char * s);</code>
RPCBMultItemGet	<code>void RPCBMultItemGet (int i, char * s, char * t);</code>
RPCBMultPropGet	<code>void RPCBMultPropGet</code>

	<code>(void * ptr, int i , char * s, char * t);</code>
<u>RPCBMultiSet</u>	<code>void RPCBMultiSet (int i, char * s, char * t);</code>
<u>RPCBMultiSortedSet</u>	<code>void RPCBMultiSortedSet (void * ptr, int i, bool v);</code>
<u>RPCBParamGet</u>	<code>void RPCBParamGet (int i, int j, char * s);</code>
<u>RPCBParamSet</u>	<code>void RPCBParamSet (int i, int j, char * s);</code>
<u>RPCBPropGet</u>	<code>void RPCBPropGet (char * s, char * t);</code>
<u>RPCBPropSet</u>	<code>void RPCBPropSet (char * s, char * t);</code>

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [C++ Guidelines](#) >
C++ Initialize the Class

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.

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [C++ Guidelines](#) >
C++ Create Broker Instances

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.

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [C++ Guidelines](#) >
C++ Connect to the Server

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", "9999");
```

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](#) returns **True**, 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;
```

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [C++ Guidelines](#) >
C++ Execute RPCs

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](#) 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](#) method 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).

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [C++ Guidelines](#) >
C++ Destroy Broker Instances

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.

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [C Guidelines](#) >
Overview: C Guidelines

Overview: Guidelines for C

The BAPI32.H header file defines the function prototypes for all functions exported in the RPC Broker 32-bit DLL.

DLL Exported Functions

To use the DLL Broker functions, using C, exported in BAPI32.DLL, do the following:

1. [Initialize—LoadLibrary and GetProcAddress](#)
2. [Create Broker Components](#)
3. [Connect to the Server](#)
4. [Execute RPCs](#)
5. [Destroy Broker Components](#)

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [C Guidelines](#) > C:
Initialize LoadLibrary and GetProcAddress

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.

```
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:

```
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");
RPCBMultiSet = (void (__stdcall*)(void *, int, char *,
char *)) GetProcAddress(hLib, "RPCBMultiSet");
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.

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [C Guidelines](#) > C:
Create Broker Components

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](#) method 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.

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [C Guidelines](#) > C:
Connect to the Server

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", "9200");
```

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](#) 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;
```

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [C Guidelines](#) > C:
Execute RPCs

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](#) 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](#) method 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).

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [C Guidelines](#) > C:
Destroy Broker Components

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);
```

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [Visual Basic Guidelines](#) > Overview: Visual Basic Guidelines

Overview: Guidelines for Visual Basic

The BAPI32.BAS header file defines the function prototypes for all functions exported in the RPC Broker 32-bit DLL.

DLL Exported Functions

To use the DLL Broker functions, using Visual Basic, exported in BAPI32.DLL, do the following:

1. [Initialize](#)
2. [Create Broker Components](#)
3. [Connect to the Server](#)
4. [Execute RPCs](#)
5. [Destroy Broker Components](#)

Sample DLL Application

The VB5EGCHO sample application, distributed with the Broker Development Kit (BDK), demonstrates use of the RPC Broker 32-bit DLL from Microsoft Visual Basic. The source code is located in the `..\BDK32\Samples\Vb5Egcho` directory.

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [Visual Basic Guidelines](#) > Visual Basic: Initialize

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.

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [Visual Basic Guidelines](#) > Visual Basic: Create Broker Components

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](#) method 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.

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [Visual Basic Guidelines](#) > Visual Basic: Connect to the Server

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", "9999")
```

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](#) returns **True**, then you are ready to call all RPCs registered to the application's "**B**"-type option:

```
intResult = RPCBCreateContext(intRPCBHandle, "MY  
APPLICATION")
```

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [Visual Basic Guidelines](#) > Visual Basic: Execute RPCs

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](#) 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](#) method 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).

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > [Visual Basic Guidelines](#) > Visual Basic: Destroy Broker Components

Visual Basic: 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 Visual Basic program, do the following:

1. Make sure the [TRPCBroker](#) component is not connected:

```
Call RPCBPropSet(intRPCBHandle, "Connected", "0")
```

2. Call the [RPCBFree](#) method to destroy the object:

```
RPCBFree(intRPCBHandle)
```

Visual Basic takes care of the details of unloading the DLL.

[Home](#) > [DLL Interfaces \(C+, C++, Visual Basic\)](#) > XUPROGMODE

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.