ActiveX Reference Manual

Updated: 3/3/2022

ActiveX Reference Manual

Table of Contents

Before You Begin:

Requirements

TDT Drivers must be installed before installing TDT ActiveX Controls.

The recommended operating systems for all TDT systems are Windows® 7 and 10.

Note: Version 7.4 and greater installations include both 32-bit and 64-bit versions of the activeX controls.

ActiveX Updates

Always ensure that you are using the same versions of ActiveX and the TDT Drivers. The version numbers should always be the same. To avoid problems, always upgrade TDT Drivers whenever you upgrade ActiveX. See the Revision History, page 99, for information about revisions and updates to the TDT ActiveX library.

Organization of the Manual

This manual is organized in the following sections:

- Overview
- Language Specific Essentials
- RPcoX Real-Time Processor Control
- PA5 Programable Attenuator Controls
- ZBus Device Controls
- Examples

TDT ActiveX Overview

TDT's ActiveX Controls provide a simple and powerful way to control TDT System 3 hardware modules from custom software applications running on a PC. ActiveX controls can be run from within an application program written in programming languages such as MATLAB, Visual Basic, Delphi, or Visual C++.

The ActiveX Controls

The TDT ActiveX programming library includes three ActiveX controllers: RPcoX, PA5x, and ZBUSx.

RPcoX

The RPcoX controller includes a versatile group of methods for the Classic Real-Time Processors (RP), Mobile Processors (RM), High Performance Processors (RX), and the Z-series Processors (RZ); making it possible to connect to hardware, load and run the RCO circuits on the hardware, and allow for flexible real-time control of the circuits loaded to the hardware.

PA5x

The PA5x controller includes methods for real-time control of the PA5 front panel parameters, such as attenuation and attenuation stepsize.

ZBUSx

The zBUSx controller includes methods that allow access to zBus control functions; such as flushing the IO, resetting the hardware, and triggering a zBus rack.

Controlling TDT Real-Time Processors using the RPcoX ActiveX Controls

Some of the most powerful ActiveX methods are those that interact with the processing chains as they are executed on TDT real-time processors. The processing chain—the most basic instructions used to control a processor are designed in RPvdsEx and saved as a Control Object, either as a Control Object File (*.rco) or embedded in the RPvds Circuit File (*.rcx). These files also contain special components called "parameter tags" that can be accessed via TDT ActiveX controls to implement real-time control. For more on RCOs, see page 20.

Using ActiveX with Common Programming Languages

Each programming language implements ActiveX controls differently. This section provides a brief explanation of programming using ActiveX controls with:

- MATLAB
- MSVC++
- Visual Basic
- Delphi

This manual also includes examples that demonstrate how to implement the TDT ActiveX controllers for MATLAB, MSVC++, and Visual Basic.

MATLAB ActiveX

MATLAB versions 5.3 and above support ActiveX controls. The primary MATLAB method call for using ActiveX controls is:

```
actxserver()
```

This method adds an ActiveX control to your program. Once the ActiveX control has been instantiated all of its ActiveX methods can be used.

Interfacing with TDT Devices through ActiveX Controls

The following three calls will get a circuit running on the processor device:

- **Connect(device type) -** establishes a connection with the processor device
- **LoadCOF** - loads a Control Object file
- **Run** - runs the circuit

Example Code

`RP=actxserver('RPco.x')`	Creates an ActiveX Control for the processor device, the second argument controls the placement of the icon in the MATLAB figure. The figure must remain open for ActiveX control methods to be called.
`RP.ConnectRP2('GB',1) % MATLAB R13 and up`	Calls the Connect function to the RP2 (a member of the RPx family) using the ActiveX control. Connects to the first RP2 via the Optical Gigabit port.
`RP.LoadCOF('C:\TDT\ActiveX\ ActXExamples\RP_files\*.rcx')`	Loads a processor device Control Object (*.rco or *.rcx) file.

`RP.Run`	Starts the processor device processing chain.

Included with the ActiveX help are several examples of programs using the ActiveX controls with the RP2. Other TDT processor devices may be used with these example files by modifying the example code to connect to the specified device. We have also included the circuit Control Object File (*.rcx). The examples include programs written for versions newer than MATLAB 6.0, specifically R13 and R14. If you are using an older version of MATLAB such as R12, please review the example files that were designed for older releases of MATLAB.

RP Example Programs

Circuit Loader, page 70

Demonstrates the basic ActiveX methods that are part of any program. The program starts an ActiveX control, connects to an RP2, and loads an *.rco or *.rcx file and runs it.

Methods used: ConnectRP2, ClearCOF, LoadCOF, Run, GetStatus

Device Checker, page 71

Checks the components in a circuit that has been loaded and is running.

Methods used: GetCycUse, GetNumOf, GetNameOf, GetTagType, GetTagSize

Band-limited Noise, page 72

Uses parameter tags to control the frequency and intensity of filtered noise.

Methods used: SetTagVal, GetTagVal

Continuous Play , page 77

Plays a continuous set of tones generated in MATLAB.

Methods used: WriteTagV, SoftTrg, GetTagVal

Continuous Acquire, page 74

Stores one channel of stream data to an f32 file.

Methods used: ReadTagV, SoftTrg, GetTagVal

FIR Filtered Noise, page 80

Uses a noise component on the DSP to generate and filter it through an FIR.

Methods used: SendSrcFile, SendParTable

Two Channel Continuous Acquire, page 81

Stores two channels of streaming data to a f32 file using ReadTagVEX.

Methods used: ReadTagVEX, SoftTrg, GetTagVal

Two Channel Continuous Play, page 83

Plays two sets of tones out of two DACs.

Methods used: WriteTagVEX

Using ActiveX Controls With More Than One TDT Module

When using ActiveX controls with multiple processor devices, create a separate ActiveX control for each module. For example, in the example code below the user can add code to talk to a different processor device by creating a second control with a different MATLAB handle (i.e. RP2_2 instead of RP2_1):

```
% TDT Module 1
RP2_1 = actxserver('RPco.x')
RP2_1.ConnectRP2('GB', 1) % This connects to RP2 module #1 via
the Optical Gigabit interface
% TDT Module 2
RP2_2 = actxserver('RPco.x')
RP2_2.ConnectRP2('GB', 2) % This connects to RP2 module #2 via
the Optical Gigabit interface
```

Using Older Versions of MATLAB

If using versions of MATLAB greater than release 12, the invoke() method is not required. If using MATLAB R12 or prior releases, the invoke() method is required. Examples of how the ConnectRP2 method should be called in older MATLAB releases are shown below.

```
invoke()
```

Calls the ActiveX methods used with a control object file (*.rco or *.rcx).

```
invoke(RP, 'ConnectRP2', 'GB', 1) % MATLAB Prior to R13
```

Important!: MATLAB 6.0 (R12) requires that all variables that are to be used in numerical operations be cast as Doubles. These operations include: +,-,.*,./,.^,: and others. Compare statements such as <,>,== do not need the variable to be of type double. Changing your MATLAB code to work with MATLAB 6.0 (R12) requires that you cast the variables as DOUBLE. MATLAB 7 (R14) supports math on integer and single-precision data.

For example:

```
freq=invoke(RPx,'GetTagVal','freq')
```

should be changed to

```
freq=double(invoke(RPx,'GetTagVal','freq')) % MATLAB 6.0 (R12)
```

Visual Basic ActiveX

Visual Basic supports ActiveX controls through a graphical interface. Controls are placed into frames in the same way that buttons and text boxes are added. The programmer then controls the circuit through calls to the ActiveX module. To use the ActiveX components for the Real-time processor family (RPcoX), the PA5 (PA5x), and the zBus (ZBUSx) you add them to your Visual Basic Program.

Interfacing with TDT Devices through ActiveX Controls

The following three calls will get a circuit running on the processor device:

- **Connect(device type)** - establishes a connection with the processor device
- **LoadCOF** - loads a Control Object file
- **Run** - runs the circuit

Example Code:

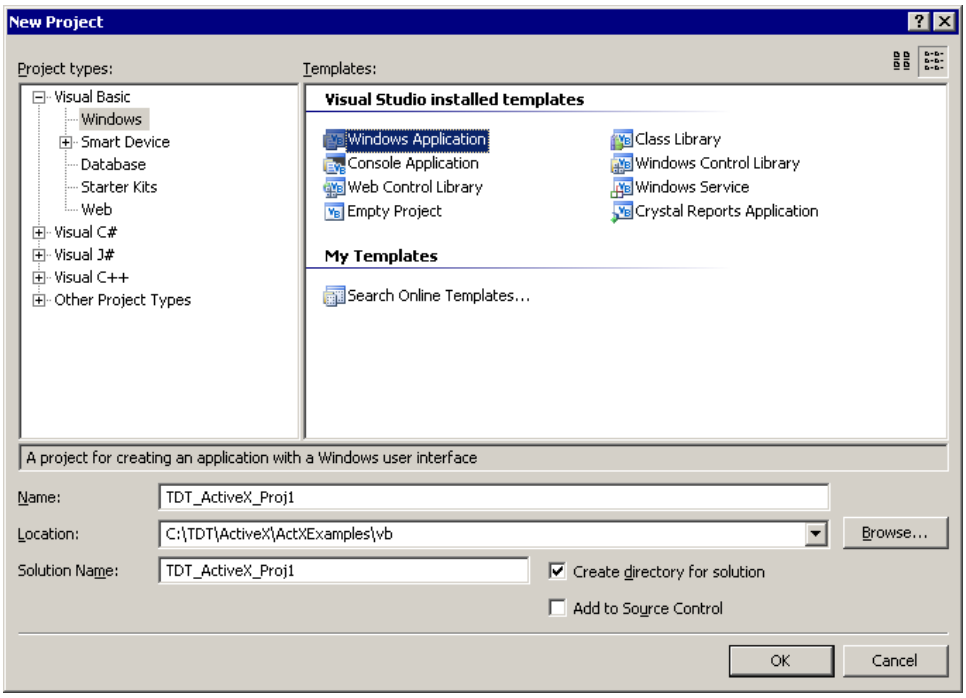
`RP2.ConnectRP2("GB",1)`	Calls the Connect method to the RP2 (a member of the RP family) using the ActiveX control. Connects to the first RP2 via the Optical Gigabit port.
`RP2.ClearCOF`	Clears any circuit on the RP2 processor device.
`RP2.LoadCOF("C:\TDT\ActiveX\ActXExamples\RP_files\*.rcx")`	Loads a processor device Control Object *.RCO (*.rco or *.rcx) File.
`RP2.Run`	Starts the processor device's processing chain.

Adding ActiveX Controls in VB.NET

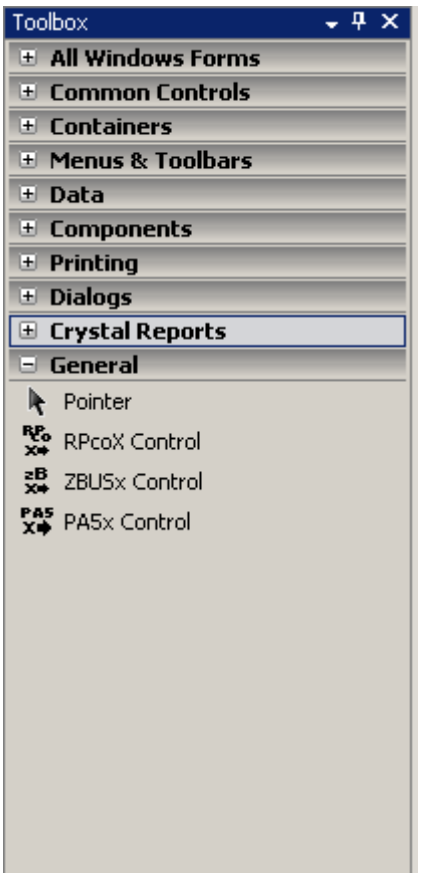
Visual Basic supports ActiveX controls through a graphical interface. Controls are placed into frames in the same way that buttons and text boxes are added. The programmer then controls the circuit through calls to the ActiveX module. To use the ActiveX components for the Real-time processor family (RPcoX), the PA5 (PA5x), and the zBus (ZBUSx) you add them to your Visual Basic Program. To use ActiveX in VB.NET you'll need to add the desired control to the **Toolbox**

To add an ActiveX Control in VB.NET:

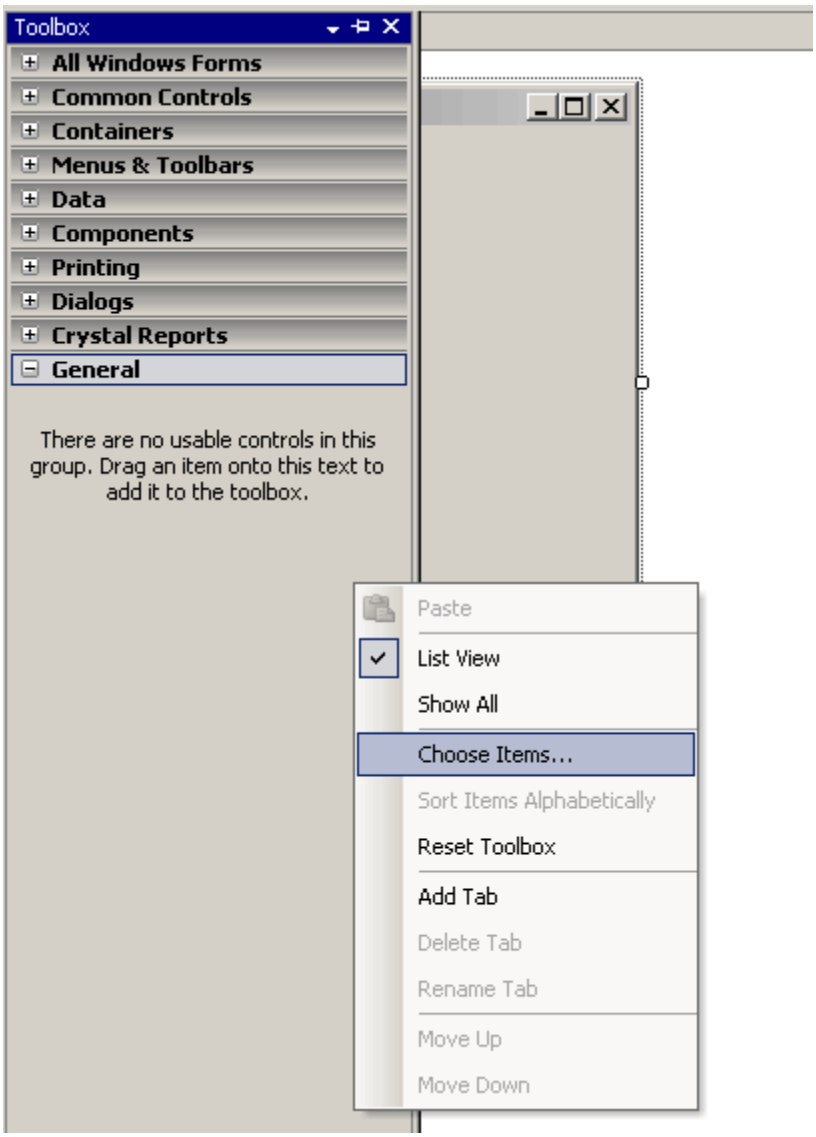
1. Create a new **Windows Application** by selecting **Visual Basic** from the **Project Types** dialog box to the left.



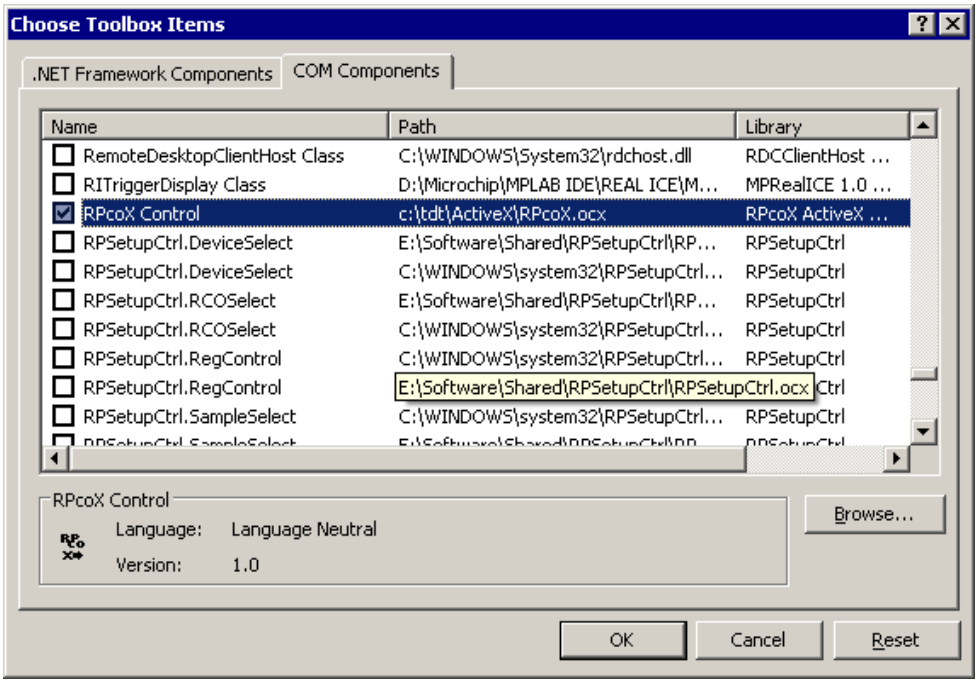
2. To display the **Toolbox,** Select **Toolbox** from the **View** menu.



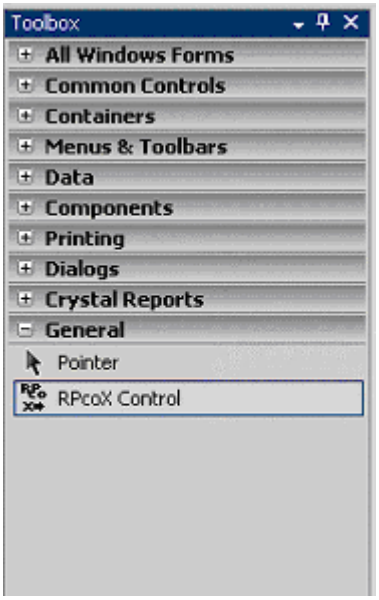
3. Next, add an ActiveX control, right-click in the **General** tab of the **Toolbox** and select **Choose Items**.



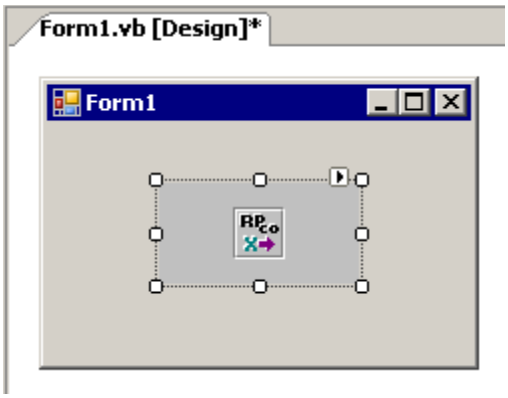
4. In the dialog box, click the **COM Components** tab. Scroll down the list and select the **RPcoX Control** check box, then click **OK**.



5. The **General** tab of the **Toolbox** should now contain the **RPcoX** control.



6. Click and drag the **RPcoX** control to your form.

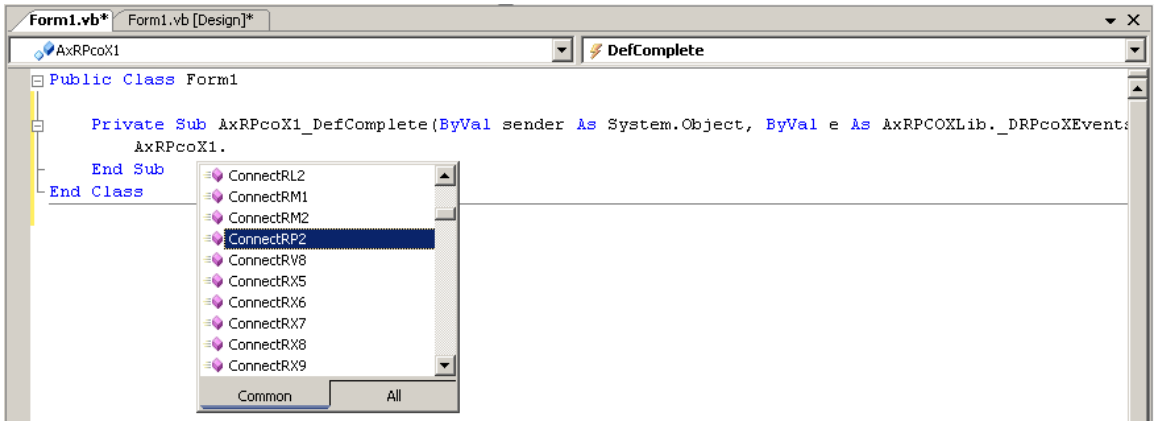


The default name for the new **RPcoX** control component is *AxRPcoX1*.

7. Repeat the steps above for any other TDT ActiveX control you wish to add (i.e. PA5x, ZBUSx).

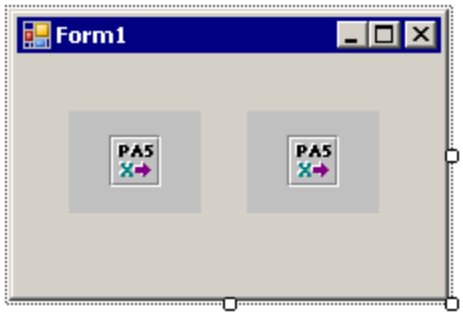
Displaying ActiveX Control Methods

In the code editor, type the name of the ActiveX control component (in this case *AxRPcoX1*) followed by a period to obtain a listing of the available methods and variable properties associated with that device.



Programming Multiple Modules

Each module should have its own ActiveX Control and its own variable. For example, to control two PA5 modules, insert two PA5x Controls. Each control will get its own variable.



Visual C++ ActiveX

Visual C++ supports ActiveX controls through a graphical interface. Controls are placed into frames in the same way that buttons and text boxes are added. The programmer then controls the circuit through calls to the ActiveX module. To use the ActiveX components for the Real-time processor family (RPcoX), the PA5 (PA5x), and the zBus (ZBUSx) you add them to your Visual C++ Program.

For Adding ActiveX Controls in VC++, see page 13.

Interfacing with TDT Devices through ActiveX Controls

The following three calls will get a circuit running on the processor device:

- **Connect (device type)** - establishes a connection with the processor device
- **LoadCOF** - loads a Control Object file
- **Run** - runs the circuit

Example Code:

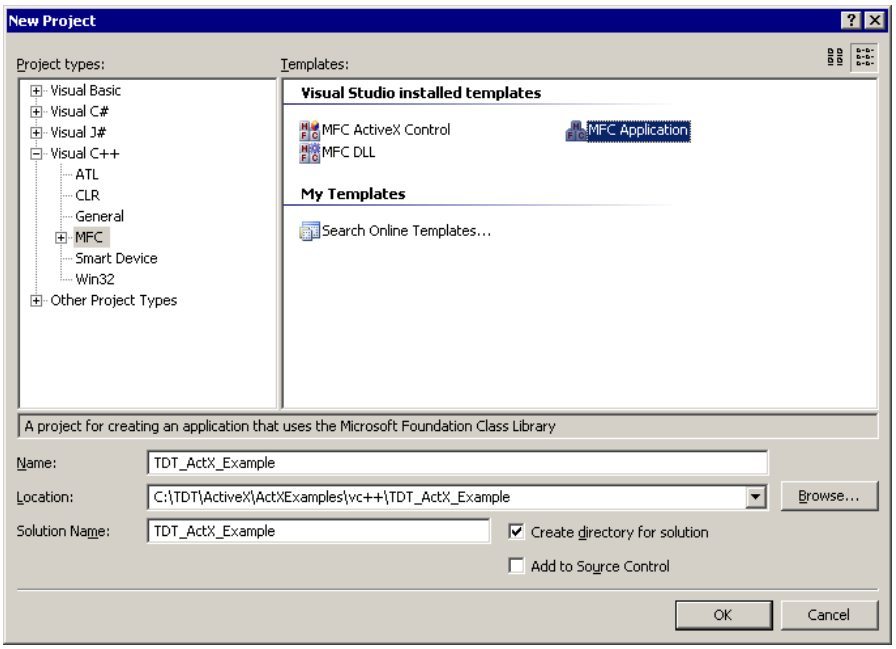
`m_rp2.ConnectRP2("GB", 1);`	Calls the Connect method to the RP2 (a member of the RP family) using the ActiveX control. Connects to the first RP2 via the Optical Gigabit port.
`m_rp2.ClearCOF();`	Clears any circuit on the RP2 processor device.
`m_rp2.LoadCOF("C:\TDT\ActiveX\ActXExamples\RP_files\*.rcx");`	Loads a processor device Control Object *.RCO (*.rco or *.rcx) File.
`m_rp2.Run();`	Starts the processor device's processing chain.

Adding ActiveX Controls in VC++

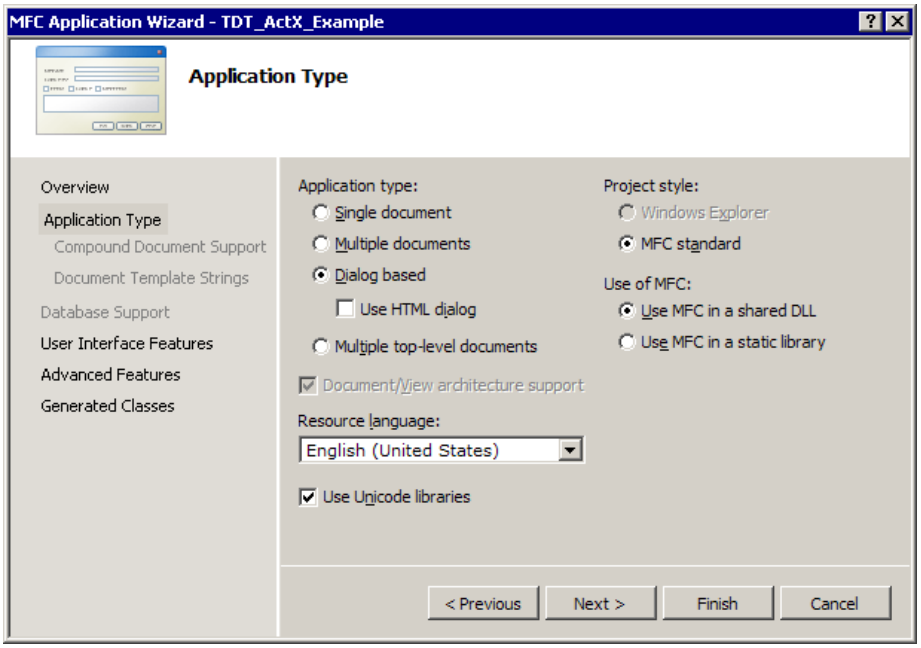
To use the TDT ActiveX controls with Visual C++, you need to make a project that uses MFC with ActiveX support. The easiest way to do this is to use the MFC Application Wizard. Make sure that support for ActiveX controls is enabled (it should be enabled by default). Then you will be able to add ActiveX controls to the dialog and make member variables for them using ClassWizard (see below for more details). This example assumes you are creating the ActiveX Control in a dialog box.

To use ActiveX in VC++:

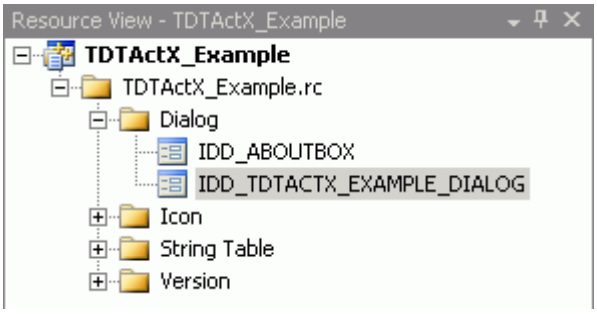
1. Create a project that uses an **MFC Application** with ActiveX support. Make sure that support for ActiveX controls is enabled (it should be enabled by default). Then you will be able to add ActiveX controls to the dialog and make member variables for them.



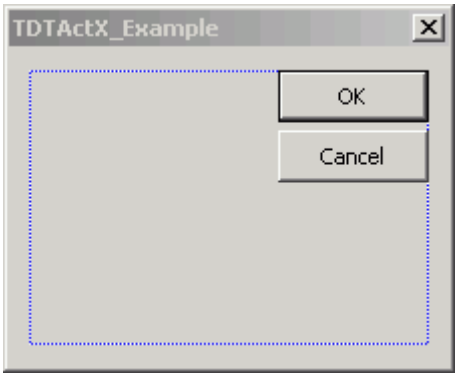
2. Follow the steps defined in the project wizard to create your **MFC Application**.
3. Under **Application type,** select the **Dialog based** radio button and click **Finish.**



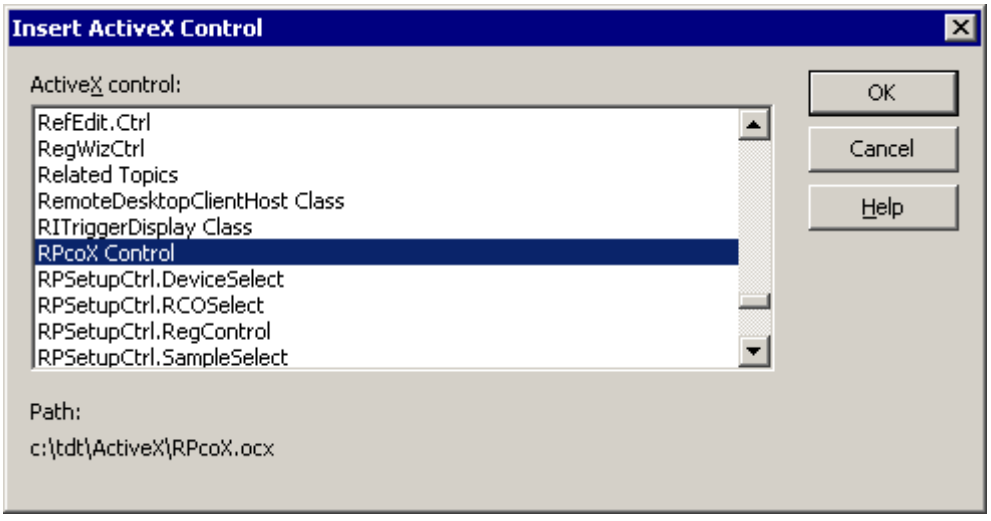
4. Under the **Resource View** dialog box, expand the **Dialog** folder and double click on **IDD_YourProjectName_DIALOG.**



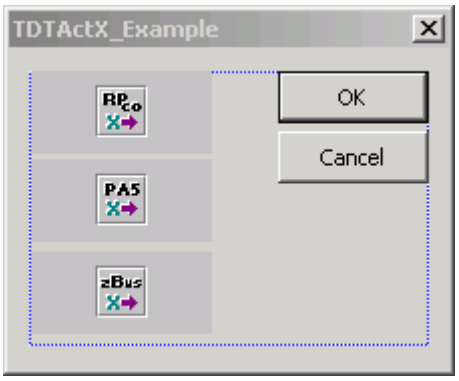
The dialog pane editor will then be shown in the workspace.



5. Right-click inside the blue dotted line on the dialog box and select **Insert ActiveX Control** from the menu.
6. Scroll down the list until you reach the desired ActiveX control (i.e. RPcoX, PA5x, or ZBUSx).

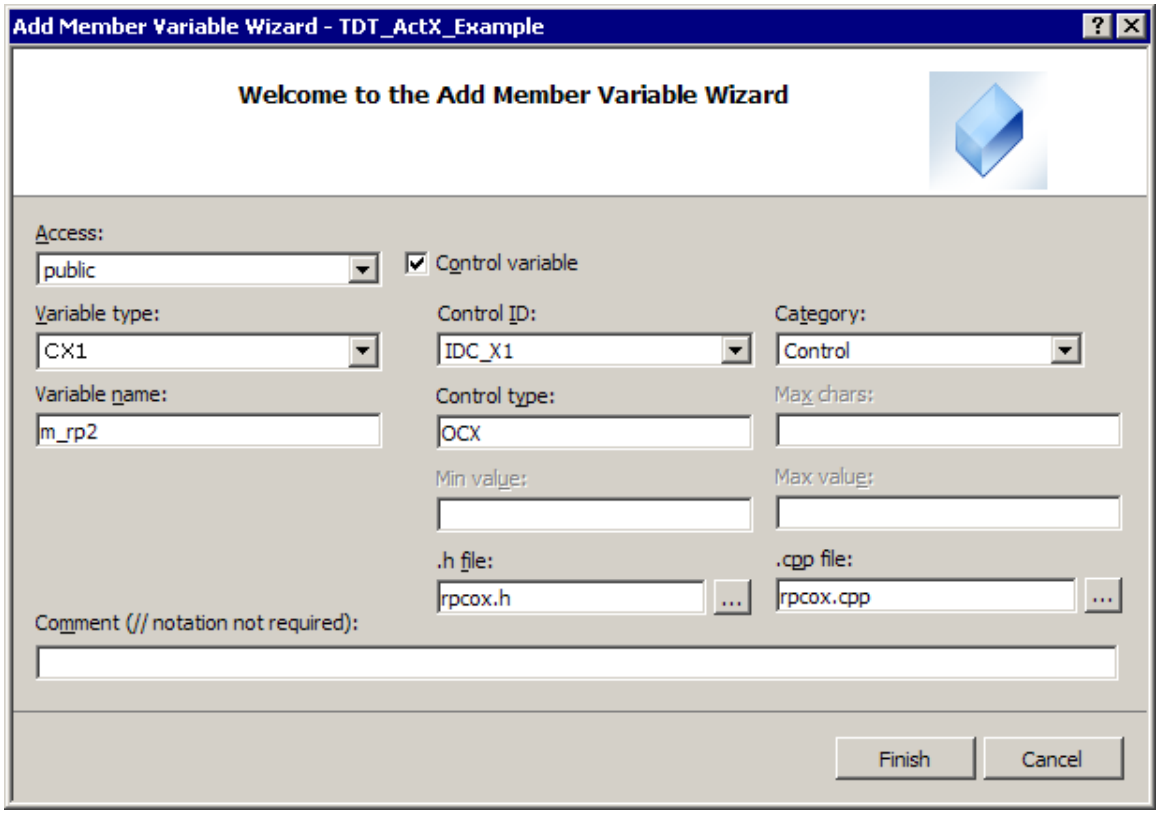


7. Click **OK.**
8. Drag the ActiveX control component to your dialog pane and place it in the desired location.



Adding a Member Variable

Right-click on the ActiveX control and select **Add Variable**. When you add a variable for the control, VC++ will create a Class wrapper for the control.

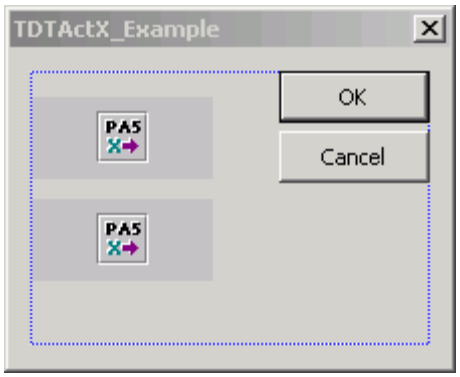


These variables are then used to call the ActiveX functions as shown below.

```
//Connect to RP2
m_rp2.ConnectRP2("GB", 1);//connect by GB to RP2 device #1
//Connect to PA5
m_pa5x1.ConnectPA5("USB", 1);//connect by USB to PA5 device #1
// Set Attenuation on PA5
m_pa5x1.SetAtten(20.0); //sets atten to 20 dB
```

Programming Multiple Modules

Each module should have its own ActiveX Control and its own variable. For example, to control two PA5 modules, insert two PA5x Controls and add a member variable for each PA5 control.



Visual C++ Examples

Included with the ActiveX help are several examples of programs using the ActiveX controls with the RP2. Other TDT processor devices may be used with these example files by modifying the example code to connect to the specified device. We have included the circuit design file *.rcx.

Circuit Loader, page 85

Demonstrates the basic ActiveX methods that are part of any program: The program starts an ActiveX control, connects to an RP2, loads a *.RCO (*.rco or *.rcx) file and runs it.

Methods used: ConnectRP2, LoadCOF, Run

Band Limited Noise, page 87

Uses parameter tags to control the frequency and intensity of filtered noise.

Methods used: ConnectRP2, ClearCOF, LoadCOF, GetStatus, Run, Halt, SetTagVal, GetTagVal, GetCycUse

Continuous Acquire, page 90

Continously acquires data and stores it on the PC at 100kH. Generates the file fnoise2.f32

Methods used: ReadTag, SoftTrg, GetTagVal

Continuous Play, page 93

Continuously plays sounds out of the RP2 that have been generated on the PC.

Methods used: WriteTag, SoftTrg, GetTagVal,GetTagSize

TDT ActiveX Console, page 96

Demonstrates the usage of the system console by connecting to an RP2, loads a *.RCO (*.rco or *.rcx) file and runs it.

Methods used: ConnectRP2, ClearCOF, LoadCOF, Run

Delphi Function Headers

All functions behave exactly the same in Delphi as they do in other programming languages. Users should refer to the RpcoX, PA5x, and ZBUSx sections of the ActiveX Help for details on how each function works. To determine the Delphi data types for each function and parameter, refer to the list below.

RpcoX

```
function GetError: WideString;
function Connect(Interface_: Integer; DevNum: Integer): Integer;
function SetTagVal(const Name: WideString; Val: Single): Integer;
function LoadCOF(const FileName: WideString): Integer;
function Run: Integer;
function Halt: Integer;
function SoftTrg(Trg_Bitn: Integer): Integer;
function GetTagVal(const Name: WideString): Single;
function ReadTag(const Name: WideString; var pBuf: Single; nOS:
Integer; nWords: Integer): Integer;
function WriteTag(const Name: WideString; var pBuf: Single; nOS:
Integer; nWords: Integer): Integer;
function SendParTable(const Name: WideString; IndexID: Single):
Integer;
function SendSrcFile(const Name: WideString; SeekOS: Integer;
nWords: Integer): Integer;
function ReadTagV(const Name: WideString; nOS: Integer; nWords:
Integer): OleVariant;
function WriteTagV(const Name: WideString; nOS: Integer; Buf:
OleVariant): Integer;
function GetTagSize(const Name: WideString): Integer;
function GetTagType(const Name: WideString): Integer;
function GetNumOf(const ObjTypeName: WideString): Integer;
function GetNameOf(const ObjTypeName: WideString; Index: Integer):
WideString;
function ReadCOF(const FileName: WideString): Integer;
function ConnectRP2(const IntName: WideString; DevNum: Integer):
Integer;
function ConnectRL2(const IntName: WideString; DevNum: Integer):
Integer;
function ConnectRA16(const IntName: WideString; DevNum: Integer):
Integer;
function ReadTagVEX(const Name: WideString; nOS: Integer; nWords:
Integer; const SrcType: WideString; const DstType: WideString;
nChans: Integer): OleVariant;
function GetStatus: Integer;
```

```
function GetCycUse: Integer;
function ClearCOF: Integer;
function WriteTagVEX(const Name: WideString; nOS: Integer; const
DstType: WideString; Buf: OleVariant): Integer;
function ZeroTag(const Name: WideString): Integer;
function GetSFreq: Single;
function ConnectRV8(const IntName: WideString; DevNum: Integer):
Integer;
function GetDevCfg(Addr: Integer; Width32: Integer): Integer;
function SetDevCfg(Addr: Integer; Val: Integer; Width32: Integer):
Integer;
function LoadCOFsf(const FileName: WideString; SampFreq: Single):
Integer;
```

ZbusX

```
function Connect(Interface_: Integer): Integer;
function GetDeviceAddr(DevType: Integer; DevNum: Integer):
Integer;
function GetDeviceVersion(DevType: Integer; DevNum: Integer):
Integer;
function HardwareReset(RackNum: Integer): Integer;
function FlushIO(RackNum: Integer): Integer;
function zBusTrigA(RackNum: Integer; zTrgMode: Integer; Delay:
Integer): Integer;
function zBusTrigB(RackNum: Integer; zTrgMode: Integer; Delay:
Integer): Integer;
function zBusSync(RackMask: Integer): Integer;
function GetError: WideString;
function GetDeviceAt(RackNum: Integer; PosNum: Integer; var DevID:
Integer; var DevNum: Integer): WideString;
function ConnectZBUS(const IntName: WideString): Integer;
```

PA5x

```
function Connect(Interface_: Integer; DevNum: Integer): WordBool;
function SetAtten(AttVal: Single): WordBool;
function GetAtten: Single;
function Reset: WordBool;
function SetUser(ParCode: Integer; Val: Single): WordBool;
function GetError: WideString;
function Display(const Text: WideString; Position: Integer):
WordBool;
function ConnectPA5(const IntName: WideString; DevNum: Integer):
Integer;
```

Working with Control Object Files (.rco and *.rcx)*

The Control Object File contains an object-oriented description of the circuit. When the circuit is loaded and run the Control Object File provides an interface between the processor device and the program using the Control Object (*.rco or *.rcx) File.

Once you have generated the circuit you can test it by running it within RPvdsEx. To check for problems Compile, Load, and Run the circuit before saving it as a Control Object File.

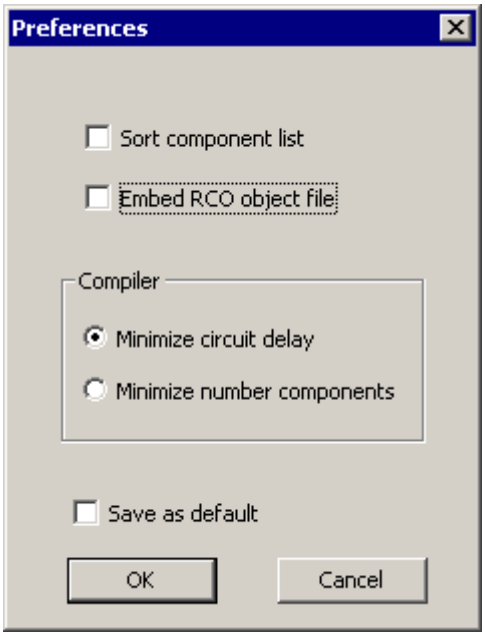
Note: The default preference for RPvdsEx is to embed the Control Object into an *.rcx file. RPvdsEx files that are compiled in this embedded format generate only one file (*.rcx) that has both the Control Object and Circuit Graphic file information.

Legacy formats use separate files for the Control Object and Circuit Graphic information. RPvdsEx preferences can be set to generate both an *.rpx and *.rco file for use with legacy formats.

Creating an RCO for Legacy Formats

To change the preferences in RPvdsEx for Legacy formats:

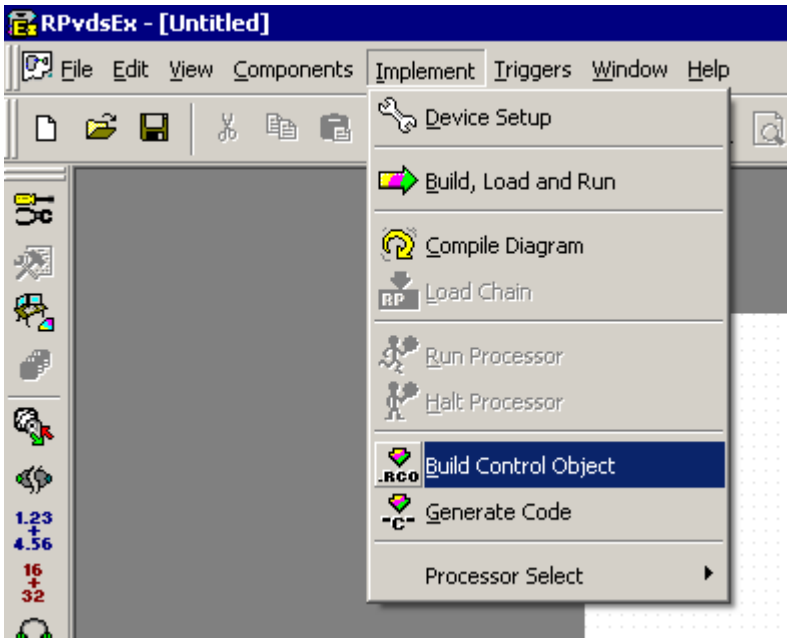
1. Click **Preferences** on the Edit menu.
2. Click to clear the **Embed RCO object file** checkbox.



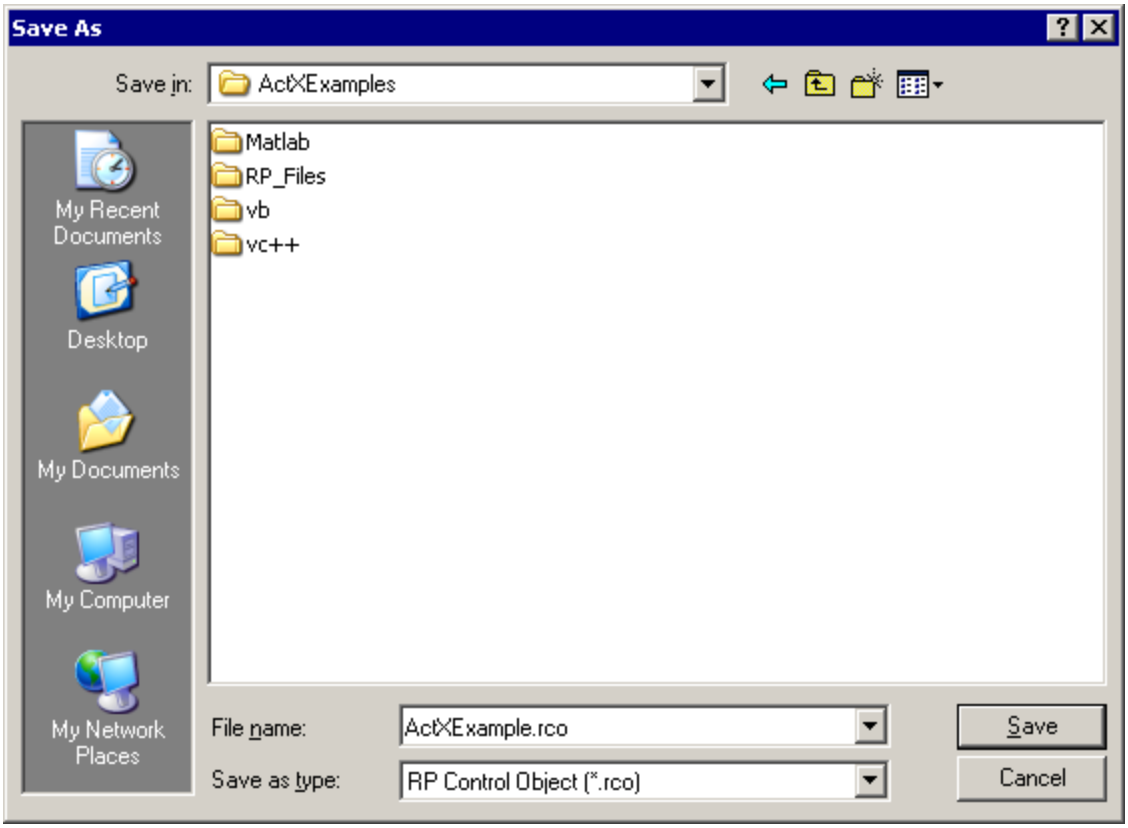
3. Click **OK**.

To save the file as a Control Object File:

1. Once the preferences above have been set, Click **Build Control Object** on the **Implement** menu or click the Build Control Object toolbar button.



2. In the Save As dialog box, enter a file name then click **Save**.



The saved *.rco file can be used by any program compatible with TDT's ActiveX controls (e.g. Matlab, Visual Basic).

RPcoX Real-Time Processor Control

About the RPcoX Methods

This section provides a listing of the available RPcoX ActiveX control methods.

Programming Steps:

- Add the RPcoX ActiveX controller to your program. The ActiveX help has examples for setting up ActiveX controllers in MATLAB, Visual Basic, and Visual C.
- Connect to a TDT processor (USB or GB) device with the matching device Connect function (i.e. for an RP2 use ConnectRP2).
- Control the device with the command and control functions using the ActiveX controller.

Device Connection

The device connection methods are used to establish an RPcoX ActiveX control to the desired device.

Important: The 'GB' argument is correct for the 'Optical Gigabit' interfaces, commonly referred to as Optibit. The PI5/FI5 are no longer supported in the current version of TDT Drivers and ActiveX.

ConnectRP2

Description: Establishes a connection with an RP2 or RP2.1 Real-time Processor through a device interface (such as Optical Gigabit or USB). A device number identifies which RP2 is connected.

'C' Prototype: `long ConnectRP2(LPCTSTR Interface, long DevNum);`

Arguments:

LPCTSTR	*Interface*	Interface to which the device is connected.

Argument	Connection	Part #s
'GB'	Optical Gigabit	PO5/FO5
'USB'	USB	UZ2, UB2, UZ1, UZ4

long	*DevNum*	Logical device number. Starts with 1 and counts upward for each device of a specified type.

Returns:

long	0	Connection not successful.
long	1	Connection successful.

Note: Invoke device connect commands only once to connect to a device, and then use ClearCOF and LoadCOF commands to upload or reload the control object to implement changes to the signal.

Sample Code:

MATLAB

```
%Connects to RP2 #1 via Optical Gigabit
RP=actxserver('RPco.x');
if RP.ConnectRP2('GB',1)
     e='connected'
else
     e='Unable to connect'
end
```

Visual Basic

```
'Connects to the RP2 via the Gigabit
If RP.ConnectRP2("GB",1) Then
  msgbox "Connection established"
Else
  msgbox "Error connecting to RP2"
End If
```

Example: Circuit Loader, page 70.

ConnectRA16

Description: Establishes a connection with the Medusa Base Station (RA16BA) via the Gigabit or USB bus interface. Invoking this method causes the control to search for the 16-channel preamplifier typically connected to the base station and establish a handle to the associated device driver. The ConnectRA16 method will return 1 if a connection was successfully established or 0 if the device is not present or is not functioning properly.

'C' Prototype:

```
long ConnectRA16(LPCTSTR Interface, long DevNum);
```

Arguments:

LPCTSTR	*Interface*	Interface to which the device is connected.

Argument	Connection	Part #s
'GB'	Optical Gigabit	PO5/FO5
'USB'	USB	UZ2, UB2, UZ1, UZ4

long	*DevNum*	Logical device number. Starts with 1 and counts upward for each device of a specified type.

Returns:

long	0	Connection not successful.
long	1	Connection successful.

Note: Invoke device connect commands only once to connect to a device, and then use ClearCOF and LoadCOF commands to upload or reload the control object to implement changes to the signal.

Sample Code:

MATLAB

```
% Connects to RA16 #1 via Optical Gigabit
RL2=actxserver('RPco.x');
if RL2.ConnectRA16('GB', 1)
   e= 'connected'
else
   e= 'Unable to connect'
end
```

ConnectRL2

Description: Establishes a connection with the Stingray Docking Station (RL2) via the Gigabit or USB bus interface. Invoking this method causes the control to search for the specified device and establish a handle to the associated device driver. The ConnectRL2 method will return 1 if a connection was successfully established or 0 if the device is not present or is not functioning properly.

'C' Prototype:
```
long ConnectRL2(LPCTSTR Interface, long DevNum);
```

Arguments:

LPCTSTR	*Interface*	Interface to which the device is connected.		
		Argument	**Connection**	**Part #s**
		'GB'	Optical Gigabit	PO5/FO5
		'USB'	USB	UZ2, UB2, UZ1, UZ4
long	*DevNum*	Logical device number. Starts with 1 and counts upward for each device of a specified type.		

Returns:

long	0	Connection not successful.
long	1	Connection successful.

Note: Invoke device connect commands only once to connect to a device, and then use ClearCOF and LoadCOF commands to upload or reload the control object to implement changes to the signal.

Sample Code:

MATLAB
```
% Connects to RL2 #1 via Optical Gigabit
RL2=actxserver('RPco.x');
if RL2.ConnectRL2('GB', 1)
   e= 'connected'
else
   e= 'Unable to connect'
end
```

ConnectRV8

Description: Establishes a connection with the Barracuda Processor (RV8) through the Gigabit or USB interface. Invoking this method causes the control to search for the Barracuda and establish a handle to the associated device driver. The ConnectRV8 method will return 1 if a connection was successfully established or 0 if the device is not present or is not functioning properly.

'C' Prototype:
```
long ConnectRV8(LPCTSTR Interface, long DevNum);
```

Arguments:

LPCTSTR	*Interface*	Interface to which the device is connected.		
		Argument	**Connection**	**Part #s**
		'GB'	Optical Gigabit	PO5/FO5
		'USB'	USB	UZ2, UB2, UZ1, UZ4
long	*DevNum*	Logical device number. Starts with 1 and counts upward for each device of a specified type.		

Returns:

long	0	Connection not successful.
long	1	Connection successful.

Note: Invoke device connect commands only once to connect to a device, and then use ClearCOF and LoadCOF commands to upload or reload the control object to implement changes to the signal.

Sample Code:

MATLAB

```
%Connects to RV8 #1 via Optical igabit
RP=actxserver('RPco.x');
if RP.ConnectRV8('GB',1)
     e='connected'
else
     e='Unable to connect'
end
```

Visual Basic

```
'Connects to the RV8 via the Optical Gigabit
If RP.ConnectRV8("GB",1) Then
  msgbox "Connection established"
Else
  msgbox "Error connecting to RV8"
End If
```

ConnectRM1

Description: Establishes a connection with a Mini Processor (RM1) using the device's built in USB interface.

'C' Prototype: `long ConnectRM1(LPCTSTR Interface, long DevNum);`

Arguments:

LPCTSTR	*Interface*	Interface to which the device is connected.

Argument	Connection	Part #s
'USB'	USB	Internal

long	*DevNum*	Logical device number. Starts with 1 and counts upward for each device of a specified type.

Returns:

long	0	Connection not successful.
long	1	Connection successful.

Note: Invoke device connect commands only once to connect to a device, and then use ClearCOF and LoadCOF commands to upload or reload the control object to implement changes to the signal.

Sample Code:

MATLAB:

```
%Connects to RM1 #1 via USB
RP=actxserver('RPco.x');
if RP.ConnectRM1('USB', 1)
    e='connected'
else
    e='Unable to connect'
end
```

Visual Basic:

```
'Connects to the RM1 via USB
```

```
If RP.ConnectRM1("USB", 1) Then
   msgbox "Connection established"
Else
   msgbox "Error connecting to RM1"
End If
```

ConnectRM2

Description: Establishes a connection with a Mobile Processor (RM2) using the device's built-in USB interface.

'C' Prototype: `long ConnectRM2(LPCTSTR Interface, long DevNum);`

Arguments:

LPCTSTR	*Interface*	Interface to which the device is connected.

Argument	Connection	Part #s
'USB'	USB	Internal

long	*DevNum*	Logical device number. Starts with 1 and counts upward for each device of a specified type.

Returns:

long	0	Connection not successful.
long	1	Connection successful.

Note: Invoke device connect commands only once to connect to a device, and then use ClearCOF and LoadCOF commands to upload or reload the control object to implement changes to the signal.

Sample Code:

MATLAB

```
%Connects to RM2 #1 via USB
RP=actxserver('RPco.x');
if RP.ConnectRM2('USB', 1)
    e='connected'
else
    e='Unable to connect'
end
```

Visual Basic

```
'Connects to the RM2 via USB
If RP.ConnectRM2("USB", 1) Then
   msgbox "Connection established"
Else
   msgbox "Error connecting to RM2"
End If
```

ConnectRX5

Description: Establishes a connection with a Pentusa Base Station (RX5) through a device interface (such as Gigabit or USB).

'C' Prototype: `long ConnectRX5(LPCTSTR Interface, long DevNum);`

Arguments:

LPCTSTR	*Interface*	Interface to which the device is connected.

		Argument	Connection	Part #s
		'GB'	Optical Gigabit	PO5/FO5
		'USB'	USB	UZ2, UB2, UZ1, UZ4
long	*DevNum*	Logical device number. Starts with 1 and counts up for each device of a specified type.		

Returns:

long	0	Connection not successful.
long	1	Connection successful.

Note: Invoke device connect commands only once to connect to a device, and then use ClearCOF and LoadCOF commands to upload or reload the control object to implement changes to the signal.

Sample Code:

MATLAB

```
%Connects to RX5 #1 via Optical Gigabit
RP=actxserver('RPco.x');
if RP.ConnectRX5('GB',1)
     e='connected'
else
     e='Unable to connect'
end
```

Visual Basic

```
'Connects to the RX5 via the Optical Gigabit
If RP.ConnectRX5("GB",1) Then
  msgbox "Connection established"
Else
  msgbox "Error connecting to RX5"
End If
```

Example: Circuit Loader, page 70.

ConnectRX6

Description: ConnectRX6 establishes a connection with a MultiFunction Processor (RX6) through a device interface (such as Gigabit or USB).

'C' Prototype:

```
long ConnectRX6(LPCTSTR Interface, long DevNum);
```

Arguments:

LPCTSTR	*Interface*	Interface to which the device is connected.		
		Argument	**Connection**	**Part #s**
		'GB'	Optical Gigabit	PO5/FO5
		'USB'	USB	UZ2, UB2, UZ1, UZ4
long	*DevNum*	Logical device number. Starts with 1 and counts up for each device of a specified type.		

Returns:

long	0	Connection not successful.
long	1	Connection successful.

Note: Invoke device connect commands only once to connect to a device, and then use ClearCOF and LoadCOF commands to upload or reload the control object to implement changes to the signal.

Sample Code:

MATLAB

```
%Connects to RX6 #1 via Optical Gigabit
RP=actxserver('RPco.x');
if RP.ConnectRX6('GB',1)
     e='connected'
else
     e='Unable to connect'
end
```

Visual Basic

```
'Connects to the RX6 via the Optical Gigabit
If RP.ConnectRX6("GB",1) Then
  msgbox "Connection established"
Else
  msgbox "Error connecting to RX6"
End If
```

Example: Circuit Loader, page 70.

ConnectRX7

Description: Establishes a connection with a MicroStimulator Base Station (RX7) through a device interface (such as Gigabit or USB).

'C' Prototype: `long ConnectRX7(LPCTSTR Interface, long DevNum);`

Arguments:

LPCTSTR	*Interface*	Interface to which the device is connected.

Argument	Connection	Part #s
'GB'	Optical Gigabit	PO5/FO5
'USB'	USB	UZ2, UB2, UZ1, UZ4

long	*DevNum*	Logical device number. Starts with 1 and counts up for each device of a specified type.

Returns:

long	0	Connection not successful.
long	1	Connection successful.

Note: Invoke device connect commands only once to connect to a device, and then use ClearCOF and LoadCOF commands to upload or reload the control object to implement changes to the signal.

Sample Code:

MATLAB

```
%Connects to RX7 #1 via Optical Gigabit
RP=actxserver('RPco.x');
if RP.ConnectRX7('GB',1)
     e='connected'
else
     e='Unable to connect'
end
```

Visual Basic

```
'Connects to the RX7 via the Gigabit
If RP.ConnectRX7("GB",1) Then
  msgbox "Connection established"
Else
  msgbox "Error connecting to RX7"
End If
```

Example: Circuit Loader, page 70.

ConnectRX8

Description: Establishes a connection with a Multi I/O Processor (RX8) through a device interface (such as Gigabit or USB).

'C' Prototype: `long ConnectRX8(LPCTSTR Interface, long DevNum);`

Arguments:

LPCTSTR	*Interface*	Interface to which the device is connected.

Argument	Connection	Part #s
'GB'	Optical Gigabit	PO5/FO5
'USB'	USB	UZ2, UB2, UZ1, UZ4

long	*DevNum*	Logical device number. Starts with 1 and counts up for each device of a specified type.

Returns:

long	0	Connection not successful.
long	1	Connection successful.

Note: Invoke device connect commands only once to connect to a device, and then use ClearCOF and LoadCOF commands to upload or reload the control object to implement changes to the signal.

Sample Code:

MATLAB

```
%Connects to RX8 #1 via Optical Gigabit
RP=actxserver('RPco.x');
if RP.ConnectRX8('GB',1)
     e='connected'
else
     e='Unable to connect'
end
```

Visual Basic

```
'Connects to the RX8 via the Optical Gigabit
If RP.ConnectRX8("GB",1) Then
  msgbox "Connection established"
Else
  msgbox "Error connecting to RX8"
End If
```

Example: Circuit Loader, page 70.

ConnectRZ2

Description: Establishes a connection with a RZ2 Base Station through a device interface (such as Gigabit or Optibit).

'C' Prototype: `long ConnectRZ2(LPCTSTR Interface, long DevNum);`

Arguments:

LPCTSTR	*Interface*	Interface to which the device is connected.

Argument	Connection	Part #s
'GB'	Optical Gigabit	PO5/PO5e

	'USB3'	USB 3.0 UZ3

long	*DevNum*	Logical device number. Starts with 1 and counts up for each device of a specified type.

Returns:

long	0	Connection not successful.
long	1	Connection successful.

Note: Invoke device connect commands only once to connect to a device, and then use ClearCOF and LoadCOF commands to upload or reload the control object to implement changes to the signal.

Sample Code:

MATLAB

```
%Connects to RZ2 #1 via Optical Gigabit
RP=actxserver('RPco.x');
if RP.ConnectRZ2('GB',1)
     e='connected'
else
     e='Unable to connect'
end
```

Visual Basic

```
'Connects to the RZ2 via the Optical Gigabit
If RP.ConnectRZ2("GB",1) Then
  msgbox "Connection established"
Else
  msgbox "Error connecting to RZ2"
End If
```

Example: Circuit Loader, page 70.

ConnectRZ5

Description: Establishes a connection with a RZ5 Base Station through a device interface (such as Gigabit or Optibit).

'C' Prototype:

```
long ConnectRZ5(LPCTSTR Interface, long DevNum);
```

Arguments:

LPCTSTR	*Interface*	Interface to which the device is connected.

Argument	Connection	Part #s
'GB'	Optical Gigabit	PO5/PO5e
'USB3'	USB 3.0	UZ3

long	*DevNum*	Logical device number. Starts with 1 and counts up for each device of a specified type.

Returns:

long	0	Connection not successful.
long	1	Connection successful.

Note: Invoke device connect commands only once to connect to a device, and then use ClearCOF and LoadCOF commands to upload or reload the control object to implement changes to the signal.

Sample Code:

MATLAB
```
%Connects to RZ5 #1 via Optical Gigabit
RP=actxserver('RPco.x');
if RP.ConnectRZ5('GB',1)
     e='connected'
else
     e='Unable to connect'
end
```

Visual Basic
```
'Connects to the RZ5 via the Optical Gigabit
If RP.ConnectRZ5("GB",1) Then
  msgbox "Connection established"
Else
  msgbox "Error connecting to RZ5"
End If
```

Example: Circuit Loader, page 70.

ConnectRZ6

Description: Establishes a connection with a RZ6 Base Station through a device interface (such as Gigabit or Optibit).

'C' Prototype:
```
long ConnectRZ6(LPCTSTR Interface, long DevNum);
```

Arguments:

LPCTSTR	*Interface*	Interface to which the device is connected.

Argument	Connection	Part #s
'GB'	Optical Gigabit	PO5/PO5e
'USB3'	USB 3.0	UZ3

long	*DevNum*	Logical device number. Starts with 1 and counts up for each device of a specified type.

Returns:

long	0	Connection not successful.
long	1	Connection successful.

Note: Invoke device connect commands only once to connect to a device, and then use ClearCOF and LoadCOF commands to upload or reload the control object to implement changes to the signal.

Sample Code:

MATLAB
```
%Connects to RZ6 #1 via Optical Gigabit
RP=actxserver('RPco.x');
if RP.ConnectRZ6('GB',1)
     e='connected'
else
     e='Unable to connect'
end
```

Visual Basic
```
'Connects to the RZ6 via the Optical Gigabit
If RP.ConnectRZ6("GB",1) Then
  msgbox "Connection established"
```

```
Else
  msgbox "Error connecting to RZ6"
End If
```

Example: Circuit Loader, page 70.

File and Program Control

About the File and Program Control Methods

The file and program methods are used to load or clear a COF (Control Object File), run the device's processing chain, or halt the device's processing chain.

File Methods

- ClearCOF
- LoadCOF
- LoadCOFsf
- ReadCOF

Program Control Methods

- Run
- Halt

ClearCOF

Description: Clears the program and data buffers on the processor.

'C' Prototype: `long ClearCOF;`

Returns:

long	0	Not successful.
long	1	Successful.

Sample Code:

Description: Clears the Control Object File (COF) and the data buffers on the processor device.

MATLAB
```
el=RP.ClearCOF
```

Visual Basic
```
errorl = RP.ClearCOF
```

Example: Circuit Loader, page 70.

LoadCOF

Description: Loads the Control Object File (*.rco or *.rcx) to the proper ActiveX control. This function/method is run after a Connectxx call and clears anything in the memory buffers on the processor device. See ReadCOF for information about

establishing a connection between an ActiveX handle and a COF without clearing a device's memory buffers.

Note: LoadCOF loads the Control Object File in real time allowing programs to utilize multiple Control Object Files if needed.

'C' Prototype:
```
long LoadCOF(LPCTSTR FileName);
```

Arguments:

LPCTSTR	**.rco file or *.rcx*	File name and extension

Note: the extension may be omitted for *.rco files but must be specified for *.rcx

Returns:

long	0	Not successful.
long	1	Successful.

Sample Code

Description: Loads a Control Object File(COF) i.e. *.rcx. and checks to see if it was properly loaded.

MATLAB
```
RP=actxserver('RPco.x');
RP.ConnectRP2('GB',1); % Connects to a RP2 via Optical
Gigabit
e=RP.LoadCOF('C:\Circuit.rcx'); % Loads circuit
if e==0
  disp 'Error loading circuit'
else
  disp 'Circuit ready to run'
end
```

Visual Basic
```
RP.ConnectRP2("GB",1)
If RP.LoadCOF("C:\Circuit.rcx") Then
 msgbox "File loaded and ready to run"
Else
 msgbox "Error loading *.rcx file to device"
```

Example: Circuit Loader, page 70.

LoadCOFsf

Description: Loads the Control Object File (*.rco or *.rcx) to the proper ActiveX control and sets the sampling frequency of the device. This function/method is run after a Connectxx call and clears anything in the memory buffers on the processor device. See ReadCOF for information about establishing a connection between an ActiveX handle and a COF without clearing a device's memory buffers.

'C' Prototype:
```
long LoadCOFsf(LPCTSTR FileName, float Sample
frequency);
```

Arguments:

LPCTSTR	**.rco file* or *.rcx	File name and extension **Note:** the extension may be omitted for *.rco files but must be specified for *.rcx
float	*Sample Frequency*	Values above 50 are used for arbitrary waveform generation.

0	1	2	3	4	5	6	>=50
6K	12K	25K	50K	100K	200K	400K	Arbitrary Sample rate
ALL	ALL	ALL	RP2 RP2.1 RL2 RV8 RX6 RX8 RZ2 RZ5 RZ6 D/A Only RA16BA RX5	RP2 RP2.1 RL2 RV8 RX6 RX8 RZ6 D/A Only RX5	RP2 RP2.1 RV8 RX6 RZ6	RV8 RX6	RV8 RX6 RX8

TechNotes: The sample frequencies are approximate and are subject to round-off error. Use GetSFreq to determine the actual sample rate.

Choosing a number greater than the maximum sample frequency for an RPx device will set that device to its maximum sample rate (for example: the maximum sample rate for an RL2 is 50 kHz (3) if the sample rate is set to 6 the devices sample rate will be 50 kHz).

PCM A/D and D/A equipped devices such as the RV8 and RX8 allow arbitrary rates to be specified. The PCM converters on these devices will adjust to the sampling rate specified without corrupting data. Sigma-Delta A/D and D/A equipped devices such as the RX6 and RX8 must specify supported realizable sampling rates in order to avoid data corruption. For more information on the realizable sampling rates supported by the Sigma-Delta converters, see Realizable Sampling Rates for the RX6. RX8 devices equipped with Sigma-Delta converters should use the realizable sampling rates up to a maximum of 97.65625 kHz.

To use the arbitrary sample frequency on the RV8, RX6, or RX8 use a value greater than 50 for the sample frequency.

Setting the sample frequency for values greater than 6 and less than 50 will generate incorrect sample rates and the circuit will fail to run.

Returns:

long	0	Not successful.
long	1	Successful.

Sample Code

Description: Loads a Control Object File(COF), sets the sample rate to 200 kHz i.e. *.rco, and checks to see if it was properly loaded. Also returns the true sample rate of the Device.

[MATLAB] In addition it loads the same COF file and sets the sampling rate to 200 Hz on an RV8

MATLAB

```
RP=actxserver('RPco.x');
% Connects to an RP2 via Optical Gigabit
RP.ConnectRP2('GB',1);
% Loads circuit sets sample rate to 200 kHz
```

```
e=RP.LoadCOFsf('C:\Circuit.rcx',5);
SFreq=RP.GetSFreq
if e==0
  disp 'Error loading circuit'
else
  disp 'Circuit ready to run'
end
```

```
RV8=actxserver('RPco.x');
RV8.ConnectRV8('GB',1);
e=RV8.LoadCOFsf('C:\Circuit.rcx',200)
```

Visual Basic

```
RP.ConnectRP2("GB",1)
If RP.LoadCOFsf("C:\Circuit.rcx",5) Then
  msgbox "File loaded and ready to run"
Else
  msgbox "Error loading *.rco file to processor
device"
End If
SFreq=RP.GetSFreq
```

ReadCOF

Description: Reads the Control Object File (*.rco or *.rcx) to the proper ActiveX control. This function gives the ActiveX handle access to circuit components and parameters without reloading the circuit or clearing the memory buffers on the device. If the ReadCOF file is not the same as the circuit running on the device, the data will be erroneous. This function is primarily for use with portable or remote processor devices such as the stingray Pocket Processor.

'C' Prototype:

```
long ReadCOF(LPCTSTR FileName);
```

Arguments:

LPCTSTR	**.rco or *.rcx file*	File name (the extension does not need to be included).

Returns:

long	0	Not successful.
long	1	Successful.

Sample Code

Description: Reads a Control Object File(COF) i.e. *.rco or *.rcx.

MATLAB

```
RP=actxserver('RPco.x');
% Connects to an RP2 via Optical Gigabit
RP.ConnectRL2('GB',1);
% Reads circuit to ActiveX handle`
e=RP.ReadCOF('C:\Circuit.rcx');
```

Visual Basic

```
RP.ConnectRP2("GB",1)
RP.ReadCOF("C:\Circuit.rcx")
```

Run

Description: Starts the processor device processing chain. Run should be called after a Connect call and LoadCOF.

'C' Prototype:

```
long Run;
```

Arguments: None.

Returns:

long	0	Not successful.
long	1	Successful.

Sample Code

Description: Goes through the connection, load and run procedure and checks to see if the circuit is running.

MATLAB

```
RP=actxserver('RPco.x');
RP.ConnectRP2('GB',1)
RP.LoadCOF('C:\Circuit.rcx');
e=RP.Run
if e==0
  disp 'error running circuit'
else
  disp 'Circuit running'
end
```

Visual Basic

```
error1 = RP.ConnectRP2("GB",1)
If error1 = 0 Then
  error1 = RP.ConnectRP2("USB", 1)
01If RP.LoadCOF("C:\Circuit.rcx") Then
  msgbox "File loaded"
Else
  msgbox "Error loading *.rcx file to device"
End If
If RP.Run Then
  msgbox "device circuit active and running"
Else
  msgbox "device circuit failed to run"
End If
```

Example: Circuit Loader, page 70.

Halt

Description: Stops the processor device's processing chain.

'C' Prototype:

```
long Halt;
```

Returns:

long	0	Not successful.
long	1	Successful.

Sample Code

Description: Stops the processor device's processing chain.

MATLAB

```
e1 = RP.Halt
```

Visual Basic

```
error1 = RP.Halt
```

Device Status

About the Device Status Methods

The device status methods return information to specific device characteristics such as the device's status, cycle usage, sampling frequency, number of total components in the COF file, and the names of any of the COF file components.

Device Status Methods

- GetStatus
- GetCycUse
- GetSFreq
- GetNumOf
- GetNameOf

GetStatus

Description: Checks the status of the device and reports the state of various status indicators. A 0 or 1 is reported for each indicator and each indicator is reported as a single bit in a binary number. The binary number, including information about all possible indicators, is returned as an integer.

While each device type can have different status indicators, the first three bits return the same basic status information about the connection and circuit status on all devices. The return values in the table below are possible for the first three status bits on all devices. However, use bit-wise operations (0/1) instead of inspecting the integer value for best results.

Bitmasks remain constant while integer values change as new Bitmasks are added to GetStatus() in the future.

Integer	0	1	3	5	7
Binary	000	001	011	101	111
Status	Nothing	Connected	Connected and loaded	Connected and running	Connected, loaded, and running

'C' Prototype: `long GetStatus;`

Arguments: None

Returns: long

Return Value (Enabled)	Status	Bitmask	Bit#	Device
1	Connected	0000000000000001	0	All
2	Circuit loaded	0000000000000010	1	All
4	Circuit running	0000000000000100	2	All

8	Battery	0000000000001000	3	RA16BA
16	Amplifier clipping on one or more channels	0000000000010000	4	RA16BA
32	Amplifier clipped since last call to GetStatus	0000000000100000	5	RA16BA
64	System Armed	0000000001000000	6	RV8
128	Circuit running (not waiting for trigger)	0000000010000000	7	RV8
256	Trigger enable	0000000100000000	8	RV8
512	Auto Clear DAC outs	0000001000000000	9	RV8
1024	Tick out	0000010000000000	10	RV8
2048	Clock out	0000100000000000	11	RV8
4096	zTrigA	0001000000000000	12	RV8
8192	zTrigB	0010000000000000	13	RV8
16384	External trigger	0100000000000000	14	RV8
32768	Multiple trigger	1000000000000000	15	RV8

Note: When checking the status of the Medusa Base Station (RA16BA), ensure that a preamplifier is properly connected and turned on. Connection status (Bit 0) will always return a 0 when a preamplifier is not properly connected. Bit 5 (amplifier clipped since last call) is reset after GetStatus is called.

Bit-0 does not report preamplifier status when using an RZ base station. Use the RZ LCD screen to determine PZ status.

Sample Code

Description: Checks if the circuit is loaded and running. Determines where in the loading routine the error occurred.

MATLAB

```
Status = double(RP.GetStatus); % Gets the status
% Checks for errors in starting circuit
if bitget(Status,1) == 0;
  er = 'Error connecting to RP'
elseif bitget(Status,2) == 0; % Checks for connection
  er = 'Error loading circuit'
elseif bitget(Status,3) == 0
  er = 'error running circuit'
else
  er = 'Circuit loaded and running'
end
```

Visual Basic

```
Status = RP.GetStatus
If (status And 7) = 7 Then MsgBox "System is running"
End If
```

Example: Circuit Loader, page 70.

GetCycUse

Description: Checks the total cycle usage of a specified processor device. GetCycUse polls the processor device and returns an integer value between (0-100).

Note: If the value returned is greater than 100, the value will fold back within the 0-100 range (for example, a cycle usage of 130% would return a value of 30). To determine if cycle usage is too high, lower the sampling rate by a factor of 2. The cycle usage should be one-half the former value. (For example, if GetCycUse returns a value of 30, halving the sample rate should reduce the cycle usage to 15%. If, after halving the sample rate, the cycle usage is 65, you know that the original cycle usage was 130% not 30%.)

'C' Prototype:
```
long GetCycUse;
```

Arguments: None.

Returns: long — Percent cycle usage.

Sample Code

Description: Warns if the cycle usage is over 90%.

MATLAB
```
if RP.GetCycUse < 90
  disp 'System within cycle usage limits'
else
  disp 'Warning: reaching upper limits of cycle usage'
end
```

Visual Basic
```
If RP.GetCycUse > 90 Then
  msgbox "Warning Cycle usage levels are to high"
End If
```

Example: Device Checker, page 71.

GetSFreq

Description: Returns the exact sampling frequency of the processor device.

'C' Prototype:
```
float GetSFreq;
```

Arguments: None

Returns: float — Sampling frequency.

Sample Code

Description: Checks the sampling frequency and warns if a tone frequency is below the nyquist value of the circuit.

MATLAB
```
if ToneFreq > RP.GetSFreq/2
  disp 'Tone above Nyquist value'
else
  disp 'Tone Freq below Nyquist'
end
```

Visual Basic
```
If ToneFreq>RP.GetSFreq/2 Then
  msgbox "Warning: Tone frequency above nyquist value"
End If
```

GetNumOf

Description: Returns the number of components, parameter tags, parameter tables, or SrcFiles in a *.rco file.

'C' Prototype: `long GetNumOf(LPCTSTR Name)`

Arguments:

LPCTSTR *Name* A string indicating the desired object type.

STRING Name	Component or Helper Type
"Component"	Number of processor device components
"ParTable"	Number of Parameter (Data) tables
"SrcFile"	Number of Source files (Data) files
"ParTag"	Number of Parameter Tags

Returns: long An integer equal to the number of objects of the specified type.

Sample Code

Description: Finds the number of Parameter Tags and returns their StringID

MATLAB

```
TagNum = double(RP.GetNumOf('ParTag'))
for loop=1:TagNum
  TagName{loop} = RP.GetNameOf('ParTag', loop)
end
```

Visual Basic

```
Dim TagNum As Integer
Dim TagName(100) As String*25
TagNum = RP.GetNumOf("ParTag")
For i = 1 to TagNum
  TagName(i)=RP.GetNameOf("ParTag", i)
Next i
```

Example: Device Checker, page 71.

GetNameOf

Description: Returns the name given to a particular parameter tag, component, data table, or source file in a processor device chain. The string 'NoName' will be returned if the object was not explicitly named in the RPvdsEx circuit. This function can be used in conjunction with GetNumOf() to return a list of all parameter tags in an RCO file.

'C' Prototype: `CString GetNameOf(LPCTSTR Name, long Component_#)`

Arguments:

LPCTSTR *Name* A string indicating the desired object type.

STRING Name	Component Type
"Component"	processor components

"ParTable"	Parameter (Data) tables
"SrcFile"	Source (Data) files
"ParTag"	Parameter Tags

long	*Component_#*	The number assigned to the component in the processing chain

Returns:

CString	String ID	The String ID of the component

Sample Code

Description: Finds the number of parameter tags and returns their source name.

MATLAB

```
TagNum=double(RP.GetNumOf('ParTag'))
for loop=1:TagNum
  TagName{loop} = RP.GetNameOf('ParTag', loop)
end
```

Visual Basic

```
Dim TagNum As Integer
Dim TagName(100) As String*25
TagName=RP.GetNumOf("ParTag");
For i = 1 to TagNum
  TagName(i) = RP.GetNameOf("ParTag", i)
Next i
```

Example: Device Checker, page 71.

Tag Status and Manipulation

About the Tag Status and Manipulation Methods

The tag status and manipulation methods are used to read in values of the COF (Control Object File) file's tags or write values to the tags themselves.

Tag Status Methods

- GetTagVal
- GetTagType
- GetTagSize
- ReadTag
- ReadTagV
- ReadTagVEX

Tag Manipulation Methods

- SetTagVal
- WriteTag
- WriteTagV
- WriteTagVEX
- ZeroTag

GetTagVal

Description: Returns the value of a specified parameter tag. Because parameter tags point to a parameter input or output, GetTagVal provides a means of determining the current value of a parameter. It can be used with all parameter types and returns a single floating point value.

'C' Prototype:
```
float GetTagVal(LPCTSTR Name)
```

Arguments:

LPCTSTR	*Name*	A string variable that matches exactly the name of a parameter tag.

Returns:

float	current value of tag	The numerical type of the parameter does not affect the return variable.

Sample Code

Description: Reads value of tag labeled 'RMS' and saves it to the variable rms.

MATLAB
```
rms = RP.GetTagVal('RMS'); % Reads rms level
```
Visual Basic
```
Dim rms As single
rms = RP.GetTagVal("RMS") 'Reads rms level
```
Visual C++
```
float rms;
rms = RP.GetTagVal("RMS"); //Reads rms level
```

Examples: Variable Band-Pass filter, page 72.

Continuous Play, page 77.

Continuous Acquire, page 74.

Two Channel Continuous Acquisition, page 81.

GetTagType

Description: Determines the data type of a parameter tag.

'C' Prototype:
```
long GetTagType(LPCTSTR Name)
```

Arguments:

LPCTSTR	*Name*	The name of a parameter tag.

Returns:

MATLAB — long — An Integer that maps to an ASCII character.

Data Type	Integer Value
Data Buffer	68
Integer	73
Logical (1 or 0)	78
Float(Single)	83
Coefficient Buffer	80
Undefined (e.g. latch output)	65

Visual Basic — char — An ASCII character.

Data Type	Ascii Map
Data Buffer	"D"

Integer	"I"
Logical (1 or 0)	"L"
Float(Single)	"S"
Coefficient Buffer	"P"
Undefined (e.g. latch output)	"A"

Sample Code

Description: Finds the data type of a particular parameter tag.

MATLAB
```
DataType = char(RP.GetTagType('RAMBuffer'));
```
Visual Basic
```
DataType = char(RP.GetTagType("RamBuffer"))
```

Example: Device Checker, page 71.

GetTagSize

Description: Returns the maximum number of data points accessible through the parameter tag.

'C' Prototype:
```
long GetTagSize(LPCTSTR Name)
```

Arguments:

LPCTSTR	*Name*	A string variable that matches the name of a parameter tag.
Returns:	long	0= error, 1=Logic, Integer, Float (Single),>1 Data type (Pointer to a buffer).

Sample Code

Description: Returns the number of points in the ram buffer.

MATLAB
```
Tagsize = RP.GetTagSize('RAMBuffer');
```
Visual Basic
```
Dim DataType As Integer
DataType = RP.GetTagSize("RamBuffer")
```

Example: Device Checker, page 71.

ReadTag

Description: Reads data from the processor device's memory into variables stored on the PC. ReadTagV should be used with MATLAB. Other programming languages should use ReadTag. See ReadTagVEX for alternative ways to read data.

ReadTag can be used with any component that has a data buffer, such as: RamBuffer, LongDynDel, FIR and so forth.

'C' Prototype:
```
long ReadTag(LPCTSTR Name, float* pBuf, long nOS, long nWords);
```

Arguments:

LPCTSTR	*Name*	Name of parameter tag.
float*	*pBuf*	Pointer to buffer to receive data.
long	*nOS*	Number of points to offset in buffer before starting read.
long	*nWords*	Number of 32-bit words to read (Samples).

Returns:

long	0	Not successful.
long	1	Successful.

Sample Code

Description: Reads 1000 points from a buffer (parameter tag labeled "datain") and stores it in a single array file (data).

Visual Basic 6

```
Dim data(0 to 999) As single
e1=RP.ReadTag("datain", data(0),0,1000)
```

Description: Reads 1000 points from parameter tag labeled 'datain' to floating point array called data.

Visual C++

```
float data[1000];
char Name[10];
Name = "datain";
ReadTag(Name, data, 0, 1000);
```

ReadTagV

Description: Reads variables stored in the processor device's memory into a PC buffer in variant format. ReadTagV should be used with MATLAB. Other programming languages should use ReadTag. See ReadTagVEX for alternative storage methods.

ReadTagV can be used with any component that has a data buffer, such as: RamBuffer, LongDynDel, FIR and so forth.

'C' Protoype:

```
VARIANT ReadTagV(LPCTSTR Name, long nOS, long nWords);
```

Arguments:

LPCTSTR	*Name*	Name of parameter tag.
long	*nOS*	Number of points to offset in buffer before starting read.
long	*nWords*	Number of 32-bit words to read (Samples).

Returns:

Variant	-1, or empty	Not successful.
Variant	Array	Successful.

Sample Code

Description: Reads 1000 points from a buffer (parameter tag 'datain') and stores it in an array file (Data_A) as variant values.

MATLAB

```
Data_A = RP.ReadTagV('datain',0,1000);
```

Example: Continuous Acquire, page 74.

ReadTagVEX

Description: Reads data that has been written to a parameter tag and stored on the processor device. Data can be converted to one of five data formats (double, float, 32-, 16-, 8-bit Integer) and stored as either an array or a matrix. The user must specify the storage format of the data to be read (F32, 32-, 16-, or 8-bit Integer) and the number of channels.

When used to read compressed or shuffled data ReadTagVEX handles data manipulation and storage. Shuffled data is separated into channels and stored in a matrix. The *nWords* argument must be set to the number of samples in the serial buffer and is used along with *nchannels* to unshuffle or expand the data. For compressed data *nWords* must be set to the number of points after compression. e.g. If the data is compressed two-folded then only 500 samples of a 1000 point signal are contained in the serial buffer and *nWords* should be set to 500 (for a compression of 4 the number of points in the buffer would be 250).

ReadTagVEX is used with components that have a data Buffer, including: RamBuffer, Serial Buffer, Average Buffer, LongDelay, LongDynDelay, ShortDelay, ShortDynDelay, Biquad, IIR, FIR, HrtfFir.

Note: ReadTagVEX and WriteTagVEX are the only read/write commands that will work in languages other than MATLAB, VB6, and VC++

'C' Protoype:

```
VARIANT ReadTagVEX(LPCTSTR Name, long nOS, long
nWords, LPCTSTR stype, LPCTSTR dtype, long nchannels);
```

Arguments:

LPCTSTR	*Name*	Name of parameter tag.
long	*nOS*	Number of points to offset in buffer before starting read.
long	*nWords*	Number of 32-bit words to read (Samples).
LPCTSTR	*Srctype*	Format type of data being read. Below is a list of the storage types.

Floating Point (32-bit)	Word (32-bit)	Integer (16-bit)	Byte (8-bit)
F32	I32	I16	I8

LPCTSTR	*Dsttype*	Format for storing data. MATLAB handles data as doubles. All other languages use a variety of formats.

Double(64-bit float)	Float (32-bit)	Word (32-bit)	Integer(16-bit)	Byte(8-bit)
F64	F32	I32	I16	I8

long	*nchannels*	Number of data channels (1-4). For compressed and standard it is 1. For Shuffled data it is 2 or 4.

Returns:

Variant	-1, or empty	Not successful.
Variant	Array	Successful.

Sample Code

MATLAB

Description: Reads 1000 points from a processor device buffer (either compressed or Standard format) and stores it in an array of 1000 points in double format.

```
Data_A=RP.ReadTagVEX('datain',0,1000,'I16','F64',1);
```

Description: Reads 1000 points from a processor device buffer that contains a shuffled data set (2-channels) and stores it in a matrix (2,500) in double format.

```
Data_A=RP.ReadTagVEX('datain',0,500,'I16','F64',2);
```

Visual Basic

Description: Reads 1000 points from a processor device buffer and stores it in an array of 1000 points in 16-bit Integer format.

```
Data_A=RP.ReadTagVEX("datain",0,1000,"I16","I16",1)
```

Description: Reads 1000 points from a processor device buffer that contains a shuffled data set (2-channels) and stores it in a matrix (2,500) in 16-bit integer format.

```
Data_A=RP.ReadTagVEX("datain",0,500,"I16","I16",2)
```

Example: Two Channel Continuous Acquisition, page 81.

SetTagVal

Description: Sets the value of the specified parameter tag.

'C' Prototype:

```
long SetTagVal(LPCTSTR Name, float Val)
```

Arguments:

LPCTSTR	*Name*	Name of a parameter tag.
float	*Val*	Parameter tag value.

Returns:

long	0	Not successful.
long	1	Successful.

Sample Code

Description: Sets the parameter Tag value to that of the variable "rms".

MATLAB

```
rms=5.0
e1=RP.SetTagVal('RMS', rms); % Set RMS Level
```

Visual Basic

```
rms=5.0
e1=RP.SetTagVal("RMS", rms) 'Set RMS Level
```

Visual C++

```
float rms=5.0;
RP.SetTagVal("RMS", rms); //Set RMS Level
```

Example: Band-Limited Noise, page 72.

WriteTag

Description: Writes data from the PC to a memory buffer pointed to by a parameter tag. WriteTagV should be used with MATLAB. Other programming languages should use WriteTag. See WriteTagVEX for alternative methods of writing data.

WriteTag is used with the following components that have a data Buffer: RamBuffer, Serial Buffer, Average Buffer, LongDelay, LongDynDelay, ShortDelay, ShortDynDelay, Biquad, IIR,FIR, HrtfFir.

'C' Prototype:

```
long WriteTag(LPCTSTR Name, float* pBuf, long nOS,
long nWords);
```

Arguments:

LPCTSTR	*Name*	Name of parameter tag.

float*	*pBuf*	Floating point array holding data to load to the processor device's memory.
long	*nOS*	Number of points to offset in the processor device's memory before starting write.
long	*nWords*	Number of 32-bit words to write.

Returns:

long	0	Not successful.
long	1	Successful.

Sample Code

Description: Writes 1000 points from an array named 'data' to a memory buffer on the processor device (parameter tag labeled 'datain').

Visual Basic 6
```
Dim data(0 to 999) As single
el=RP.WriteTag("datain", data(0), 0, 1000)
```

Visual C++
```
float data[1000];
char Name[10];
Name = "datain"; // fill data array with data to load
RP.WriteTag(Name, data, 0, 1000);
```

WriteTagV

Description: Writes variables from the PC to a memory buffer on the processor device. WriteTagV should be used with MATLAB. Other programming languages should use WriteTag. WriteTagV is designed to take data in a standard MATLAB row vector. Column vectors should be transposed.

WriteTagV is used with the following components that have a data Buffer: RamBuffer, Serial Buffer, Average Buffer, LongDelay, LongDynDelay, ShortDelay, ShortDynDelay, Biquad, IIR,FIR, HrtfFir.

Note: WriteTagV is to be used in Matlab *only* with data type double. Attempting to write vectors of any other type will fail and return a zero.

See WriteTagVEX for alternative methods of writing vectors of all other data types.

'C' Prototype:
```
long WriteTagV(LPCTSTR Name, long nOS, Variant
&buffer);
```

Arguments:

LPCTSTR	*Name*	Name of parameter tag.
long	*nOS*	Number of points to offset in buffer before starting write.
Variant	*&buffer*	Data array with the samples.

Returns:

long	0	Not successful.
long	1	Successful.

Sample Code

MATLAB

Description: Writes 10000 points from an array (data) to a memory buffer on a processor device (pointed to by the parameter tag (datain)).

```
e1=RP.WriteTagV('datain', 0, data(1000:11000));
```

Description: Writes 1000 points from an array (data) to a memory buffer on a processor device (pointed to by the parameter tag (datain)).

```
e1=RP.WriteTagV('datain', 0, data(0:1000));
```

Example: Continuous Play, page 77.

WriteTagVEX

Description: WriteTagVEX writes data stored in array or matrix format to a memory buffer on the processor device. The data format for storage in the memory buffer can be one of the following: 32-bit Float, 32-,16-, and 8-bit Integer formats. In addition, data is not limited to a single array format. The organization of variables stored in a matrix is preserved.

WriteTagVEX is used with the following components that have a data Buffer: RamBuffer, Serial Buffer, Average Buffer, LongDelay, LongDynDelay, ShortDelay, ShortDynDelay, Biquad, IIR, FIR, and HrtfFir.

Note: ReadTagVEX and WriteTagVEX are the only read/write commands that will work in languages other than MATLAB, VB6, and VC++

'C' Protoype:

```
long WriteTagVEX(LPCTSTR Name, long nOS, LPCTSTR
dtype, Variant &buffer);
```

Arguments:

LPCTSTR	*Name*	Name of parameter tag.
long	*nOS*	Number of points to offset in buffer before starting write.
LPCTSTR	*dtype*	One of four data types that the data is stored in.

Floating Point(32-bit)	Word(32-bit)	Integer(16-bit)	Byte(8-bit)
F32	I32	I16	I8

Variant	&buffer	Data array/matrix with the samples.

Returns:

long	0	Not successful.
long	1	Successful.

Sample Code

MATLAB

Description: Writes 10000 points from an array to a memory buffer on a processor device in floating point format.

```
e1=RP.WriteTagVEX('datain', 0, 'F32',
data(1000:11000));
```

Description: Writes 2000 points from a matrix (data) to a memory buffer on the processor device (pointed to by the parameter tag (datain) in integer format (16-bit).

```
e1=RP.WriteTagVEX('datain', 0, 'I16',
data(1:2,1:1000));
```

Visual Basic

Description: Writes 2000 points from a matrix (data) into a data buffer on a processor device (pointed to by parameter tag datain).

```
Dim data(1 to 2,0 to 999) As Variant
e1=RP.WriteTagVEX("datain", 0, "I16", data)
```

Description: Writes a 1000 points from an array as float variables to a data buffer on a processor device.

```
Dim data(0 to 999) As Variant
e1=RP.WriteTagVEX("datain", 0, "F32", data)
```

Example: Two-channel Playback, page 83.

ZeroTag

Description: Sets a parameter tag value to zero. When the parameter tag points to a memory buffer, all values in the buffer are set to zero.

'C' Prototype:

```
long ZeroTag(LPCTSTR Name);
```

Arguments:

LPCTSTR	*Name*	Name of parameter tag.

Returns:

long	0	Not successful.
long	1	Successful.

Sample Code

Description: Sets membuf values to zero.

MATLAB
```
error1=RP.ZeroTag('membuf')
```

Visual Basic
```
error1=RP.ZeroTag("membuf")
```

Other

GetDevCfg

Description: GetDevCfg is used with the RV8. After setting the number of sweeps with SetDevCfg, you can use this function to determine the number of sweeps remaining on the RV8. At this time, only the information pertaining to the remaining sweeps can be retrieved from the device.

'C' Prototype:

```
long GetDevCfg(long Address, long Wide32)
```

Arguments:

long	*address*	Position of a particular data value.

Address	**Configuration information**
9	Sweep Count

long	*Wide32*	Set Wide32 = 0

Returns: long — The value at the memory location.

Sample Code

Description: Finds the number of sweeps left on the RV8.

MATLAB
```
Sweeps_Left = RP.GetDevCfg(9, 0);
```
Visual Basic
```
Sweeps_left = RP.GetDevCfg(9, 0)
```

SetDevCfg

Description: SetDevCfg is used with the RV8. It allows direct access to memory locations for the control of the RV8 special modes, sample number, trigger counter and bit logic.

'C' Prototype:
```
long SetDevCfg(long Address, long Value, long Wide32)
```

Arguments:

long	*address*	Position of a device configuration value.
long	*value*	Sets the value of the device.
long	*Wide32*	Setting Wide32=1 enables modification of the upper and lower registers of the sample counter simultaneously.

Returns:

long	1	Successful.
long	0	Not successful.

Tech Notes:

Address ***Configuration information***

0 Special Mode value for the RV8. The bitmask for the special mode is as follows: The top row is the bit number, the middle row contains the integer value for setting the bit number, and the bottom row describes the Configuration property.

0	1	2	3	4	5	6	7
1	2	4	8	16	32	64	128
Trigger Enabled	AutoClr DACs	Tick Out	Clk Out	UseZtrigA	UseZTrigB	Ext Trig	Multiple Trigger

1 Integer value allows user to set sample rate. Make sure the RV8 is halted before using.

2 CountLo. The Lower 16-bits of the Sample Counter. Note use Wide32 to write to the upper and lower counter simultaneously.

3 CountHi. The Upper 16-bits of the Sample Counter. See Note above

09 Sweep Count. Sets the number of times the RV8 can be triggered in mTrig mode.

0a/10 OutLogic: Sets the value for a logical high. The default value for each output channel is 0 (logical high = 1 or 'high true'). Setting OutLogic = 1 inverts the logic (logical high = 0 or 'low true').

0b/11 InLogic: Sets the value for a logical high. See OutLogic for a description.

Enabling the Trigger Mode

The Trigger mode requires that you set two components of the Special Mode: Bitmask 1 and one of the Three trigger types (zBUSA, zBUSB or External). Note only one of the three trigger types

(zBUSA, zBUSB or External) can be enabled at any time. Additional modes that might be enabled are multiple trigger and AutoClr.

Multiple trigger allows users to trigger the RV8 with out halting and running the chain again. In addition It allows users to set the maximum number of times a system can be triggered. To set the Multiple Trigger requires that you also set the Sweep Count. Sweep Count can be set to any value between 1 and 4,294,967,296.

AutoClr: AutoClr sets the DAC outs to 0. If AutoClr is not set the last value sent to the DAC's is played out.

Setting the Sample Count

In the Trigger mode the sample count needs to be a value greater than zero otherwise the signal will play for a long time. There are two ways to set the Sample Count. The Lower and Upper Count addresses can be set separately or by setting wide32=1 in the SetDevCfg it allows users to set the value for both upper and lower addresses. TDT recommends that wide32 be used to set the value. The example below shows the difference.

Setting the Sample Count for 300,000 with wide32. In this case it is a matter of using the actual value.

```
RV8.SetDevCfg(2,300000,1)
```

Setting the Sample Count for 300,000 without wide32.

300,000 needs to be converted to a hexadecimal value and then split into the lower and upper 16-bit values. In this cause the lower 16-bit value is 37,856. The Upper 16-bit value is 4.

```
RV8.SetDevCfg(2,37856,0)
RV8.SetDevCfg(3,4,0)
```

Setting Multiple Triggering

In Single Trigger mode the circuit needs to be halted and run after each trigger. In Multiple trigger mode the circuit can be configured to be triggered several times before the circuit needs to be halted. The code below sets the circuit to trigger 5 times before it needs to be reset. The maximum number of times a circuit can be triggered is by setting this variable is 65535. If sweep count is set to 0 (default) the circuit will trigger a near infinite number of times.

```
RV8.SetDevCfg(09,5,0)
```

Using the zTRIG Option

To use the zTRIGB or UsezTRIGA option you need to use the zBUS ActiveX controls. Your code should include a connection to the zBUS. The example code below shows how this would work. Make sure that the ActiveX control is active in your program. Note it is not necessary to have a Trigger component in the circuit.

MATLAB

```
zBUS.ConnectZBUS('GB')
RV8.ConnectRV8('GB',1)
RV8.LoadCOF('C:\Circuit.rcx')
%Triggers off zBUSA, AutoClr, and multiple trigger
RV8.SetDevCfg(0,147,0)
RV8.SetDevCfg(2,30000,1) %Plays out 30000 samples
%Triggers up to 10 times before stopping
RV8.SetDevCfg(9,10,1)
RV8.Run
zBUS.zBUSTrigA(0,0,10) %Triggers the RV8
%Returns the number of sweeps left
sweepcount=RV8.GetDevCfg(9,0)
```

SoftTrg

Description: Sends a software trigger to the processor device. There are ten software triggers for each processor device.

Note: Do not use software triggers for signal generation or acquisition that requires precise timing. Software triggers are affected by USB transfer times. Expect a 2-4 ms delay for each call to the processor device from the SoftTrg(). If multiple devices need to be triggered simultaneously use zBusTrigA/B().

'C' Prototype:
```
long SoftTrg(long Trg_Bitn);
```

Arguments:

long	*Trg_Bitn*	Software trigger number to send.

Returns:

long	0	Not successful.
long	1	Successful.

Sample Code

MATLAB

Description: This starts one of ten possible software triggers.

```
error1=RP.run
error2=RP.SoftTrg(1)
```

Visual Basic

Description: This starts one of ten possible software triggers. It then starts another software trigger.

```
error1=RP.run
error2=RP.SoftTrg(1)
error3=RP.SoftTrg(10)
```

Examples: Continuous Play, page 77.

Continuous Acquire, page 74.

Two Channel Continuous Acquire, page 81.

SendParTable

Description: Sends data from a DataTable to its output.

'C' Prototype:
```
long SendParTable(LPCTSTR Name, float IndexID);
```

Arguments:

LPCTSTR	*Name*	Name of DataTable component (not a parameter tag).
float	*IndexID*	ID number of column of data to send.

Returns:

long	0	Not successful.
long	1	Successful.

Sample Code

Description: Cycles through three filters for the same input. Allows changes in the filter coefficients from a data table.

MATLAB
```
for n = 1:3
```

```
    e1 = RP.SendParTable('PTab', n);
    if e1==0
      disp 'Filter incorrectly loaded'
    else
      disp ['Filter' num2str(n) ' loaded']
  end
```

Visual Basic

```
  For n = 1 To 3
    e1 = RP2.SendParTable("PTab", n)
    If e1 = 0 Then
      msgBox "Filter incorrectly loaded"
    Else
      msgBox "Filter " & n & " loaded"
    End If
```

Example: FIR filtered noise, page 80.

SendSrcFile

Description: Sends data from a data file (specified in a SourceFile Component) into the processing chain. This allows programmers to load a data file from the PC directly to a RAM buffer.

Tech Notes: SourceFile supports the following data types: Float Point (32-bit), Long Int(32 bit), Int (16-bit), Ascii, and Wav formats.

SendSrcFile gives you control over the size of data transferred and the position in the data file that SendSrcFile starts. A file can contain many waveforms that are played at different times or in different circumstances.

16-bit words are padded to fit the 32-bit format of the Data Buffers.

Note that this method does not let you specify a new filename to load. This can only be done in the RPvds circuit. If you need to load data from different files, you would first load it into a PC buffer and then use WriteTag() to send the data to a buffer on the processor device.

'C' Prototype:

```
long SendSrcFile(LPCTSTR Name, long SeekOS, long nWords)
```

Arguments:

LPCTSTR	*Name*	Name of DataFile Component in RPvdsEx circuit (not a parameter tag).
long	*SeekOS*	Position in the Data file to start writing to the buffer.
long	*nWords*	Number of 32-bit words to send.

Returns:

long	0	Not successful.
long	1	Successful.

Sample Code

Description: This code finds the number of SrcFiles, gets the String ID of the last SrcFile and sends a portion of the PC data file to the processor device.

MATLAB

```
SrcFile1=RP.GetNumof('SrcFile')
SFile=RP.GetNameOf('SrcFile', SrcFile1)
test=RP.SendSrcFile(SFile, 1000, 50000);
```

Visual Basic

```
SrcFile1=RP.GetNumof("SrcFile")
SFile=RP.GetNameOf("SrcFile", SrcFile1)
```

```
test=RP.SendSrcFile(SFile, 1000, 50000);
```

PA5 Programmable Attenuator

About the PA5x Methods

This section provides a listing of the available PA5x ActiveX control methods.

Programming Steps

- Add the PA5x ActiveX controller to your program. The ActiveX help has examples for setting up ActiveX controllers in MATLAB, Visual Basic, and Visual C.
- Connect to a PA5 (USB or GB) device with the connectPA5 function.
- Control the PA5 with the command and control functions using the ActiveX controller.

ConnectPA5

Description: Establishes a connection with the specified device. The connection is established through either the Optical Gigabit or USB interface. Invoking this method causes the control to search for the specified device and establish a handle to the associated device driver. The method will return a '1' if a connection was successfully established or a '0' if the device is not present or if it is not functioning properly.

'C' Prototype:

```
Long ConnectPA5(LPCTSTR Interface, long DevNum);
```

Arguments:

LPCTSTR	*Interface*	Interface to which the device is connected.

Argument	**Connection**	**Part #s**
'GB'	Optical Gigabit	PO5/PO5e
'USB'	USB	UZ2, UB2, UZ1, UZ4

long	*DevNum*	Logical device number. Starts with 1 and counts upward for each device of a specified type.

Returns:

long	0	Connection not successful.
long	1	Connection successful.

Sample Code

MATLAB

Description: Connects to PA5#1 via Optical Gigabit

```
% Connects to PA5 #1 via Optical Gigabit
PA5x1=actxserver('PA5.x');
if PA5x1.ConnectPA5('GB',1)==1
  e= 'connected'
else
  e= 'Unable to connect'
end
```

Visual Basic

Description: Connects to PA5 #2 via Optical Gigabit

```
If PA5x1.ConnectPA5("GB", 2) Then
  MsgBox "Connection established"
Else
  MsgBox "Unable to connect"
End If
```

Display

Description: Prints text to the PA5's LED display.

'C' Prototype:

```
BOOL Display(LPCTSTR Text, long Position);
```

Arguments:

LPCTSTR	*Text*	String to be printed to the display (max length eight characters).
long	*Position*	Position in display: 0=left, 7=right.

Returns:

Boolean	False (0)	Not successful.
Boolean	True (-1)	Successful.

Sample Code

Description: Displays a warning.

MATLAB

```
PA5x1.Display('Check Attenuation', 0);
```

Visual Basic

```
PA5x1.Display("Check Attenuation", 0)
```

GetError

Description: Use this call to retrieve an error message or to test for an error. Returns a string containing one of the following error messages:

zBus Error: - This shows where the error occurred

Call:PA5setatt - What function call was attempted

zError:One or more arguments out of range. - Error message

'C' Prototype:

```
CString GetError;
```

Returns:

CString	error	Error message.

Sample Code

Description: Checks for an error message and displays it on the PA5 if one is returned.

MATLAB

```
ErrMess = PA5x1.GetError
if length(ErrMess) > 0
  disp ErrMess
end
```

Visual Basic

```
Dim ErrMess As String
ErrMess = PA5x1.GetError
If Len(ErrMess) > 0 Then
  MsgBox ErrMess
End If
```

GetAtten

Description: Returns the current level of attenuation on the PA5 as a value from 0-120. It is not altered by user-defined attenuation levels.

'C' Prototype:
```
float GetAtten;
```

Returns:

float — attenuation on PA5

Sample Code

MATLAB

Description: Starts an active X control for the PA5, connects to PA5 #1 through the GB port and gets the current attenuation setting for the PA5.

```
PA5x1=actxserver('PA5.x')
PA5x1.ConnectPA5('GB', 1) % Connects PA5 via Optical
Gigabit
z=PA5x1.GetAtten
```

Visual Basic

Description: Connects PA5 #1 through the GB and gets the current attenuation setting.

```
PA5x1.ConnectPA5("GB", 1)
z=PA5x1.GetAtten
```

Reset

Description: Resets the PA5 and restores the factory defaults.

Factory defaults are:
Attenuation=0.0, Step size =3.0, Update=Dynamic.

'C' Prototype:
```
BOOL Reset;
```

Arguments: None

Returns:

Boolean	False (0)	Not successful.
Boolean	True (-1)	Successful.

Sample Code

Description: Starts ActiveX control, connects to the PA5 via the GB interface, and resets the PA5 to the factory defaults (0.0 attenuation).

MATLAB
```
PA5x1=actxserver('PA5.x');
PA5x1.ConnectPA5('GB', 1) % Connects PA5 via Optibit
PA5x1.Reset
```

Visual Basic
```
PA5x1.ConnectPA5("GB",1)
PA5x1.Reset
```

SetAtten

Description: Sets attenuation on the PA5. Attenuation is a floating point value between 0.0 and 120. Values higher and lower than these values will set an error flag. You can use GetError() to check for error messages.

'C' Prototype: `BOOL SetAtten(float AttVal);`

Arguments:

float	*AttVal*	Attenuation (0.0...120.0).

Returns:

Boolean	False (0)	Not successful.
	True (-1)	Successful.

Sample Code

Description: Sets the Attenuation to the value given by "Atten" and checks for an error. If "Atten" is greater than 120 or less than zero an error message is generated.

MATLAB

```
PA5x1.SetAtten(Atten);
error1=PA5x1.GetError()
if length(error1)==0
  disp 'Attenuation set correctly'
else
  PA5x1.Display(error1, 0)
end
```

Visual Basic

```
PA5x1.SetAtten(Atten)
error1 = PA5x1.GetError
If error1 = "" Then
  msgbox "Attenuation set correctly"
Else
  msgbox error1
End If
```

SetUser

Description: Sets parameters for User Attenuation mode. For a complete description of how these work, check the TDT online help. A brief description of each function is given below along with an ActiveX example.

Note: User values are used for comparison and display purposes only. They do not affect the values for SetAtten() or GetAtten(). They should only be used to quickly assess signals from several PA5's using the front panel display.

'C' Prototype: `BOOL SetUser(long ParCode, float Val);`

Arguments:

long	*ParCode*	Code for specific parameter.

ParCode	Parameter Constants (Name)
1	`PA5_USERPAR_BASE:` Set base attenuation (0.0 .. 120.0 dB). Base attenuation is used when several stimulus devices (speakers) vary in signal intensity. Setting the base for each speaker will display the same attenuations.
2	`PA5_USERPAR_STEP:` Set dB step size (0.0 .. 120.0 dB)
3	`PA5_USERPAR_REFERENCE:` Set Reference dB Level (0.0 .. 120.0 dB). Reference attenuation allows the user to display smaller numbers

	(including negative ones). For example, for a Reference of 120 the most intense signal would display 120 dB on the front panel and the least intense signal would display 0.0 under user settings.
4	`PA5_USERPAR_UPDATE:` Sets User Update Parameter. DYNAMIC updates produce a continuous change in attenuation. MANUAL update only changes the attenuation when the Select button (dial) is pressed. The display is dimmed while changing the attenuation. `Use PA5_USERUPDATE_DYNAMIC` or `PA5_USERUPDATE_MANUAL` for Val argument. 0 `PA5_USERUPDATE_DYNAMIC` Set User Update mode to Dynamic, where attenuation is changed as the dial is turned. 1 `PA5_USERUPDATE_MANUAL` Set User Update mode to Manual, where attenuation is not changed while dial is turned. Attenuation updates only when the user presses the dial to SELECT the new value.
5	`PA5_USERPAR_ABSMIN`: Set minimum level of attenuation allowed on the PA5. Used to prevent accidental output of very loud sounds.

float	*Val*	Value for given parameter code.
Returns:	0	Not successful.
	-1	Successful.

Sample Code

Description: Sets up a series of constants that match the values used for 'SetUser'. Some parameter values for the different User functions are set. Finally the code sends all of the variables to the PA5. The user is given 5 seconds to change the setting and see the difference between the value that the userAtten displays and the base value for the attenuator.

MATLAB

```
PA5x1=actxserver('PA5.x');
PA5x1.ConnectPA5('GB', 1) % Connects PA5 via Optibit
% Constants used by Setuser
PA5_USERPAR_BASE=1;
PA5_USERPAR_STEP=2;
PA5_USERPAR_REFERENCE=3;
PA5_USERPAR_UPDATE=4;
PA5_USERUPDATE_DYNAMIC=0;
PA5_USERUPDATE_MANUAL=1;
PA5_USERPAR_ABSMIN=5;
% Parameter values used for Setuser
Base=5;
Step=5;
Reference=120;
Absmin=20.0;
% Invoke commands for SetUser
PA5x1.SetUser(PA5_USERPAR_BASE, Base);
PA5x1.SetUser(PA5_USERPAR_STEP, Step);
PA5x1.SetUser(PA5_USERPAR_REFERENCE, Reference);
PA5x1.SetUser(PA5_USERPAR_ABSMIN, Absmin);
```

```
PA5x1.SetUser(PA5_USERPAR_UPDATE,PA5_USERUPDATE_DYNAMI
C);
pause(1)
% Check the values
PA5x1.GetAtten
```

Visual Basic

```
'Constants used by Setuser
const PA5_USERPAR_BASE=1
const PA5_USERPAR_STEP=2
const PA5_USERPAR_REFERENCE=3
const PA5_USERPAR_UPDATE=4
const PA5_USERUPDATE_DYNAMIC=0
const PA5_USERUPDATE_MANUAL=1
const PA5_USERPAR_ABSMIN=5
'Parameter values used for Setuser
Base=5
Step=5
Reference=120
Absmin=20.0
PA5x1.ConnectPA5("GB", 1) 'Connects PA5x1 via Optibit
'Invoke commands for SetUser
PA5x1.SetUser(PA5_USERPAR_BASE, Base)
PA5x1.SetUser(PA5_USERPAR_STEP, Step)
PA5x1.SetUser(PA5_USERPAR_REFERENCE, Reference)
PA5x1.SetUser(PA5_USERPAR_ABSMIN, Absmin)
PA5x1.SetUser(PA5_USERPAR_UPDATE,PA5_USERUPDATE_DYNAMI
C);
'Check values
atten=PA5x1.GetAtten
```

zBUS Device

About the zBUSx Methods

This section provides a listing of the available zBUSx ActiveX control methods.

Programming Steps

- Add the zBUSx ActiveX controller to your program. The ActiveX help has examples for setting up ActiveX controllers in MATLAB, Visual Basic, and Visual C.
- Connect to a zBUS (USB or GB) device caddie (rack) with the connectZBUS function.
- Control the zBUS with the command and control functions using the ActiveX controller.

ConnectZBUS

Description: Establishes a connection with a ZBUS device interface (GB or USB). ConnectZBUS returns 0 if unsuccessful and 1 when successful.

'C' Prototype:

```
long ConnectZBUS(LPCTSTR Interface);
```

Arguments:

LPCTSTR *Interface* — Interface to which the device is connected.

Argument	Connection	Part #s
'GB '	Optical Gigabit	PO5/FO5
'USB'	USB	UZ1, UZ2, UB2, UZ4
'USB3'	USB 3.0	UZ3

Returns:

long	0	Connection not successful.
long	1	Connection successful.

Sample Code

Description: Connects to the ZBUS device via the Gigabit interface

MATLAB

```
% Connects to the ZBUS via Optical Gigabit
zBUS=actxserver('ZBUS.x');
if zBUS.ConnectZBUS('GB')
  e= 'connected'
else
  e= 'Unable to connect'
end
```

Visual Basic

```
'Connects to the ZBUS via the Optical Gigabit
If zBUS.ConnectZBUS("GB") Then
  msgbox "Connection established"
Else
  msgbox "Error connecting to ZBUS"
End If
```

FlushIO

Description: Clears the input and output values on the zBUS in order to remove bad data from the buffers.

'C' Prototype:
```
long FlushIO(long racknum);
```

Arguments:

long	*racknum*	Rack number of the IO line to flush.

Returns:

long	0	Unable to Flush I/O lines.
long	1	Successfully Flushed I/O lines.

Sample Code

Description: Flushes the IO lines of zBUS device caddie 1.

MATLAB
```
% Flushes the zbus I/O lines
zBUS.FlushIO(1)
```

Visual Basic
```
'Flushes the zbus I/O lines
zBUS.FlushIO(1)
```

GetDeviceAddr

Description: Returns the address of a device, given the device type and device number.

'C' Prototype:
```
long GetDeviceAddr(long Devtype, long devnum);
```

Arguments:

long *Devtype* — ID number of the device.

PA5	RP2	RL2	RA16	RV8	RX5	RX6	RX7	RX8	RZ2	RZ5	RZ6
33	35	36	37	38	45	46	47	48	50	53	54

long devnum — Device number (1-16) e.g. RP2_1 is the first RP2 in the system (Note: Device number and physical position on the racks can differ).

Returns:

long	0	No such device type or device number.
long	n>2	Even numbers indicate position 1 and odd numbers position 2 of the device caddie (rack). For example: 2 = rack 1 position 1 3 = rack 1 position 2 4 = rack 2 position 1

Sample Code

Description: Gets the address of PA5_1.

MATLAB
```
% Gets the device address
zBUS.GetDeviceAddr(33,1)
```

Visual Basic
```
Dim address As Integer
'Get the device address
address=zBUS.GetDeviceAddr(33,1)
```

GetDeviceVersion

Description: Checks the version of the device, or microcode of the device (programming information).

'C' Prototype: `long GetDeviceVersion(long Devtype, long devnum);`

Arguments:

long *Devtype* ID number of the device.

PA5	RP2	RL2	RA16	RV8	RX5	RX6	RX7	RX8	RZ2	RZ5	RZ6
33	35	36	37	38	45	46	47	48	50	53	54

long *devnum* Device number (1-16) e.g. RP2_1 is the first RP2 in the system (Note: Device number and physical position of the racks can differ).

Returns:

long 0 No such device type or device number.

long >16 Version of the microcode.

TechNote: RP2.1 returns a value of 1xx (xx=version number) for the version identification.

RL2 Base stations return a value of 135 for the version identification.

Sample Code

Description: Checks to see if the Device has version 50 or greater of the microcode.

MATLAB

```
% Gets the device version
if zBUS.GetDeviceVersion(35, 1) < 50
  disp ' Update your microcode to run with this
ActiveX '
end
```

Visual Basic

```
Dim Vnum As Integer
'Gets the device version
If zBUS.GetDeviceVersion(35,1) < 50 Then
  msgbox "Update your microcode to run with this
ActiveX"
End If
```

GetError

Description: Returns an error description from the zBUS.

Note: unsuccessful returns are not always the result of a zBUS error. For example, if a device does not exist at that address a return of zero is valid. The ActiveX controls are designed to produce few error calls.

'C' Prototype: `LPCTSTR GetError;`

Arguments: None

Returns:

LPCTSTR "" No Error

LPCTSTR "(LPCTSTR)" Possible Error descriptions: All Errors begin with ZERR
ARG_OUT_OF_RANGE
UNABLE_TO_GET_XBUS_LOCK

UNKNOWN_ERROR
XBUS_COMMINICATION_ERROR
NO_INTERFACE_INITIALIZED
XBUS_GENERATED_ERROR
ACTIVE_ACCESS_UNAVAIL
PASSIVE_ACCESS_NOT_ALLOWED
MEMORY_ALLOC_FAILED
FAILED_READ_FROM_DEVICE
DEVICE_DRIVER_CODE_ERROR
SPECED_MEMORY_NOT_VALID
ILLEGAL_USB_DEVICE_SPECED
ZUSB_COM_ERROR
ZUSB_DEVICE_NOT_RESPOND
ZUSB_START_FAILURE
ZUSB_UNABLE_TO_ACC_DEV
CALL_NOT_SUPPORT_ON_INTER
DEVICE_SPEC_ERR

Sample Code

Description: Checks the Version number of the PA5 and returns a possible zBUS error.

MATLAB
```
% Gets the error string
if zBUS.GetDeviceVersion(34, 1)==0
  e=zBUS.GetError
end
```

Visual Basic
```
'Gets the error string
If zBUS.GetDeviceVersion(34, 1)=0 Then
  msgbox zBUS.GetError
End If
```

HardwareReset

Description: Resets the logical connection of the device caddie (rack) to the computer and returns a 0. Used to clear data lines and restore connections to the devices.

'C' Prototype: `long HardwareReset(long racknum);`

Arguments:

long	*racknum*	Caddie number to Reset.

Returns:

long	0	Successfully performed a Hardware Reset.

Sample Code

Description: Hardware reset of device caddie number 1.

MATLAB
```
% Hardware Reset of the zbus I/O lines
zBUS.HardwareReset(1)
```

Visual Basic
```
'Hardware Reset of the zbus I/O lines
zBUS.HardwareReset(1)
```

Important Note!: See Tech Note #181 for updated information on HardwareReset.

zBusTrigA/zBusTrigB

Description: Triggers several processor devices simultaneously either in one rack or over all racks. Trigger types include a single pulse varying in length (the length is dependant on the sampling rate), a permenant logical high, or a permenant logical low.

Note: To generate a single sample pulse, connect an EdgeDectect component after the zTrig component in your RPvdsEx circuit.

Minimum delay time is 2 milliseconds per rack, e.g. if you trigger five racks the zBusTrig requires 10 milliseconds.

Note: Differences in sample rates will cause differences in the triggering of the clock.

'C' Prototype: `long zBusTrigA/B(long racknum, long Trig type, long delay);`

Arguments:

long	*Racknum*	0=all device caddies (racks) triggered n=racknum triggered.
long	*Trig type*	0=pulse, 1=high, 2=low.
long	*delay*	delay before trigger event occurs, must be a minimum of 2msec per rack.

Returns:

long	0	Unsuccessful.
long	1	Successful.

long **Note:** In v57 and above, a zero will be returned even if the trigger is actually generated correctly. There are two ways to monitor the actual results.

In your RPvdsEx circuit:

Link the output of the zTrig component to a digital output on the device. This will allow the trigger result to be viewed on the front panel of the device.

Link a parameter tag to the output of the zTrig component and read this tag in MATLAB, to view the results.

Sample Code

MATLAB

Description: Two RP2 (devices 1 and 2) are loaded with the same circuit. They are triggered simultaneously using zBusTrigA. Only rack 1 receives the trigger. The delay is set for 3 msec just as a precaution and the trigger is a pulse. Both circuits are triggered simultaneously.

```
zBus=actxserver('ZBUS.x');
zBus.ConnectZBUS('GB')
RP2_1=actxserver('RPco.x');
RP2_2=actxserver('RPco.x');
RP2_1.ConnectRP2('GB', 1)
RP2_1.LoadCOF('C:\Circuit')
RP2_1.Run
RP2_2.ConnectRP2('GB', 2)
RP2_2.LoadCOF('C:\Circuit')
RP2_2.Run
zBus.zBusTrigA(1, 0, 5)
```

Visual Basic

Description: Two RP2 (devices 1 and 2) are loaded with the same circuit. They are triggered simultaneously using zBusTrigA across all possible racks. The delay is set for 5 msec just as a precaution and the trigger is a pulse. Both circuits are triggered simultaneously.

```
zBus.ConnectZBUS("GB")
RP2_1.ConnectRP2("GB", 1)
RP2_1.LoadCOF("C:\Circuit")
RP2_1.Run
RP2_2.ConnectRP2("GB", 2)
RP2_2.LoadCOF("C:\Circuit")
RP2_2.Run
zBus.zBusTrigA(0, 0, 5)
```

zBusSync

Description: Synchronizes the clocks across several device caddies (racks) to minimize drift. The clocks that drive the DSP can drift by as little as 0.01% over several seconds, producing clock differences of several microseconds. zBusSync ensures synchronization across devices.

To use zBusSync, connect the Sync lines on the UB1/UZ4 to be synchronized, using short BNC cables and T-connectors to minimize noise.

zBusSync uses a bitmask to identify a master and slave clocks. The first rack 'turned on' in the bitmask (according to the logical order of devices) is master and the rest are slaves, i.e. they get their clock signal from the master device.

This command should only be used with the UB1/UZ4 USB 1.1 interfaces. It will always return a zero when used with any other interface type.

'C' Prototype: `long zBusSync(long Bitmask Racknum);`

Arguments:

long	*BitMask Racknum*	Bitmask values for the racknum. e.g. 5 means that device caddie 1 is the master and device caddie 3 is the slave synchronized clock. 6 means that device caddie 2 is the master and device caddie 3 is the slave.

Returns:

long	0	Unsuccessful.
long	1	Successful.

Sample Code

MATLAB

Description: Synchronizes the clocks of zBus device caddies 1 and 2.

```
zBUS=actxserver('ZBUS.x');
zBUS.ConnectZBUS('USB')
zBus.zBusSync(3)
```

Visual Basic

Description: Synchronizes the clocks of zBus device caddies 1, 2, 3, and 4.

```
zBus.ConnectZBUS("USB")
zBus.zBusSync(15)
```

ActiveX Examples

The example programs included with the ActiveX disk are general programs that can be modified for other purposes. Most of the examples have been developed with MATLAB and Visual Basic, which produces very compact code without a great deal of Windows related code. Programmers using other languages would benefit from the MATLAB and VB examples as well.

The steps generally used to develop example programs include:

Step 1: Design a circuit using RPvdsEx.

Step 2: Create a Control Object File (*.rco or *.rcx).

Step 3: Create a program that implements TDT ActiveX controls.

Complete documentation for each example is provided in the MATLAB, Visual Basic, and Visual C++ example sections that follow.

MATLAB Examples

MATLAB Example: Circuit Loader

This example documents a MATLAB program that lets the user load RPvdsEx control object files (*.rcx) and run them on Real-Time Processors.

ActiveX Methods Used

- ConnectRP2
- LoadCOF
- Run
- GetStatus

Files Used

The file required for this example can be found in: *C:\TDT\ActiveX\ActXExamples\matlab*

- **Circuit_Loader.m**: MATLAB (R13+) script file for loading a control object file (*.rcx)

  or

- **Circuit_Loader_R12.m**: MATLAB (R12) script file for loading a control object file (*.rcx)

Required Hardware

- RP2

Running the Application

To run the application:

- At the MATLAB prompt type "**Circuit_Loader**" and press the **Enter** key.

Program Description

The program prompts the user for the following information: Connection type (GB...), Device number, and COF (*.rcx) name. The program then loads and runs the RPvdsEx circuit and checks for errors using GetStatus. It also returns the ActiveX object that is controlling the device.

Relevant Code

The first line of code below sets up a processor device ActiveX control in MATLAB. The next line connects the control to an RP2; the fourth line clears that processor device of its COF file and any memory buffers (this call is not required). The sixth line loads a COF (*.rcx file) with the proper path and name designated. The seventh line of code starts the circuit. The eighth line checks the status of the circuit (7=loaded and running). All programs will use the Connect, LoadCOF, and Run when using ActiveX controls.

```
% Load circuit onto device and run
RP = actxserver('RPco.x');

RP.ConnectRP2(connectionType, deviceNumber);
% Connects RP2 via USB or GB given the proper device number
RP.Halt; % Stops any processing chains running on RP2
RP.ClearCOF; % Clears all the buffers and circuits on that RP2
disp(['Loading ' circuitPath]);
RP.LoadCOF(circuitPath); % Loads circuit
RP.Run; % Starts circuit

status=double(RP.GetStatus); % Gets the status
if bitget(status,1)==0; % Checks for connection
     disp('Error connecting to RP2'); return;
elseif bitget(status,2)==0; % Checks for errors in loading circuit
     disp('Error loading circuit'); return;
elseif bitget(status,3)==0 % Checks for errors in running circuit
     disp('Error running circuit'); return;
else
     disp('Circuit loaded and running'); return;
end
```

MATLAB Example: Device Checker

This example uses ActiveX controls to load an RPvdsEx circuit. It checks the cycle usage to see if the circuit uses too much of the Real-Time Processor's processing time. High cycle usage (>90%) causes erratic behavior on the Real-Time Processor. It then finds the name of each component type and the name, data type, and size of each parameter tag.

ActiveX Methods Used

- GetCycUse
- GetNumOf
- GetNameOf
- GetTagType
- GetTagSize

Files Used

The files required for this example can be found in: *C:\TDT\ActiveX\ActXExamples\matlab*

- **Circuit_Loader.m**: MATLAB (R13+) script file for loading a control object file (*.rcx)
- **Device_Checker.m**: MATLAB (R13+) script file for checking device properties

  or

- **Circuit_Loader_R12.m**: MATLAB (R12) script file for loading a control object file (*.rcx)
- **Device_Checker_R12.m**: MATLAB (R12) script file for checking device properties

Required Hardware

- RP2

Running the Application

To run the application:

- At the MATLAB prompt type "**Device_Checker**" and the **Enter** key.

This example uses Circuit_Loader.m to load the circuit.

Relevant Code

The first line checks the cycle usage of the Real-Time Processor. The second line of code finds the number of parameter tags. A loop then determines the String ID, Data type, and Data size for each parameter tag. The MATLAB example uses similar code for other types of components.

```
Cycle_Usage = RP.GetCycUse; % Checks cycle usage

% Gets the number of each of the component types:
NumParTags = RP.GetNumOf('ParTag');

% Gets the names of the Parameter Tags, The TagType (data type), %
and TagSize
for z = 1:NumParTags
     PName = RP.GetNameOf('ParTag',z);
     % Returns the Parameter name
     PType = char(RP.GetTagType(PName));
     % Returns the Tag Type: Single, Integer, Data, Logical
     PSize = RP.GetTagSize(PName);
     % Returns TagSize (size of Data Buffer or 1)
     disp(['    ' PName '    type ' PType '   size '
num2str(PSize)]);
end
```

MATLAB Example: Band Limited Noise

This example loads an RPvdsEx circuit that generates variable intensities of band limited noise and checks the output for clipping. User control of the frequency and intensity of the noise can be set through the MATLAB command window.

ActiveX Methods Used

- SetTagVal
- GetTagVal

Files Used

The file required for this example can be found in: *C:\TDT\ActiveX\ActXExamples\matlab*

- **Band_Limited_Noise.m**: MATLAB (R13+) script file for running *.rcx file

  or

- **Band_Limited_Noise_R12.m**: MATLAB (R12) script file for running *.rcx file

The RPvdsEx file used can be found in: *C:\TDT\ActiveX\ActXExamples\RP_files*

- **Band_Limited_Noise.rcx**

Required Hardware

- RP2

Required Applications

- RPvdsEx
- MATLAB

Running the Application

To run the application:

- In the Command Window type "**Band_Limited_Noise**" at the prompt.

Making the RPvdsEx Circuit

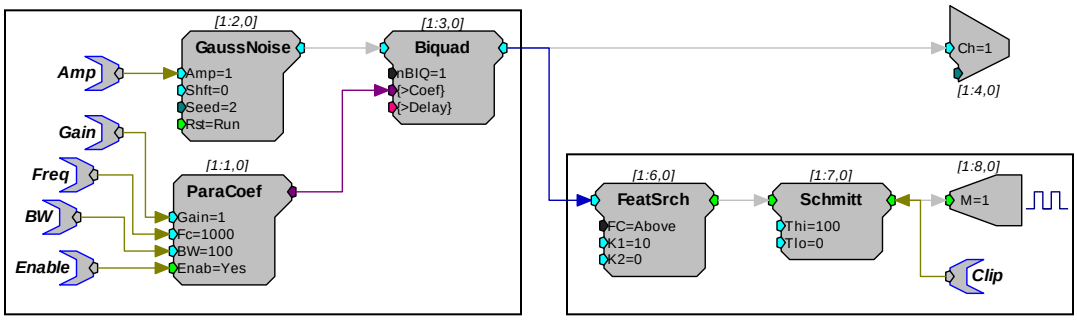
Component types required:

- Five parameter tags. To change the name, double-click the parameter tag and type a new name.
  - Gain - Increases the relative bandpass filtering in dB
  - Freq - Center frequency of the bandpass
  - BW - Width of the bandpass (3dB roll off)
  - Amp - Changes the amplitude of the noise
  - Enable - Starts and stops generation of the filter coefficients
  - Clip - Checks to see if the signal is clipped
- Gaussian noise generator (waveform generator)
- Parametric filter coefficient generator
- Biquad filter
- Feature search
- Schmitt trigger
- DacOut
- BitOut

Connect the circuit as shown below:

Note: Double click on any RPvdsEx and then click on the help button to access the RPvdsEx components help.

The two boxes represent the different parts of the circuit. The box on the left includes components that generate (GaussNoise) and filter (ParaCoef/Biquad) the waveform. The parameter tags are used to set the amplitude of the noise and filter parameters. The second part of the circuit, found in the box on right, checks for clipping (signal values greater than +/- 10 Volts) and generates a high signal on Bit 0 (M=1) of the processor device.



Program Description

This program controls a circuit that generates band-limited noise. The user controls the center frequency, bandwidth, and the intensity of the filtered noise. If the parameters produce clipping the user is prompted to change some of the parameters. The relevant code controls or receives information about the circuit through parameter tags.

Relevant Code

The code below sets the values of a series of tags. Each tag sends the value to the component port(s) (e.g. Gain) to which they are connected.

```
% User gives information about the center frequency, Bandwidth,
% gain of filter, and amplitude of noise
Freq=input('Enter the center frequency for the filter: ');
Gain=input('Enter the dB gain for the filter: ');
Bandwidth=input('Enter the bandwidth for the filter: ');
Amp=input('Enter the intensity for the noise: ');
% Sets the initial settings for the filter coefficients and the
% noise
RP.SetTagVal('Gain', Gain); % Gain of band limited filter
RP.SetTagVal('Freq', Freq); % CenterFrequency
RP.SetTagVal('BW', Bandwidth); % Bandwidth of filter
RP.SetTagVal('Amp', Amp); % Amplitude of the Gaussian Noise
% Loads Coefficients to Biquad Filter
RP.SetTagVal('Enable', 1);
RP.SetTagVal('Enable', 0); % Stops Coefficient generator from
sending signal (saves on cycle usage)
```

GetTagVal

This code checks for clipping. A parameter tag is polled once every 100 msec. It returns a one if the signal is clipped and a zero if it is not. The GetTagVal returns the state of the Schmitt trigger (high or low).

```
while quit==0
    Clip=RP.GetTagVal('Clip'); % Checks to see if signal is
clipped (top light on panel is on while clipping occurs)
    if Clip==1
        disp('Gain of filter or noise intensity is too high');
```

MATLAB Example: Continuous Acquire

This example uses a circuit that continuously saves data to a 100,000 sample buffer at 100 kHz. It continuously reads from a serial buffer in 50,000 sample chunks and saves the first 1,000,000 samples to a f32 file.

ActiveX Methods Used

- ReadTagV
- SoftTrg
- GetTagVal

Files Used

The file required for this example can be found in: *C:\TDT\ActiveX\ActXExamples\matlab*

- Continuous_Acquire.m: MATLAB (R13+) script file for running *.rcx file

  or
- Continuous_Acquire_R12.m: MATLAB (R12) script file for running *.rcx file

The RPvdsEx file used can be found in: *C:\TDT\ActiveX\ActXExamples\RP_files*

- Continuous_Acquire.rcx: RPvdsEx circuit

Required Hardware

- RP2

Required Applications

- RPvdsEx
- MATLAB

Running the Application

To run the application:

- In the Command Window type "**Continuous_Acquire**" at the prompt.

Making the RPvdsEx Circuit

Component types required:

- Two Parameter tags. To change the name of a parameter tag, double-click on the parameter and type a new name.
  - dataout - Points to the memory buffer
  - index - Points to the index of the serial buffer.
- Two soft triggers (Soft1 and Soft2). To change the trigger to a soft trigger, double-click the trigger and click on the drop down menu under Trigger type. Change one to Soft1 and the other to Soft2.
- AdcIn
- RSFlipFlop
- SerialBuf. To change the size of the serial buffer's memory, double-click the serial buffer and change Size to 100000.

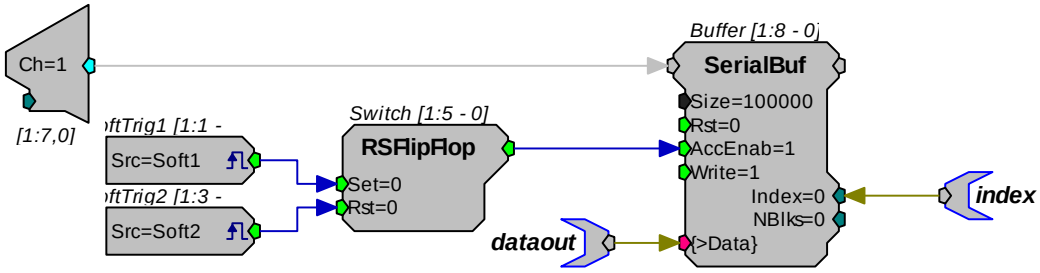
Connect the circuit as shown below:

Note: Double click on any RPvdsEx and then click on the help button to access the RPvdsEx components help.

The circuit below uses a Serial Buffer to acquire a signal. The signal is captured to a serial buffer, downloaded to the PC and stored in a file named fnoise2. To demonstrate the circuit, a Gaussian

noise signal is generated (not shown). Removal of the noise generator portion of the circuit does not affect the MATLAB script.

Data is continuously acquired by channel one but is only saved to the serial buffer when the AccEnab line is set high. The two soft-triggers control the start and stop of the data acquisition. When Soft1 goes high, the RSFlipFlop goes high and stays high. This sets the AccEnab line high and the serial buffer starts saving the data. The serial buffer holds 100,000 samples. When the buffer captures more than 100,000 points the end of memory is reached, the index is reset to 0, and any data in memory is written over. When data cannot be downloaded to the PC fast enough it gets overwritten in the buffer.



Program Description

The program acquires 10 seconds of signal at 100 kHz sampling rate and stores it in a file. A software trigger starts the counter and the signal is stored in the serial buffer. The serial buffer index is polled until 50,000 points are read into the buffer. The data is then sent to an array using ReadTagV and the data array is stored in a data file. The counter is polled until the next 50,000 points are read and the cycle is repeated. Each time the data is sent to the PC the program checks to see if the transfer rate is fast enough. A final software trigger ends the data acquisition. The last half second of data acquisition is plotted.

Relevant Code

This part of the code starts acquisition of the data by the serial buffer. It then checks to see if the buffer is half-filled. Half of the buffer is acquired while the other half is being filled. This method is called double buffering and allows for continuous acquisition data to be written to the *fnoise2* file in separate half-buffer partitions. Double buffering allows the circuit to continuously acquire data while it also writes the older data to the *fnoise2* file.

```
% Begin acquiring
RP.SoftTrg(1);

% Main Looping Section
for i = 1:10
      curindex=RP.GetTagVal('index');
      disp(['Current index: ' num2str(curindex)]);

      % Wait until first half of Buffer fills
      while(curindex<bufpts)
      % Check to see if it has read into half the buffer
            curindex=RP.GetTagVal('index');
      end

      % Read first segment
      noise=RP.ReadTagV('dataout', 0, bufpts);
      % Read from the buffer
```

```
        disp(['Wrote ' num2str(fwrite(fnoise,noise,'float32')) '
        points to file']); % Writes to a file

        % Check to see if the data transfer rate is fast enough
        curindex=RP.GetTagVal('index');
        disp(['Current index: ' num2str(curindex)]);
        if (curindex<bufpts)
               disp('Transfer rate is too slow');
        end

        % Wait until second half of buffer fills
        while(curindex>bufpts)
               curindex=RP.GetTagVal('index');
        end

        % Read second segment
        noise=RP.ReadTagV('dataout', bufpts, bufpts);
        % Reads from the buffer
        disp(['Wrote ' num2str(fwrite(fnoise,noise,'float32')) '
        points to file']); % Writes to a file

        % Check to see if the data transfer rate is fast enough
        curindex=RP.GetTagVal('index');
        disp(['Current index: ' num2str(curindex)]);
        if(curindex>bufpts)
               disp('Transfer rate is too slow');
        end

% Loop back to start of data capture routine.
end
```

MATLAB Example: Continuous Play

This example uses a circuit that continuously loads data to a 100,000 sample buffer at 100 kHz and sends the signal out for play to a DAC and a MATLAB script file that continuously writes to a serial buffer in 50,000 sample chunks.

ActiveX Methods Used

- WriteTagV
- SoftTrg
- GetTagVal

Files Used

The file required for this example can be found in: *C:\TDT\ActiveX\ActXExamples\matlab*

- Continuous_Play.m: MATLAB (R13+) script file for running *.rcx file

  or

- Continuous_Play_R12.m: MATLAB (R12) script file for running *.rcx file

The RPvdsEx file used can be found in: *C:\TDT\ActiveX\ActXExamples\RP_files*

- Continuous_Play.rcx: RPvdsEx circuit

Required Hardware

- RP2

Required Applications

- RPvdsEx
- MATLAB

Running the Application

To run the application:

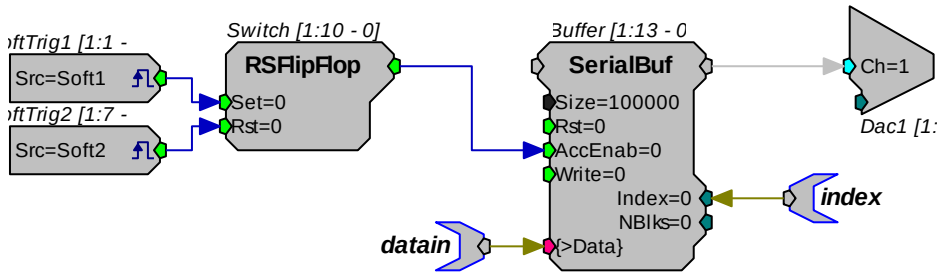
- In the Command Window type "**Continuous_Play**" at the prompt.

Making the RPvdsEx Circuit

Component types required:

- Two parameter tags. To change the name of a parameter tag, double-click on the parameter and type a new name.
  - datain - Points to the memory buffer
  - index - Points to the index of the serial buffer
- Two soft triggers (Soft1 and Soft2). To change the trigger to soft trigger, double-click the trigger and select the trigger type from the drop down menu under Trigger Type. Change one to Soft1 and the other to Soft2.
- DacOut
- RSFlipFlop
- SerialBuf. To change the size of the serial buffer's memory, double-click the serial buffer and set Size to 100000.

Your circuit should look like the one below. The signal is generated on the PC and then loaded into the serial buffer for play out.



When the Soft1 trigger goes high, the FlipFlop goes high and stays high. This sets the AccEnab line high and the serial buffer starts sending the data out to the DAC. When the serial buffer has played out 100,000 points the index is reset and the data at the beginning of the buffer is played out. As long as the AccEnab is high the serial buffer will play the signal.

Program Description

The program plays a series of tones for 10 seconds. The first second of tones is loaded to the serial buffer. A software trigger starts the counter and the signal is played out through the DAC. The serial buffer index is polled until 50,000 points are played from the buffer. Another tone is generated and loaded to the first half of the buffer. The counter is polled until the next 50,000

points are played out and the cycle is repeated. The program checks to see if the transfer rate is fast enough when the data is written to the buffer. A final software trigger ends the play out.

Relevant Code

This section writes the tones to the serial buffer. The first call to WriteTagV writes the signal named s1 to the first half of the buffer and the second call writes signal s2 to the second half of the buffer. Half of the buffer is written to, while the other half is being read to play out a tone.

This method is called double buffering and is used to read the data values of the tones into one half of the buffer while the other half is being played out. This allows the example to play tones continuously.

```
RP.WriteTagV('datain', 0, s1);
RP.WriteTagV('datain', bufpts-1, s2);
% This section starts the signal playout. Once half the buffer
% is played out it loads the next signal. After ten seconds,
% the second software trigger sets the AccEnab line low and
% stops play out.

% Start Playing
    RP.SoftTrg(1);
    curindex=RP.GetTagVal('index');

    % Main Looping Section
    for i = 1:10

        % Wait until done playing A
        while(curindex < bufpts) % Checks to see if it has
        % played from half the buffer
          curindex=RP.GetTagVal('index');
        end

        % Loads the next signal segment
        freq1=freq1+1000;
        s1=sin(2*pi*t*freq1);
        RP.WriteTagV('datain', 0, s1);

        % Checks to see if the data transfer rate is fast
        % enough
        curindex=RP.GetTagVal('index');
        if(curindex < bufpts)
          disp('Transfer rate is too slow');
        end

        % Wait until start playing A
        while(curindex > bufpts)
          curindex=RP.GetTagVal('index');
        end

        % Load B
        freq2=freq2+1000;
        s2=sin(2*pi*t*freq2);
        RP.WriteTagV('datain', bufpts, s2);

        % Make sure still playing A
        curindex=RP.GetTagVal('index');
        if(curindex > bufpts)
```

```
            disp('Transfer rate is too slow');
        end

        % Loop back to wait until done playing A
    end

    % Stop playing
    RP.SoftTrg(2);
    RP.Halt;
end
```

MATLAB Example: FIR Filtered Noise

This example documents a program that uses a noise generator to output a signal. An FIR filters the signal and the filtered and unfiltered signals are played out of two DACs.

ActiveX Methods

- SendSrcFile
- SendParTable

Files Used

The file required for this example can be found in: *C:\TDT\ActiveX\ActXExamples\matlab*

- FIR_filtered_noise.m: MATLAB (R13+) script file for running *.rcx file

  or
- FIR_filtered_noise_R12.m: MATLAB (R12) script file for running *.rcx file

The RPvdsEx file used can be found in: *C:\TDT\ActiveX\ActXExamples\RP_files*

- FIR_Filtered_Noise.rcx: RPvdsEx circuit

Required Hardware

- RP2

Required Applications

- RPvdsEx
- MATLAB

Running the Application

To run the application:

- In the Command Window type "**FIR_Filtered_Noise**" at the prompt.

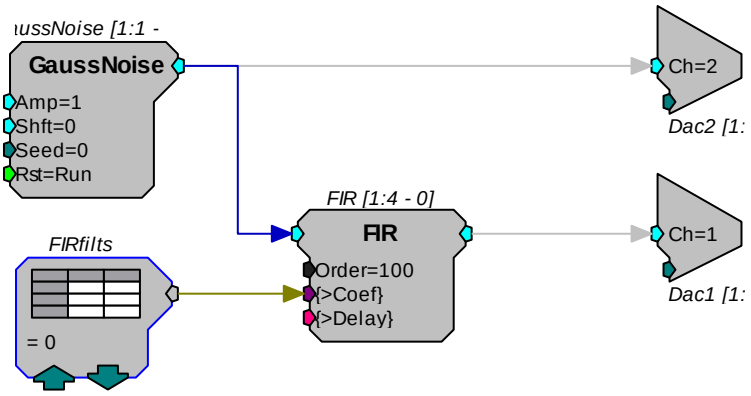
Making the RPvdsEx Circuit

Component types required:

- GaussNoise. To change the parameters of the noise signal double click the icon and edit the values
- Data Table. The data table provided with the circuit contains the FIR filter coefficients for low, high, and bandpass filters. The coefficients were generated in MATLAB and pasted into the data table. You will need to use the RPvdsEx file

"FIR_Filtered_Noise.rcx" which is provided in the ActiveX installation to use these filter coefficients.

- FIR filter. To change the order of the FIR, double-click the component and change Order to 100.
- Two DacOuts. Channel 2 plays out the unfiltered signal. Channel 1 plays out the filtered signal.



The circuit uses a GaussNoise component to output a signal. The signal is then filtered with an FIR (low, high or bandpass) filter whose coefficients are loaded from a data table. The signals are played out on Channel 1(filtered) and Channel 2(unfiltered) for comparison purposes.

Each section of signal is filtered three times: a low pass filter, high pass filter, and band pass filter. The program cycles through these three filter settings. Filters were generated in MATLAB as FIR filters with 100 taps.

Relevant Code

```
% Cycles through the three FIR filters
for i = 1:3
    % Loads one set of filter coefficients to an FIR
    RP.SendParTable('FIRfilts', i);
    pause(2);
end
% Stop playing
RP.Halt;
```

MATLAB Example: Two Channel Acquisition with ReadTagVEX

This example uses a circuit that continuously acquires data from two channels at 100 kHz per channel. It continuously reads from a serial buffer in 50,000 sample chunks and saves the data in matrix format to disk.

ActiveX Methods

- ReadTagVEX
- SoftTrg
- GetTagVal

Files Used

The file required for this example can be found in: *C:\TDT\ActiveX\ActXExamples\matlab*

- TwoCh_Continuous_Acquire.m: MATLAB (R13+) script file for running *.rcx file

  or

- TwoCh_Continuous_Acquire_R12.m: MATLAB (R12) script file for running *.rcx file

The RPvdsEx file used can be found in: *C:\TDT\ActiveX\ActXExamples\RP_files*

- TwoCh_Continuous_Acquire.rcx: RPvdsEx circuit

Required Hardware

- RP2

Required Applications

- RPvdsEx
- MATLAB

Running the Application

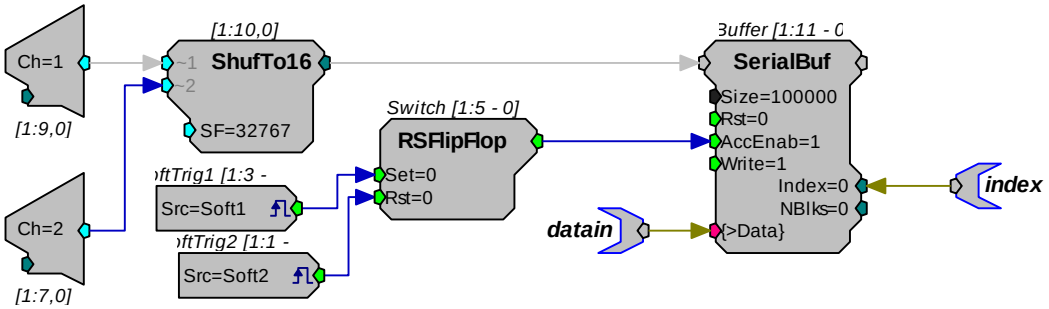
To run the application:

- In the Command window type "**TwoCh_Continuous_Acquire**" at the prompt.

Making the RPvdsEx Circuit

Component types required:

- Two Parameter tags. To change the name, double-click the parameter tag and type a new name.
  - dataout - Points to the memory buffer
  - index - Points to the index of the serial buffer.
- Two soft triggers (Soft1 and Soft2). To change the trigger to a soft trigger, double-click the trigger and select a trigger from the drop down menu under Trigger Type. Change one to Soft1 and the other to Soft2.
- Two AdcIns
- RSFlipFlop
- ShufTo16. This component reduces two 32-bit floating point input values to 16 bits each. The 16-bit values are then stored in the upper and lower half of a 32-bit output. At a 100 kHz sampling rate, it is possible to stream two channels to disk in real-time.
- SerialBuf. To change the size of the serial buffer's memory, double-click the serial buffer component and change the Size to 100000.



The circuit uses a SerialBuf and ShufTo16 to acquire two channels of data continuously at a 100 kHz sampling rate. The signal is captured to a serial buffer, downloaded to the PC, and stored in a file.

Note: This circuit contains a Gaussian noise generator that is output to DAC OUT-1 and a tone generator that is output to DAC OUT-2.

Program Description

The program is very similar to the Continuous Acquire MATLAB example. It acquires 10 seconds of signal from two channels at 100 kHz sampling rate and stores it in a file.

A software trigger starts the counter and a signal is stored in the serial buffer. The counter is polled until 50,000 points are read into the buffer. The data is then downloaded to a MATLAB array, which is stored in a data file. The counter is polled until the next 50,000 points are read and the cycle is repeated. Each time the data is sent to the PC the program checks to see if the transfer rate is fast enough. A final software trigger ends the data acquisition. The last half second of the acquired data is plotted.

Relevant Code

Check Continuous Acquire, page 74, for a description of the general program. This code reads the data from the buffer. I16 is the source type of the data on the processor device; F64 is how the data is stored on the PC.

```
noise=RP2.ReadTagVEX('dataout', 0, bufpts, 'I16', 'F64', 2);
% Reads from the buffer
```

MATLAB example: Two Channel Play with WriteTagVEX

This example uses a circuit that continuously plays a signal out of two channels at 100 kHz per channel. It continuously writes to a serial buffer in 50,000 sample chunks.

ActiveX Methods

- WriteTagVEX
- SoftTrg
- GetTagVal

Files Used

The file required for this example can be found in: *C:\TDT\ActiveX\ActXExamples\matlab*

- TwoCh_Continuous_Play.m: MATLAB (R13+) script file for running *.rcx file

  or
- TwoCh_Continuous_Play_R12.m: MATLAB (R12) script file for running *.rcx file

The RPvdsEx file used can be found in: *C:\TDT\ActiveX\ActXExamples\RP_files*

- TwoCh_Continuous_Play.rcx: RPvdsEx circuit

Required Hardware

- RP2

Required Applications

- RPvdsEx
- MATLAB

Running the Application

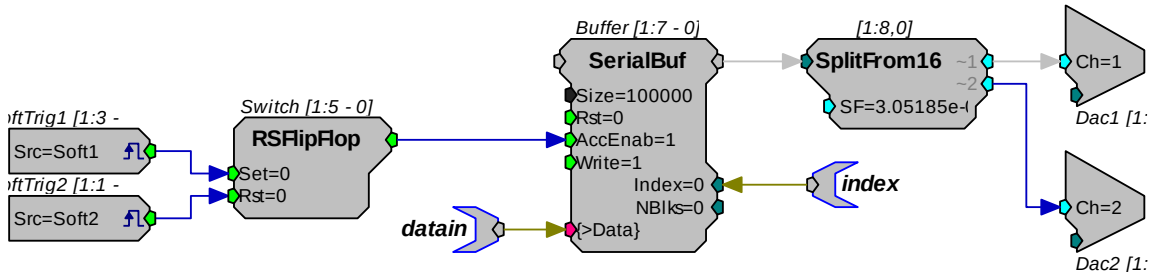
To run the application:

- In the Command Window type "**TwoCh_Continuous_Play**" at the prompt.

Making the RPvdsEx Circuit

Component types required:

- Two parameter tags: To change the name of a parameter tag, double-click it then type a new name.
  - Datain - Points to the memory buffer
  - Index - Points to the index of the serial buffer.
- Two software triggers (Soft1 and Soft2). To change the trigger to soft trigger, double-click on the trigger and click on the drop down menu under Trigger Type. Change one to Soft1 and the other to Soft2.
- Two DacOuts
- RSFlipFlop
- SplitFrom16
- A serial buffer (SerialBuf). To change the size of the serial buffer's memory, double-click the component and change the Size to 100000.



The circuit below uses a SerialBuf and SplitFrom16 to play out signals to two channels continuously. The signal is generated on the PC and then loaded into the serial buffer memory.

The circuit is similar to the Continuous Play example. When Soft Trigger 1 goes high the FlipFlop goes high and stays high. This sets the AccEnab line high and the serial buffer starts sending the data out to the DAC. When the serial buffer has played out 100,000 points, the index is reset and the data at the beginning of the buffer is played out. As long as the AccEnab is high the serial buffer will play the signal. The signal from the serial buffer memory is split into two channels with SplitFrom16 and both channels are played out on DAC OUT-1 and DAC OUT-2.

Program Description

This program generates two tones in MATLAB, stores them in a matrix, and loads them to the serial buffer's memory with WriteTagVEX. The general format for generating the signal with WriteTagVEX is shown below. Otherwise, this example is similar to the Continuous Play example.

Relevant Code

The signals must be generated and scaled to fit the format for WriteTagVEX and SplitFrom16. For SplitFrom16 the format must be 16-bit integer. The scaling factor determines the amplitude of the signal; in this case the scaling factor assumes a +/- 1.0 V input signal to a +/- 10 V output. The

floating point signals are converted to integer format with a 16-bit range. The two signals are then placed in a matrix.

```
% Two-Channel Continuous Play example using a serial buffer
% This program writes to the rambuffer once it has cycled half
% way through the buffer.
Npts=100000; % Size of the serial buffer
bufpts=npts/2; % Number of points to write to buffer
RP=Circuit_Loader('C:\TDT\ActiveX\ActXExamples\RP_files\TwoCh_C
ontinuous_Play.rcx');
if all(bitget(RP.GetStatus,1:3))
   % Generate two tone signals to play out in MATLAB
    freq1=1000;
    freq2=5000;
    fs=97656.25;
    t=(1:bufpts)/fs;
    s1=round(sin(2*pi*t*freq1)*32760);
    s2=round(sin(2*pi*t*freq2)*32760);
    % Serial buffer will be divided into two buffers A & B
    % Load up entire buffer with segments A and B
    s=[s1;s2]; % Concatenate two arrays into a matrix
```

The signals are loaded with WriteTagVEX. The format below with 'I16' indicates 16-bit integer format. WriteTagVEX determines the properties of the variant used for signal generation.

```
RP2.WriteTagVEX('datain', 0, 'I16', s);
```

Visual C++ Examples

Visual C++ Example: Circuit Loader

This example documents a Visual C++ program that lets the user load RPvdsEx control object files *.RCO(*.rco or *.rcx) and run them on Real-Time Processors. Up to 32 processors can be controlled at once by this program (up to 8 RP2/RP2.1s, up to 8 RA16s, up to 8 RV8s, and up to 8 RL2s).

ActiveX Methods Used

- ConnectRP2
- LoadCOF
- GetStatus
- ClearCOF
- Run
- Halt

Files Used

The files required for this example can be found in:
C:\TDT\ActiveX\ActXExamples\vc++\CircuitLoader

- CircuitLoader.vcproj: Visual C++ project file

- CircuitLoaderDlg.cpp: Visual C++ code that controls the graphical user interface and communicates with the RPvdsEx circuit; contains ActiveX components for the processor devices
- CircuitLoader.exe: compiled executable; for running the example without having to start up Visual C++

Required Hardware

- At least one Real-Time Processor (either RP2, RP2.1, RA16, RV8, or RL2)

Required Applications

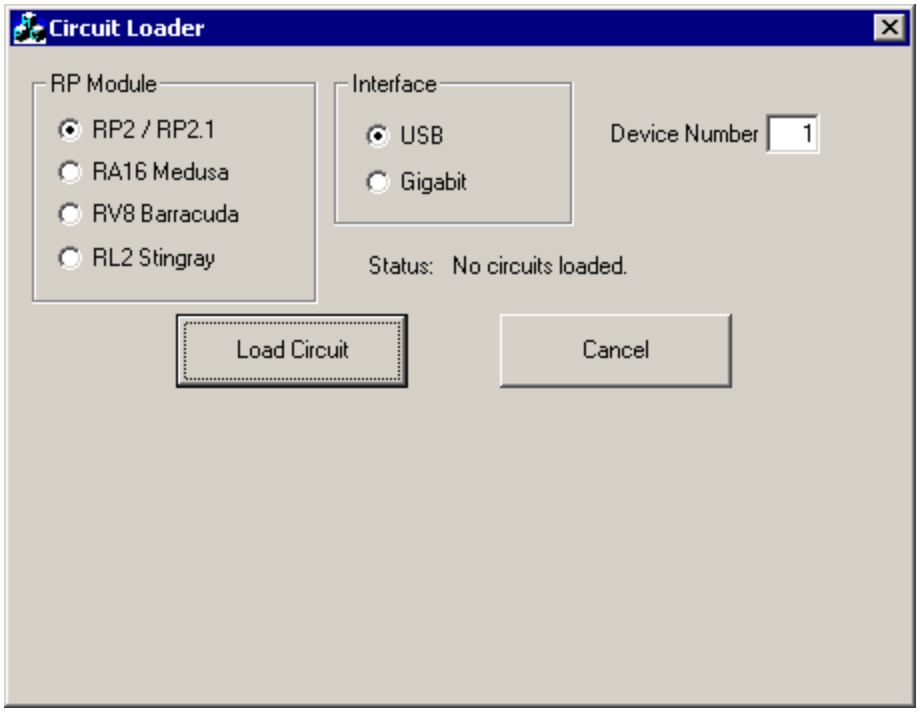
- Visual C++

Running the Application

- Run the CircuitLoader.exe executable file from the CircuitLoader directory, or load the CircuitLoader.vcproj project into Visual C++ and compile and run it from there.

Program Description

The Visual C++ program presents a graphical interface through which the user can load various circuits to Real-Time processors. The user selects the type of processor device, the interface (USB or Optical Gigabit), and the device number (from 1 to 8) through radio buttons and input boxes. When the Load Circuit button is clicked, a CommonDialog control lets the user choose the *.rcx file, and then it is loaded to the correct device based on the current settings of the user interface. A label is updated to show whether the circuit was loaded successfully or if an error occurred. 32 ActiveX controls are used in the program, one for each device that can potentially be used.



Relevant Code

The code below is run when the user clicks on the "Load Circuit" button. It displays a dialog window to select the *.rcx file, and then connects to the appropriate processor device, and loads and runs the circuit.

```
m_openfile_dialog.ShowOpen();
CString filepath = m_openfile_dialog.GetFileName();
CString interface_str;
switch(GetCheckedRadioButton(RADIO_USB, RADIO_GIGABIT)) {
  case RADIO_USB:
    interface_str = "USB";
    break;
  default:
    interface_str = "GIGABIT";
}
CString device_type;
int devnum = atoi(m_devnum_text);
long status;
switch(GetCheckedRadioButton(RADIO_RP2, RADIO_RL2)) {
  case RADIO_RP2:
    device_type = "RP2";
    break;
  case RADIO_RA16:
    device_type = "RA16";
    break;
  case RADIO_RV8:
    device_type = "RV8";
    break;
  default:
    device_type = "RL2";
}
status = RunCircuit(GetRP(device_type, devnum), filepath,
device_type, interface_str, devnum);
```

Visual C++ Example: Band Limited Noise

This example uses a circuit that produces band-limited noise and a Visual C++ program that lets the user control filter and noise settings, start and stop playing, and view results.

ActiveX Methods Used

- ConnectRP2
- Run
- GetSFreq
- ClearCOF
- Halt
- GetCycUse
- LoadCOF
- SetTagVal
- GetStatus
- GetTagVal

Files Used

The files required for this example can be found in:
C:\TDT\ActiveX\ActXExamples\vc++\BandLimitedNoise

- BandLimitedNoise.vcproj: Visual C++ project file

- BandLimitedNoiseDlg.cpp: Visual C++ code that controls the graphical user interface and communicates with the RPvdsEx circuit; contains an ActiveX component for the RP2
- BandLimitedNoise.exe: compiled executable; for running the example without having to start up Visual C++

The RPvdsEx file used can be found in: *C:\TDT\ActiveX\ActXExamples\RP_files*

- Band_Limited_Noise.rcx: Control File of the RPvdsEx designed circuit

Required Hardware

- RP2

Required Applications

- RPvdsEx
- Visual C++

Running the Application

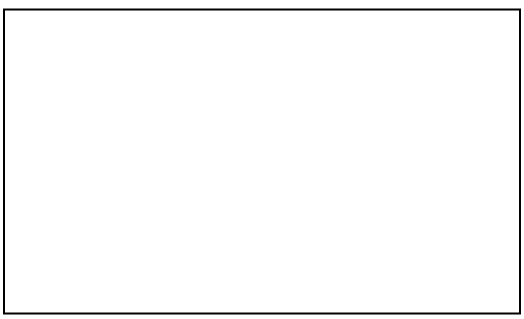
- Run the BandLimitedNoise.exe executable file from the BandLimitedNoise directory, or load the BandLimitedNoise.vcproj project into Visual C++ and compile and run it from there.

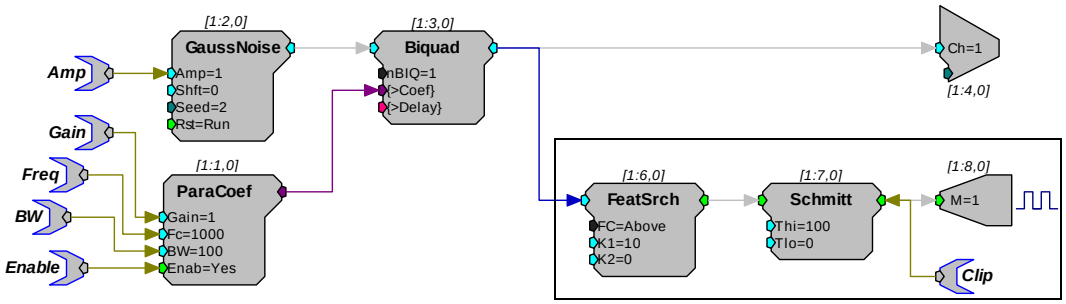
Making the RPvdsEx Circuit

Required components:

- Six parameter tags (ParTag). To change the name of a parameter tag, double-click the parameter and type a new name.
  - Gain - increases the relative bandpass filtering in dB
  - Freq - center frequency of the bandpass filter
  - BW - width of the bandpass filter (3 dB rolloff)
  - Amp - changes the amplitude of the noise
  - Enable - toggles generation of the filter coefficients
  - Clip - checks to see whether the signal was clipped or not
- Gaussian noise generator (GaussNoise)
- Parametric filter coefficient generator (ParaCoef)
- Biquad filter (Biquad)
- Feature search (FeatSrch)
- Schmitt trigger (Schmitt)
- Digital-to-analog converter (DacOut)
- Digital bit output (BitOut)

Connect the circuit as shown below. The RPx online help is accessible from within RPvdsEx if it is required. The two boxes represent the different parts of the circuit.

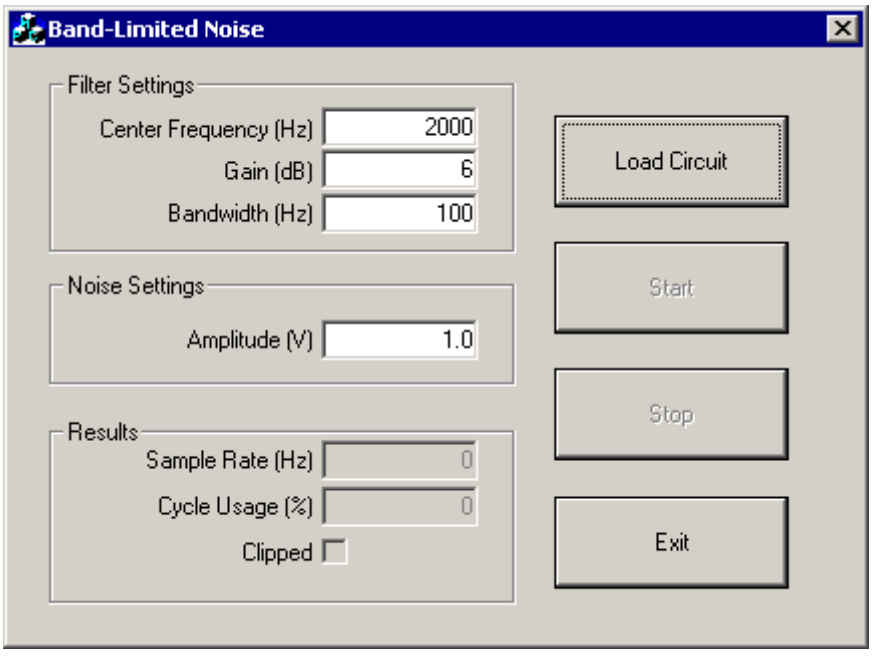




The box on the left has components that generate (GaussNoise) and filter (ParaCoef/Biquad) the waveform. The parameter tags are used to set the amplitude of the noise and filter parameters. The second part of the circuit (box on right) checks for clipping (signal values greater than +/- 10 volts) and generates a high signal on Bit 0 (M=1) of the processor device if clipping occurs.

Program Description

The Visual C++ program controls a circuit that generates band-limited noise. Buttons allow the user to load the circuit and start and stop playing of the noise. Through input boxes, the user controls the center frequency, bandwidth, filter gain, coefficient generation, and the intensity of the filtered noise. The sample rate and cycle usage are displayed, along with a checkbox that is marked if the parameters produce clipping (values beyond +/- 10 volts). The relevant code controls or receives information about the circuit through parameter tags. An ActiveX control is used for the RP2 device.



Relevant Code

The code below is run when the user clicks the Load Circuit button. It connects to the RP2, loads the circuit, and makes sure everything was loaded successfully.

```
if (m_rp2.ConnectRP2("GB", 1) ==0)
  if (m_rp2.ConnectRP2("USB", 1) == 0) {
    AfxMessageBox("Error connecting to RP2.");
    return;
  }
m_rp2.ClearCOF();
```

```
if(m_rp2.LoadCOF("C:\TDT\ActiveX\ActXExamples\RP_files\Band_Lim
ited_Noise.rcx") == 0) {
  AfxMessageBox("Error loading file");
  return;
}
// enable start button, disable stop button
m_start_button.EnableWindow(TRUE);
m_stop_button.EnableWindow(FALSE);
```

The code below is run when the user clicks the Start Circuit button. It sets the values of each parameter based on the values in the input boxes of the graphical interface. It then starts the circuit running, which plays the noise out of the RP2 on output channel number 1.

```
// set parameter values
UpdateData(TRUE);
m_rp2.SetTagVal("Amp", (float)atof(m_amplitude_text));
m_rp2.SetTagVal("Freq", (float)atof(m_centerfreq_text));
m_rp2.SetTagVal("BW", (float)atof(m_bandwidth_text));
m_rp2.SetTagVal("Gain", (float)atof(m_gain_text));
m_rp2.SetTagVal("Enable", (float)m_check_enable);
m_rp2.Run();
long status = m_rp2.GetStatus();
if (!(status && 4)) {
  AfxMessageBox("Error running circuit.");
  m_rp2.Halt();
}
```

Visual C++ Example: Continuous Acquire

This example uses a circuit that continually acquires data from an input channel into a 100,000 sample serial buffer at a rate of 100 kHz and a Visual C++ program that continually reads from the serial buffer in blocks of 50,000 samples and saves the data to a file.

ActiveX Methods Used

- ConnectRP2
- Run
- GetTagVal
- LoadCOF
- Halt
- ReadTag
- GetStatus
- SoftTrg

Files Used

The files required for this example can be found in:
C:\TDT\ActiveX\ActXExamples\vc++\ContinuousAcquire

- ContinuousAcquire.vcproj: Visual C++ project file
- ContinuousAcquireDlg.cpp: Visual C++ code that controls the graphical user interface and communicates with the RPvdsEx circuit; contains an ActiveX component for the processor devices

- ContinuousAcquire.exe: compiled executable; for running the example without having to start up Visual C++

The RPvdsExfile used can be found in: *C:\TDT\ActiveX\ActXExamples\RP_files*

- Continuous_Acquire.rcx: Control File of the RPvdsEx designed circuit

Required Hardware

- RP2

Required Applications

- RPvdsEx
- Visual C++

Running the Application

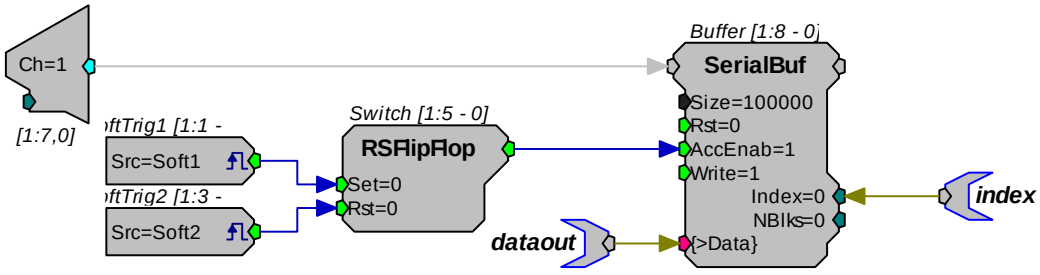
Run the ContinuousAcquire.exe executable file from the ContinuousAcquire directory, or load the ContinuousAcquire.vcproj project into Visual C++ and compile and run it from there. The program will produce an output file *C:\TDT\ActiveX\ActXExamples\VC++\fnoise2.f32*.

Making the RPvdsEx Circuit

Required components for acquisition:

- Two parameter tags (ParTag). To change the name of a parameter tag, double-click the parameter and type a new name.
  - dataout - points to the memory buffer
  - index - points to the index of the serial buffer
- Two software triggers (TrgIn, set to Soft1 and Soft2)
- Analog-to-digital converter (AdcIn)
- RS flip-flop (RSFlipFlop)
- Serial buffer (SerialBuf). To change the size of the serial buffer's memory, double-click the component and change the value for "Size" to 100000 (for this example)

Connect the circuit as shown below. The RPx online help is accessible from within RPvdsEx if it is required.

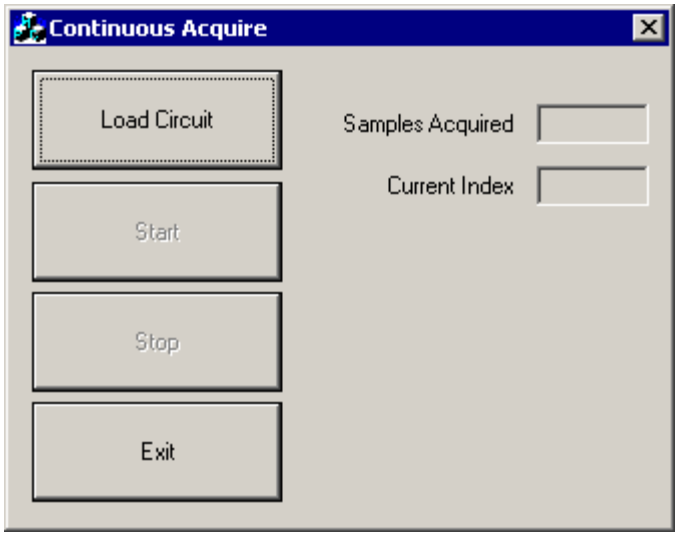


Data is continuously acquired on channel one but is only saved to the Serial buffer when the AccEnab line is set high. The two software triggers control the start and stop of the data acquisition. When Soft1 goes high the RSFlipFlop goes and stays high. This sets the AccEnab line high and the serial buffer starts saving the data. The serial buffer holds 100000 samples. When the buffer captures more than 100000 points the end of memory is reached, the index is reset to 0, and any data in memory is written over. When data cannot be downloaded to the PC fast enough it gets overwritten in the buffer.

To simulate real acquisition for this example, noise is played out on output channel 1 from the same circuit. This should be fed back in to input channel 1 to test acquisition.

Program Description

The Visual C++ program controls the continuous acquisition circuit described above. The graphical interface to the program consists of buttons for loading the RPvdsEx circuit, starting acquisition, stopping acquisition, and exiting the program. The number of samples acquired and the current index of the serial buffer are displayed while acquisition is taking place. The data is written to an output file called "fnoise2.f32". An ActiveX control is used for the RP2 device. A timer is used to synchronize reading of data from the buffer.



Relevant Code

The code below is run when the user clicks the Start Acquire button. It enables the timer and performs a software trigger to start acquisition.

```
m_rp2.SoftTrg(1);
SetTimer(1, 10, NULL);
```

The code below is run when the acquisition timer goes off (every 10 ms). It alternates between reading from the first half of the buffer and the second half of the buffer. There is also code to check the data transfer rate and make sure it is keeping up with the acquisition input.

```
if(acquire) {
  curindex = m_rp2.GetTagVal("index");
  m_index_text.Format("%f", curindex);
  UpdateData(FALSE);
  if(high) {
    while(curindex > bufpts) {
      curindex = m_rp2.GetTagVal("index");
      m_index_text.Format("%f", curindex);
      UpdateData(FALSE);
    }
  }
}
else {
  while(curindex < bufpts) {
    curindex = m_rp2.GetTagVal("index");
    m_index_text.Format("%f", curindex);
    UpdateData(FALSE);
```

```
  }
}
// Read segment and write it to file
if(m_rp2.ReadTag("dataout", data, offset, bufpts) == 0)
  AfxMessageBox("Error transferring data.");
WriteToFile(data, bufpts);
samples_acquired += bufpts;
m_samples_text.Format("%d", samples_acquired);
UpdateData(FALSE);
```

Visual C++ Example: Continuous Play

This example uses a circuit that continually plays to an output channel data from a 100,000 sample serial buffer at a rate of 100 kHz and a Visual C++ program that continually writes to the serial buffer in blocks of 50,000 samples.

ActiveX Methods Used

- ConnectRP2
- SoftTrg
- GetTagVal
- LoadCOF
- GetTagSize
- WriteTag
- Run

Files Used

The files required for this example can be found in:
C:\TDT\ActiveX\ActXExamples\vc++\ContinuousPlay

- ContinuousPlay.vcproj: Visual C++ project
- ContinuousPlayDlg.cpp: Visual C++ form; includes graphical interface and VB code; contains an ActiveX component for the RP2
- ContinuousPlay.exe: compiled executable; for running the example without having to start up Visual C++

The RPvdsEx file used can be found in: *C:\TDT\ActiveX\ActXExamples\RP_files*

- Continuous_Play.rcx: Control File of the RPvdsEx designed circuit

Required Hardware

- RP2

Required Applications

- RPvdsEx
- Visual C++

Running the Application

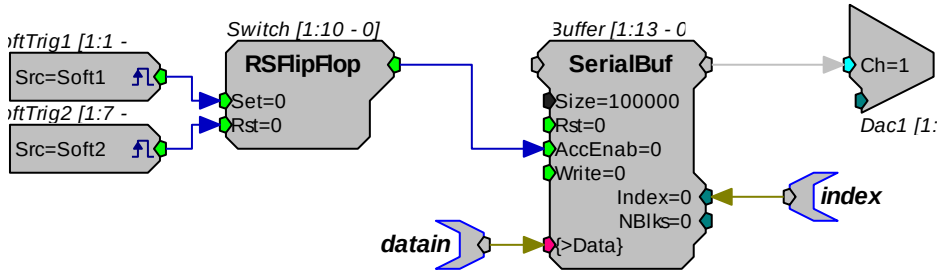
Run the ContinuousPlay.exe executable file from the ContinuousPlay directory, or load the ContinuousPlay.vcproj project into Visual C++ and compile and run it from there.

Making the RPvdsEx Circuit

Required components for acquisition:

- Two parameter tags (ParTag). To change the name of a parameter tag, double-click the parameter and type a new name.
  - datain - points to the memory buffer
  - index - points to the index of the serial buffer
- Two software triggers (TrgIn, set to Soft1 and Soft2)
- Digital-to-analog converter (DacOut)
- RS flip-flop (RSFlipFlop)
- Serial buffer (SerialBuf). To change the size of the serial buffer's memory, double-click the component and change the value for Size to 100000 (for this example).

Connect the circuit as shown below. The RPx online help is accessible from within RPvdsEx if it is required.

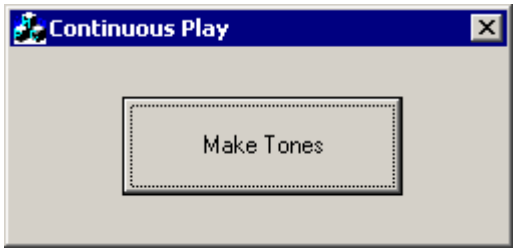


When software trigger 1 goes high the RSFlipFlop goes and stays high. This sets the AccEnab line high and the serial buffer starts sending the data out to the DAC. When the serial buffer has played out 100000 points the index is reset and the data at the beginning of the buffer is played out. As long as the AccEnab is high the Serial Buffer will play the signal.

Program Description

The program plays a series of tones for 10 seconds. The first second of tones is loaded to the serial buffer. A software trigger starts the counter and the signal is played out through the DAC. The Serial buffer index is polled until 50,000 points are played from the buffer. Another tone is generated and loaded to the second half of the buffer. The counter is polled until the next 50,000 points are played out and the cycle is repeated. The program checks to see if the transfer rate is fast enough when the data is written to the buffer. A final software trigger ends the play out.

The interface to the program consists of only a single button, which starts the playing process. An ActiveX control is used for the RP2 device.



Relevant Code

The code below contains the main playing loop. Each time through the loop, the tones are created at different frequencies. The first time through, the tones are written to the buffer immediately.

Each time after that, the sendtones() function is called. The playing process is terminated by the software trigger 2.

```
// For each iteration, load tones into arrays and send
   for (i = 0; i < num_iterations; i++) {
     freq1 += 500;
     freq2 += 500;
     for (int j = 0; j < bufpts; j++) {
       tone1[j] = (float) sin(2*PI*time[j]*freq1);
       tone2[j] = (float) sin(2*PI*time[j]*freq2);
     }
     if (i == 0) {
       // First time through
       m_rp2.WriteTag("datain", tone1, 0, bufpts);
       m_rp2.WriteTag("datain", tone2, bufpts, bufpts);
       m_rp2.SoftTrg(1);
     }
     else {
       SendTones(bufpts, tone1, tone2);
     }
   }
   // All done
   m_rp2.SoftTrg(2);
   m_rp2.Halt();
```

The code for the sendtones() function is shown below. It waits until the first half of the buffer is done playing, then writes the new tone to the first half of the buffer while the second half is being played. Then it ensures that the data was written to the buffer fast enough (otherwise the output is unreliable because the index buffer keeps looping continuously). After that, it waits until the second half is done playing, and then writes the new tone to the second half of the buffer. Again, the transfer rate is checked.

```
   // Send first tone to first half of buffer
   curindex = m_rp2.GetTagVal("index");
   while (curindex < bufpts) {
     curindex = m_rp2.GetTagVal("index");
     sleep(20);
   }
   m_rp2.WriteTag("datain", tone1, 0, bufpts);
   curindex = m_rp2.GetTagVal("index");
   if (curindex < bufpts) {
     AfxMessageBox("Error: transfer rate too slow.");
     m_rp2.SoftTrg(2);
     return;
   }
   // Send second tone to second half of buffer
   while (curindex > bufpts) {
     curindex = m_rp2.GetTagVal("index");
     sleep(20);
   }
   m_rp2.WriteTag("datain", tone2, bufpts, bufpts);
   curindex = m_rp2.GetTagVal("index");
   if (curindex > bufpts) {
     AfxMessageBox("Error: transfer rate too slow.");
     m_rp2.SoftTrg(2);
   }
```

Visual C++ Example: TDT ActiveX Console

This example documents a Visual C++ program that loads the Band_Limited_Noise.rcx control object file and runs it on the RP2 processor through the system console. This example illustrates how to create a formless application in Visual C++.

ActiveX Methods Used

- ConnectRP2
- LoadCOF
- ClearCOF
- Run

Files Used

The files required for this example can be found in:
C:\TDT\ActiveX\ActXExamples\vc++\TDT_ActiveX_Console

- TDT_ActiveX_Console.vcproj: Visual C++ project file
- TDT_ActiveX_Console.cpp: Visual C++ code that communicates with the RPvdsEx circuit; contains ActiveX components for the processor devices
- TDT_ActiveX_Console.exe: compiled executable; for running the example without having to start up Visual C++

Required Hardware

- RP2

Required Applications

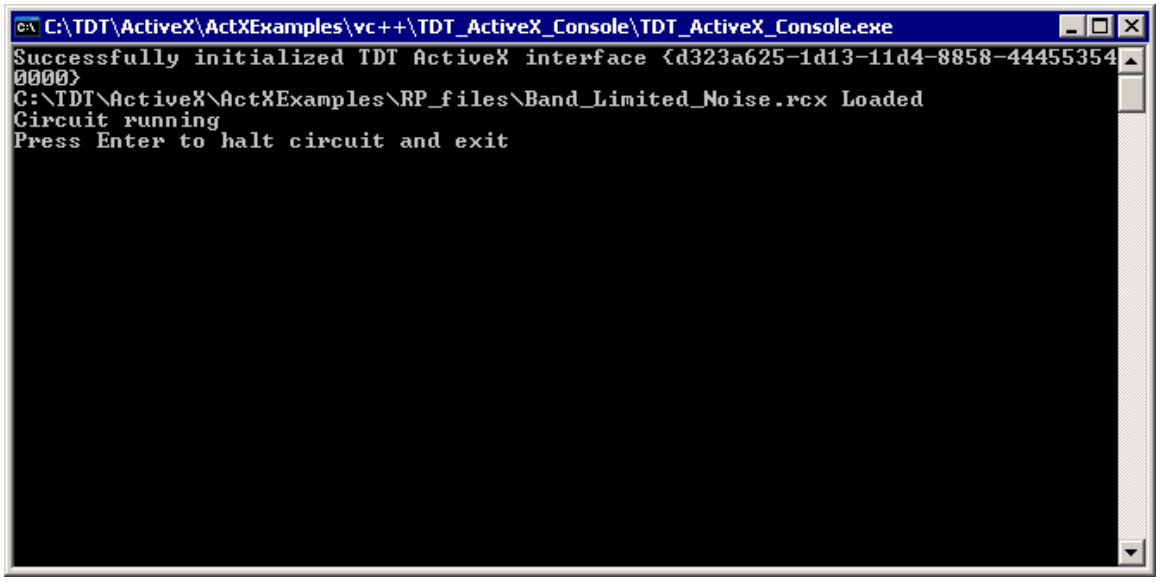
- Visual C++

Running the Application

Run the TDT_ActiveX_Console.exe executable file from the TDT_ActiveX_Console directory, or load the TDT_ActiveX_Console.vcproj project into Visual C++ and compile and run it from there.

Program Description

The Visual C++ program loads the Band_Limited_Noise.rcx control file and runs it on an RP2 processor device. The system console is used to connect to the device through an initialization to the RPcoX ActiveX control. Once initialized, the ActiveX control is used to control the RP2 processor.



Relevant Code

The code below is run when the user double clicks on the TDT_ActiveX_Console.exe executable file. It displays the system console and all connection information. Once the RPcoX ActiveX control has been initialized, the circuit can be loaded and run on the RP2 processor.

```
int _tmain(int argc, _TCHAR* argv[])
{
const char* circuitPath =
"C:\\TDT\\ActiveX\\ActXExamples\\RP_files\\Band_Limited_Noise.r
cx";
//Initialize ActiveX object
HRESULT hr;
hr = CoInitialize(NULL);
if (FAILED(hr)) {
  printf("Failed to initialize COM!\n");
}
const char* appId = "{d323a625-1d13-11d4-8858-444553540000}";
//"RPcoX.ocx"
hr = RP.CreateInstance(appId);
if (FAILED(hr)) {
  printf("CreateInstance for %s failed!\n", appId);
}
else {
  printf("Successfully initialized TDT ActiveX interface %s\n",
appId);
}
if (0 == RP) return -1;
```

Revision History

Version 7.3 – February 2012

Addition of support for 64-bit operating systems

Version 7.1 – May 4, 2010

Addition of support for the RZ6 Processor and ConnectRZ6 Method

Version 6.6 - August 15, 2007

Version 6.4 - January 23, 2007

Version 6.2 - September, 8, 2006

Version 6.0 - January 18, 2006

November 11, 2004 Version 5.8

Addition of New ActiveX controls to support RXn devices:

ConnectRX5, ConnectRX6, ConnectRX7, and ConnectRX8

April 15, 2003 Version 5.0

Addition of New ActiveX controls to support RMx devices:

ConnectRM1 and ConnectRM2

ActiveX Examples has been updated and expanded to include more Visual C++ examples.

Febuary 8, 2002 Version 4.2

Addition of Gigabit interface support and removal of XBUS interface support. See Connect device RPcoX. For how to connect to a device through the Gigabit interface.

January 8, 2002 Version 4.2

Fixed errors in ActiveX help relating to zBUS ActiveX methods.

Addition of a QuickStart Guide.

Addition of New ActiveX control:

LoadCOFsf: Allows users to select the sample rate of an rco(COF) file when the file is loaded.

August 8, 2001 Version 4.1

ActiveX controls for the RPx families of devices.

ConnectRA16: Connects to the medusa amplifiers.

ConnectRV8: Connects to the Barracuda precision event timer.

GetDevCfg: Accesses Device settings for the Barracuda.

SetDevCfg: Sets the Device settings for the Barracuda.

ActiveX and MATLAB 6.0

MATLAB 6.0 requires that all variables that are to be used in numerical operations be cast as Doubles. These operations include: +,-,.*,./,.^,: and others. Compare statements such as <,>,== do not need the variable to be of type double. To change your MATLAB code to work with MATLAB 6.0 requires that you cast the variable is a DOUBLE. For example freq=invoke(RPx,'GetTagVal','freq') should be changed to freq=double(invoke(RPx,'GetTagVal','freq'))in MATLAB 6.0. Note that the above values work in MATLAB 5.3. Matlab 7 supports math on integer and single-precision data.

March 5, 2001 Version 3.7

New Feature

Stingray Reader. A program for acquiring data from your RPx device.

ActiveX controls for the RP family of devices

ConnectRL2: Connects to the RL2 (Stingray device)

ReadCOF: Maps the parameter tags and memory of an rco file for access by the PC. Used with portable RPx devices.

ActiveX: controls for the zBUS.

ConnectzBUS: Makes a connection between the PC and the zBus.

FlushIO: Flushes the data buffer on the zBus.

GetDeviceAddr: Gets the address of a device type.

GetDeviceAt: Gets the device type at a particular address.

GetDeviceVer: Gets the correct version of the devices microcode.

GetError: Returns an error string.

HardwareReset: Resets the Stingray and deletes any processing chain running on the system.

zBusTrigA/B: Triggers multiple zBus racks/RPx components simultaneously

zSync: Synchronizes the zBus clocks across several racks.

Bug fixes

zBusTrig fully functional

zSync fully functional

Problems with ReadTagVex

Example Additions

Detect Circuit for use with the Stingray.

Sept. 05, 2000 Version 3.5

Folder with ActiveX examples for MATLAB.

Revision of Connect method: Each member of the Real-time Processor family has its Connect method. Use ConnectRP2 to connect to an RP2. Device type is a String variable ("XBUS", "USB" etc...)

New Methods:

GetStatus: Used to check device status.

GetCycUse: Checks the cycle usage of the device.

GetNameOf: Returns the String ID of a component

GetNumOf: Returns the number of Components in the *.rco file.

GetSFreq: Returns the sampling rate of the RP.

GetTagType: Determines the data type of the parameter tag.

GetTagSize: Returns the size of the data type.

ReadTagVEX: Reads data from a memory buffer and stores it in multiple data types and formats.

WriteTagVEX: Writes several types of formatted data to a memory buffer.

ZeroTag: Sets Parameter Tag values to zero.

Known Anomalies

Note: Anomalies and tech notes are also available on the Web at: https://www.tdt.com/technotes/.

When using the GetStatus method with RX devices, the method returns erroneous values. RX devices return higher bit information and this causes issues with the status values described in the ActiveX help documentation. To access relevant status information in Matlab, use 'bitget' (or the equivalent in other programming languages) to read each bit directly.

e.g. If all(bitget(RP.GetStatus,1:3));.

When using Delphi, ActiveX controls cannot be updated. Delphi remembers the older version of the ActiveX controls. To update to a new version of ActiveX controls, first uninstall the earlier version (i.e. remove it from the Delphi interface) and then install the new version.

Several errors occur when using ActiveX with MATLAB 6.0 and above. The main problem occurs when calling the invoke function, e.g. status = invoke(RP, 'GetStatus');. When using the return value of some of these calls, errors such as "function ___ not defined for variables of class 'int32'." result. To solve this problem cast the return values as doubles,

e.g. status = double(invoke(RP, 'GetStatus'));.

Although ActiveX seems to connect and properly load a circuit to the RA16BA (Medusa Base Station), the GetStatus method will consistently return a 0 for connection status when a preamplifier is not properly connected to the base station. Connection Status is located in the least significant bit for the GetStatus command. When checking the status of the base station, ensure that the preamplifier is properly connected and turned on.

The zBusSync ActiveX Command is used for synchronizing caddies with USB1.1 (UZ1/UZ4) interfaces and should not be used with other types of interfaces.

Calling ReadTagV with Matlab 6.5 with the characters 'readtagv' (all lowercase) will cause a memory leak of 8 bytes per point returned. Calling GetTagVal with Matlab 7.0 with the characters 'gettagval' (all lowercase) will cause a memory leak of 40 bytes per function invocation.

Using the ActiveX methods ConnectRxx (e.g. ConnectRX6, ConnectRP2 etc.) more than once can sometimes cause a communication failure.

Version 57 or greater

Invoking the ActiveX zTrigA or zTrigB calls always returns a zero, irrespective of the actual result.

HardwareReset returns a 0 if the hardware reset was performed successfully or not.

Index