COM Automation

Online Help

KEYSIGHT TECHNOLOGIES

Notices

Revision History

May 2017

Available in electronic format only

Keysight Technologies.
1900 Garden of the Gods Road
Colorado Springs, CO 80907 USA

Warranty

THE MATERIAL CONTAINED IN THIS DOCUMENT IS PROVIDED "AS IS," AND IS SUBJECT TO BEING CHANGED, WITHOUT NOTICE, IN FUTURE EDITIONS. FURTHER, TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, KEYSIGHT DISCLAIMS ALL WARRANTIES, EITHER EXPRESS OR IMPLIED WITH REGARD TO THIS MANUAL AND ANY INFORMATION CONTAINED HEREIN, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. KEYSIGHT SHALL NOT BE LIABLE FOR ERRORS OR FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES IN CONNECTION WITH THE FURNISHING, USE, OR PERFORMANCE OF THIS DOCUMENT OR ANY INFORMATION CONTAINED HEREIN. SHOULD KEYSIGHT AND THE USER HAVE A SEPARATE WRITTEN AGREEMENT WITH WARRANTY TERMS COVERING THE MATERIAL IN THIS DOCUMENT THAT CONFLICT WITH THESE TERMS, THE WARRANTY TERMS IN THE SEPARATE AGREEMENT WILL CONTROL.

Technology Licenses

The hardware and/or software described in this document are furnished under a license and may be used or copied only in accordance with the terms of such license.

U.S. Government Rights

The Software is "commercial computer software," as defined by Federal Acquisition Regulation ("FAR") 2.101. Pursuant to FAR 12.212 and 27.405-3 and Department of Defense FAR Supplement ("DFARS") 227.7202, the U.S. government acquires commercial computer software under the same terms by which the software is cu tomarily provided to the public. Accordingly, Keysight provides the Software to U.S. government customers under its standard commercial license, which is embodied in its End User License Agreement (EULA), a copy of which can be found at http://www.keysight.com/find/sweula. The license set forth in the EULA represents the exclusive authority by which the U.S. government may use, modify, distribute, or disclose the Software. The EULA and the license set forth therein, does not require or permit, among other things, that Keysight: (1) Furnish technical information related to commercial computer software or commercial computer software documentation that is not customarily provided to the public; or (2) Relinquish to, or otherwise provide, the government rights in excess of these rights customarily provided to the public to use, modify, reproduce, release, perform, display, or disclose commercial computer software or commercial computer software documentation. No additional government requirements beyond those set forth in the EULA shall apply, except to the extent that those terms, rights, or licenses are explicitly required from all providers of commercial computer software pursuant to the FAR and the DFARS and are set forth specifically in writing elsewhere in the EULA. Keysight shall be under no obligation to update, revise or otherwise modify the Software. With respect to any technical data as defined by FAR 2.101, pursuant to FAR 12.211 and 27.404.2 and DFARS 227.7102, the U.S. government acquires no greater than Limited Rights as defined in FAR 27.401 or DFAR 227.7103-5 (c), as applicable in any technical data. 52.227-14 (June 1987) or DFAR 252.227-7015 (b)(2) (November 1995), as applicable in any technical data.

Safety Notices

CAUTION

A CAUTION notice denotes a hazard. It calls attention to an operating procedure, practice, or the like that, if not correctly performed or adhered to, could result in damage to the product or loss of important data. Do not proceed beyond a CAUTION notice until the indicated conditions are fully understood and met.

WARNING

A WARNING notice denotes a hazard. It calls attention to an operating procedure, practice, or the like that, if not correctly performed or adhered to, could result in personal injury or death. Do not proceed beyond a WARNING notice until the indicated conditions are fully understood and met.

COM Automation—At a Glance

The *Keysight Logic Analyzer* application includes the COM Automation Server. This software lets you write programs that control the *Keysight Logic Analyzer* application from remote computers on the Local Area Network (LAN).

Contents

4 COM Automation Reference

1 COM Automation Overview

In some test and measurement environments, the process of making a measurement and analyzing the results become routine or repetitive. In other environments, it may be more convenient to make measurements or analyze data from a remote PC. Whatever the situation, you can benefit from performing measurement tasks programmatically through a Visual C++ or Visual Basic program.

The COM Automation Server is part of the *Keysight Logic Analyzer* application. It gives PC applications a COM interface to the logic analyzer (see Figure 1). This lets you write programs that communicate with the logic analyzer using a COM model definition and take advantage of the ease of programming offered by the Visual Studio Environment (that is, Visual Basic or Visual C++).

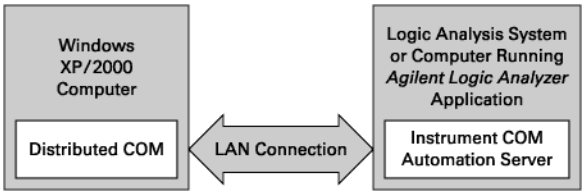


Figure 1 COM Automation Architecture

By executing programs using the Instrument COM Automation Server, you manipulate the logic analysis environment and its functional components as Objects. You manipulate objects by using the properties and methods associated with the objects. Methods represent actions you take against the objects. Properties represent characteristics of the objects, such as their type or size.

Each object implements a dual interface through which you can manipulate the object. Each object implements an IDispatch interface for Automation and a Component Object Model (COM) interface for direct access to object members (properties and methods). By importing the Instrument Automation Server's type library, you can employ early binding by using the COM interface. Early binding makes all calls into interface members faster at run time.

For more information on Logic Analyzer Objects and the components that manipulate them, refer to Object Hierarchy Overview (see page 77).

2 Setting Up for COM Automation

A remote computer connected to the logic analyzer via LAN uses Distributed COM (DCOM) to control the logic analyzer. COM is used when the logic analyzer is controlled from within the logic analyzer itself. Because COM connections work without any additional configuration, this getting started section only pertains to Distributed COM.

The following assumptions are made in this getting started section. Verify the validity of each before proceeding.

- Both the remote computer and logic analyzer are on the same LAN and both can "see" each other in "My Network Places" (or communicate with each other using the ping command in a Command Prompt window). See supported networking configurations (see page 14) for more information.
- The logic analyzer is running version 02.00 or later of the *Keysight Logic Analyzer* application.
- You are reasonably familiar with Windows.

Setting up your remote computer to communicate with the logic analyzer requires the following three summarized steps:

1 Install the LA COM Automation client software (see page 15) on your remote computer.

2 Test your Distributed COM connection (see page 16).

See Also

- Using COM Automation (see page 21)

Supported Networking Configurations

There are many ways of setting up your remote computer to communicate with the logic analyzer, but due to the security requirements of Distributed COM, only two configurations are supported. The first configuration is when both the remote computer and the logic analyzer are members of a "Workgroup", and the second configuration is when both the remote computer and the logic analyzer are members of a "Domain".

Keysight 16850-series and 16860-series logic analyzers are shipped from the factory such that "Everyone" has permission to launch and access the *Keysight Logic Analyzer* application via Distributed COM. The term "Everyone" refers to a different range of users depending on whether the logic analyzer is a member of a Domain or Workgroup. By default, the logic analyzer is configured as a member of a workgroup. Therefore, "Everyone" includes only those users who have been given logon accounts on the logic analyzer.

Workgroup

A workgroup is established by the logic analyzer administrator declaring the workgroup name and declaring the logic analyzer as a member of the workgroup. A workgroup does not require a network administrator to create it or control membership.

"Everyone" includes only those users who have been given logon accounts on the logic analyzer. By default, the logic analyzer is configured as members of a workgroup named WORKGROUP.

NOTE To set up a logon account for a new user, see the operating system's online help. For Distributed COM access, the user's account name and password must EXACTLY match their remote computer logon account name and password.

NOTE Recent Windows security patches require passwords to be set on accounts before Distributed COM access is allowed.

Domain

A domain is typically a large organizational group of computers. Network administrators maintain the domain and control which machines have membership in it.

"Everyone" includes those people who have membership in the domain. In addition, those with logon accounts can also access the analyzer.

See Also

- "*Keysight Logic Analyzers Isolated Network Setup White Paper*"

Step 1. Install the LA COM Automation client software

You can install either the Keysight Logic and Protocol Analyzer software package (on instruments or PCs hosting instruments) or the Keysight Logic and Protocol Analyzer COM Automation software package (on machines that do not need the full LPA software installed).

Because the logic analyzer and remote computer are on the same LAN, you can install the COM automation client software from the logic analyzer:

1. If you have changed the Windows firewall settings on the logic analyzer to allow remote access to shared folders, then on your remote computer, map a network drive to the logic analyzer (for example, \\computer-name\C$).
2. Navigate to the \Program Files\Keysight Technologies\Logic Analyzer directory on the mapped network drive.
3. Locate the "SetupLACOM.exe" file, and run it to install the LA COM Automation client software on your remote computer.

Next

- Step 2. Test your Distributed COM connection (see page 16)

Step 2. Test your Distributed COM connection

Once you have installed Keysight Logic and Protocol Analyzer install package or the Keysight Logic and Protocol Analyzer COM Automation install package, you can run the following tools to test your COM/Distributed COM connection to the logic analyzer or diagnose COM/DCOM connection issues.

- COM testing for 64-bit, unmanaged applications/scripts - C:\Program Files\Keysight Technologies\Logic Analyzer\LA COM Automation\agTestClient.exe
- COM testing for 32-bit, unmanaged applications/scripts: C:\Program Files\Keysight Technologies\Logic Analyzer\LA COM Automation\agTestClient_x86.exe
- COM testing for managed applications: C:\Program Files\Keysight Technologies\Logic Analyzer\COMConnectionTool.exe

If the COM Connection Tool does not resolve the problem, see also Distributed COM Troubleshooting (see page 17).

Distributed COM Troubleshooting

If the Client Test program (see page 16) fails to connect to the logic analyzer:

- Make sure the remote computer and logic analyzer can "see" each other in "My Network Places" (or communicate with each other using the ping command in a Command Prompt window).
- Make sure the logic analyzer is running version 02.00 or later of the *Keysight Logic Analyzer* application.
- See supported networking configurations (see page 14). If you are in a Workgroup, check that both the account name and password used on both the logic analyzer and remote computer match EXACTLY. Also, if you are in a Workgroup and have the Windows XP operating system, you must turn off simple file sharing (see page 17).

NOTE **Recent Windows security patches require passwords to be set on accounts before Distributed COM access is allowed.**

- Make sure you are logged in to the logic analyzer so the user and privileges are assigned correctly. The *Keysight Logic Analyzer* application does not have to be running; if it isn't, connecting via COM will automatically start it.
- To verify logic analyzer machine-wide Distributed COM properties (see page 17)
- To verify logic analyzer application Distributed COM properties (see page 18)
- To verify remote computer application Distributed COM properties (see page 19)
- Make sure the logic analyzer allows DCOM access though the firewall. Some IT departments are now automatically installing firewalls onto remote client computers. Verify your remote client computer also allows DCOM access through the firewall if one is installed.
- If you have a remote client computer that cannot connect to the logic analyzer and one that can, run ipconfig -all in the Command Prompt window on both computers to see how their LAN configurations differ. This may help in troubleshooting the problem.

To turn off simple file sharing

If both the remote computer and the logic analyzer are members of a "Workgroup" (see supported networking configurations (see page 14)) and the logic analyzer has the Windows XP operating system, you must turn off simple file sharing.

1 Open Windows Explorer (or double-click My Computer).
2 From the Windows Explorer menu, choose Tools>Folder Options....
3 In the Folder Options dialog, select the View tab.
4 In the "Advanced settings" options list, uncheck Use simple file sharing (Recommended).
5 Click OK to close the Folder Options dialog.

To verify logic analyzer machine-wide Distributed COM properties

Normally, the logic analyzer Distributed COM configuration is set at the factory. If this has been changed, you may have to set it back to the default settings.

To verify the machine-wide Distributed COM properties on the computer that runs the *Keysight Logic Analyzer* application:

1 From the Windows task bar choose Start>Run..., enter DCOMCNFG.EXE as the name of the program to open, and click OK.
2 Access the machine-wide Distributed COM properties, security, and protocols tabs:

- In the left-side pane of the Component Services window, browse to the Console Root, Component Services, Computers folder; then, right-click on My Computer and choose Properties from the popup menu.

3 In the Default Properties tab:
   a Check the Enable Distributed COM on this computer option.
   b For the Default Authentication Level, select Connect.
   c For the Default Impersonation Level, select Identify.
4 In the COM Security tab:
   a Under Access Permissions, click Edit Limits....
   b In the Access Permission dialog, make sure the Everyone account has "Allow" checked for both Local Access and Remote Access.
   c Click OK to close the Access Permission dialog.
   d Under Launch and Activation Permissions, click Edit Limits....
   e In the Launch Permission dialog, make sure the MACHINE\Administrators and Everyone accounts have "Allow" checked for: Local Launch, Remote Launch, Local Activation, and Remote Activation.
   f Click OK to close the Launch Permission dialog.
5 In the Default Protocols tab:
   a Make sure Connection-oriented TCP/IP is listed first.
6 Click OK to close the properties dialog.

To verify logic analyzer application Distributed COM properties

Normally, the logic analyzer Distributed COM configuration is set at the factory. If this has been changed, you may have to set it back to the default settings.

To verify the application's Distributed COM properties on the computer that runs the *Keysight Logic Analyzer* application:

1 From the Windows task bar choose Start>Run..., enter DCOMCNFG.EXE as the name of the program to open, and click OK.
2 Open the Keysight 168x/169x/169xx Logic Analyzer Properties dialog:
   - In the left-side pane of the Component Services window, browse to Console Root, Component Services, Computers, My Computer, DCOM Config, Keysight 168x/169x/169xx Logic Analyzer; then, right-click and choose Properties from the popup menu.
3 In the Keysight 168x/169x/169xx Logic Analyzer Properties dialog, verify the following settings under each Tab heading indicated below.

Tab	Settings
General	Authentication Level should be set to "Default".
Location	Set to "Run application on this computer".

Security	Use custom launch and activation permissions. Verify "Everyone" has launch and activation permissions. If not: • Add "Everyone" and make sure "Allow" is checked for Local Launch, Remote Launch, Local Activation, and Remote Activation. Use custom access permissions. Verify "Everyone" has access permission. If not: • Add "Everyone" and make sure "Allow" is checked for Local Access and Remote Access. Use default configuration permissions.
Endpoints	Leave at default system protocols.
Identity	Set to "The interactive user".

4 In the Keysight 168x/169x/169xx Logic Analyzer Properties dialog, click OK.

To verify remote computer application Distributed COM properties

Normally, the remote computer Distributed COM configuration is set when you install the LA COM Automation client software. If this has been changed, you may have to set it back to the default settings.

To verify the application's Distributed COM properties on the remote computer:

1 From the Windows task bar choose Start>Run..., enter DCOMCNFG.EXE as the name of the program to open, and click OK.

2 Open the Properties dialog for Keysight 168x/169x/169xx Logic Analyzer:

a In the Component Services window, navigate the hierarchy tree to Component Services>Computers>My Computer>DCOM Config.

b In the DCOM Config folder, right-click on Keysight 168x/169x/169xx Logic Analyzer and choose Properties from the popup menu.

3 In the Keysight 168x/169x/169xx Logic Analyzer Properties dialog, verify the following settings under each Tab heading indicated below.

Tab	**Settings**
General	Authentication Level should be set to "Default".
Location	Normally, none of these options are selected, and the name of the computer on which to run the application is specified in the remote program (see the **Connect (see** page 90) object's **Instrument (see** page 202) property). However, the "Run application on the following computer" option can be checked, with the logic analyzer's computer name entered in the field that follows.
Security	Use default launch (and activation if on Windows XP) permissions. Use default access permissions. Use Custom configuration permissions. Verify that you have "Full Control".
Endpoints	Leave at default system protocols.
Identity	Normally, this tab does not appear (but it can if the *Keysight Logic Analyzer* application has been previously installed on the remote computer). If this tab appears, set to "The interactive user".

4 In the Keysight 168x/169x/169xx Logic Analyzer Properties dialog, click OK.

5 Close the Component Services window.

3 Using COM Automation

To programmatically control the logic analyzer via COM automation, you can use the Microsoft Visual Basic for Applications (VBA) or you can install any other COM aware client software package like Visual Basic, Visual C++, LabVIEW, VEE, etc.

Using Visual Basic for Applications (VBA) in Microsoft Excel

1 Install the Microsoft Excel software.

2 Import the type library:

a In the Visual Basic Editor, choose the Tools>References... menu item.

b In the References dialog, select the library "Keysight 168x/169x/169xx Logic Analyzer Object Library".

3 In the Excel Visual Basic Editor, copy and paste the GetLAData() code below. (In Excel 2000: Execute Tools>Macro>Macros.... In the Macro Name box, type in "GetLAData". Then, press the Create button.)

4 Optionally, call the GetLAData() macro from a custom toolbar button, and watch the Worksheet update with the logic analysis data.

Example

```
Sub GetLAData()

' This Excel macro example transfers all of the bus/signal's
' from the first module in a 168x/9x/9xx Logic Analysis System to
' the Active Excel Worksheet.  Variables to modify are:
'
'  myInst    -> change to the hostname or IP address of the LA
'               you're connecting to (default is 'localhost'
'               if you're running Excel directly on the LA)
'  mySheet   -> change the worksheet to copy the data to
'               (default is the active worksheet)
'  myAnalyzer-> change to the analyzer name to transfer data from
'               (default is the first module)
'  myStartSample, myEndSample -> change to the data range to upload
'               (default is -10 and 10 respectively)
'

' Get the active Excel worksheet
Dim mySheet As Worksheet
Set mySheet = ActiveWorkbook.ActiveSheet

' Clear all of the cells in the worksheet
mySheet.Cells.ClearContents

' Create the 168x/9x/9xx Logic Analyzer Instrument object
' and connect to the Logic Analyzer
'
Dim myConnect As AgtLA.Connect
Dim myInst As AgtLA.Instrument
Set myConnect = CreateObject("AgtLA.Connect")
Set myInst = myConnect.Instrument("localhost")

' Run the measurement, wait for completion or time out
myInst.Run
myInst.WaitComplete (10)

' Get the first analyzer module
Dim myAnalyzer As AgtLA.AnalyzerModule
Set myAnalyzer = myInst.Modules(0)

' Upload a range of acquired data and copy to the Excel worksheet
Dim myBusSignal As AgtLA.BusSignal
```

```
Dim myData As AgtLA.SampleBusSignalData

Dim myNumDataRows As Long
Dim myStartSample As Long
Dim myEndSample As Long

colNum = 1           ' Start putting the data in the first column
myStartSample = -10  ' Sample data range to upload
myEndSample = 10

' Copy over all bus/signals
For Each myBusSignal In myAnalyzer.BusSignals
  Set myData = myBusSignal.BusSignalData
  mySheet.Cells(1, colNum) = myBusSignal.Name

  Select Case myBusSignal.BusSignalType
    Case AgtBusSignalSampleNum
      Dim lArray() As Long
      lArray = myData.GetDataBySample(myStartSample, myEndSample, _
          AgtDataLong, myNumDataRows)
      For rowNum = 0 To myNumDataRows - 1
        ' Rows start with 1, and the bus/signal name is on the first
        ' row; so, add 2.
        mySheet.Cells(rowNum + 2, colNum) = lArray(rowNum)
      Next rowNum
    Case AgtBusSignalTime
      Dim dArray() As Double
        dArray = myData.GetDataBySample(myStartSample, myEndSample, _
            AgtDataTime, myNumDataRows)
      For rowNum = 0 To myNumDataRows - 1
        mySheet.Cells(rowNum + 2, colNum) = dArray(rowNum)
      Next rowNum
    Case AgtBusSignalGenerated
      Dim vArray As Variant   ' Decimal holds max 96 bits unsigned.
      vArray = myData.GetDataBySample(myStartSample, myEndSample, _
          AgtDataDecimal, myNumDataRows)
      For rowNum = 0 To myNumDataRows - 1
        mySheet.Cells(rowNum + 2, colNum) = vArray(rowNum)
      Next rowNum
    Case AgtBusSignalProbed
      ' Long holds a maximum of 31 bits unsigned.
      lArray = myData.GetDataBySample(myStartSample, myEndSample, _
          AgtDataLong, myNumDataRows)
      For rowNum = 0 To myNumDataRows - 1
        ' format has hex for display purposes
        mySheet.Cells(rowNum + 2, colNum) = Hex$(lArray(rowNum))
      Next rowNum
 End Select

  ' Go to the next bus/signal
  colNum = colNum + 1
Next

End Sub
```

See Also

- Example Visual Basic Programs (see page 25)

Using Visual Basic (in Visual Studio)

Before you can use the Visual Basic programming environment to control the *Keysight Logic Analyzer* application, you must first import the Instrument Automation Server's type library into your project.

1 Choose the Project>References... menu item.

2 In the References dialog, select the library "Keysight 168x/169x/169xx Logic Analyzer Object Library".

See Also

- Example Visual Basic Programs (see page 25)

Example Visual Basic and Visual C++ Programs

- Loading, Running, Storing (see page 25)
- Setting Up Simple Triggers (see page 29)
- Setting Up Advanced Triggers (see page 32)
- Changing the Sampling Mode (see page 36)
- Checking the Logic Analyzer Software Version (see page 40)

See Also

- Additional Visual Basic Examples (see page 47)
- Using Visual Basic for Applications (VBA) in Microsoft Excel (see page 22)
- Using Visual Basic (in Visual Studio) (see page 24)

Loading, Running, Storing

In order to create an easy to use, yet powerful remote control mechanism, the design of the COM Automation Server adheres to the basic use model of "load-run-store".

In other words, to create a remote control application or a program that runs repetitive tests:

1 Use the *Keysight Logic Analyzer* application to go through each test once, and save the logic analyzer configurations and trigger setup specifications to files.

2 Then, from your program, load the appropriate logic analyzer configuration and trigger setup specification files, run the measurement, and store or act on the results as appropriate.

Example

If you encounter any name collisions (in other words, if you already have an object defined that uses the same name as an object in the Instrument Automation Server library), you can use the "AgtLA" library name prefix to resolve the conflict. For example, if you have a "Module" object defined, you can use "AgtLA.Module" to refer to the Instrument Automation Server's "Module" object.

Visual Basic

```
' You must create the Connect object (see page 90) and use it
' to access the Instrument object.  In this example, "myInst"
' represents the Instrument object.
'
' Load the configuration file.
myInst.Open ("c:\LA\Configs\mpc860_demo_compare.ala")

' Load the logic analyzer trigger file.
Dim myAnalyzer As AgtLA.AnalyzerModule
Set myAnalyzer = myInst.GetModuleByName("My 1690A-1")
myAnalyzer.RecallTriggerByFile ("c:\LA\Triggers\TrigSpecFile.xml")

' Run the measurement, wait for it to complete.
myInst.Run
myInst.WaitComplete (20)

' Process/display/store captured data.
Dim myBusSignal As AgtLA.BusSignal
Dim myData As AgtLA.SampleBusSignalData

For Each myBusSignal In myAnalyzer.BusSignals
   ' Get Data from "ADDR".
   If myBusSignal.Name = "ADDR" Then
      Set myData = myBusSignal.BusSignalData

      'Upload a range of acquired data.
      Dim myArray() As Long   ' The size is defined in GetDataBySample.
```

```
      Dim NumRows As Long
      myArray = myData.GetDataBySample(-10, 10, AgtDataLong, NumRows)

      ' Find the largest bus/signal value.
      Dim LongValue As Long
      Dim LargestValue As Long
      LargestValue = 0
      For i = 0 To NumRows - 1
         LongValue = myArray(i)
         If LongValue > LargestValue Then
            LargestValue = LongValue
         End If
      Next i
      MsgBox "Largest value is: " + Str(LargestValue)

   End If
Next
```

Visual C++

```
//
//  This simple Console application demonstrates how to use the
//  Keysight 168x/9x/9xx COM interface from Visual C++.
//
//  This project was created in Visual C++ Developer.  To create a
//  similar project:
//
//    - Execute File -> New
//    - Select the Projects tab
//    - Select "Win32 Console Application"
//    - Select A "hello,World!" application (Visual Studio 6.0)
//
//  To make this buildable, you need to specify your "import" path
//  in stdafx.h (search for "TODO")
//
//  To run, you need to specify the host Logic Analyzer to connect
//  to (search for "TODO")
//

#include "stdafx.h"

////////////////////////////////////////////////////////////////////////
//
//  Forward declarations.
//
void DisplayError(_com_error& err);

////////////////////////////////////////////////////////////////////////
//
//  main() entry point.
//
int main(int argc, char* argv[])
{
   printf("*** Main()\n");

   //
   //  Initialize the Microsoft COM/ActiveX library.
   //
   HRESULT hr = CoInitialize(0);
```

```
if (SUCCEEDED(hr))
{
   try {   // Catch any unexpected run-time errors.
      _bstr_t hostname = "mtx33";   // myLAHostname.
      printf("Connecting to instrument '%s'\n", (char*) hostname);

      // Create the connect object and get the instrument object.
      AgtLA::IConnectPtr pConnect =
            AgtLA::IConnectPtr(__uuidof(AgtLA::Connect));
      AgtLA::IInstrumentPtr pInst =
            pConnect->GetInstrument(hostname);

      // Load the configuration file.
      _bstr_t configFile = "C:\\LA\\Configs\\config.ala";
      printf("Loading the config file '%s'\n", (char*) configFile);
      pInst->Open(configFile, FALSE, "", TRUE);

      // Get a specific analyzer module.
      _bstr_t moduleName = "MPC860 Demo Board";
      AgtLA::IAnalyzerModulePtr pAnalyzer =
            pInst->GetModuleByName(moduleName);

      // Load the logic analyzer trigger file.
      _bstr_t triggerFile = "C:\\LA\\Configs\\trigger.xml";
      printf("Loading the trigger file '%s'\n", (char*) triggerFile);
      pAnalyzer->RecallTriggerByFile(triggerFile);

      // Run the measurement, wait for it to complete.
      pInst->Run(FALSE);
      pInst->WaitComplete(20);

      // Process/display/store captured data.
      _bstr_t busSignal;
      _variant_t varArray;
      long numRowsRet;
      long numBytesPerRow;

      AgtLA::IBusSignalsPtr pBusSignals = pAnalyzer->GetBusSignals();
      for (long i = 0; i < pBusSignals->GetCount(); i++)
      {
         busSignal = pBusSignals->GetItem(i)->GetName();

         // Get data from "ADDR" bus.
         if (strcmp(busSignal, "ADDR") == 0)
         {
            long numSamples;
            long lBound;

            AgtLA::IBusSignalPtr pBusSignal =
                  pAnalyzer->GetBusSignals()->GetItem(busSignal);
            AgtLA::ISampleBusSignalDataPtr pSampleData =
                  pBusSignal->GetBusSignalData();
            varArray = pSampleData->GetDataBySample(-10, 10,
                  AgtLA::AgtDataRaw, &numRowsRet);
            numBytesPerRow = pBusSignal->GetByteSize();
            HRESULT hr = SafeArrayGetLBound(varArray.parray, 1,
```

```
                        &lBound);

                    if (SUCCEEDED(hr))
                    {
                       long uBound;
                       hr = SafeArrayGetUBound(varArray.parray, 1, &uBound);

                       if (SUCCEEDED(hr))
                       {
                          byte* pByteArray;
                          hr = SafeArrayAccessData(varArray.parray,
                                (void**) &pByteArray);

                          if (SUCCEEDED(hr))
                          {
                             numSamples =
                                   (uBound - lBound + 1) / numBytesPerRow;
                             byte* pByte = pByteArray;
                             printf("Displaying '%s' ", (char*) moduleName);
                             printf("module\'s ADDR bus samples from -10 ");
                             printf("to 10:\n");

                             for (int i = 0; i < numSamples; i++)
                             {
                                printf("  sample[%d]: ", -10 + i);

                                for (int j = 0; j < numBytesPerRow; j++)
                                {
                                   printf("%02x ", pByte[j]);
                                }

                                pByte += numBytesPerRow;
                                printf("\n");
                             }

                             printf("\n");
                             SafeArrayUnaccessData(varArray.parray);
                          }
                       }
                    }
                 }
              }
              catch (_com_error& e) {
                 DisplayError(e);
              }

              // Uninitialize the Microsoft COM/ActiveX library.
              CoUninitialize();
           }
           else
           {
              printf("CoInitialize failed\n");
           }

           return 0;
        }
```

```
////////////////////////////////////////////////////////////////////
//
//  Displays the last error -- used to show the last exception
//  information.
//
void DisplayError(_com_error& error)
{
   printf("*** DisplayError()\n");

   printf("Fatal Unexpected Error:\n");
   printf("  Error Number = %08lx\n", error.Error());

   static char errorStr[1024];
   _bstr_t desc = error.Description();

   if (desc.length() == 0)
   {
      // Don't have a description string.
      strcpy(errorStr, error.ErrorMessage());
      int nLen = lstrlen(errorStr);

      // Remove funny carriage return ctrl<M>.
      if (nLen > 2 && (errorStr[nLen - 2] == 0xd))
      {
         errorStr[nLen - 2] = '\0';
      }
   }
   else
   {
      strcpy(errorStr, desc);
   }

   printf("  Error Message = %s\n", (char*) errorStr);
}
```

Setting Up Simple Triggers

This example shows how to set up simple triggers using the SimpleTrigger (see page 177) method of the AnalyzerModule (see page 80) object.

Visual Basic

```
'You must create the Connect object (see page 90) and use it
' to access the Instrument object.  In this example, "myInst"
' represents the Instrument object.
'
' Load the configuration file.
myInst.Open ("c:\LA\Configs\mpc860_demo_compare.ala")

' Declare trigger variables.
Dim mySimpleTriggers(2) As String
mySimpleTriggers(0) = "ADDR=hfff034d8"
mySimpleTriggers(1) = "ADDR=h00004088 And DATA=h46xxxxxx"
mySimpleTriggers(2) = "ADDR=h000041ad And DATA=h47xxxxxx"
Dim I As Integer

' Set up triggers using the SimpleTrigger method.
Dim myAnalyzer As AgtLA.AnalyzerModule
```

```
Set myAnalyzer = myInst.GetModuleByName("My 1690A-1")

For I = 0 To 2
   myAnalyzer.SimpleTrigger mySimpleTriggers(I)

   ' Run the measurement, wait for it to complete.
   myInst.Run
   myInst.WaitComplete (20)

   ' Process/display/store captured data.
Next
```

Visual C++

```
//
//  This simple Visual C++ Console application demonstrates how to use
//  simple triggers with the Keysight 168x/9x/9xx COM interface.
//
//  This project was created in Visual C++ Developer.  To create a
//  similar project:
//
//    - Execute File -> New
//    - Select the Projects tab
//    - Select "Win32 Console Application"
//    - Select A "hello,World!" application (Visual Studio 6.0)
//
//  To make this buildable, you need to specify your "import" path
//  in stdafx.h (search for "TODO" in that file).  For example, add:
//  #import "C:/Program Files/Keysight Technologies/Logic Analyzer/LA \
//  COM Automation/agClientSvr.dll"
//
//  To run, you need to specify the host logic analyzer to connect
//  to (search for "TODO" below).
//

#include "stdafx.h"


////////////////////////////////////////////////////////////////////////
//
//  Forward declarations.
//
void DisplayError(_com_error& err);


////////////////////////////////////////////////////////////////////////
//
//  main() entry point.
//
int main(int argc, char* argv[])
{
   printf("*** Main()\n");

   //
   //  Initialize the Microsoft COM/ActiveX library.
   //
   HRESULT hr = CoInitialize(0);

   if (SUCCEEDED(hr))
```

```
{
   try {   // Catch any unexpected run-time errors.
      _bstr_t hostname = "mtx33";   // TODO, use your logic
                                    // analysis system hostname.
      printf("Connecting to instrument '%s'\n", (char*) hostname);

      // Create the connect object and get the instrument object.
      AgtLA::IConnectPtr pConnect =
            AgtLA::IConnectPtr(__uuidof(AgtLA::Connect));
      AgtLA::IInstrumentPtr pInst =
            pConnect->GetInstrument(hostname);

      // Load the configuration file.
      _bstr_t configFile = "C:\\LA\\Configs\\config.ala";
      printf("Loading the config file '%s'\n", (char*) configFile);
      pInst->Open(configFile, FALSE, "", TRUE);

      // Declare trigger variables.
      _bstr_t mySimpleTriggers[] = {
         "ADDR=hfff034d8",
         "ADDR=h00004088 And DATA=h46xxxxxx",
         "ADDR=h000041ad And DATA=h47xxxxxx"
      };

      // Set up triggers using the SimpleTrigger method.
      _bstr_t moduleName = "MPC860 Demo Board";
      AgtLA::IAnalyzerModulePtr pAnalyzer =
            pInst->GetModuleByName(moduleName);
      for (long i = 0; i < 3; i++)
      {
         printf("Trigger when '%s' occurs once, store anything.\n",
               (char*) mySimpleTriggers[i]);
         pAnalyzer->SimpleTrigger(mySimpleTriggers[i], 1,
               "Anything");

         // Run the measurement, wait for it to complete.
         pInst->Run(FALSE);
         pInst->WaitComplete(20);

         // Process/display/store captured data.
      }
   }
   catch (_com_error& e) {
      DisplayError(e);
   }

   // Uninitialize the Microsoft COM/ActiveX library.
   CoUninitialize();
}
else
{
   printf("CoInitialize failed\n");
}

return 0;
}
```

```
////////////////////////////////////////////////////////////////////
//
//  Displays the last error -- used to show the last exception
//  information.
//
void DisplayError(_com_error& error)
{
   printf("*** DisplayError()\n");

   printf("Fatal Unexpected Error:\n");
   printf("  Error Number = %08lx\n", error.Error());

   static char errorStr[1024];
   _bstr_t desc = error.Description();

   if (desc.length() == 0)
   {
      // Don't have a description string.
      strcpy(errorStr, error.ErrorMessage());
      int nLen = lstrlen(errorStr);

      // Remove funny carriage return ctrl<M>.
      if (nLen > 2 && (errorStr[nLen - 2] == 0xd))
      {
         errorStr[nLen - 2] = '\0';
      }
   }
   else
   {
      strcpy(errorStr, desc);
   }

   printf("  Error Message = %s\n", (char*) errorStr);
}
```

The SimpleTrigger (see page 177) method cannot set complex, multiple-step trigger sequences. To do that, you must set the AnalyzerModule (see page 80) object's Trigger (see page 217) property to an XML-format trigger specification string.

See Also

- Setting Up Advanced Triggers (see page 32)

Setting Up Advanced Triggers

This example shows how to set up advanced triggers by setting the Trigger (see page 217) property of the AnalyzerModule (see page 80) object to an XML-format trigger specification string.

Visual Basic

```
'You must create the Connect object (see page 90) and use it
' to access the Instrument object.  In this example, "myInst"
' represents the Instrument object.
'
' Load the configuration file.
myInst.Open ("c:\LA\Configs\mpc860_demo_compare.ala")

' Declare trigger variables.
Dim myTriggerFiles(2) As String
myTriggerFiles(0) = "c:\LA\Triggers\TrigSpecFile0.xml"
```

```
myTriggerFiles(1) = "c:\LA\Triggers\TrigSpecFile1.xml"
myTriggerFiles(2) = "c:\LA\Triggers\TrigSpecFile2.xml"
Dim I As Integer

' Set triggers using the logic analyzer Trigger property.
Dim myAnalyzer As AgtLA.AnalyzerModule
Set myAnalyzer = myInst.GetModuleByName("My 1690A-1")
Dim myTrigger As String
Dim myTrigFileNum
myTrigFileNum = FreeFile

For I = 0 To 2
   ' Get trigger spec. from local file.
   Open myTriggerFiles(I) For Input As myTrigFileNum
   ' InputB copies bytes from a file into a variable.
   ' StrConv converts the ANSI string to a UNICODE string.
   myTrigger = StrConv(InputB(LOF(myTrigFileNum), myTrigFileNum), _
         vbUnicode)
   Close myTrigFileNum

   ' Set up the logic analyzer trigger.
   myAnalyzer.Trigger = myTrigger

   ' Or, to get trigger spec. from file on the
   ' instrument (and set the Trigger property):
   'myAnalyzer.RecallTriggerByFile (myTriggerFiles(I))

   ' Display the logic analyzer trigger specification.
   myTrigger = myAnalyzer.Trigger
   MsgBox myTrigger

   ' Run the measurement, wait for it to complete.
   myInst.Run
   myInst.WaitComplete (20)

   ' Process/display/store captured data.
Next
```

Visual C++

```
//
//  This simple Visual C++ Console application demonstrates how to use
//  advanced triggers with the Keysight 168x/9x/9xx COM interface.
//
//  This project was created in Visual C++ Developer.  To create a
//  similar project:
//
//    - Execute File -> New
//    - Select the Projects tab
//    - Select "Win32 Console Application"
//    - Select A "hello,World!" application (Visual Studio 6.0)
//
//  To make this buildable, you need to specify your "import" path
//  in stdafx.h (search for "TODO" in that file).  For example, add:
//  #import "C:/Program Files/Keysight Technologies/Logic Analyzer/LA \
//  COM Automation/agClientSvr.dll"
//
//  To run, you need to specify the host logic analyzer to connect
//  to (search for "TODO" below).
```

```
//

#include "stdafx.h"
#include <iostream>
#include <fstream>
#include <sstream>


////////////////////////////////////////////////////////////////////////
//
//  Forward declarations.
//
void DisplayError(_com_error& err);


////////////////////////////////////////////////////////////////////////
//
//  main() entry point.
//
int main(int argc, char* argv[])
{
   printf("*** Main()\n");

   //
   //  Initialize the Microsoft COM/ActiveX library.
   //
   HRESULT hr = CoInitialize(0);

   if (SUCCEEDED(hr))
   {
      try {   // Catch any unexpected run-time errors.
         _bstr_t hostname = "mtx33";   // TODO, use your logic
                                       // analysis system hostname.
         printf("Connecting to instrument '%s'\n", (char*) hostname);

         // Create the connect object and get the instrument object.
         AgtLA::IConnectPtr pConnect =
               AgtLA::IConnectPtr(__uuidof(AgtLA::Connect));
         AgtLA::IInstrumentPtr pInst =
               pConnect->GetInstrument(hostname);

         // Load the configuration file.
         _bstr_t configFile = "C:\\LA\\Configs\\config.ala";
         printf("Loading the config file '%s'\n", (char*) configFile);
         pInst->Open(configFile, FALSE, "", TRUE);

         // Declare trigger variables.
         _bstr_t myTriggerFiles[] = {
            "c:\\LA\\Triggers\\TrigSpecFile0.xml",
            "c:\\LA\\Triggers\\TrigSpecFile1.xml",
            "c:\\LA\\Triggers\\TrigSpecFile2.xml"
         };

         // Set up triggers using the SimpleTrigger method.
         _bstr_t moduleName = "MPC860 Demo Board";
         AgtLA::IAnalyzerModulePtr pAnalyzer =
               pInst->GetModuleByName(moduleName);
```

```
            for (long i = 0; i < 3; i++)
            {
               _bstr_t myTriggerSpec;

               // Get trigger spec. from local file.
               std::wifstream inFile(myTriggerFiles[i]);
               std::wstringstream inBuffer;   // Intermediate buffer.
               inBuffer << inFile.rdbuf();   // Read entire file.
               // Create bstr.
               myTriggerSpec = SysAllocString(inBuffer.str().c_str());

               // Set up the logic analyzer trigger.
               printf("Loading local trigger file '%s'\n",
                     (char*) myTriggerFiles[i]);
               pAnalyzer->PutTrigger(myTriggerSpec);

               // Or, to load trigger spec. from instrument (and set the
               // Trigger property):
               //printf("Loading trigger from instrument '%s'\n",
               //      (char*) myTriggerFiles[i]);
               //pAnalyzer->RecallTriggerByFile(myTriggerFiles[i]);

               // Display the logic analyzer trigger specification.
               myTriggerSpec = pAnalyzer->GetTrigger();
               printf("XML trigger spec: '%s'\n", (char*) myTriggerSpec);

               // Run the measurement, wait for it to complete.
               pInst->Run(FALSE);
               pInst->WaitComplete(20);

               // Process/display/store captured data.
            }
         }
         catch (_com_error& e) {
            DisplayError(e);
         }

         // Uninitialize the Microsoft COM/ActiveX library.
         CoUninitialize();
      }
      else
      {
         printf("CoInitialize failed\n");
      }

      return 0;
   }


   ////////////////////////////////////////////////////////////////////
   //
   //  Displays the last error -- used to show the last exception
   //  information.
   //
   void DisplayError(_com_error& error)
   {
      printf("*** DisplayError()\n");
```

```
   printf("Fatal Unexpected Error:\n");
   printf("  Error Number = %08lx\n", error.Error());

   static char errorStr[1024];
   _bstr_t desc = error.Description();

   if (desc.length() == 0)
   {
      // Don't have a description string.
      strcpy(errorStr, error.ErrorMessage());
      int nLen = lstrlen(errorStr);

      // Remove funny carriage return ctrl<M>.
      if (nLen > 2 && (errorStr[nLen - 2] == 0xd))
      {
         errorStr[nLen - 2] = '\0';
      }
   }
   else
   {
      strcpy(errorStr, desc);
   }

   printf("  Error Message = %s\n", (char*) errorStr);
}
```

See Also

- Setting Up Simple Triggers (see page 29)

Changing the Sampling Mode

This example shows how to change the logic analyzer sampling mode by using XML-format strings with the Setup (see page 211) property of the AnalyzerModule (see page 80) object.

Visual Basic

```
'You must create the Connect object (see page 90) and use it
' to access the Instrument object.  In this example, "myInst"
' represents the Instrument object.

' Create the logic analyzer object.
Dim myAnalyzer As AgtLA.AnalyzerModule
Set myAnalyzer = myInst.GetModuleByName("My 1690A-1")
Dim mySetup As String

' Set the timing (asynchronous) sampling mode.
Dim myTimingSamplingSetup As String
myTimingSamplingSetup = "<Module>" + _
  "<SamplingSetup>" + _
    "<Sampling ChannelMode='Full' MaxSpeed='400' " + _
      "SamplePeriod='2.5 ns' Type='Standard' Acquisition='Timing' " + _
      "AcquisitionDepth='256K' TriggerPosition='50'/>" + _
  "</SamplingSetup>" + _
```

```
"</Module>"
myAnalyzer.Setup = myTimingSamplingSetup

' Display the complete logic analyzer setup.
mySetup = myAnalyzer.Setup
MsgBox mySetup

' Set the state (synchronous) sampling mode.
Dim myStateSamplingSetup As String
myStateSamplingSetup = "<Module>" + _
  "<SamplingSetup>" + _
    "<Sampling ChannelMode='Full' Acquisition='State' " + _
      "AcquisitionDepth='256K' MaxSpeed='200' " + _
      "TriggerPosition='50'/>" + _
    "<StateClockSpec Mode='Master'>" + _
      "<Clear/>" + _
      "<Master>" + _
        "<ClockGroup>" + _
          "<Edges>" + _
            "<Edge PodIndex='1' Value='Rising'/>" + _
          "</Edges>" + _
          "<Qualifiers Operator='And'>" + _
            "<Qualifier Level='Low' PodIndex='2'/>" + _
          "</Qualifiers>" + _
        "</ClockGroup>" + _
      "</Master>" + _
    "</StateClockSpec>" + _
  "</SamplingSetup>" + _
"</Module>"
myAnalyzer.Setup = myStateSamplingSetup

' Display the complete logic analyzer setup.
mySetup = myAnalyzer.Setup
MsgBox mySetup
```

Visual C++

```
//
//  This simple Visual C++ Console application demonstrates
//  how to change the logic analyzer sampling mode with the
//  Keysight 168x/9x/9xx COM interface.
//
//  This project was created in Visual C++ Developer.  To create a
//  similar project:
//
//    - Execute File -> New
//    - Select the Projects tab
//    - Select "Win32 Console Application"
//    - Select A "hello,World!" application (Visual Studio 6.0)
//
//  To make this buildable, you need to specify your "import" path
//  in stdafx.h (search for "TODO" in that file).  For example, add:
//  #import "C:/Program Files/Keysight Technologies/Logic Analyzer/LA \
//  COM Automation/agClientSvr.dll"
//
//  To run, you need to specify the host logic analyzer to connect
//  to (search for "TODO" below).
//
```

```
#include "stdafx.h"


////////////////////////////////////////////////////////////////////
//
//  Forward declarations.
//
void DisplayError(_com_error& err);


////////////////////////////////////////////////////////////////////
//
//  main() entry point.
//
int main(int argc, char* argv[])
{
   printf("*** Main()\n");

   //
   //  Initialize the Microsoft COM/ActiveX library.
   //
   HRESULT hr = CoInitialize(0);

   if (SUCCEEDED(hr))
   {
      try {   // Catch any unexpected run-time errors.
         _bstr_t hostname = "mtx33";   // TODO, use your logic
                                       // analysis system hostname.
         printf("Connecting to instrument '%s'\n", (char*) hostname);

         // Create the connect object and get the instrument object.
         AgtLA::IConnectPtr pConnect =
               AgtLA::IConnectPtr(__uuidof(AgtLA::Connect));
         AgtLA::IInstrumentPtr pInst =
               pConnect->GetInstrument(hostname);

         // Get the logic analyzer object.
         _bstr_t moduleName = "My 16910A-1";
         AgtLA::IAnalyzerModulePtr pAnalyzer =
               pInst->GetModuleByName(moduleName);

         // Set the timing (asynchronous) sampling mode.
         _bstr_t myTimingSamplingSetup = " \
            <Module> \
               <SamplingSetup> \
                  <Sampling ChannelMode='Full' MaxSpeed='400' \
                        SamplePeriod='2.5 ns' Type='Standard' \
                        Acquisition='Timing' AcquisitionDepth='256K' \
                        TriggerPosition='50'/> \
               </SamplingSetup> \
            </Module>";
         pAnalyzer->PutSetup(myTimingSamplingSetup);

         // Display the complete logic analyzer setup.
         _bstr_t mySetup;
         mySetup = pAnalyzer->GetSetup();
         printf("Logic analyzer setup: '%s'\n", (char*) mySetup);
```

```
            // Set the state (synchronous) sampling mode.
            _bstr_t myStateSamplingSetup = " \
               <Module> \
                  <SamplingSetup> \
                     <Sampling ChannelMode='Full' Acquisition='State' \
                           AcquisitionDepth='256K' MaxSpeed='200' \
                           TriggerPosition='50'/> \
                     <StateClockSpec Mode='Master'> \
                        <Clear/> \
                        <Master> \
                           <ClockGroup> \
                              <Edges> \
                                 <Edge PodIndex='1' Value='Rising'/> \
                              </Edges> \
                              <Qualifiers Operator='And'> \
                                  <Qualifier Level='Low' PodIndex='2'/> \
                              </Qualifiers> \
                           </ClockGroup> \
                        </Master> \
                     </StateClockSpec> \
                  </SamplingSetup> \
               </Module>";
            pAnalyzer->PutSetup(myStateSamplingSetup);

            // Display the complete logic analyzer setup.
            mySetup = pAnalyzer->GetSetup();
            printf("Logic analyzer setup: '%s'\n", (char*) mySetup);

         }
         catch (_com_error& e) {
            DisplayError(e);
         }

         // Uninitialize the Microsoft COM/ActiveX library.
         CoUninitialize();
      }
      else
      {
         printf("CoInitialize failed\n");
      }

      return 0;
}


//////////////////////////////////////////////////////////////////////
//
//  Displays the last error -- used to show the last exception
//  information.
//
void DisplayError(_com_error& error)
{
   printf("*** DisplayError()\n");

   printf("Fatal Unexpected Error:\n");
   printf("  Error Number = %08lx\n", error.Error());
```

```
   static char errorStr[1024];
   _bstr_t desc = error.Description();

   if (desc.length() == 0)
   {
      // Don't have a description string.
      strcpy(errorStr, error.ErrorMessage());
      int nLen = lstrlen(errorStr);

      // Remove funny carriage return ctrl<M>.
      if (nLen > 2 && (errorStr[nLen - 2] == 0xd))
      {
         errorStr[nLen - 2] = '\0';
      }
   }
   else
   {
      strcpy(errorStr, desc);
   }

   printf("  Error Message = %s\n", (char*) errorStr);
}
```

Checking the Logic Analyzer Software Version

This example shows you how to check for the correct logic analyzer software version before using recently added objects, methods, and properties.

Visual Basic

```
Public Sub CheckVersion()
  Dim strVersion As String
  bResult = DoesCurrentVersionMatchRequiredVersion("03.06.01")
End Sub

Private Function DoesCurrentVersionMatchRequiredVersion(ByVal _
      strRequiredVersion As String) As Boolean

  ' Get the version number, broken down into:
  '
  ' xx.yyy.zzz
  ' where xx is the major version
  '       yy is the minor version
  '       zz is the SubMinor version (which may not be present)
  '
  ' Example :
  ' 03.00.01 has major version 3, minor version 0, sub-minor version 1
  '

  Dim nRequiredMajor As Integer
  Dim nRequiredMinor As Integer
  Dim nRequiredSubMinor As Integer
  Dim nCurrentMajor As Integer
  Dim nCurrentMinor As Integer
  Dim nCurrentSubMinor As Integer

  ' If the two versions are identical, we're done.
  If (strRequiredVersion = AgtLA.Version) Then
```

```
    DoesCurrentVersionMatchRequiredVersion = True
    Exit Function
  End If

  Call GetNumbersForVersionString(strRequiredVersion, _
                                  nRequiredMajor, _
                                  nRequiredMinor, _
                                  nRequiredSubMinor)
  Call GetNumbersForVersionString(AgtLA.Version, _
                                  nCurrentMajor, _
                                  nCurrentMinor, _
                                  nCurrentSubMinor)

  ' Check Major Version first.
  If (nCurrentMajor > nRequiredMajor) Then
    DoesCurrentVersionMatchRequiredVersion = True
    Exit Function
  Else
    If (nCurrentMajor < nRequiredMajor) Then
      DoesCurrentVersionMatchRequiredVersion = False
      Exit Function
    End If
  End If

  ' Check Minor Version.
  If (nCurrentMinor > nRequiredMinor) Then
    DoesCurrentVersionMatchRequiredVersion = True
    Exit Function
  Else
    If (nCurrentMinor < nRequiredMinor) Then
      DoesCurrentVersionMatchRequiredVersion = False
      Exit Function
    End If
  End If

  ' Check SubMinor Version.
  If (nCurrentSubMinor > nRequiredSubMinor) Then
    DoesCurrentVersionMatchRequiredVersion = True
    Exit Function
  Else
    If (nCurrentSubMinor < nRequiredSubMinor) Then
      DoesCurrentVersionMatchRequiredVersion = False
      Exit Function
    End If
  End If
  DoesCurrentVersionMatchRequiredVersion = True
End Function

Private Sub GetNumbersForVersionString(ByVal strVersion As String, _
                                       ByRef nMajor As Integer, _
                                       ByRef nMinor As Integer, _
                                       ByRef nSubMinor As Integer)
  Dim nDash As Integer
  Dim nFirstPeriod As Integer
  Dim nSecondPeriod As Integer

  On Error GoTo invalidStr
```

```
  ' If there's a dash, eliminate.
  nDash = InStr(1, strVersion, "-")
  If (nDash > 0) Then
    strVersion = Mid(strVersion, 1, nDash - 1)
  End If

  ' Get the  Version first.  Put up a message if the string is wrong.
  nFirstPeriod = InStr(1, strVersion, ".")

  ' If there's no period, we need to exit.
  If (nFirstPeriod = 0) Then
    MsgBox "The version string " + strVersion + " is not valid.  " + _
           "Examples are 03.02.01 or 03.00."
    Exit Sub
  End If

  nSecondPeriod = InStr(nFirstPeriod + 1, strVersion, ".")
  nMajor = CInt(Mid(strVersion, 1, nFirstPeriod - 1))
  If (nSecondPeriod = 0) Then
    nMinor = CInt(Mid(strVersion, nFirstPeriod + 1, _
        Len(strVersion) - nFirstPeriod))
    nSubMinor = 0
  Else
    nMinor = CInt(Mid(strVersion, nFirstPeriod + 1, _
        nSecondPeriod - nFirstPeriod - 1))
    nSubMinor = CInt(Mid(strVersion, nSecondPeriod + 1, _
        Len(strVersion) - nSecondPeriod))
  End If

  Exit Sub

invalidStr:
  MsgBox "The version string " + strRequiredVersion + _
      " is not valid.  Examples are 03.02.01 or 03.00."

End Sub
```

Visual C++

```
//
//  This simple Visual C++ Console application demonstrates how to use
//  the Keysight 168x/169x/169xx COM interface to check the system
//  software version.
//
//  This project was created in Visual C++ Developer.  To create a
//  similar project:
//
//    - Execute File -> New
//    - Select the Projects tab
//    - Select "Win32 Console Application"
//    - Select A "hello,World!" application (Visual Studio 6.0)
//
//  To make this buildable, you need to specify your "import" path
//  in stdafx.h (search for "TODO" in that file).  For example, add:
//  #import "C:/Program Files/Keysight Technologies/Logic Analyzer/LA \
//  COM Automation/agClientSvr.dll"
//
//  To run, you need to specify the host logic analyzer to connect
```

```
//  to (search for "TODO" below).
//

#include "stdafx.h"
#include <string>
using namespace std;


////////////////////////////////////////////////////////////////////////
//
//  Forward declarations.
//
boolean GetNumbersForVersionString(
  _bstr_t&   strVersion,
  long&      nMajor,
  long&      nMinor,
  long&      nSubminor);

boolean DoesCurrVersionMatch(
  _bstr_t&     strReqdVersion);

void DisplayError(_com_error& err);


////////////////////////////////////////////////////////////////////////
//
//  main() entry point.
//
int main(int argc, char* argv[])
{
   printf("*** Main()\n");

   //
   //  Initialize the Microsoft COM/ActiveX library.
   //
   HRESULT hr = CoInitialize(0);

   if (SUCCEEDED(hr))
   {
      _bstr_t strReqdVersion = "03.00.0001";
      if (DoesCurrVersionMatch(strReqdVersion)) {
         printf("Current version matches required '%s'\n",
               (char*) strReqdVersion);
      }
      else {
         printf("Current version does not match required '%s'\n",
               (char*) strReqdVersion);
      }

      // Uninitialize the Microsoft COM/ActiveX library.
      CoUninitialize();
   }
   else
   {
      printf("CoInitialize failed\n");
   }
```

```
   return 0;
}


//////////////////////////////////////////////////////////////////////
//
//  Given a version string, return major, minor, and subminor numbers.
//
boolean GetNumbersForVersionString(_bstr_t& version,
                                   long& nMajor,
                                   long& nMinor,
                                   long& nSubminor)
{
   string strVersion = version;
   string strMajor;
   string strMinor;
   string strSubminor;

   string::size_type nDash;
   string::size_type nFirstPeriod;
   string::size_type nSecondPeriod;

   // If there's a dash, eliminate from dash to end of string.
   nDash = strVersion.find("-");

   if (nDash != string::npos) {
      strVersion = strVersion.substr(0, nDash);
   }

   // Get the  Version first.  Put up a message if the string is wrong.
   nFirstPeriod = strVersion.find(".");

   // If there's no period, we need to exit.
   if (nFirstPeriod == string::npos) {
      printf("The version string '%s' is not valid.",
            strVersion.c_str());
      printf("  Examples are 03.02.01 or 03.00.\n");
      return (false);
   }

   strMajor = strVersion.substr(0, nFirstPeriod);
   nMajor = atol(strMajor.c_str());

   nSecondPeriod = strVersion.find(".", nFirstPeriod + 1);
   if (nSecondPeriod == string::npos) {   // No second period.
      strMinor = strVersion.substr(nFirstPeriod+1);
      nMinor = atol(strMinor.c_str());
      strSubminor = "";
      nSubminor = 0;
   }
   else {   // There is a second period.
      strMinor = strVersion.substr(nFirstPeriod+1,
            nSecondPeriod-nFirstPeriod-1);
      nMinor = atol(strMinor.c_str());
      strSubminor = strVersion.substr(nSecondPeriod+1);
      nSubminor = atol(strSubminor.c_str());
   }
```

```
   return (true);
}


////////////////////////////////////////////////////////////////////
//
//  Returns true if the current system software version matches
//  the required version.
//
boolean DoesCurrVersionMatch(_bstr_t& strReqdVersion)
{
   // Get the version number, broken down into:
   //
   // xx.yyy.zzz
   // where xx is the major version
   //       yy is the minor version
   //       zz is the Subminor version (which may not be present)
   //
   // Example :
   // 03.00.01 has the major version 3, minor version 0, sub-minor
   // version 1
   //

   long nReqdMajor;
   long nReqdMinor;
   long nReqdSubminor;
   long nCurrMajor;
   long nCurrMinor;
   long nCurrSubminor;

   try {   // Catch any unexpected run-time errors.
      _bstr_t hostname = "mtx33";   // TODO, use your logic analysis
                                    // system hostname.
      printf("Connecting to instrument '%s'\n", (char*) hostname);

      // Create the connect object and get the instrument object.
      AgtLA::IConnectPtr pConnect =
            AgtLA::IConnectPtr(__uuidof(AgtLA::Connect));
      AgtLA::IInstrumentPtr pInst = pConnect->GetInstrument(hostname);

      // Get current system software version.
      _bstr_t strCurrVersion = pInst->GetVersion();

      // If the two versions are identical, we're done.
      if (strReqdVersion == strCurrVersion) {
         return (true);
      }

      // Get the individual numbers for the version string.
      if (!GetNumbersForVersionString(strReqdVersion, nReqdMajor,
            nReqdMinor, nReqdSubminor)) {
         return (false);
      }
      if (!GetNumbersForVersionString(strCurrVersion, nCurrMajor,
            nCurrMinor, nCurrSubminor)) {
         return (false);
      }
```

```
      // Check Major version first.
      if (nCurrMajor > nReqdMajor) {
         return (true);
      }
      else {
         if (nCurrMajor < nReqdMajor) {
            return (false);
         }
      }

      // Check Minor version next.
      if (nCurrMinor > nReqdMinor) {
         return (true);
      }
      else {
         if (nCurrMinor < nReqdMinor) {
            return (false);
         }
      }

      // Check Subminor version last.
      if (nCurrSubminor > nReqdSubminor) {
         return (true);
      }
      else {
         if (nCurrSubminor < nReqdSubminor) {
            return (false);
         }
      }

      // All numbers the same, return true.
      return (true);
   }
   catch (_com_error& e) {
      DisplayError(e);
      return (false);
   }
}


////////////////////////////////////////////////////////////////////////
//
//  Displays the last error -- used to show the last exception
//  information.
//
void DisplayError(_com_error& error)
{
   printf("*** DisplayError()\n");

   printf("Fatal Unexpected Error:\n");
   printf("  Error Number = %08lx\n", error.Error());

   static char errorStr[1024];
   _bstr_t desc = error.Description();

   if (desc.length() == 0)
```

```
    {
       // Don't have a description string.
       strcpy(errorStr, error.ErrorMessage());
       int nLen = lstrlen(errorStr);

       // Remove funny carriage return ctrl<M>.
       if (nLen > 2 && (errorStr[nLen - 2] == 0xd))
       {
          errorStr[nLen - 2] = '\0';
       }
    }
    else
    {
       strcpy(errorStr, desc);
    }

    printf("  Error Message = %s\n", (char*) errorStr);
}
```

Additional Visual Basic Examples

Visual Basic example projects can be found in your install directory. The default installation example directory is C:/Program Files/Keysight Technologies/Logic Analyzer/LA COM Automation/Visual Basic Examples.

Before running each example, read the documentation at the top of each source file for an explanation of how the logic analyzer must be set up before the example will run successfully.

Using Visual C++

This online help is geared mainly towards Visual Basic not Visual C++ programmers. Although you'll find the documentation is not directly applicable, the explanation of object, methods, properties and parameters can still be helpful.

Visual C++ Examples

Visual C++ example projects can be found in your install directory. The default installation example directory is C:/Program Files/Keysight Technologies/Logic Analyzer/LA COM Automation/Visual C++ Examples.

Before running each example, read the documentation at the top of each source file for an explanation of how the logic analyzer must be set up before the example will run successfully.

Simple Visual C++ Example

This example connects to the logic analyzer hardware and starts a measurement.

This C++ example uses COM smart pointers _comptr_t to integrate the ActiveX/COM automation server into the Visual C++ environment. The declaration of the Instrument specific smart pointers is in the "agClientSvr.tlh" header file which is automatically created and included when the instrument type library is imported using the #import directive.

NOTE For detailed method and property parameter types, see the "agClientSvr.tlh" header file. Note that Get/Put methods are generated for properties that you can get or set (as described in this online help). For example, the GetInstrument method in the following example accesses the Instrument (see page 202) property.

Using smart pointers is not the only way to integrate COM into the Visual C++ environment. For more details, refer to the Microsoft Visual C++ documentation.

```
// Import the Instrument Automation Server's type library.
// Replace <install_dir> with your installation directory.
// Default is:
//   C:/Program Files/Keysight Technologies/Logic Analyzer/LA COM
//   Automation/
#import "<install_dir>/agClientSvr.dll"

// Before using the Automation Server, initialize the COM/OLE
// libraries in your MFC application's InitInstance() method.
if (!AfxOleInit())
{
   ::AfxMessageBox("OLE initialization failed");
   return FALSE;
}

// If you're not using MFC, you should call
//CoInitialize(0)/CoUninitialize()

// For detailed method and property parameter types,
// see the header file "agClientSvr.tlh generated automatically
// by the #import directive in your project's configuration directory.

// Place the following C++ code in your class method.
try {
  // create the connect object and get the instrument
  AgtLA::IConnectPtr pConnect =
      AgtLA::IConnectPtr(__uuidof(AgtLA::Connect));
```

```
  AgtLA::IInstrumentPtr pInst = pConnect->GetInstrument("");
  // run all modules
  pInst->Run(FALSE);
}
catch (_com_error& e) {
   // Display any error messages returned.
   _bstr_t msg(e.Description());
   if (msg.length() == 0)
      msg = e.ErrorMessage();
   ::AfxMessageBox(msg);
}
```

Using LabVIEW

Tutorial - To programmatically control the logic analyzer in LabVIEW

Descriptions of the basic LabVIEW interface, its operation, and general use is not covered here. Refer to your LabVIEW online help for this information. The following steps are intended only as a guideline.

LabVIEW 8.0

1 Open the logic analyzer Connect object:

a In the Labview Front Panel, go to the Refnum and choose Automation Refnum.

b Right click select "ActiveX" class, choose browse.

c Select Object from Type library dialog, click/select "show creatable objects only" box.

d Choose "Keysight 168x/169x/169xx Logic Analyzer Object Library Version 1.0".

e Choose the "Connect(AgtLA.Connect.1)" object.

2 Get the logic analyzer Instrument Object:

a In the LabVIEW block diagram, go to the "Function" Palette and choose "Connectivity" then "ActiveX".

b Inside the "ActiveX" palette, drag the "Invoke Node" icon into the window.

c Press the "Connect Wire" button in the "Tools" Palette and connect the "Automation Open" icon's "Automation Refnum" output to the "Invoke Node" icon's "reference" input. The "Invoke Node" name displayed is now "IConnect".

d Right click on the "IConnect" icon and choose Methods->Instrument

e Connect a String Constant to the "HostNameOrIPAddress".

f The output of the "Instrument" method is an "IInstrument" object. Use this object to call the instrument's methods and properties.

3 Call a logic analyzer Instrument object method:

a In the LabVIEW diagram window, go to the "Function" Palette and choose "Communication", then "ActiveX".

b Inside the "ActiveX" palette, drag the "Invoke Node" icon onto the window.

c Press the "Connect Wire" button in the "Tools" Palette and connect the "IConnect" icon's "Instrument" method output to the "Invoke Node" icon's "reference" input. The "Invoke Node" name displayed is now "IInstrument".

d Right-click on the "IInstrument" icon and choose "Methods" to call any method.

4 Call a logic analyzer Instrument object property:

a In the LabVIEW diagram window, go to the "Function" Palette and choose "Communication", then "ActiveX".

b Inside the "ActiveX" palette, drag the "Property Node" icon onto the window.

c Press the "Connect Wire" button in the "Tools" Palette and connect the "IConnect" icon's "Instrument" method output to the "Property Node" icon's "reference" input. The "Property Node" name displayed is now "IInstrument".

d Right-click on the "IInstrument" icon and choose "Properties" to call any property.

LabVIEW 6.0

NOTE The process documented here assumes you are using LabVIEW 6.0. This process may change for different versions of LabVIEW.

1 Open the logic analyzer Connect object:
   a In the LabVIEW diagram window, go to the "Function" Palette and choose "Communication", then "ActiveX".
   b Inside the "ActiveX" palette, drag the "Automation Open" icon onto the window.
   c Right-click on the "Automation Open" icon and choose "Select ActiveX Class"; then, choose "Browse...".
   d The Select Object From Type Library dialog will be displayed.
   e Choose "Keysight 168x/169x/169xx Logic Analyzer Object Library Version 1.0".
   f Choose the "Connect (AgtLA.Connect.1)" object.
2 Get the logic analyzer Instrument object:
   a In the LabVIEW diagram window, go to the "Function" Palette and choose "Communication", then "ActiveX".
   b Inside the "ActiveX" palette, drag the "Invoke Node" icon onto the window.
   c Press the "Connect Wire" button in the "Tools" Palette and connect the "Automation Open" icon's "Automation Refnum" output to the "Invoke Node" icon's "reference" input. The "Invoke Node" name displayed is now "IConnect".
   d Right click on the "IConnect" icon and choose Methods->Instrument
   e Connect a String Constant to the "HostNameOrIPAddress".
   f The output of the "Instrument" method is an "IInstrument" object. Use this object to call the instrument's methods and properties.
3 Call a logic analyzer Instrument object method:
   a In the LabVIEW diagram window, go to the "Function" Palette and choose "Communication", then "ActiveX".
   b Inside the "ActiveX" palette, drag the "Invoke Node" icon onto the window.
   c Press the "Connect Wire" button in the "Tools" Palette and connect the "IConnect" icon's "Instrument" method output to the "Invoke Node" icon's "reference" input. The "Invoke Node" name displayed is now "IInstrument".
   d Right-click on the "IInstrument" icon and choose "Methods" to call any method.
4 Call a logic analyzer Instrument object property:
   a In the LabVIEW diagram window, go to the "Function" Palette and choose "Communication", then "ActiveX".
   b Inside the "ActiveX" palette, drag the "Property Node" icon onto the window.
   c Press the "Connect Wire" button in the "Tools" Palette and connect the "IConnect" icon's "Instrument" method output to the "Property Node" icon's "reference" input. The "Property Node" name displayed is now "IInstrument".
   d Right-click on the "IInstrument" icon and choose "Properties" to call any property.

See the LabVIEW examples (see page 51) for a detailed view of how to use Invoke and Property Nodes.

LabVIEW Examples

LabVIEW examples can be found in your install directory. The default installation example directories are:

- C:/Program Files/Keysight Technologies/Logic Analyzer/LA COM Automation/LabVIEW 6.0 Examples
- C:/Program Files/Keysight Technologies/Logic Analyzer/LA COM Automation/LabVIEW 7.0 Examples

Using Perl

1 Install the Perl software.

2 Copy and paste the example code below into a file (PrintLAData.pl).

3 Run the example from the Command Prompt by entering the command: "perl PrintLAData.pl".

Example

```
#
# This Perl example prints all of the bus/signal's from the
# first module in a 168x/9x/9xx Logic Analysis System.
#
#   $LAHostNameOrIP -> change to the hostname or IP address of the LA
#                      you're connecting to (default is 'localhost'
#                      if you're running Perl directly on the LA)
#   $LAAnalyzer     -> change to the analyzer name to transfer data
#                      from (default is the first module)
#   $LAStartRange, $LAEndRange -> change to the data range to upload
#                      (default is -10 and 10 respectively)
#
# This example was tested using ActiveState Perl version 5.6.1


# when using strict, declare *all* globals
use strict qw(vars refs subs);

# libraries needed to interface with the Logic Analyzer COM interface
use Win32::OLE;
use Win32::OLE::Variant;
use Win32::OLE::Const;

### ======================================================================
###                       * Begin Subroutines *
### ----------------------------------------------------------------------

##-----------------------------------------------------------------------
## FUNCTION:
##   PrintArrays -- prints the Logic Analyzer Data Arrays
##
## SYNOPSIS:
##
## ARGUMENTS:
##   arrays - array of data arrays to format and print.  The
##            first array contains the bus/signal names.
##
##-----------------------------------------------------------------------

sub PrintArrays
{
  my @arrays = @_;

  my @nameArray = @{$arrays[0]};

  # Calculate the max width for each column and store in an array
  my @maxWidthArray;

  print("\n");
```

```
  for my $i ( 1 .. $#arrays )
  {
    my $maxlen = length($nameArray[$i-1]);
    for my $j ( 0 .. $#{ $arrays[1] } )
    {
      my $data = $arrays[$i][$j];
      if (length($data) > $maxlen) {
        $maxlen = length($data);
      }
    }
    push(@maxWidthArray, $maxlen);
  }

  # Print the header row
  for my $i ( 0 .. $#nameArray )
  {
    my $hdr = $nameArray[$i];
    my $firstSpaces = ($maxWidthArray[$i] - length($hdr))/2;
    print "  " . " " x $firstSpaces;
    print $hdr;
    print " " x ($maxWidthArray[$i] - length($hdr) - $firstSpaces);
  }
  print "\n";
  for my $i ( 0 .. $#nameArray )
  {
    print "  " . "-" x $maxWidthArray[$i];
  }
  print "\n";

  # Print the data rows
  for my $i ( 0 .. $#{ $arrays[1] } )
  {
    for my $j ( 1 .. $#arrays )
    {
      my $data = $arrays[$j][$i];
      print "  " . " " x ($maxWidthArray[$j-1] - length($data)) . $data;
    }
    print "\n";
  }
}

### ------------------------------------------------------------------
###                         End of Subroutines
### ==================================================================

### ==================================================================
###                       * Begin Main Routine *
### ------------------------------------------------------------------

# Logic Analyzer host name or IP address
my $LAHostNameOrIP = "localhost";

# Create the logic analyzer client server
my $LAConnect = Win32::OLE->new('AgtLA.Connect');
if (! $LAConnect)
{
  print ("Connection failed: ");
```

```
  print ("please install the LA COM Automation client software\n");
  exit 1;
}

# Get the typedef constants
my $LAConstants = Win32::OLE::Const->Load($LAConnect);

# Connect to the remote logic analyzer instrument
print("\nConnecting to '$LAHostNameOrIP'\n");
my $LAInst = $LAConnect->Instrument($LAHostNameOrIP);
if (Win32::OLE->LastError != 0)
{
  print("Connection failed: ");
  print("please verify the LA hostname or IP address ");
  print("'$LAHostNameOrIP'\n");
  exit 1;
}

# Optionally load a configuration file that exists on the logic
# analyzer.  If the file only exists on your client PC, then use the
# $LAConnect->CopyFile() method to copy the file onto your logic
# analyzer
#
# $LAInst->Open("test.xml");

# Run the analyzer and wait for the measurement to complete before
# getting data
$LAInst->Run();
$LAInst->WaitComplete(10);  # time out after 10 seconds

# Get the first module's data
my $LAAnalyzer = $LAInst->Modules(0);

#
# Get the module's bus/signal names and store into 'LANameArray'
#
my $LABusSignals = $LAAnalyzer->BusSignals();
my $LABusSignalsCount = $LABusSignals->count;
my @LANameArray;
my @LADataArrays;
my $numRows = Variant(VT_I4 | VT_BYREF, 0);
my $LAStartRange = -10;
my $LAEndRange = 10;

foreach my $index (0..$LABusSignalsCount-1)
{
  my $name = $LABusSignals->Item($index)->Name;
  push(@LANameArray, $name);
}
push (@LADataArrays, [@LANameArray]);

#
# Get the module's bus/signal data and store into 'LADataArrays'
#
foreach my $index (0..$LABusSignalsCount-1)
{
  my $LAData = $LABusSignals->Item($index)->BusSignalData;
```

```
  my $LABusSignalType = $LABusSignals->Item($index)->BusSignalType();
  my $LADataType = $LAConstants->{AgtDataLong};

  if ($LABusSignalType == $LAConstants->{AgtBusSignalTime})
  {
    $LADataType = $LAConstants->{AgtDataTime};
  }

  my $LADataArray = $LAData->GetDataBySample($LAStartRange,
      $LAEndRange, $LADataType, $numRows);
  push(@LADataArrays, $LADataArray);
}

#
# Print the Arrays
#
PrintArrays(@LADataArrays);
```

Using Python

1 Install the Python and Python for Windows extension software.

2 Set up early binding for COM objects by running the MakePy utility. MakePy is a normal Python module that lives in the *win32com\client* directory of the PythonCOM package. There are two ways to run this script:

- Start PythonWin, and from the Tools menu, select the item COM Makepy utility.
- Using the Windows Explorer, locate the *client* subdirectory under the main *win32com* directory and double-click the file *makepy.py*.

In both cases, you are presented with a list of objects MakePy can use to support early binding. Select Keysight 168x/169x/169xx Logic Analyzer Object Library and click OK.

3 Copy and paste the example code below into a file (PrintLAData.py).

4 Run the example from the Command Prompt by entering the command: "python PrintLAData.py".

Example

```
#
# This Python example prints all of the bus/signal's from the
# first module in a 168x/9x/9xx Logic Analysis System.
#
#   LAHostNameOrIP -> change to the hostname or IP address of the LA
#                     you're connecting to (default is 'localhost'
#                     if you're running Python directly on the LA)
#   LAModule       -> change to the analyzer name to transfer data from
#                     (default is the first module)
#   LAStartRange, LAEndRange -> change to the data range to upload
#                       (default is -10 and 10 respectively)
#
# This example was tested using Python version 2.2.3 and
# the Python for Windows extensions build 200.

import sys
import string

# Libraries needed to interface with the Logic Analyzer COM interface.
import win32com.client
from win32com.client import constants
import pythoncom

### ================================================================
###                       * Begin Subroutines *
### ----------------------------------------------------------------

##------------------------------------------------------------------
## FUNCTION:
##   PrintArrays -- prints the Logic Analyzer Data Arrays
##
## SYNOPSIS:
##
## ARGUMENTS:
##   arrays - array of data arrays to format and print.  The
##            first array contains the bus/signal names.
##
##------------------------------------------------------------------

def PrintArrays(arrays):
```

```
  NameArray = arrays[0]
  TypeArray = arrays[1]
  DataArrays = arrays[2:]
  DataStringArrays = []

  # Calculate the max width for each column and store in an array.
  MaxWidthArray = []
  for name in NameArray:
    MaxWidthArray.append(len(name))

  for column in range(len(DataArrays)):
     DataStringArray = []
     for dataValue in DataArrays[column]:
       if TypeArray[column] == constants.AgtBusSignalProbed:
         dataValueString = "%X" % dataValue
       elif TypeArray[column] == constants.AgtBusSignalGenerated:
         dataValueString = "%s" % dataValue
       elif TypeArray[column] == constants.AgtBusSignalSampleNum:
         dataValueString = "%d" % dataValue
       elif TypeArray[column] == constants.AgtBusSignalTime:
         dataValueString = "%E" % dataValue
       DataStringArray.append(dataValueString)
       dataValueStringLength = len(dataValueString)
       if dataValueStringLength > MaxWidthArray[column]:
         MaxWidthArray[column] = dataValueStringLength
     DataStringArrays.append(DataStringArray)

  # Print the header row.
  print ""
  for column in range(len(NameArray)):
    print "  " + string.center(NameArray[column], \
        MaxWidthArray[column]),
  print ""
  for column in range(len(NameArray)):
    print "  " + "-" * MaxWidthArray[column],
  print ""

  # Print the data rows.
  for row in range(len(DataStringArrays[1])):
    for column in range(len(DataStringArrays)):
      print "  " + string.rjust(DataStringArrays[column][row], \
          MaxWidthArray[column]),
    print ""

### ------------------------------------------------------------------
###                         End of Subroutines
### ==================================================================

### ==================================================================
###                      * Begin Main Routine *
### ------------------------------------------------------------------

# Logic Analyzer host name or IP address.
LAHostNameOrIP = "localhost"

# Create the logic analyzer client server, and
# connect to the remote logic analyzer instrument.
```

```
print "\nConnecting to '%s'" % LAHostNameOrIP ;
try:
  LAConnect = win32com.client.Dispatch("AgtLA.Connect")
  LAInst = LAConnect.GetInstrument(LAHostNameOrIP)
except pythoncom.com_error, (hr, msg, exc, arg):
  print "The AgtLA call failed with code: %d: %s" % (hr, msg)
  if exc is None:
    print "There is no extended error information"
  else:
    wcode, source, text, helpFile, helpId, scode = exc
    print "The source of the error is", source
    print "The error message is", text
    print "More info can be found in %s (id=%d)" % (helpFile, helpId)
  sys.exit(1)

# Optionally, load a configuration file that exists on the
# logic analyzer.  If the file only exists on your client PC,
# then use the LAConnect.CopyFile() method to copy the file
# onto your logic analyzer.
#
# LAInst.Open("test.xml")

# Get the logic analyzer module.
#
# LAModule = LAInst.GetModuleByName("My 1691D-1")
LAModule = LAInst.Modules(0)

# Optionally, set up a trigger before running the analyzer.
#
# if LAModule.Type == "Analyzer":
#   LAAnalyzerModule = \
#       win32com.client.CastTo(LAModule, "IAnalyzerModule")
#   LAAnalyzerModule.SimpleTrigger("My Bus 1=hff")

# Run the analyzer and wait for the measurement to complete
# before getting data.
LAInst.Run()
LAInst.WaitComplete(10)   # Time out after 10 seconds.

# Get the module's bus/signal names and types, and store
# into 'LABusSignalNameArray' and 'LABusSignalTypeArray'.
LABusSignals = LAModule.BusSignals
LABusSignalsCount = LABusSignals.Count
LABusSignalNameArray = []
LABusSignalTypeArray = []

for index in range(LABusSignalsCount):
  LABusSignalName = LABusSignals.Item(index).Name
  LABusSignalType = LABusSignals.Item(index).BusSignalType
  LABusSignalNameArray.append(LABusSignalName)
  LABusSignalTypeArray.append(LABusSignalType)

# Get the module's bus/signal data and store into 'LADataArrays'.
LADataArrays = []
LADataArrays.append(LABusSignalNameArray)
LADataArrays.append(LABusSignalTypeArray)
LAStartRange = -10
```

```
LAEndRange = 10

for index in range(LABusSignalsCount):
  LAData = LABusSignals.Item(index).BusSignalData
  LABusSignalType = LABusSignalTypeArray[index]
  LADataType = constants.AgtDataLong

  if LABusSignalType == constants.AgtBusSignalTime:
    LADataType = constants.AgtDataTime

  if LAData.Type == "Sample":
    SampleBusSignalData = \
        win32com.client.CastTo(LAData, "ISampleBusSignalData")
    (LADataArray, NumRows) = \
        SampleBusSignalData.GetDataBySample(LAStartRange,
                                            LAEndRange,
                                            LADataType)
    LADataArrays.append(LADataArray)

#
# Print the Arrays.
#
PrintArrays(LADataArrays)
```

Using Tcl

1 Install the Tcl software.

2 Copy and paste the example code below into a file (PrintLAData.tcl).

3 Run the example from the Command Prompt by entering the command: "tclsh.exe PrintLAData.tcl".

Example

```
#
# This Tcl example prints all of the bus/signal's from the
# first module in a 168x/9x/9xx Logic Analysis System.
#
#  This example was tested using ActiveState ActiveTcl 8.4.5.0
#

set usage "16900.tcl \[-e\] \[-f <hostname or IP>\] \[-c <config file>\]
Does a simple run command to the specified 16900 frame
-e : print out interface information
-f : specify the frame name on the command line
-c : specify a config file to load (optional)
";

package require cmdline;   # argument processing
package require tcom;      # Use the ActiveState tcom package

### ================================================================
###                      * Begin Procedures *
### ----------------------------------------------------------------

##-----------------------------------------------------------------
##  Procedure to explore COM interfaces
##-----------------------------------------------------------------

proc explore_tcom {handle} {
   #  Explore the handle we got back....
   set ihandle [ ::tcom::info interface $handle ];
   set iname [ $ihandle name ];
   puts [ concat "Interface name: " $iname ];

   set methodlist [ $ihandle methods ];

   puts [ concat "There are " [ llength $methodlist ] \
         " elements in the method list" ];

   set index 0;
   while { [ llength $methodlist] > $index } {
      set one [lindex $methodlist $index];

      set memberid [ lindex $one 0 ];
      set returntype [ lindex $one 1 ];
      set methodname [ lindex $one 2 ];
      set parmlist [ lindex $one 3 ];
      puts [ concat "name: " $methodname ", memberid: " $memberid ", \
            parmlist " $parmlist ];

      set index [ expr $index + 1 ];
   }
```

```
   #  Explore properties of the handle
   set proplist [ $ihandle properties ];

   puts [ concat "\nThere are " [ llength $proplist ] \
         " elements in the properties list" ];

   set index 0;
   while { [ llength $proplist] > $index } {
      set one [lindex $proplist $index];

      set memberid [ lindex $one 0 ];
      set rwmode [ lindex $one 1 ];
      set datatype [ lindex $one 2 ];
      set propname [ lindex $one 3 ];
      set descriptions [ lindex $one 4 ];
      puts [ concat "name: " $propname ", memberid: " $memberid ", \
            rwmode " $rwmode ", datatype: " $datatype ];

      set index [ expr $index + 1 ];
   }
}; # explore_tcom

### ------------------------------------------------------------------
###                          End of Procedures
### ==================================================================

### ==================================================================
###                       * Begin Main Routine *
### ------------------------------------------------------------------

set framename "empty";           # No default frame name
set configfile "";
set exploreinterfaces 0;

# Check command line arguments
while {[ ::cmdline::getopt argv {"e" "f.arg" "c.arg"} c valvar ] > 0} {
   switch $c \
      "c"         { set configfile $valvar} \
      "e"         { set exploreinterfaces 1}\
      "f"         { set framename $valvar} \
      "default"   { puts $usage; exit 1; } ;
}; # while statement


# Ask for the hostname or IP address of the 16900 frame
if {$framename == "empty"} {
   puts "What is the hostname or IP of the 16900 frame? ";
   set framename [ gets stdin ];
};

puts "Connecting to '$framename'";

# Open the connection to the logic analyzer
set lahandle [ ::tcom::ref createobject "AgtLA.Connect" ];
if {$lahandle == 0} {
   puts "Error opening AgtLA.Connect";
```

```
   exit 1;
}

if {$exploreinterfaces == 1 } {
   puts "******************************
   AgtLA.Connect handle information:
   ";
   explore_tcom $lahandle;
   puts "******************************";
}

# Attach to the frame....
set laframe [ $lahandle Instrument $framename];

if {$exploreinterfaces == 1 } {
   puts "******************************
   Instrument handle information:
   ";
   explore_tcom $laframe;
   puts "******************************";
}

# If they specified a config file, load it
if {$configfile ne ""} {
   set openreturn [ $laframe Open $configfile 0];
   puts [ concat "Open returned " $openreturn ];
}

# Do the run command
$laframe Run 0;      # Non repetitive run

# Wait for the run to finish
$laframe WaitComplete 10;  # Wait until meas complete or 10 seconds

#  Get the first analyzer's data
set analyzers [ $laframe Modules ];

if {$exploreinterfaces == 1 } {
   puts "******************************
   Analyzers handle information:
   ";
   explore_tcom $analyzers;
   puts "******************************";
}

# In order to pass an integer to the COM object, must
# force the internal representation to an integer using
# the following two lines:
#
set intval -1;
incr intval;
set analyzer [ $analyzers Item $intval ];

if {$exploreinterfaces == 1 } {
   puts "******************************
   analyzer handle information:
   ";
```

```
   explore_tcom $analyzer;
   puts "******************************";
}


#  Get the first analyzer's bus/signal names
set bus_signal_names [ $analyzer BusSignals ];

if {$exploreinterfaces == 1 } {
   puts "******************************
   bus_signal_names handle information:
   ";
   explore_tcom $bus_signal_names;
   puts "******************************";
}

#  Walk through the bus/signal names.  Find out:
#     type of data (probed, samplenum, time)
#     max width of column (max of bus/signal name and printed value)
#
set num_bus_signal_names [$bus_signal_names Count];
set index -1;
incr index;       # Force to be integer 0
while {$index < $num_bus_signal_names} {
   set bus_signal_name [$bus_signal_names Item $index];

   if {$exploreinterfaces == 1 && $index == 0} {
      puts "******************************
      bus_signal_name handle information:
      ";
      explore_tcom $bus_signal_name;
      puts "******************************";
   }


   set name [$bus_signal_name Name];
   set namewidth [expr [string length $name] + 1];

   set datahandle [$bus_signal_name BusSignalData];

   if {$exploreinterfaces == 1 && $index == 0} {
      puts "******************************
      data handle information:
      ";
      explore_tcom $datahandle;
      puts "******************************";
   }

   set bits [$bus_signal_name BitSize];

   # The bustype is an integer. For each type, convert to the type
   # of print-out we want
   set bustype [$bus_signal_name BusSignalType];


   switch $bustype {
      1        {  #AgtBusSignalProbed
```

```
                  set datatype 6;   # StringHex
                  set bitwidth [expr $bits / 4 + 1];
               }
      2        {  # AgtBusSignalGenerated
                  set datatype 7;   # DataString
                  set bitwidth 30;  # Arbitrary....
               }

      3        {  # AgtBusSignalSampleNum
                  set datatype 3;   # StringDecimal
                  set bitwidth [expr $bits / 10 + 1 ];
               }
      4        {  # AgtBusSignalTime
                  set datatype 4;   # DataTime
                  set bitwidth 30;  # Arbitrary....
               }
      }



   set width $namewidth;
   if {$width < $bitwidth} {set width $bitwidth};


   lappend bus_signal_info $name $bits $width $datatype $datahandle;
   puts "$name: $bits bits, $width width, $datatype datatype";

   incr index;
}

#  Print out the bus/signal names,
foreach {name bits width datatype datahandle} $bus_signal_info {
   puts -nonewline [ format "%*s " $width $name ];
}
puts "";

#  The most efficient way to get data is to get a large
#  chunk of data for the first bus/signal name, then the next, and
#  so on.  For the purposes of this example, just grab
#  one sample at a time and go across the row....
#
for {set i 0} {$i < 10} {incr i} {
   foreach {name bits width datatype datahandle} $bus_signal_info {
      set value [$datahandle GetDataBySample $i $i $datatype numrows];
      puts -nonewline [ format "%*s " $width $value ];
   }
   puts "";
}
```

4 COM Automation Reference

Objects, Methods, and Properties Quick Reference

[Automation Home (see page 3)] [Objects (see page 79)]

Objects	Methods/Properties	Description
AnalyzerModule (see page 80)		A state/timing analyzer hardware measurement module.
methods	GetRawData (see page 155)	Given a range, returns the raw analyzer data.
	GetRawTimingZoomData (see page 156)	Given a range, returns the raw analyzer timing zoom data.
	RecallTriggerByName (see page 174)	Loads a named trigger from the recall buffer.
	RecallTriggerByFile (see page 173)	Loads a previously saved trigger file on the instrument file system.
	SimpleTrigger (see page 177)	Trigger on a simple condition with optional occurrence and storage qualification.
	WaitComplete (see page 180)	Waits until the analyzer module, tool, and viewer measurements are complete.
properties	Setup (see page 211)	Gets or sets the logic analyzer's XML-format setup specification.
	Trigger (see page 217)	Gets or sets the logic analyzer's XML-format trigger specification.
BusSignal (see page 81)		A named and grouped set of pod channels.
methods	IsTimingZoom (see page 167)	Is this a timing zoom bus/signal?
properties	Activity (see page 191)	Gets the activity indicators of the bus/signal.
	BitSize (see page 192)	Gets the number of channels in the bus/signal.
	BusSignalData (see page 192)	Gets the acquisition data associated with a bus/signal.
	BusSignalType (see page 193)	Gets the type of bus/signal.
	ByteSize (see page 194)	Gets the size of the bus/signal in bytes.
	Channels (see page 195)	Gets the channels defined in the bus/signal.
	CreatorName (see page 199)	Gets the name of the module, tool, or viewer that created this bus/signal.
	Name (see page 205)	Gets or sets the name of the bus/signal.
	Polarity (see page 207)	Gets the polarity of the bus/signal.
	Symbols (see page 214)	Gets or sets the symbols associated with a bus/signal.
BusSignalData (see page 81)		A generic bus/signal data object.
properties	Type (see page 217)	Gets the specific bus/signal data type.
BusSignalDifference (see page 81)		Represents the different values for a particular bus/signal within a sample that has differences.
properties	Name (see page 205)	Gets the bus/signal name associated with the sample difference.
	Reference (see page 208)	Gets the reference buffer value associated with the sample difference.
	Value (see page 218)	Gets the data value associated with the sample difference.

Objects	Methods/Properties	Description
BusSignalDifferences (see page 82)		A collection object that contains all of the SampleDifference object's buses/signals with differences.
properties	_NewEnum (see page 219)	Used by Visual Basic to support the implementation of **For Each ... Next**.
	Count (see page 196)	Gets the number of bus/signal differences in the collection.
	Item (see page 203)	Given an index into the collection, gets a BusSignalDifference (see page 81) object from the collection.
BusSignals (see page 86)		A collection of the hardware module's defined BusSignals.
methods	Add (see page 127)	Adds a new bus/signal to the collection.
	Remove (see page 175)	Removes a bus/signal from the collection.
properties	_NewEnum (see page 219)	Used by Visual Basic to support the implementation of **For Each ... Next**.
	Count (see page 196)	Gets the number of BusSignal (see page 81) objects in the collection.
	Item (see page 203)	Gets one of the BusSignal (see page 81) objects in the collection given either an index or name.
CompareWindow (see page 90)		A window that compares bus/signal data.
methods	Execute (see page 134)	Executes the compare using the current options.
properties	Options (see page 206)	Gets or sets the Compare window options.
	SampleDifferences (see page 210)	Gets a collection of all the samples with differences found in the last comparison.
Connect (see page 90)		A connection to the logic analyzer instrument.
methods	CopyFile (see page 129)	Copies a file to the instrument file system.
	GetRemoteInfo (see page 157)	Gets the logic analyzer's remote user login and computer name.
properties	Instrument (see page 202)	Gets the logic analyzer instrument object.
ConnectSystem (see page 91)		A connection to the logic analyzer system.
methods	Connect (see page 129)	Connects to the remote logic analyzer system.
	RecvFile (see page 174)	Copies a file from the remote logic analyzer system to your local system.
	SendFile (see page 177)	Copies a file from your local system to the remote logic analyzer system.
Exerciser (see page 91) For a detailed description of its methods, refer to the *Using COM Commands for U4421A Module* section in the *U4421A D-PHY Analyzer and Exerciser* online help.		A hardware measurement module for sending defined stimulus to a DUT.

Objects	Methods/Properties	Description
methods	ApplyToHardware(string cmd)	Applies the stimulus related settings loaded in the Setup dialog box to the stimulus hardware module.
	ExecuteCommand(string cmd, out string result)	Loads the specified stimulus settings to the Setup dialog box of the hardware module in the Logic and Protocol Analyzer GUI. In addition, the command also immediately applies these settings to the hardware module even when the module is in the Running state. The command is useful for changing stimulus settings of the module at runtime and for inserting packets dynamically.
	GetXmlFormat(string cmdKey, out string currentSettings)	Returns the XML command format for the given command name.
	LoadExerciserParameters(string cmd, out string result)	Loads the stimulus related settings for the hardware module to the Setup dialog box of the module in the Logic and Protocol Analyzer GUI.
	StartExerciser	Starts the transmission of stimulus from the hardware module to a DUT.
	StopExerciser	Stops the transmission of stimulus from the hardware module to a DUT.
FindResult (see page 92)		Gets the results from the Find (see page 136), FindNext (see page 143), and FindPrev (see page 143) method calls.
properties	Found (see page 201)	Gets the found status.
	OccurrencesFound (see page 206)	Gets the number of occurrences found.
	SubrowFound (see page 214)	Gets the subrow number if found on a subrow.
	TimeFound (see page 216)	Gets the time found as a double.
	TimeFoundString (see page 216)	Gets the time found as a string.
Frame (see page 92)		A logic analyzer frame.
properties	ComputerName (see page 196)	Gets the computer name of the logic analyzer frame.
	Description (see page 200)	Gets a description of the logic analyzer frame.
	IPAddress (see page 202)	Gets the frame's IP address(es).
	TargetControlPort (see page 215)	Gets or sets the target control port value.
Frames (see page 93)		A collection of logic analyzer frames connected via the multiframe connector.
properties	_NewEnum (see page 219)	Used by Visual Basic to support the implementation of **For Each ... Next**.
	Count (see page 196)	Gets the number of frames in the collection.
	Item (see page 203)	Gets one of the frames in the collection given either an index, computer name, or IP address.
Instrument (see page 93)		The logic analyzer instrument.
methods	Close (see page 128)	Closes the current configuration.

Objects	Methods/Properties	Description
	DeleteFile (see page 129)	Deletes a file on the instrument file system.
	DoAction (see page 130)	Execute a specific XML-based command action.
	DoCommands (see page 130)	Execute a particular XML-based command.
	Export (see page 134)	Exports data to a file on the instrument file system.
	ExportEx (see page 135)	Exports data to a file on the instrument file system.
	GetProbeByName (see page 154)	Given a probe name, returns its corresponding probe object.
	GetModuleByName (see page 149)	Given a module name, returns its corresponding hardware module object.
	GetToolByName (see page 159)	Given a tool name, returns its corresponding tool object.
	GetWindowByName (see page 160)	Given a window name, returns its corresponding window object.
	Import (see page 165)	Imports data from a file located on the instrument file system.
	ImportEx (see page 166)	Imports data from a file located on the instrument file system into a particular module.
	GoOffline (see page 160)	Disconnects the user interface from the logic analyzer frame.
	GoOnline (see page 164)	Connects the user interface to a specific logic analyzer frame.
	IsOnline (see page 166)	Tells whether the user interface is connected to a logic analyzer frame.
	New (see page 167)	Creates a new instrument Overview.
	Open (see page 168)	Loads a previously saved configuration file on the instrument file system.
	PanelLock (see page 168)	Disables user access to the instrument front panel or remote display.
	PanelUnlock (see page 171)	Re-enables user access to the instrument front panel or remote display.
	QueryCommand (see page 171)	Query for XML-based commands.
	Run (see page 176)	Starts running all modules.
	Save (see page 176)	Saves the current configuration to a file on the instrument file system.
	Stop (see page 179)	Stops all currently running modules.
	WaitComplete (see page 180)	Waits until all module, tool, and viewer measurements are complete.

Objects	Methods/Properties	Description
properties	Frames (see page 202)	Gets a collection of logic analyzer frames connected via the multiframe connector.
	Markers (see page 204)	Gets a collection of all the markers in the instrument.
	Model (see page 204)	Gets the model number.
	Modules (see page 205)	Gets a collection of all the hardware modules in the instrument.
	Overview (see page 207)	Gets the XML-format Overview window specification.
	PanelLocked (see page 207)	Indicates the front panel is locked. If locked, the message is returned.
	Probes (see page 208)	Gets a collection of all currently defined probes.
	RemoteComputerName (see page 209)	Gets or sets the remote computer name.
	RemoteUserName (see page 209)	Gets or sets the remote user login name.
	SelfTest (see page 211)	Gets the SelfTest object.
	Status (see page 213)	Gets the status of all hardware modules.
	Tools (see page 216)	Gets a collection of all active software tools.
	VBE (see page 218)	Gets the VBE extensibility object.
	Version (see page 219)	Gets the version number of the system software.
	Windows (see page 219)	Gets a collection of all active windows.
Marker (see page 95)		A reference point in the captured data.
properties	Comments (see page 195)	Gets or sets the marker comments.
	Name (see page 205)	Gets or sets the name of the marker.
	TextColor (see page 215)	Gets or sets the marker text color.
	BackgroundColor (see page 191)	Gets or sets the marker background color.
	Position (see page 208)	Gets or sets the marker position.
Markers (see page 95)		A collection of all the defined markers.
methods	Add (see page 128)	Adds a new marker to the collection using specific values.
	AddXML (see page 128)	Adds multiple markers to the collection using an XML string.
	Remove (see page 175)	Removes a marker from the collection.
	RemoveAll (see page 175)	Removes all markers from the collection.
	RemoveXML (see page 175)	Removes multiple markers from the collection using an XML string.
properties	_NewEnum (see page 219)	Used by Visual Basic to support the implementation of **For Each ... Next**.
	Count (see page 196)	Gets the number of markers in the collection.
	Item (see page 203)	Gets one of the markers in the collection given either an index or name.
Module (see page 99)		A generic hardware module.

Objects	Methods/Properties	Description
methods	DoAction (see page 130)	Execute a specific XML-based command action.
	DoCommands (see page 130)	Execute a particular XML-based command.
	QueryCommand (see page 171)	Query for XML-based commands.
	WaitComplete (see page 180)	Waits until the module, tool, and viewer measurements are complete.
properties	BusSignals (see page 194)	Gets a collection of the module's defined bus/signals.
	CardModels (see page 194)	Gets the card model numbers.
	Description (see page 200)	Gets a description of the module.
	Frame (see page 201)	Gets the frame in which the module resides.
	Model (see page 204)	Gets the model number.
	Name (see page 205)	Gets or sets the name of the module.
	RunningStatus (see page 210)	Gets the detailed running status of the module.
	Slot (see page 212)	Gets the module's slot location in the frame.
	Status (see page 213)	Gets the status of the module.
	StatusMsg (see page 213)	Gets the formatted status message.
	Type (see page 217)	Gets the specific module type.
Modules (see page 100)		A collection of all the hardware modules installed in the instrument.
properties	_NewEnum (see page 219)	Used by Visual Basic to support the implementation of **For Each ... Next**.
	Count (see page 196)	Gets the number of modules in the collection.
	Item (see page 203)	Gets one of the modules in a collection given either a slot, index, or name.
Probe (see page 100)		A generic probe.
methods	DoAction (see page 130)	Execute a specific XML-based command action.
	DoCommands (see page 130)	Execute a particular XML-based command.
	QueryCommand (see page 171)	Query for XML-based commands.
properties	Name (see page 205)	Gets or sets the name of the probe.
	Type (see page 217)	Gets the probe's type.
Probes (see page 101)		A collection of all active probes.
properties	_NewEnum (see page 219)	Used by Visual Basic to support the implementation of **For Each ... Next**.
	Count (see page 196)	Gets the number of probes in the collection.
	Item (see page 203)	Gets one of the probes in the collection given either an index or name.
ProtocolWindow (see page 104)		A window which displays protocol data from a serial analyzer.

Objects	Methods/Properties	Description
methods	GetProtocolDataFields (see page 154)	Gets the acquisition data for the fields displayed in the Protocol Viewer.
	WriteProtocolDataFieldsToFile (see page 188)	Writes the acquisition data displayed in various fields in the Protocol Viewer to a specified CSV file.
	GetTriggerSampleNumber (see page 159)	Gets the packet number and channel of the trigger packet.
	FindImages (see page 140)	Searches acquisition data for frames containing images and puts them into the Protocol Viewer's list of images.
	GetImageList (see page 147)	Returns a string containing the Protocol Viewer's list of images. Optionally, it writes this list to a text file.
	GetPacketCount (see page 152)	Given a channel name, returns the number of packets for that particular channel.
	WriteAllImagesToFiles (see page 184)	Writes all the images in the Protocol Viewer's list to separate bitmap files in a specified folder.
	WriteImageToFile (see page 186)	Writes a single image to a bitmap file. The image is selected using the 'Frame Number' string from the image list.
SampleBusSignalData (see page 104)		The data for a specific bus/signal captured in the state or timing sampling modes.
methods	GetDataBySample (see page 144)	Given a sample range, returns an array of data.
	GetDataByTime (see page 146)	Given a time range, returns an array of data.
	GetNumSamples (see page 152)	Given a range, returns the number of samples stored.
	GetSampleNumByTime (see page 158)	Gets the closest sample number corresponding to the time given.
	GetTime (see page 158)	Given a range, returns the time for this bus/signal in the format specified by the data type given.
properties	DataType (see page 199)	Gets the recommended bus/signal data type.
	EndSample (see page 201)	Gets the data's ending sample number relative to trigger.
	EndTime (see page 201)	Gets the data's ending time relative to trigger.
	StartSample (see page 212)	Gets the data's starting sample number relative to trigger.
	StartTime (see page 212)	Gets the data's starting time relative to trigger.
SampleDifference (see page 114)		Represents a sample containing a CompareWindow difference.
properties	BusSignalDifferences (see page 193)	Gets a collection of all the buses/signals with differences for this sample.
	SampleNum (see page 211)	Gets the sample number at which differences occurred.
SampleDifferences (see page 115)		A collection object of all the sample differences in the CompareWindow object.
properties	_NewEnum (see page 219)	Used by Visual Basic to support the implementation of **For Each ... Next**.
	Count (see page 196)	Gets the number of bus/signal differences in the collection.
	Item (see page 203)	Given an index into the collection, gets a SampleDifference (see page 114) object from the collection.

Objects	Methods/Properties	Description
SelfTest (see page 115)		Object for running instrument self-tests.
methods	TestAll (see page 180)	Runs an instrument's self-tests.
SerialModule (see page 117)		A hardware measurement module for viewing and analyzing serial data.
methods	RecallTriggerByFile (see page 173)	Loads a previously saved trigger file located on the instrument file system.
Tool (see page 118)		A generic instrument software tool.
methods	DoAction (see page 130)	Execute a specific XML-based command action.
	DoCommands (see page 130)	Execute a particular XML-based command.
	QueryCommand (see page 171)	Query for XML-based commands.
properties	BusSignals (see page 194)	Gets a collection of the tool's defined bus/signals.
	Name (see page 205)	Gets or sets the name of the tool.
	Type (see page 217)	Gets the specific tool type.
Tools (see page 118)		A collection of all of the instrument's software tools.
properties	_NewEnum (see page 219)	Used by Visual Basic to support the implementation of **For Each ... Next**.
	Count (see page 196)	Gets the number of tools in the collection.
	Item (see page 203)	Gets one of the tools in the collection given either an index or name.
Window (see page 122)		A generic instrument display window.
methods	DoAction (see page 130)	Execute a specific XML-based command action.
	DoCommands (see page 130)	Execute a particular XML-based command.
	Find (see page 136)	Finds a specified data event with optional occurrence and time duration.
	FindNext (see page 143)	Finds the next event by searching forward from the last event found using the event specified by the last call to Find.
	FindPrev (see page 143)	Finds the previous event by searching backward from the last event found using the event specified by the last call to Find.
	GoToPosition (see page 165)	Moves the center of the window to a new position.
	QueryCommand (see page 171)	Query for XML-based commands.
properties	BusSignals (see page 194)	Gets a collection of the window's defined bus/signals.
	Name (see page 205)	Gets or sets the name of the window.
	Type (see page 217)	Gets the window's type.
Windows (see page 122)		A collection of all of the instrument's display windows.
properties	_NewEnum (see page 219)	Used by Visual Basic to support the implementation of **For Each ... Next**.
	Count (see page 196)	Gets the number of windows in the collection.
	Item (see page 203)	Gets one of the windows in the collection given either an index or name.

See Also

- Logic Analyzer Object Hierarchy Overview (see page 77)

Object Hierarchy Overview

The Instrument object represents the logic analysis system. From the Instrument object, you can directly access objects by using the Instrument objects properties and methods, or, you can indirectly access objects through other objects obtained by these properties and methods.

The Instrument object contains collections of: Modules (see page 100) (that represent the hardware installed in the instrument), Probes (see page 101) (that organize probes connected to a DUT), Tools (see page 118) (that filter or decode captured data), and Windows (see page 122) (that display captured data). When the instrument is initially powered up, the Probes and Tools collections are empty and are not available until they are created in the user interface or restored by opening configuration files.

Module (see page 99), Tool (see page 118), and Window (see page 122) objects return data through a BusSignalData (see page 81) object, which returns information about:

- Directly acquired data when obtained from a Module (see page 99) object.
- Created and/or manipulated data when obtained from a Tool (see page 118) object.
- Displayed data when obtained from a Window (see page 122) object.

The following tree illustrates the hierarchy of the Instrument object:

- Connect (see page 90)
  - Instrument (see page 93)
    - Frames (see page 93)
      - Frame (see page 92)
    - Markers (see page 95)
      - Marker (see page 95)
    - SelfTest (see page 115)
    - Modules (see page 100) – Collection of all modules in the system.
      - *Module* (see page 99) – Generic module object.
        - BusSignals (see page 86)
          - BusSignal (see page 81)
            - *BusSignalData* (see page 81)
            - Sample*BusSignalData* (see page 104)
      - Analyzer*Module* (see page 80) – Logic analyzer specific object.
      - SerialModule (see page 117) – Serial analyzer specific object.
      - Exerciser (see page 91) - A hardware measurement module for sending defined stimulus to a DUT.
    - Probes (see page 101)
      - Probe (see page 100)
    - Tools (see page 118)
      - Tool (see page 118)
        - BusSignals (see page 86)
          - BusSignal (see page 81)
            - *BusSignalData* (see page 81) – Generic bus/signal data object.
            - Sample*BusSignalData* (see page 104) – Sample bus/signal data specific object.
    - Windows (see page 122)

        - *Window* (see page 122) – Generic window object.
            - BusSignals (see page 86)
                - BusSignal (see page 81)
                    - *BusSignalData* (see page 81)
                    - Sample*BusSignalData* (see page 104)
            - FindResult (see page 92)
        - Compare*Window* (see page 90) – Compare window specific object.
            - SampleDifferences (see page 115)
                - SampleDifference (see page 114)
                    - BusSignalDifferences (see page 82)
                        - BusSignalDifference (see page 81)
        - ProtocolWindow (see page 104) - Protocol Viewer window specific object.
- ConnectSystem (see page 91)

Italics = This denotes a generic/specific relationship known as inheritance.

Containment and Inheritance

There are generic and specific objects. For example:

- The Module (see page 99) object is generic; it contains the methods and properties for the AnalyzerModule (see page 80) object.
- The AnalyzerModule (see page 80) object contains logic analyzer specific properties and methods (such as the GetDataBySample (see page 144) method), but it also has access to all of the generic properties and methods in the Module (see page 99) object.

If you know what type of object you have, use the specific objects (like AnalyzerModule (see page 80)).

If you don't know what type of object you have, use the generic object (like Module (see page 99)); then, depending on the type, you can use the more specific objects. For example:

```
Dim myModule As AgtLA.Module
Set myModule = AgtLA.Modules(0)    ' Start generic module.

Dim myAnalyzerModule As AgtLA.AnalyzerModule
If (myModule.Type = "Analyzer") Then    ' Once you know the type,
  Set myAnalyzerModule = myModule       ' use the more specific object
End If                                  ' (coerce).
```

See Also

- Object Quick Reference (see page 79)

Object Quick Reference

[Automation Home (see page 3)] [Objects]

Object	Description
AnalyzerModule (see page 80)	A state/timing analyzer hardware measurement module.
BusSignal (see page 81)	A named and grouped set of pod channels.
BusSignalData (see page 81)	A generic bus/signal data object.
BusSignalDifference (see page 81)	Represents the different values for a particular bus/signal within a sample that has differences.
BusSignalDifferences (see page 82)	A collection object that contains all of the SampleDifference object's buses/signals with differences.
BusSignals (see page 86)	A collection of the hardware module's defined BusSignals.
CompareWindow (see page 90)	A window that compares bus/signal data.
Connect (see page 90)	A connection to the logic analyzer instrument.
ConnectSystem (see page 91)	A connection to the logic analyzer system.
Exerciser (see page 91)	A hardware measurement module for sending defined stimulus to a DUT.
FindResult (see page 92)	Gets the results from the Find (see page 136), FindNext (see page 143), and FindPrev (see page 143) method calls.
Frame (see page 92)	A logic analyzer frame.
Frames (see page 93)	A collection of logic analyzer frames connected via the multiframe connector.
Instrument (see page 93)	The logic analyzer instrument.
Marker (see page 95)	A reference point in the captured data.
Markers (see page 95)	A collection of all the defined markers.
Module (see page 99)	A generic hardware module.
Modules (see page 100)	A collection of all the hardware modules installed in the instrument.
Probe (see page 100)	A generic probe.
Probes (see page 101)	A collection of all active probes.
ProtocolWindow (see page 104)	A window which displays protocol data from a serial analyzer.
SampleBusSignalData (see page 104)	The data for a specific bus/signal captured in the state or timing sampling modes.
SampleDifference (see page 114)	Represents a sample containing a CompareWindow difference.
SampleDifferences (see page 115)	A collection object of all the sample differences in the CompareWindow object.
SelfTest (see page 115)	Provides methods for running instrument self-tests.
SerialModule (see page 117)	A hardware measurement module for viewing and analyzing serial data.
Tool (see page 118)	A generic instrument software tool.

Object	Description
Tools (see page 118)	A collection of all of the instrument's software tools.
Window (see page 122)	A generic instrument display window.
Windows (see page 122)	A collection of all of the instrument's display windows.

See Also

- Logic Analyzer Object Hierarchy Overview (see page 77)

AnalyzerModule Object

[Automation Home (see page 3)] [Objects (see page 79)]

To Access

```
Dim variable As AgtLA.AnalyzerModule
Set variable = Module (see page 99)
```

Description

The AnalyzerModule object represents a state/timing analyzer hardware measurement module.

Since this object is derived from the Module (see page 99) object, a Set must be used to get to these specific methods and properties. For example:

```
' You must create the Connect object (see page 90) and use it
' to access the Instrument object.  In this example, "myInst"
' represents the Instrument object.
'
' Get the AnalyzerModule specific object.
Dim myAnalyzer As AgtLA.AnalyzerModule
Set myAnalyzer = myInst.Modules(0)
MsgBox "Trigger: " + myAnalyzer.Trigger
```

Methods

Method	Description
GetRawData (see page 155)	Given a range, returns the raw analyzer data.
GetRawTimingZoomData (see page 156)	Given a range, returns the raw analyzer timing zoom data.
RecallTriggerByName (see page 174)	Loads a named trigger from the recall buffer.
RecallTriggerByFile (see page 173)	Loads a previously saved trigger file on the instrument file system.
SimpleTrigger (see page 177)	Trigger on a simple condition with optional occurrence and storage qualification.
WaitComplete (see page 180)	Waits until the analyzer module, tool, and viewer measurements are complete.

(Also Includes Module (see page 99) object methods)

Properties

Property	Description
Setup (see page 211)	Gets or sets the logic analyzer's XML-format setup specification.
Trigger (see page 217)	Gets or sets the logic analyzer's XML-format trigger specification.

(Also Includes Module (see page 99) object properties)

BusSignal Object

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 86)]

To Access

- BusSignals.Item (see page 203) IndexOrName
- BusSignals IndexOrName

Methods

Method	Description
IsTimingZoom (see page 167)	Is this a timing zoom bus/signal?

Properties

Property	Description
Activity (see page 191)	Gets the activity indicators of the bus/signal.
BitSize (see page 192)	Gets the number of channels in the bus/signal.
BusSignalData (see page 192)	Gets the acquisition data associated with a bus/signal.
BusSignalType (see page 193)	Gets the type of bus/signal.
ByteSize (see page 194)	Gets the size of the bus/signal in bytes.
Channels (see page 195)	Gets the channels defined in the bus/signal.
CreatorName (see page 199)	Gets the name of the module, tool, or viewer that created this bus/signal.
Name (see page 205)	Gets or sets the name of the bus/signal.
Polarity (see page 207)	Gets the polarity of the bus/signal.
Symbols (see page 214)	Gets or sets the symbols associated with a bus/signal.

See Also

- BusSignals (see page 86) object

BusSignalData Object

[Automation Home (see page 3)] [Objects (see page 79)]

To Access

- BusSignal.BusSignalData (see page 192)

Methods

There are no methods.

Properties

Property	Description
Type (see page 217)	Gets the specific bus/signal data type.

See Also

- BusSignalData (see page 192) property

BusSignalDifference Object

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 82)]

To Access

- BusSignalDifferences.Item (see page 203) IndexOrName
- BusSignalDifferences IndexOrName

Methods

There are no methods.

Properties

Property	Description
Name (see page 205)	Gets the bus/signal name associated with the sample difference.
Reference (see page 208)	Gets the reference buffer value associated with the sample difference.
Value (see page 218)	Gets the data value associated with the sample difference.

See Also

- BusSignalDifferences (see page 82) object

BusSignalDifferences Object

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 82)]

To Access

- SampleDifference.BusSignalDifferences (see page 193)

Methods

There are no methods.

Properties

Property	Description
_NewEnum (see page 219)	Used by Visual Basic to support the implementation of **For Each ... Next**.
Count (see page 196)	Gets the number of bus/signal differences in the collection.
Item (see page 203)	Given an index into the collection, gets a BusSignalDifference (see page 81) object from the collection.

See Also

- BusSignalDifferences (see page 193) property

BusSignalDifferences Example

Visual Basic

```
' You must create the Connect object (see page 90) and use it
' to access the Instrument object.  In this example, "myInst"
' represents the Instrument object.
'
'
' Load the configuration file.
myInst.Open ("c:\LA\Configs\mpc860_demo_compare.ala")

' Run the measurement, wait for it to complete.
myInst.Run
myInst.WaitComplete (20)

' Get the CompareWindow object.
Dim myCompare As AgtLA.CompareWindow
Set myCompare = myInst.GetWindowByName("Compare-1")

' Set the CompareWindow options.
Dim myOptions As String
myOptions = "<Options ReferenceOffset='0' Range='M1..M2' " + _
      "MaxDifferences='0'/>"
myCompare.Options = myOptions

' Display the CompareWindow options.
myOptions = myCompare.Options
```

```
MsgBox "CompareWindow Options: " + myOptions

' Execute the compare.
myCompare.Execute

' Display the bus/signal differences between M1 and M2.
Dim mySampleDiff As AgtLA.SampleDifference
Dim myBusSignalDiff As AgtLA.BusSignalDifference
Dim myString As String

For Each mySampleDiff In myCompare.SampleDifferences
   myString = myString + vbNewLine + "Sample: " + _
         Str(mySampleDiff.SampleNum)
   For Each myBusSignalDiff In mySampleDiff.BusSignalDifferences
      ' Add the bus signal difference information to the string.
      myString = myString + ", Bus/signal: " + myBusSignalDiff.Name
      myString = myString + ", Value: " + myBusSignalDiff.Value
      myString = myString + ", Ref: " + myBusSignalDiff.Reference
   Next
Next
MsgBox "BusSignal difference values: " + vbNewLine + myString
```

Visual C++

```
//
//  This simple Visual C++ Console application demonstrates how to use
//  the Keysight 168x/169x/169xx COM interface to display bus/signal
//  differences in the Compare window.
//
//  This project was created in Visual C++ Developer.  To create a
//  similar project:
//
//    - Execute File -> New
//    - Select the Projects tab
//    - Select "Win32 Console Application"
//    - Select A "hello,World!" application (Visual Studio 6.0)
//
//  To make this buildable, you need to specify your "import" path
//  in stdafx.h (search for "TODO" in that file).  For example, add:
//  #import "C:/Program Files/Keysight Technologies/Logic Analyzer/LA \
//  COM Automation/agClientSvr.dll"
//
//  To run, you need to specify the host logic analyzer to connect
//  to (search for "TODO" below).
//

#include "stdafx.h"


////////////////////////////////////////////////////////////////////
//
//  Forward declarations.
//
void DisplayError(_com_error& err);


////////////////////////////////////////////////////////////////////
//
//  main() entry point.
```

```
//
int main(int argc, char* argv[])
{
   printf("*** Main()\n");

   //
   //  Initialize the Microsoft COM/ActiveX library.
   //
   HRESULT hr = CoInitialize(0);

   if (SUCCEEDED(hr))
   {
      try {   // Catch any unexpected run-time errors.
         _bstr_t hostname = "mtx33";   // TODO, use your logic
                                       // analysis system hostname.
         printf("Connecting to instrument '%s'\n", (char*) hostname);

         // Create the connect object and get the instrument object.
         AgtLA::IConnectPtr pConnect =
               AgtLA::IConnectPtr(__uuidof(AgtLA::Connect));
         AgtLA::IInstrumentPtr pInst =
               pConnect->GetInstrument(hostname);

         // Load the configuration file.
         _bstr_t configFile = "C:\\LA\\Configs\\compare.ala";
         printf("Loading the config file '%s'\n", (char*) configFile);
         pInst->Open(configFile, FALSE, "", TRUE);

         // Run the measurement, wait for it to complete.
         pInst->Run(FALSE);
         pInst->WaitComplete(20);

         // Get the CompareWindow object.
         AgtLA::ICompareWindowPtr pCompareWindow =
               pInst->GetWindowByName("Compare-1");

         // Set the CompareWindow options.
         _bstr_t myCompareOptions = "<Options ReferenceOffset='0' \
               Range='M1..M2' MaxDifferences='0'/>";
         pCompareWindow->PutOptions(myCompareOptions);

         // Display the CompareWindow options.
         myCompareOptions = pCompareWindow->GetOptions();
         printf("CompareWindow options: '%s'\n",
               (char*) myCompareOptions);

         // Execute the compare.
         pCompareWindow->Execute();

         // Display the bus/signal differences between M1 and M2.
         AgtLA::ISampleDifferencesPtr pSampleDifferences =
               pCompareWindow->GetSampleDifferences();
         for (long i = 0; i < pSampleDifferences->GetCount(); i++)
         {
            AgtLA::ISampleDifferencePtr pSampleDifference =
                  pSampleDifferences->GetItem(i);
            printf("Sample: '%d'\n", pSampleDifference->GetSampleNum());
```

```
            AgtLA::IBusSignalDifferencesPtr pBusSignalDifferences =
                  pSampleDifference->GetBusSignalDifferences();
            for (long j = 0; j < pBusSignalDifferences->GetCount(); j++)
            {
               // Print the bus signal difference information.
               AgtLA::IBusSignalDifferencePtr pBusSignalDifference =
                     pBusSignalDifferences->GetItem(j);
               printf("  Bus/signal: '%s'\n",
                     (char*) pBusSignalDifference->GetName());
               printf("  Value: '%s'\n",
                     (char*) pBusSignalDifference->GetValue());
               printf("  Ref: '%s'\n",
                     (char*) pBusSignalDifference->GetReference());
            }
         }
      }
      catch (_com_error& e) {
         DisplayError(e);
      }

      // Uninitialize the Microsoft COM/ActiveX library.
      CoUninitialize();
   }
   else
   {
      printf("CoInitialize failed\n");
   }

   return 0;
}


//////////////////////////////////////////////////////////////////////
//
//  Displays the last error -- used to show the last exception
//  information.
//
void DisplayError(_com_error& error)
{
   printf("*** DisplayError()\n");

   printf("Fatal Unexpected Error:\n");
   printf("  Error Number = %08lx\n", error.Error());

   static char errorStr[1024];
   _bstr_t desc = error.Description();

   if (desc.length() == 0)
   {
      // Don't have a description string.
      strcpy(errorStr, error.ErrorMessage());
      int nLen = lstrlen(errorStr);

      // Remove funny carriage return ctrl<M>.
      if (nLen > 2 && (errorStr[nLen - 2] == 0xd))
      {
```

```
            errorStr[nLen - 2] = '\0';
        }
    }
    else
    {
        strcpy(errorStr, desc);
    }

    printf("  Error Message = %s\n", (char*) errorStr);
}
```

BusSignals Object

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 86)]

To Access

- Module.BusSignals (see page 194)
- Tool.BusSignals (see page 194)
- Window.BusSignals (see page 194)

Methods

Method	Description
Add (see page 127)	Adds a new bus/signal to the collection.
Remove (see page 175)	Removes a bus/signal from the collection.

Properties

Property	Description
_NewEnum (see page 219)	Used by Visual Basic to support the implementation of **For Each ... Next**.
Count (see page 196)	Gets the number of BusSignal (see page 81) objects in the collection.
Item (see page 203)	Gets one of the BusSignal (see page 81) objects in the collection given either an index or name.

Remarks

NOTE The "Time" data is no longer returned as a bus/signal (see What's Changed (see page 223)).

See Also

- BusSignals (see page 194) property

BusSignals Example

Visual Basic

```
' You must create the Connect object (see page 90) and use it
' to access the Instrument object.  In this example, "myInst"
' represents the Instrument object.
'
'
' Display bus/signal information.
Dim myBusSignals As AgtLA.BusSignals
Set myBusSignals = myInst.GetModuleByName("My 1690A-1").BusSignals

Dim myBusSignal As AgtLA.BusSignal
Dim myString As String
```

```
For Each myBusSignal In myBusSignals
   myString = myString + "Name: " + myBusSignal.Name + vbNewLine
   myString = myString + "  BitSize=" + Str(myBusSignal.BitSize)
   myString = myString + ", ByteSize=" + Str(myBusSignal.ByteSize)
   myString = myString + ", Type="
   Select Case myBusSignal.BusSignalType
      Case AgtBusSignalSampleNum
         myString = myString + "AgtBusSignalSampleNum" + vbNewLine
      Case AgtBusSignalTime
         myString = myString + "AgtBusSignalTime" + vbNewLine
      Case AgtBusSignalGenerated
         myString = myString + "AgtBusSignalGenerated" + vbNewLine
      Case AgtBusSignalProbed
         myString = myString + "AgtBusSignalProbed" + vbNewLine
         myString = myString + "  Channels=" + _
               myBusSignal.Channels + vbNewLine
         myString = myString + "  Polarity=" + _
               myBusSignal.Polarity + vbNewLine
         myString = myString + "  Activity=" + _
               myBusSignal.Activity + vbNewLine
      Case Else
         myString = myString + "Unknown" + vbNewLine
   End Select
Next

MsgBox "Bus/signal information: " + myString
```

Visual C++

```
//
//  This simple Visual C++ Console application demonstrates how to use
//  the Keysight 168x/169x/169xx COM interface to display bus/signal
//  information.
//
//  This project was created in Visual C++ Developer.  To create a
//  similar project:
//
//    - Execute File -> New
//    - Select the Projects tab
//    - Select "Win32 Console Application"
//    - Select A "hello,World!" application (Visual Studio 6.0)
//
//  To make this buildable, you need to specify your "import" path
//  in stdafx.h (search for "TODO" in that file).  For example, add:
//  #import "C:/Program Files/Keysight Technologies/Logic Analyzer/LA \
//  COM Automation/agClientSvr.dll"
//
//  To run, you need to specify the host logic analyzer to connect
//  to (search for "TODO" below).
//

#include "stdafx.h"


////////////////////////////////////////////////////////////////////
//
//  Forward declarations.
//
void DisplayError(_com_error& err);
```

```
////////////////////////////////////////////////////////////////////
//
//  main() entry point.
//
int main(int argc, char* argv[])
{
   printf("*** Main()\n");

   //
   //  Initialize the Microsoft COM/ActiveX library.
   //
   HRESULT hr = CoInitialize(0);

   if (SUCCEEDED(hr))
   {
      try {   // Catch any unexpected run-time errors.
         _bstr_t hostname = "mtx33";   // TODO, use your logic
                                       // analysis system hostname.
         printf("Connecting to instrument '%s'\n", (char*) hostname);

         // Create the connect object and get the instrument object.
         AgtLA::IConnectPtr pConnect =
               AgtLA::IConnectPtr(__uuidof(AgtLA::Connect));
         AgtLA::IInstrumentPtr pInst =
               pConnect->GetInstrument(hostname);

         // Load the configuration file.
         _bstr_t configFile = "C:\\LA\\Configs\\config.ala";
         printf("Loading the config file '%s'\n", (char*) configFile);
         pInst->Open(configFile, FALSE, "", TRUE);

         // Get module whose bus/signal information will be displayed.
         _bstr_t moduleName = "MPC860 Demo Board";
         AgtLA::IModulePtr pModule = pInst->GetModuleByName(moduleName);

         // For each bus/signal, display a range of data.
         AgtLA::IBusSignalsPtr pBusSignals = pModule->GetBusSignals();
         for (long i = 0; i < pBusSignals->GetCount(); i++)
         {
            // Get the data for the bus/signal.
            AgtLA::IBusSignalPtr pBusSignal =
                  pModule->GetBusSignals()->GetItem(i);
            _bstr_t busSignal = pBusSignal->GetName();
            printf("Name: '%s'\n", (char*) busSignal);
            printf("  BitSize: '%d'\n", pBusSignal->GetBitSize());
            printf("  ByteSize: '%d'\n", pBusSignal->GetByteSize());

            switch(pBusSignal->GetBusSignalType())
            {
            case AgtLA::AgtBusSignalProbed:
               {
                  printf("  Type: 'AgtBusSignalProbed'\n");
               }
               break;
```

```
            case AgtLA::AgtBusSignalGenerated:
               {
                  printf("  Type: 'AgtBusSignalGenerated'\n");
               }
               break;

            case AgtLA::AgtBusSignalSampleNum:
               {
                  printf("  Type: 'AgtBusSignalSampleNum'\n");
               }
               break;

            case AgtLA::AgtBusSignalTime:
               {
                  printf("  Type: 'AgtBusSignalTime'\n");
               }

            default:
               break;

            }
         }
      }
      catch (_com_error& e) {
         DisplayError(e);
      }

      // Uninitialize the Microsoft COM/ActiveX library.
      CoUninitialize();
   }
   else
   {
      printf("CoInitialize failed\n");
   }

   return 0;
}


//////////////////////////////////////////////////////////////////////
//
//  Displays the last error -- used to show the last exception
//  information.
//
void DisplayError(_com_error& error)
{
   printf("*** DisplayError()\n");

   printf("Fatal Unexpected Error:\n");
   printf("  Error Number = %08lx\n", error.Error());

   static char errorStr[1024];
   _bstr_t desc = error.Description();

   if (desc.length() == 0)
   {
      // Don't have a description string.
```

```
      strcpy(errorStr, error.ErrorMessage());
      int nLen = lstrlen(errorStr);

      // Remove funny carriage return ctrl<M>.
      if (nLen > 2 && (errorStr[nLen - 2] == 0xd))
      {
         errorStr[nLen - 2] = '\0';
      }
   }
   else
   {
      strcpy(errorStr, desc);
   }

   printf("  Error Message = %s\n", (char*) errorStr);
}
```

CompareWindow Object

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 82)]

To Access

```
Dim variable As AgtLA.CompareWindow
Set variable = Window (see page 122)
```

Description

The CompareWindow object represents an instrument window that compares bus/signal data.

Methods

Method	Description
Execute (see page 134)	Executes the compare using the current options.

(Also Includes Window (see page 122) methods)

Properties

Property	Description
Options (see page 206)	Gets or sets the Compare window options.
SampleDifferences (see page 210)	Gets a collection of all the samples with differences found in the last comparison.

(Also Includes Window (see page 122) properties)

Connect Object

[Automation Home (see page 3)] [Objects (see page 79)]

To Access

```
Dim variable As AgtLA.Connect
Set variable = CreateObject("AgtLA.Connect")
```

Methods

Method	Description
CopyFile (see page 129)	Copies a file to the instrument file system.
GetRemoteInfo (see page 157)	Gets the logic analyzer's remote user login and computer name.

Properties

Property	Description
Instrument (see page 202)	Gets the logic analyzer instrument object.

ConnectSystem Object

[Automation Home (see page 3)] [Objects (see page 79)]

To Access

·

```
Dim variable As AgtLA.ConnectSystem
Set variable = CreateObject("AgtLA.ConnectSystem")
```

Methods

Method	Description
Connect (see page 129)	Connects to the remote logic analyzer system.
RecvFile (see page 174)	Copies a file from the remote logic analyzer system to your local system.
SendFile (see page 177)	Copies a file from your local system to the remote logic analyzer system.

Properties

There are no properties.

Exerciser Object

[Automation Home (see page 3)] [Objects (see page 79)]

To Access

```
Dim variable As AgtLA.Exerciser
Set variable = Module (see page 99)
```

Description

The Exerciser object represents a hardware measurement module for sending defined stimulus to a DUT.

Since this object is derived from the Module (see page 99) object, a Set must be used to get to these specific methods and properties. For example:

```
' You must create the Connect object (see page 90) and use it
' to access the Instrument object.  In this example, "myInst"
' represents the Instrument object.
'
'
' Get the Exerciser specific object.
Dim myExerciser As AgtLA.Exerciser
Set myExerciser = myInst.Modules(0)
```

Methods

The Exerciser object provides generic methods that accept command string parameters.

The object is currently used for sending D-PHY stimulus from the U4421A D-PHY Analyzer and Exerciser module to a D-PHY DUT.

For a detailed description of each of these methods and the XML format command string parameters that these methods accept, refer to the *Using COM Commands for U4421A Module* section in the *U4421A D-PHY Analyzer and Exerciser online help*.

Method	Description
ApplyToHardware(string cmd)	Applies the stimulus related settings loaded in the Setup dialog box to the stimulus hardware module.
ExecuteCommand(string cmd, out string result)	Loads the specified stimulus settings to the Setup dialog box of the hardware module in the Logic and Protocol Analyzer GUI. In addition, the command also immediately applies these settings to the hardware module even when the module is in the Running state. The command is useful for changing stimulus settings of the module at runtime and for inserting packets dynamically.
GetXmlFormat(string cmdKey, out string currentSettings)	Returns the XML command format for the given command name.
LoadExerciserParameters(string cmd, out string result)	Loads the stimulus related settings for the hardware module to the Setup dialog box of the module in the Logic and Protocol Analyzer GUI.
StartExerciser	Starts the transmission of stimulus from the hardware module to a DUT.
StopExerciser	Stops the transmission of stimulus from the hardware module to a DUT.

(Also Includes Module (see page 99) object methods)

FindResult Object

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 137)]

To Access

- Window.Find(Event) (see page 136)
- Window.FindNext() (see page 143)
- Window.FindPrev() (see page 143)

Methods

There are no methods.

Properties

Property	Description
Found (see page 201)	Gets the found status.
OccurrencesFound (see page 206)	Gets the number of occurrences found.
SubrowFound (see page 214)	Gets the subrow number if found on a subrow.
TimeFound (see page 216)	Gets the time found as a double.
TimeFoundString (see page 216)	Gets the time found as a string.

See Also

- Find (see page 136) method
- FindNext (see page 143) method
- FindPrev (see page 143) method
- GetDataByTime (see page 146) method

Frame Object

[Automation Home (see page 3)] [Objects (see page 79)]

To Access

- Frames.Item (see page 203) IndexOrName
- Frames IndexOrName

Methods There are no methods.

Properties

Property	Description
ComputerName (see page 196)	Gets the computer name of the logic analyzer frame.
Description (see page 200)	Gets a description of the logic analyzer frame.
IPAddress (see page 202)	Gets the frame's IP address(es).
TargetControlPort (see page 215)	Gets or sets the target control port value.

See Also

- Frames (see page 93) object

Frames Object

[Automation Home (see page 3)] [Objects (see page 79)]

To Access

- Instrument.Frames (see page 202)

Methods There are no methods.

Properties

Property	Description
_NewEnum (see page 219)	Used by Visual Basic to support the implementation of For Each ... Next.
Count (see page 196)	Gets the number of frames in the collection.
Item (see page 203)	Gets one of the frames in the collection given either an index, computer name, or IP address.

See Also

- Frames (see page 202) property

Instrument Object

[Automation Home (see page 3)] [Objects (see page 79)]

To Access

- When using Visual Basic outside of the *Keysight Logic Analyzer* application:

```
Dim variable As AgtLA.Instrument
Set variable = Connect.Instrument (see page 202) HostNameOrIpAddress
```

- When "using the Advanced Customization Environment (ACE)" (in the online help):

  AgtLA

Methods

Method	Description
Close (see page 128)	Closes the current configuration.
DeleteFile (see page 129)	Deletes a file on the instrument file system.
DoAction (see page 130)	Execute a specific XML-based command action.
DoCommands (see page 130)	Execute a particular XML-based command.
Export (see page 134)	Exports data to a file on the instrument file system.

Method	Description
ExportEx (see page 135)	Exports data to a file on the instrument file system.
GetProbeByName (see page 154)	Given a probe name, returns its corresponding probe object.
GetModuleByName (see page 149)	Given a module name, returns its corresponding hardware module object.
GetToolByName (see page 159)	Given a tool name, returns its corresponding tool object.
GetWindowByName (see page 160)	Given a window name, returns its corresponding window object.
Import (see page 165)	Imports data from a file located on the instrument file system.
ImportEx (see page 166)	Imports data from a file located on the instrument file system into a particular module.
GoOffline (see page 160)	Disconnects the user interface from the logic analyzer frame.
GoOnline (see page 164)	Connects the user interface to a specific logic analyzer frame.
IsOnline (see page 166)	Tells whether the user interface is connected to a logic analyzer frame.
New (see page 167)	Creates a new instrument Overview.
Open (see page 168)	Loads a previously saved configuration file on the instrument file system.
PanelLock (see page 168)	Disables user access to the instrument front panel or remote display.
PanelUnlock (see page 171)	Re-enables user access to the instrument front panel or remote display.
QueryCommand (see page 171)	Query for XML-based commands.
Run (see page 176)	Starts running all modules.
Save (see page 176)	Saves the current configuration to a file on the instrument file system.
Stop (see page 179)	Stops all currently running modules.
WaitComplete (see page 180)	Waits until all module, tool, and viewer measurements are complete.

Properties

Property	Description
Frames (see page 202)	Gets a collection of logic analyzer frames connected via the multiframe connector.
Markers (see page 204)	Gets a collection of all the markers in the instrument.
Model (see page 204)	Gets the model number.
Modules (see page 205)	Gets a collection of all the hardware modules in the instrument.
Overview (see page 207)	Gets the XML-format Overview window specification.
PanelLocked (see page 207)	Indicates the front panel is locked. If locked, the message is returned.
Probes (see page 208)	Gets a collection of all currently defined probes.
RemoteComputerName (see page 209)	Gets or sets the remote computer name.
RemoteUserName (see page 209)	Gets or sets the remote user login name.
SelfTest (see page 211)	Gets the SelfTest object.
Status (see page 213)	Gets the status of all hardware modules.
Tools (see page 216)	Gets a collection of all active software tools.

Property	Description
VBE (see page 218)	Gets the VBE extensibility object.
Version (see page 219)	Gets the version number of the system software.
Windows (see page 219)	Gets a collection of all active windows.

See Also

- Instrument (see page 202) property

Marker Object

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 96)]

To Access

- Markers.Item (see page 203) IndexOrName
- Markers IndexOrName

Methods

There are no methods.

Properties

Property	Description
Comments (see page 195)	Gets or sets the marker comments.
Name (see page 205)	Gets or sets the name of the marker.
TextColor (see page 215)	Gets or sets the marker text color.
BackgroundColor (see page 191)	Gets or sets the marker background color.
Position (see page 208)	Gets or sets the marker position.

See Also

- Markers (see page 95) object

Markers Object

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 96)]

To Access

- Instrument.Markers (see page 204)

Methods

Method	Description
Add (see page 128)	Adds a new marker to the collection using specific values.
AddXML (see page 128)	Adds multiple markers to the collection using an XML string.
Remove (see page 175)	Removes a marker from the collection.
RemoveAll (see page 175)	Removes all markers from the collection.
RemoveXML (see page 175)	Removes multiple markers from the collection using an XML string.

Properties

Property	Description
_NewEnum (see page 219)	Used by Visual Basic to support the implementation of **For Each ... Next**.
Count (see page 196)	Gets the number of markers in the collection.
Item (see page 203)	Gets one of the markers in the collection given either an index or name.

Remarks Window specific markers such as "Beginning of Data", "End of Data", and "Trigger" and markers with position other than by time are not part of the collection.

See Also

- Markers (see page 204) property

Markers Example

Visual Basic

```
' You must create the Connect object (see page 90) and use it
' to access the Instrument object.  In this example, "myInst"
' represents the Instrument object.
'
' Add a marker and set its position.
myInst.Markers.Add "Loc4", , , 0.00000001

' Change a marker's position and color.
Dim myMarker As AgtLA.Marker
Set myMarker = myInst.Markers("Loc4")
myMarker.Position = 0.000000005
myMarker.BackgroundColor = &HFF00
myMarker.TextColor = &HFF

' Add multiple markers to the collection using an XML string.
myInst.Markers.AddXML "<Markers>" + _
  "<Marker Name='XML M1' Comments='My Marker' " + _
      "ForegroundColor='hff00ff' BackgroundColor='h00ffff' " + _
      "Position='10 ns' LockPosition='T' />" + _
  "<Marker Name='XML M2' ForegroundColor='hff00ff' " + _
      "BackgroundColor='h00ffff' Position='15 ns' " + _
      "LockPosition='F' />" + _
  "<Marker Name='XML M3' BackgroundColor='h00ffff' " + _
      "Position='20 ns' LockPosition='T' />" + _
  "<Marker Name='XML M4' Position='25 ns' LockPosition='F' />" + _
  "<Marker Name='XML M5' LockPosition='T' />" + _
  "<Marker Name='XML M6' />" + _
"</Markers>"

' Display all of the markers.
Dim myMarkerNames As String
For Each myMarker In myInst.Markers
   ' Add the marker name to the string.
   myMarkerNames = myMarkerNames + vbNewLine + myMarker.Name
Next
MsgBox "Marker names: " + myMarkerNames

' Remove a marker from the collection.
myInst.Markers.Remove "Loc4"

' Remove multiple markers from the collection using an XML string.
myInst.Markers.RemoveXML "<Markers>" + _
  "<Marker Name='XML M1' />" + _
  "<Marker Name='XML M3' />" + _
  "<Marker Name='XML M5' />" + _
"</Markers>"

' Remove all markers from the collection.
myInst.Markers.RemoveAll
```

Visual C++

```
//
//  This simple Visual C++ Console application demonstrates how to use
//  the Keysight 168x/169x/169xx COM interface to set up markers.
//
//  This project was created in Visual C++ Developer.  To create a
//  similar project:
//
//    - Execute File -> New
//    - Select the Projects tab
//    - Select "Win32 Console Application"
//    - Select A "hello,World!" application (Visual Studio 6.0)
//
//  To make this buildable, you need to specify your "import" path
//  in stdafx.h (search for "TODO" in that file).  For example, add:
//  #import "C:/Program Files/Keysight Technologies/Logic Analyzer/LA \
//  COM Automation/agClientSvr.dll"
//
//  To run, you need to specify the host logic analyzer to connect
//  to (search for "TODO" below).
//

#include "stdafx.h"


////////////////////////////////////////////////////////////////////////
//
//  Forward declarations.
//
void DisplayError(_com_error& err);


////////////////////////////////////////////////////////////////////////
//
//  main() entry point.
//
int main(int argc, char* argv[])
{
   printf("*** Main()\n");

   //
   //  Initialize the Microsoft COM/ActiveX library.
   //
   HRESULT hr = CoInitialize(0);

   if (SUCCEEDED(hr))
   {
      try {   // Catch any unexpected run-time errors.
         _bstr_t hostname = "mtx33";   // TODO, use your logic
                                       // analysis system hostname.
         printf("Connecting to instrument '%s'\n", (char*) hostname);

         // Create the connect object and get the instrument object.
         AgtLA::IConnectPtr pConnect =
               AgtLA::IConnectPtr(__uuidof(AgtLA::Connect));
         AgtLA::IInstrumentPtr pInst =
               pConnect->GetInstrument(hostname);
```

```
   // Run the measurement, wait for it to complete.
   pInst->Run(FALSE);
   pInst->WaitComplete(20);

   // Get the markers object.
   AgtLA::IMarkersPtr pMarkers = pInst->GetMarkers();

   // Add a marker and set its position.
   pMarkers->Add("Loc4", 0x000000, 0xffff00, 0.00000001);

   // Change a marker's position and color.
   AgtLA::IMarkerPtr pMarker = pMarkers->GetItem("Loc4");
   pMarker->PutPosition(0.000000005);
   pMarker->PutBackgroundColor(0xFF00);
   pMarker->PutTextColor(0xFF);

   // Add multiple markers to the collection using an XML string.
   _bstr_t myAddMarkers = "<Markers> \
         <Marker Name='XML M1' Comments='My Marker' \
            ForegroundColor='hff00ff' BackgroundColor='h00ffff' \
            Position='10 ns' LockPosition='T' /> \
         <Marker Name='XML M2' ForegroundColor='hff00ff' \
            BackgroundColor='h00ffff' Position='15 ns' \
            LockPosition='F' /> \
         <Marker Name='XML M3' BackgroundColor='h00ffff' \
            Position='20 ns' LockPosition='T' /> \
         <Marker Name='XML M4' Position='25 ns' \
            LockPosition='F' /> \
         <Marker Name='XML M5' LockPosition='T' /> \
         <Marker Name='XML M6' /> \
         </Markers>";
   pMarkers->AddXML(myAddMarkers);

   // Display all of the markers.
   for (long i = 0; i < pMarkers->GetCount(); i++)
   {
      pMarker = pMarkers->GetItem(i);
      _bstr_t name = pMarker->GetName();
      printf("Marker name = %s\n", (char*) name);
   }

   // Remove a marker from the collection.
   pMarkers->Remove("Loc4");
   // Remove multiple markers from the collection using an XML
   // string.
   _bstr_t myRemoveMarkers = "<Markers> \
         <Marker Name='XML M1' /> \
         <Marker Name='XML M3' /> \
         <Marker Name='XML M5' /> \
         </Markers>";
   pMarkers->RemoveXML(myRemoveMarkers);

   // Remove all markers from the collection.
   pMarkers->RemoveAll();
}
catch (_com_error& e) {
   DisplayError(e);
```

```
      }

      // Uninitialize the Microsoft COM/ActiveX library.
      CoUninitialize();
   }
   else
   {
      printf("CoInitialize failed\n");
   }

   return 0;
}


////////////////////////////////////////////////////////////////////
//
//  Displays the last error -- used to show the last exception
//  information.
//
void DisplayError(_com_error& error)
{
   printf("*** DisplayError()\n");

   printf("Fatal Unexpected Error:\n");
   printf("  Error Number = %08lx\n", error.Error());

   static char errorStr[1024];
   _bstr_t desc = error.Description();

   if (desc.length() == 0)
   {
      // Don't have a description string.
      strcpy(errorStr, error.ErrorMessage());
      int nLen = lstrlen(errorStr);

      // Remove funny carriage return ctrl<M>.
      if (nLen > 2 && (errorStr[nLen - 2] == 0xd))
      {
         errorStr[nLen - 2] = '\0';
      }
   }
   else
   {
      strcpy(errorStr, desc);
   }

   printf("  Error Message = %s\n", (char*) errorStr);
}
```

Module Object

[Automation Home (see page 3)] [Objects (see page 79)]

To Access

- Instrument.GetModuleByName(Name) (see page 149)
- Modules.Item (see page 203) IndexOrName
- Modules IndexOrName

Methods

Method	Description
DoAction (see page 130)	Execute a specific XML-based command action.
DoCommands (see page 130)	Execute a particular XML-based command.
QueryCommand (see page 171)	Query for XML-based commands.
WaitComplete (see page 180)	Waits until the module's measurement completes.

Properties

Property	Description
BusSignals (see page 194)	Gets a collection of the module's defined bus/signals.
Description (see page 200)	Gets a description of the module.
Frame (see page 201)	Gets the frame in which the module resides.
Model (see page 204)	Gets the model number.
Name (see page 205)	Gets or sets the name of the module.
RunningStatus (see page 210)	Gets the detailed running status of the module.
Slot (see page 212)	Gets the module's slot location in the frame.
Status (see page 213)	Gets the status of the module.
StatusMsg (see page 213)	Gets the formatted status message.
Type (see page 217)	Gets the specific module type.

See Also

- Modules (see page 100) object

Modules Object

[Automation Home (see page 3)] [Objects (see page 79)]

To Access

- Instrument.Modules (see page 205)

Methods

There are no methods.

Properties

Property	Description
_NewEnum (see page 219)	Used by Visual Basic to support the implementation of **For Each ... Next**.
Count (see page 196)	Gets the number of modules in the collection.
Item (see page 203)	Gets one of the modules in a collection given either a slot, index, or name.

See Also

- Modules (see page 205) property
- Module (see page 99) object
- GetModuleByName (see page 149) method

Probe Object

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 101)]

To Access

- Instrument.GetProbeByName(Name) (see page 154)

- Probes.Item (see page 203) IndexOrName
- Probes IndexOrName

Methods

Method	Description
DoAction (see page 130)	Execute a specific XML-based command action.
DoCommands (see page 130)	Execute a particular XML-based command.
QueryCommand (see page 171)	Query for XML-based commands.

Properties

Property	Description
Name (see page 205)	Gets or sets the name of the probe.
Type (see page 217)	Gets the probe's type.

See Also

- Probes (see page 101) object

Probes Object

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 101)]

To Access

- Instrument.Probes (see page 208)

Methods

There are no methods.

Properties

Property	Description
_NewEnum (see page 219)	Used by Visual Basic to support the implementation of **For Each ... Next**.
Count (see page 196)	Gets the number of probes in the collection.
Item (see page 203)	Gets one of the probes in the collection given either an index or name.

See Also

- Probes (see page 208) property
- Probe (see page 100) object
- GetProbeByName (see page 154) method

Probes Example

Visual Basic

```
'You must create the Connect object (see page 90) and use it
' to access the Instrument object.  In this example, "myInst"
' represents the Instrument object.
'
' Display all of the probe names.
Dim myProbeNames As String
Dim myProbe As AgtLA.Probe

For Each myProbe in myInst.Probes
   ' Add the probe's name to the string.
   myProbeNames = myProbeNames + vbNewLine + myProbe.Name
Next
MsgBox "Probe names: " + myProbeNames
```

Visual C++

```
//
//  This simple Visual C++ Console application demonstrates how to use
//  the Keysight 168x/169x/169xx COM interface to display all the probe
//  names.
//
//  This project was created in Visual C++ Developer.  To create a
//  similar project:
//
//    - Execute File -> New
//    - Select the Projects tab
//    - Select "Win32 Console Application"
//    - Select A "hello,World!" application (Visual Studio 6.0)
//
//  To make this buildable, you need to specify your "import" path
//  in stdafx.h (search for "TODO" in that file).  For example, add:
//  #import "C:/Program Files/Keysight Technologies/Logic Analyzer/LA \
//  COM Automation/agClientSvr.dll"
//
//  To run, you need to specify the host logic analyzer to connect
//  to (search for "TODO" below).
//

#include "stdafx.h"


////////////////////////////////////////////////////////////////////////
//
//  Forward declarations.
//
void DisplayError(_com_error& err);


////////////////////////////////////////////////////////////////////////
//
//  main() entry point.
//
int main(int argc, char* argv[])
{
   printf("*** Main()\n");

   //
   //  Initialize the Microsoft COM/ActiveX library.
   //
   HRESULT hr = CoInitialize(0);

   if (SUCCEEDED(hr))
   {
      try {   // Catch any unexpected run-time errors.
         _bstr_t hostname = "mtx33";   // TODO, use your logic
                                       // analysis system hostname.
         printf("Connecting to instrument '%s'\n", (char*) hostname);

         // Create the connect object and get the instrument object.
         AgtLA::IConnectPtr pConnect =
               AgtLA::IConnectPtr(__uuidof(AgtLA::Connect));
         AgtLA::IInstrumentPtr pInst =
               pConnect->GetInstrument(hostname);
```

```
         // Load the configuration file.
         _bstr_t configFile = "C:\\LA\\Configs\\probes.ala";
         printf("Loading the config file '%s'\n", (char*) configFile);
         pInst->Open(configFile, FALSE, "", TRUE);

         // Display all of the probe names.
         AgtLA::IProbesPtr pProbes = pInst->GetProbes();
         for (long i = 0; i < pProbes->GetCount(); i++)
         {
            AgtLA::IProbePtr pProbe = pProbes->GetItem(i);
            _bstr_t name = pProbe->GetName();
            printf("Probe name = %s\n", (char*) name);
         }
      }
      catch (_com_error& e) {
         DisplayError(e);
      }

      // Uninitialize the Microsoft COM/ActiveX library.
      CoUninitialize();
   }
   else
   {
      printf("CoInitialize failed\n");
   }

   return 0;
}


//////////////////////////////////////////////////////////////////////
//
//  Displays the last error -- used to show the last exception
//  information.
//
void DisplayError(_com_error& error)
{
   printf("*** DisplayError()\n");

   printf("Fatal Unexpected Error:\n");
   printf("  Error Number = %08lx\n", error.Error());

   static char errorStr[1024];
   _bstr_t desc = error.Description();

   if (desc.length() == 0)
   {
      // Don't have a description string.
      strcpy(errorStr, error.ErrorMessage());
      int nLen = lstrlen(errorStr);

      // Remove funny carriage return ctrl<M>.
      if (nLen > 2 && (errorStr[nLen - 2] == 0xd))
      {
         errorStr[nLen - 2] = '\0';
      }
```

```
    }
    else
    {
       strcpy(errorStr, desc);
    }

    printf("  Error Message = %s\n", (char*) errorStr);
}
```

ProtocolWindow Object

[Automation Home (see page 3)] [Objects (see page 79)]

To Access

```
Dim variable As AgtLA.ProtocolWindow
Set variable = Window (see page 122)
```

Description

The ProtocolWindow object represents an output window for viewing protocol data from a serial analyzer.

Methods

Method	Description
GetProtocolDataFields (see page 154)	Gets the acquisition data for the fields displayed in the Protocol Viewer.
WriteProtocolDataFieldsToFile (see page 188)	Writes the acquisition data displayed in various fields in the Protocol Viewer to a specified CSV file.
GetTriggerSampleNumber (see page 159)	Gets the packet number and channel of the trigger packet.
FindImages (see page 140)	Searches acquisition data for frames containing images and puts them into the Protocol Viewer's list of images.
GetImageList (see page 147)	Returns a string containing the Protocol Viewer's list of images. Optionally, it writes this list to a text file.
GetPacketCount (see page 152)	Given a channel name, returns the number of packets for that particular channel.
WriteAllImagesToFiles (see page 184)	Writes all the images in the Protocol Viewer's list to separate bitmap files in a specified folder.
WriteImageToFile (see page 186)	Writes a single image to a bitmap file. The image is selected using the 'Frame Number' string from the image list.

(Also includes Window object's methods)

SampleBusSignalData Object

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 105)]

To Access

```
Dim variable As AgtLA.SampleBusSignalData
Set variable = BusSignalData (see page 81)
```

Description

The SampleBusSignalData object represents the data associated with a bus/signal. The data can be uploaded using the methods GetDataBySample (see page 144) and GetDataByTime (see page 146).

Since this object is derived from the BusSignalData (see page 81) object, a Set must be used to get to these specific methods and properties. For example:

```
' You must create the Connect object (see page 90) and use it
' to access the Instrument object.  In this example, "myInst"
' represents the Instrument object.
'
'' Get the AnalyzerModule specific object.
Dim myData As AgtLA.SampleBusSignalData
Set myData = myInst.Modules(0).BusSignals(0).BusSignalData
MsgBox "Sample Range: " + myData.StartSample + ".." + myData.EndSample
```

Methods

Method	Description
GetDataBySample (see page 144)	Given a sample range, returns an array of data.
GetDataByTime (see page 146)	Given a time range, returns an array of data.
GetNumSamples (see page 152)	Given a range, returns the number of samples stored.
GetSampleNumByTime (see page 158)	Gets the closest sample number corresponding to the time given.
GetTime (see page 158)	Given a range, returns the time for this bus/signal in the format specified by the data type given.

Properties

Property	Description
DataType (see page 199)	Gets the recommended bus/signal data type.
EndSample (see page 201)	Gets the data's ending sample number relative to trigger.
EndTime (see page 201)	Gets the data's ending time relative to trigger.
StartSample (see page 212)	Gets the data's starting sample number relative to trigger.
StartTime (see page 212)	Gets the data's starting time relative to trigger.

SampleBusSignalData Example

Visual Basic

```
'You must create the Connect object (see page 90) and use it
' to access the Instrument object.  In this example, "myInst"
' represents the Instrument object.
'
' Run the measurement, wait for it to complete.
myInst.Run
myInst.WaitComplete (20)

' Display all of the bus/signal data.
Dim myString As String
Dim printHeader As Boolean
Dim myBusSignals As AgtLA.BusSignals
Set myBusSignals = myInst.GetModuleByName("My 1690A-1").BusSignals
Dim myBusSignal As AgtLA.BusSignal
Dim myData As AgtLA.SampleBusSignalData

Dim myNumDataRows As Long
Dim myStartSample As Long
Dim myEndSample As Long
```

```
Dim i As Long

printHeader = True
myStartSample = -5   ' Sample range to upload.
myEndSample = 5

For Each myBusSignal In myBusSignals
   Set myData = myBusSignal.BusSignalData

   If printHeader = True Then
      myString = myString + "Sample range: " + _
            Str(myData.StartSample) + ".." + Str(myData.EndSample)
      myString = myString + ", Time range: " + _
            Str(myData.StartTime) + ".." + Str(myData.EndTime) + _
            vbNewLine
      printHeader = False
   End If

   ' Print the bus/signal information.
   myString = myString + vbNewLine + "Name: " + myBusSignal.Name
   myString = myString + ", BitSize=" + Str(myBusSignal.BitSize) + _
         ", ByteSize=" + Str(myBusSignal.ByteSize) + vbNewLine

   ' Print the bus/signal data.
   Select Case myBusSignal.BusSignalType
      Case AgtBusSignalSampleNum
         Dim lArray() As Long
         lArray = myData.GetDataBySample(myStartSample, myEndSample, _
               AgtDataLong, myNumDataRows)
         For i = 0 To myNumDataRows - 1
            myString = myString + Str(lArray(i)) + " "
         Next i
      Case AgtBusSignalTime
         Dim dArray() As Double
         dArray = myData.GetDataBySample(myStartSample, myEndSample, _
               AgtDataTime, myNumDataRows)
         For i = 0 To myNumDataRows - 1
            myString = myString + Str(dArray(i)) + " "
         Next i
      Case AgtBusSignalGenerated
         ' Decimal holds a maximum of 96 bits unsigned.
         Dim vArray As Variant
         vArray = myData.GetDataBySample(myStartSample, myEndSample, _
               AgtDataDecimal, myNumDataRows)
         For i = 0 To myNumDataRows - 1
            myString = myString + Str(vArray(i)) + " "
         Next i
      Case AgtBusSignalProbed
         ' Use the data type that is appropriate for your bus width:
         '  - AgtDataLong holds a maximum of 31 bits unsigned.
         '  - AgtDataDouble holds a maximum of 52 bits unsigned.
         '  - AgtDataDecimal holds a maximum of 96 bits unsigned.
         lArray = myData.GetDataBySample(myStartSample, myEndSample, _
               AgtDataLong, myNumDataRows)
         For i = 0 To myNumDataRows - 1
            myString = myString + Hex$(lArray(i)) + " "
         Next i
```

```
   End Select
Next
MsgBox myString
```

Visual C++

```
//
//  This simple Visual C++ Console application demonstrates how to use
//  the Keysight 168x/169x/169xx COM interface to display captured data.
//
//  This project was created in Visual C++ Developer.  To create a
//  similar project:
//
//    - Execute File -> New
//    - Select the Projects tab
//    - Select "Win32 Console Application"
//    - Select A "hello,World!" application (Visual Studio 6.0)
//
//  To make this buildable, you need to specify your "import" path
//  in stdafx.h (search for "TODO" in that file).  For example, add:
//  #import "C:/Program Files/Keysight Technologies/Logic Analyzer/LA \
//  COM Automation/agClientSvr.dll"
//
//  To run, you need to specify the host logic analyzer to connect
//  to (search for "TODO" below).
//

#include "stdafx.h"


//////////////////////////////////////////////////////////////////////
//
//  Forward declarations.
//
void DisplayRawData(
  _bstr_t&     busSignalName,
  _variant_t&  rawDataArray,
  long         numBytesPerRow);

void DisplayBusSignalData(
  _bstr_t&     busSignalName,
  _variant_t&  busSignalArray);

void DisplayError(_com_error& err);


//////////////////////////////////////////////////////////////////////
//
//  main() entry point.
//
int main(int argc, char* argv[])
{
   printf("*** Main()\n");

   //
   //  Initialize the Microsoft COM/ActiveX library.
   //
   HRESULT hr = CoInitialize(0);
```

```
if (SUCCEEDED(hr))
{
   try {   // Catch any unexpected run-time errors.
      _bstr_t hostname = "mtx33";   // TODO, use your logic
                                    // analysis system hostname.
      printf("Connecting to instrument '%s'\n", (char*) hostname);

      // Create the connect object and get the instrument object.
      AgtLA::IConnectPtr pConnect =
            AgtLA::IConnectPtr(__uuidof(AgtLA::Connect));
      AgtLA::IInstrumentPtr pInst =
            pConnect->GetInstrument(hostname);

      // Load the configuration file.
      _bstr_t configFile = "C:\\LA\\Configs\\config.ala";
      printf("Loading the config file '%s'\n", (char*) configFile);
      pInst->Open(configFile, FALSE, "", TRUE);

      // Run the measurement, wait for it to complete.
      pInst->Run(FALSE);
      pInst->WaitComplete(20);

      // Get data from the Listing window.
      _bstr_t windowName = "Listing-1";
      AgtLA::IWindowPtr pWindow = pInst->GetWindowByName(windowName);

      // For each bus/signal, display a range of data.
      long start = -10;
      long end = 10;
      _variant_t data;
      long numRowsRet;
      long numBytesPerRow;
      AgtLA::IBusSignalsPtr pBusSignals = pWindow->GetBusSignals();
      printf("\n");
      for (long i = 0; i < pBusSignals->GetCount(); i++)
      {
         // Get the data for the bus/signal.
         AgtLA::IBusSignalPtr pBusSignal =
               pWindow->GetBusSignals()->GetItem(i);
         _bstr_t busSignal = pBusSignal->GetName();
         printf("Bus/signal: '%s'\n", (char*) busSignal);
         AgtLA::ISampleBusSignalDataPtr pSampleData =
               pBusSignal->GetBusSignalData();

         switch(pBusSignal->GetBusSignalType())
         {
         case AgtLA::AgtBusSignalProbed:
            {
               printf("  Type: 'AgtBusSignalProbed'\n");
               // "raw" and "long" formats supported.
               if (pBusSignal->GetBitSize() > 32) {
                  data = pSampleData->GetDataBySample(start, end,
                        AgtLA::AgtDataRaw, &numRowsRet);
                  numBytesPerRow = pBusSignal->GetByteSize();
                  printf("  Data type: 'AgtDataRaw' ");
                  printf("(%d bytes/row)\n", numBytesPerRow);
                  DisplayRawData(busSignal, data, numBytesPerRow);
```

```
                }
                else {
                   data = pSampleData->GetDataBySample(start, end,
                         AgtLA::AgtDataLong, &numRowsRet);
                   printf("  Data type: 'AgtDataLong'\n");
                   DisplayBusSignalData(busSignal, data);
                }
             }
             break;

          case AgtLA::AgtBusSignalGenerated:
             {
                printf("  Type: 'AgtBusSignalGenerated'\n");
                data = pSampleData->GetDataBySample(start, end,
                      AgtLA::AgtDataStringHex, &numRowsRet);
                printf("  Data type: 'AgtDataStringHex'\n");
                DisplayBusSignalData(busSignal, data);
             }
             break;

          case AgtLA::AgtBusSignalSampleNum:
             {
                printf("  Type: 'AgtBusSignalSampleNum'\n");
                data = pSampleData->GetDataBySample(start, end,
                      AgtLA::AgtDataLong, &numRowsRet);
                printf("  Data type: 'AgtDataLong'\n");
                DisplayBusSignalData(busSignal, data);
             }
             break;

          case AgtLA::AgtBusSignalTime:
             {
                printf("  Type: 'AgtBusSignalTime'\n");
                data = pSampleData->GetDataBySample(start, end,
                      AgtLA::AgtDataTime, &numRowsRet);
                printf("  Data type: 'AgtDataTime'\n");
                DisplayBusSignalData(busSignal, data);
             }

          default:
             break;

          }
       }
    }
    catch (_com_error& e) {
       DisplayError(e);
    }

    // Uninitialize the Microsoft COM/ActiveX library.
    CoUninitialize();
}
else
{
    printf("CoInitialize failed\n");
}
```

```
   return 0;
}


////////////////////////////////////////////////////////////////////
//
//  Displays the data in raw data format.
//
void DisplayRawData(_bstr_t& busSignalName,
                    _variant_t& varArray,
                    long numBytesPerRow)
{
   long numSamples;
   long lBound;
   HRESULT hr = SafeArrayGetLBound(varArray.parray, 1, &lBound);

   if (SUCCEEDED(hr))
   {
      long uBound;
      hr = SafeArrayGetUBound(varArray.parray, 1, &uBound);

      if (SUCCEEDED(hr))
      {
         printf("  Variant data format: VT_UI1 (unsigned char)\n");
         byte* pByteArray;
         hr = SafeArrayAccessData(varArray.parray,
               (void**) &pByteArray);

         if (SUCCEEDED(hr))
         {
            numSamples = (uBound - lBound + 1) / numBytesPerRow;
            byte* pByte = pByteArray;

            for (int i = 0; i < numSamples; i++)
            {
               printf("    dataArray[%d]: ", i);

               for (int j = 0; j < numBytesPerRow; j++)
               {
                  printf("%02x ", pByte[j]);
               }

               pByte += numBytesPerRow;
               printf("\n");
            }

            printf("\n");
            SafeArrayUnaccessData(varArray.parray);
         }
      }
   }
}


////////////////////////////////////////////////////////////////////
//
//  Displays bus/signal data in the given array.
```

```
//
void DisplayBusSignalData(_bstr_t& busSignalName,
                          _variant_t& varArray)
{
   signed   _int8*   pArrayInt8   = NULL;
   unsigned _int8*   pArrayUInt8  = NULL;
   signed   _int16*  pArrayInt16  = NULL;
   unsigned _int16*  pArrayUInt16 = NULL;
   signed   _int32*  pArrayInt32  = NULL;
   unsigned _int32*  pArrayUInt32 = NULL;
   signed   _int64*  pArrayInt64  = NULL;
   unsigned _int64*  pArrayUInt64 = NULL;
   double*           pArrayDouble = NULL;
   BSTR*             pArrayBstr   = NULL;

   SAFEARRAY* pDataArray = varArray.parray;

   long lBound;
   HRESULT hr = SafeArrayGetLBound(pDataArray, 1, &lBound);

   if (FAILED(hr))
   {
      printf("SafeArrayGetLBound failed\n");
   }

   long uBound;
   hr = SafeArrayGetUBound(pDataArray, 1, &uBound);

   if (FAILED(hr))
   {
      printf("SafeArrayGetUBound failed\n");
   }

   long numSamples = uBound - lBound + 1;

   switch (varArray.vt - VT_ARRAY)
   {
   case VT_I1:
      {
         printf("  Variant data format: VT_I1 (char)\n");
         hr = SafeArrayAccessData(pDataArray, (void**) &pArrayInt8);

         for (int i = lBound; i <= uBound; i++)
         {
            printf("    dataArray[%d]: %02cx\n", i, pArrayInt8[i]);
         }
         break;
      }

   case VT_UI1:
      {
         printf("  Variant data format: VT_UI1 (unsigned char)\n");
         hr = SafeArrayAccessData(pDataArray, (void**) &pArrayUInt8);

         for (int i = lBound; i <= uBound; i++)
         {
            printf("    dataArray[%d]: %02cx\n", i, pArrayUInt8[i]);
```

```
      }
      break;
   }

case VT_I2:
   {
      printf("  Variant data format: VT_I2 (short)\n");
      hr = SafeArrayAccessData(pDataArray, (void**) &pArrayInt16);

      for (int i = lBound; i <= uBound; i++)
      {
         printf("    dataArray[%d]: %04hx\n", i, pArrayInt16[i]);
      }
      break;
   }

case VT_UI2:
   {
      printf("  Variant data format: VT_UI2 (unsigned short)\n");
      hr = SafeArrayAccessData(pDataArray, (void**) &pArrayUInt16);

      for (int i = lBound; i <= uBound; i++)
      {
         printf("    dataArray[%d]: %04hx\n", i, pArrayUInt16[i]);
      }
      break;
   }

case VT_I4:
   {
      printf("  Variant data format: VT_I4 (long)\n");
      hr = SafeArrayAccessData(pDataArray, (void**) &pArrayInt32);

      for (int i = lBound; i <= uBound; i++)
      {
         printf("    dataArray[%d]: %08lx\n", i, pArrayInt32[i]);
      }
      break;
   }

case VT_UI4:
   {
      printf("  Variant data format: VT_UI4 (unsigned long)\n");
      hr = SafeArrayAccessData(pDataArray, (void**) &pArrayUInt32);

      for (int i = lBound; i <= uBound; i++)
      {
         printf("    dataArray[%d]: %08lx\n", i, pArrayUInt32[i]);
      }
      break;
   }

case VT_I8:
   {
      printf("  Variant data format: VT_I8 (__int64)\n");
      hr = SafeArrayAccessData(pDataArray, (void**) &pArrayInt64);
```

```
            for (int i = lBound; i <= uBound; i++)
            {
               printf("    dataArray[%d]: %016I64x\n", i, pArrayInt64[i]);
            }
            break;
         }

      case VT_UI8:
         {
            printf("  Variant data format: VT_UI8 (unsigned __int64)\n");
            hr = SafeArrayAccessData(pDataArray, (void**) &pArrayUInt64);

            for (int i = lBound; i <= uBound; i++)
            {
               printf("    dataArray[%d]: %016I64x\n", i, pArrayUInt64[i]);
            }
            break;
         }

      case VT_R8:
         {
            printf("  Variant data format: VT_R8 (double)\n");
            hr = SafeArrayAccessData(pDataArray, (void**) &pArrayDouble);

            for (int i = lBound; i <= uBound; i++)
            {
               printf("    dataArray[%d]: %0le\n", i, pArrayDouble[i]);
            }
            break;
         }

      case VT_BSTR:
         {
            printf("  Variant data format: VT_BSTR (_bstr_t)\n");
            hr = SafeArrayAccessData(pDataArray, (void**) &pArrayBstr);

            for (int i = lBound; i <= uBound; i++)
            {
               printf("    dataArray[%d]: %S\n", i, pArrayBstr[i]);
            }
            break;
         }

      default:
         {
            printf("  Variant data format: unknown\n");
            hr = E_FAIL;
            break;
         }
   }

   if (FAILED(hr))
   {
      printf("SafeArrayAccessData failed\n");
   }

   printf("\n");
```

```
  SafeArrayUnaccessData(pDataArray);
}


////////////////////////////////////////////////////////////////////
//
//  Displays the last error -- used to show the last exception
//  information.
//
void DisplayError(_com_error& error)
{
   printf("*** DisplayError()\n");

   printf("Fatal Unexpected Error:\n");
   printf("  Error Number = %08lx\n", error.Error());

   static char errorStr[1024];
   _bstr_t desc = error.Description();

   if (desc.length() == 0)
   {
      // Don't have a description string.
      strcpy(errorStr, error.ErrorMessage());
      int nLen = lstrlen(errorStr);

      // Remove funny carriage return ctrl<M>.
      if (nLen > 2 && (errorStr[nLen - 2] == 0xd))
      {
         errorStr[nLen - 2] = '\0';
      }
   }
   else
   {
      strcpy(errorStr, desc);
   }

   printf("  Error Message = %s\n", (char*) errorStr);
}
```

SampleDifference Object

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 82)]

To Access

- SampleDifferences.Item (see page 203) IndexOrName
- SampleDifferences IndexOrName

Methods

There are no methods.

Properties

Property	Description
BusSignalDifferences (see page 193)	Gets a collection of all the buses/signals with differences for this sample.
SampleNum (see page 211)	Gets the sample number at which differences occurred.

See Also

- SampleDifferences (see page 115) object

SampleDifferences Object

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 82)]

To Access

- CompareWindow.SampleDifferences (see page 210)

Methods

There are no methods.

Properties

Property	Description
_NewEnum (see page 219)	Used by Visual Basic to support the implementation of **For Each ... Next**.
Count (see page 196)	Gets the number of bus/signal differences in the collection.
Item (see page 203)	Given an index into the collection, gets a SampleDifference (see page 114) object from the collection.

See Also

- SampleDifferences (see page 210) property

SelfTest Object

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 115)]

To Access

- Instrument.SelfTest (see page 211)

Methods

Method	Description
TestAll (see page 180)	Runs an instrument's self-tests.

Properties

There are no properties.

See Also

- SelfTest (see page 211) property

SelfTest Example

The following script runs all of the tests available on the target instrument:

Visual Basic

```
Dim result As String
result = theInstrument.SelfTest.TestAll()
```

Visual C++

```
//
//  This simple Visual C++ Console application demonstrates how to
//  use the Keysight 168x/169x/169xx COM interface to run the logic
//  analysis system's self tests.
//
//  This project was created in Visual C++ Developer.  To create a
//  similar project:
//
//    - Execute File -> New
//    - Select the Projects tab
//    - Select "Win32 Console Application"
//    - Select A "hello,World!" application (Visual Studio 6.0)
//
//  To make this buildable, you need to specify your "import" path
//  in stdafx.h (search for "TODO" in that file).  For example, add:
//  #import "C:/Program Files/Keysight Technologies/Logic Analyzer/LA \
//  COM Automation/agClientSvr.dll"
```

```
//
//  To run, you need to specify the host logic analyzer to connect
//  to (search for "TODO" below).
//

#include "stdafx.h"


////////////////////////////////////////////////////////////////////////
//
//  Forward declarations.
//
void DisplayError(_com_error& err);


////////////////////////////////////////////////////////////////////////
//
//  main() entry point.
//
int main(int argc, char* argv[])
{
   printf("*** Main()\n");

   //
   //  Initialize the Microsoft COM/ActiveX library.
   //
   HRESULT hr = CoInitialize(0);

   if (SUCCEEDED(hr))
   {
      try {   // Catch any unexpected run-time errors.
         _bstr_t hostname = "col-mil20";   // TODO, use your logic
                                           // analysis system hostname.
         printf("Connecting to instrument '%s'\n", (char*) hostname);

         // Create the connect object and get the instrument object.
         AgtLA::IConnectPtr pConnect =
               AgtLA::IConnectPtr(__uuidof(AgtLA::Connect));
         AgtLA::IInstrumentPtr pInst =
               pConnect->GetInstrument(hostname);


         // Run the logic analysis system self tests and print the
         // results.
         AgtLA::ISelfTestPtr pSelfTest = pInst->GetSelfTest();
         _bstr_t testResults = pSelfTest->TestAll();
         printf("Self test results: %s\n", (char*) testResults);
      }
      catch (_com_error& e) {
         DisplayError(e);
      }

      // Uninitialize the Microsoft COM/ActiveX library.
      CoUninitialize();
   }
   else
   {
```

```
      printf("CoInitialize failed\n");
   }

   return 0;
}


////////////////////////////////////////////////////////////////////
//
//  Displays the last error -- used to show the last exception
//  information.
//
void DisplayError(_com_error& error)
{
   printf("*** DisplayError()\n");

   printf("Fatal Unexpected Error:\n");
   printf("  Error Number = %08lx\n", error.Error());

   static char errorStr[1024];
   _bstr_t desc = error.Description();

   if (desc.length() == 0)
   {
      // Don't have a description string.
      strcpy(errorStr, error.ErrorMessage());
      int nLen = lstrlen(errorStr);

      // Remove funny carriage return ctrl<M>.
      if (nLen > 2 && (errorStr[nLen - 2] == 0xd))
      {
         errorStr[nLen - 2] = '\0';
      }
   }
   else
   {
      strcpy(errorStr, desc);
   }

   printf("  Error Message = %s\n", (char*) errorStr);
}
```

SerialModule Object

[Automation Home (see page 3)] [Objects (see page 79)]

To Access

```
Dim variable As AgtLA.SerialModule
Set variable = Module (see page 99)
```

Description

The SerialModule object represents a hardware measurement module for viewing and analyzing serial data.

Since this object is derived from the Module (see page 99) object, a Set must be used to get to these specific methods and properties. For example:

```
'You must create the Connect object (see page 90) and use it
' to access the Instrument object.  In this example, "myInst"
' represents the Instrument object.
'
' Get the SerialModule specific object.
Dim myAnalyzer As AgtLA.SerialModule
Set myAnalyzer = myInst.Modules(0)
MsgBox "Trigger: " + myAnalyzer.Trigger
```

Methods

Method	Description
RecallTriggerByFile (see page 173)	Loads a previously saved trigger file located on the instrument file system.

(Also Includes Module (see page 99) object methods)

Tool Object

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 119)]

To Access

- Instrument.GetToolByName(Name) (see page 159)
- Tools.Item (see page 203) IndexOrName
- Tools IndexOrName

Methods

Method	Description
DoAction (see page 130)	Execute a specific XML-based command action.
DoCommands (see page 130)	Execute a particular XML-based command.
QueryCommand (see page 171)	Query for XML-based commands.

Properties

Property	Description
BusSignals (see page 194)	Gets a collection of the tool's defined bus/signals.
Name (see page 205)	Gets or sets the name of the tool.
Type (see page 217)	Gets the specific tool type.

See Also

- Tools (see page 118) object

Tools Object

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 119)]

To Access

- Instrument.Tools (see page 216)

Methods

There are no methods.

Properties

Property	Description
_NewEnum (see page 219)	Used by Visual Basic to support the implementation of **For Each ... Next**.
Count (see page 196)	Gets the number of tools in the collection.
Item (see page 203)	Gets one of the tools in the collection given either an index or name.

See Also

- Tools (see page 216) property
- Tool (see page 118) object
- GetToolByName (see page 159) method

Tools Example

Visual Basic

```
'You must create the Connect object (see page 90) and use it
' to access the Instrument object.  In this example, "myInst"
' represents the Instrument object.
'
' Display all of the tool names.
Dim myToolNames As String
Dim myTool As AgtLA.Tool

For Each myTool in myInst.Tools
   ' Add the tool's name to the string.
   myToolNames = myToolNames + vbNewLine + myTool.Name
Next
MsgBox "Tool names: " + myToolNames

' Get the MPC8XX Inverse Assembler tool; then,
' execute the QueryCommand.
Dim XMLCmdResponse As String
If myInst.GetToolByName("Motorola PowerQUICC (MPC8XX) Inverse " + _
      "Assembler-1").QueryCommand("GetProperties", XMLCmdResponse) _
      Then
   MsgBox "MPC8XX IA Properties: " + XMLCmdResponse
End If
```

Visual C++

```
//
//  This simple Visual C++ Console application demonstrates how to use
//  the Keysight 168x/169x/169xx COM interface to display all the tool
//  names and, with a specific tool, execute a QueryCommand to get the
//  tool's properties.
//
//  This project was created in Visual C++ Developer.  To create a
//  similar project:
//
//    - Execute File -> New
//    - Select the Projects tab
//    - Select "Win32 Console Application"
//    - Select A "hello,World!" application (Visual Studio 6.0)
//
//  To make this buildable, you need to specify your "import" path
//  in stdafx.h (search for "TODO" in that file).  For example, add:
//  #import "C:/Program Files/Keysight Technologies/Logic Analyzer/LA \
//  COM Automation/agClientSvr.dll"
//
```

```
//  To run, you need to specify the host logic analyzer to connect
//  to (search for "TODO" below).
//

#include "stdafx.h"


////////////////////////////////////////////////////////////////////
//
//  Forward declarations.
//
void DisplayError(_com_error& err);


////////////////////////////////////////////////////////////////////
//
//  main() entry point.
//
int main(int argc, char* argv[])
{
   printf("*** Main()\n");

   //
   //  Initialize the Microsoft COM/ActiveX library.
   //
   HRESULT hr = CoInitialize(0);

   if (SUCCEEDED(hr))
   {
      try {   // Catch any unexpected run-time errors.
         _bstr_t hostname = "mtx33";   // TODO, use your logic
                                       // analysis system hostname.
         printf("Connecting to instrument '%s'\n", (char*) hostname);

         // Create the connect object and get the instrument object.
         AgtLA::IConnectPtr pConnect =
               AgtLA::IConnectPtr(__uuidof(AgtLA::Connect));
         AgtLA::IInstrumentPtr pInst =
               pConnect->GetInstrument(hostname);

         // Load the configuration file.
         _bstr_t configFile = "C:\\LA\\Configs\\config.ala";
         printf("Loading the config file '%s'\n", (char*) configFile);
         pInst->Open(configFile, FALSE, "", TRUE);

         // Display all of the tool names.
         AgtLA::IToolsPtr pTools = pInst->GetTools();
         for (long i = 0; i < pTools->GetCount(); i++)
         {
            AgtLA::IToolPtr pTool = pTools->GetItem(i);
            _bstr_t name = pTool->GetName();
            printf("Tool name = %s\n", (char*) name);
         }

         // Get the MPC8XX Inverse Assembler tool; then,
         // execute the QueryCommand.
         _bstr_t myTool =
```

```
                "Motorola PowerQUICC (MPC8XX) Inverse Assembler-1";
            AgtLA::IToolPtr pTool = pInst->GetToolByName(myTool);
            BSTR cmdResponseXML;
            if (pTool->QueryCommand("GetProperties", &cmdResponseXML)) {
               printf("MPC8XX IA Properties '%S'\n",
                     (char*) cmdResponseXML);
            }
            SysFreeString(cmdResponseXML);
         }
         catch (_com_error& e) {
            DisplayError(e);
         }

         // Uninitialize the Microsoft COM/ActiveX library.
         CoUninitialize();
      }
      else
      {
         printf("CoInitialize failed\n");
      }

      return 0;
}


//////////////////////////////////////////////////////////////////////
//
//  Displays the last error -- used to show the last exception
//  information.
//
void DisplayError(_com_error& error)
{
   printf("*** DisplayError()\n");

   printf("Fatal Unexpected Error:\n");
   printf("  Error Number = %08lx\n", error.Error());

   static char errorStr[1024];
   _bstr_t desc = error.Description();

   if (desc.length() == 0)
   {
      // Don't have a description string.
      strcpy(errorStr, error.ErrorMessage());
      int nLen = lstrlen(errorStr);

      // Remove funny carriage return ctrl<M>.
      if (nLen > 2 && (errorStr[nLen - 2] == 0xd))
      {
         errorStr[nLen - 2] = '\0';
      }
   }
   else
   {
      strcpy(errorStr, desc);
   }
```

```
    printf("  Error Message = %s\n", (char*) errorStr);
}
```

Window Object

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 123)]

To Access

- Instrument.GetWindowByName(Name) (see page 160)
- Windows.Item (see page 203) IndexOrName
- Windows IndexOrName

Methods

Method	Description
DoAction (see page 130)	Execute a specific XML-based command action.
DoCommands (see page 130)	Execute a particular XML-based command.
Find (see page 136)	Finds a specified data event with optional occurrence and time duration.
FindNext (see page 143)	Finds the next event by searching forward from the last event found using the event specified by the last call to Find (see page 136).
FindPrev (see page 143)	Finds the previous event by searching backward from the last event found using the event specified by the last call to Find (see page 136).
GoToPosition (see page 165)	Moves the center of the window to a new position.
QueryCommand (see page 171)	Query for XML-based commands.

Properties

Property	Description
BusSignals (see page 194)	Gets a collection of the window's defined bus/signals.
Name (see page 205)	Gets or sets the name of the window.
Type (see page 217)	Gets the window's type.

See Also

- Windows (see page 122) object

Windows Object

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 123)]

To Access

- Instrument.Windows (see page 219)

Methods

There are no methods.

Properties

Property	Description
_NewEnum (see page 219)	Used by Visual Basic to support the implementation of **For Each ... Next**.
Count (see page 196)	Gets the number of windows in the collection.
Item (see page 203)	Gets one of the windows in the collection given either an index or name.

See Also

- Windows (see page 219) property
- Window (see page 122) object

- GetWindowByName (see page 160) method

Windows Example

Visual Basic

```
'You must create the Connect object (see page 90) and use it
' to access the Instrument object.  In this example, "myInst"
' represents the Instrument object.
'

' Display all of the window names.
Dim myWindowNames As String
Dim myWindow As AgtLA.Window

For Each myWindow in myInst.Windows
   ' Add the window's name to the string.
   myWindowNames = myWindowNames + vbNewLine + myWindow.Name
Next
MsgBox "Window names: " + myWindowNames

' Get the compare window using the Windows default
' Item method, then execute the compare.
myInst.Windows("Compare-1").Execute
```

Visual C++

```
//
//  This simple Visual C++ Console application demonstrates how to
//  use the Keysight 168x/169x/169xx COM interface to display all the
//  window names and to perform an execute in the Compare window.
//
//  This project was created in Visual C++ Developer.  To create a
//  similar project:
//
//    - Execute File -> New
//    - Select the Projects tab
//    - Select "Win32 Console Application"
//    - Select A "hello,World!" application (Visual Studio 6.0)
//
//  To make this buildable, you need to specify your "import" path
//  in stdafx.h (search for "TODO" in that file).  For example, add:
//  #import "C:/Program Files/Keysight Technologies/Logic Analyzer/LA \
//  COM Automation/agClientSvr.dll"
//
//  To run, you need to specify the host logic analyzer to connect
//  to (search for "TODO" below).
//

#include "stdafx.h"


////////////////////////////////////////////////////////////////////////
//
//  Forward declarations.
//
void DisplayError(_com_error& err);


////////////////////////////////////////////////////////////////////////
//
```

```
//  main() entry point.
//
int main(int argc, char* argv[])
{
   printf("*** Main()\n");

   //
   //  Initialize the Microsoft COM/ActiveX library.
   //
   HRESULT hr = CoInitialize(0);

   if (SUCCEEDED(hr))
   {
      try {   // Catch any unexpected run-time errors.
         _bstr_t hostname = "mtx33";   // TODO, use your logic
                                       // analysis system hostname.
         printf("Connecting to instrument '%s'\n", (char*) hostname);

         // Create the connect object and get the instrument object.
         AgtLA::IConnectPtr pConnect =
               AgtLA::IConnectPtr(__uuidof(AgtLA::Connect));
         AgtLA::IInstrumentPtr pInst =
               pConnect->GetInstrument(hostname);

         // Load the configuration file.
         _bstr_t configFile = "C:\\LA\\Configs\\compare.ala";
         printf("Loading the config file '%s'\n", (char*) configFile);
         pInst->Open(configFile, FALSE, "", TRUE);

         // Display all of the window names.
         AgtLA::IWindowsPtr pWindows = pInst->GetWindows();
         for (long i = 0; i < pWindows->GetCount(); i++)
         {
            AgtLA::IWindowPtr pWindow = pWindows->GetItem(i);
            _bstr_t name = pWindow->GetName();
            printf("Window name = %s\n", (char*) name);
         }

         // Get the Compare window, then execute the compare.
         AgtLA::ICompareWindowPtr pCompareWindow =
               pInst->GetWindowByName("Compare-1");
         _bstr_t myCompareOptions = "<Options ReferenceOffset='-2' \
               Range='M1..M2' MaxDifferences='0'/>";
         pCompareWindow->PutOptions(myCompareOptions);
         pCompareWindow->Execute();
         printf("Compare executed using options '%s'\n",
               (char*) myCompareOptions);
      }
      catch (_com_error& e) {
         DisplayError(e);
      }

      // Uninitialize the Microsoft COM/ActiveX library.
      CoUninitialize();
   }
   else
   {
```

```
      printf("CoInitialize failed\n");
   }

   return 0;
}


////////////////////////////////////////////////////////////////////
//
//  Displays the last error -- used to show the last exception
//  information.
//
void DisplayError(_com_error& error)
{
   printf("*** DisplayError()\n");

   printf("Fatal Unexpected Error:\n");
   printf("  Error Number = %08lx\n", error.Error());

   static char errorStr[1024];
   _bstr_t desc = error.Description();

   if (desc.length() == 0)
   {
      // Don't have a description string.
      strcpy(errorStr, error.ErrorMessage());
      int nLen = lstrlen(errorStr);

      // Remove funny carriage return ctrl<M>.
      if (nLen > 2 && (errorStr[nLen - 2] == 0xd))
      {
         errorStr[nLen - 2] = '\0';
      }
   }
   else
   {
      strcpy(errorStr, desc);
   }

   printf("  Error Message = %s\n", (char*) errorStr);
}
```

Methods

Add Method (BusSignals Object)

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- BusSignals (see page 86) object

Description

Adds a BusSignal (see page 81) object to the BusSignals (see page 86) collection using specific values.

VB Syntax

object.Add Name, Channels, [Polarity="+"]

Parameters	Definition
object	An expression that evaluates to a **BusSignals (see** page 86) object.
Name	A **String** containing the name of the bus/signal to be added.
Channels	A **String** containing *MultiplePodChannels* or the **String** "*None*" if no channels are assigned. **MultiplePodChannels** contains a comma separated list of *PodChannels*. **PodChannels** contains a *PodNumber* followed by the **String** "*[*" followed by a comma separated list of individual channel *Number(s)* and *NumberRange(s)* followed by the **String** "*]*". ***Note:****Pod channels normally are in MSB to LSB notation unless you are trying to reorder channels.* **NumberRange** contains a *Number* followed by the **String** "**:**" followed by a *Number*. PodChannels Example: `Pod 1[9:7,5,3:1]` – this bus consists of 1 single channel *Number* and 2 *NumberRange's* for a total of 7 channels. They are Pod 1[9], Pod 1[8], Pod 1[7], Pod 1[5], Pod 1[3], Pod 1[2], Pod 1[1]. MultiplePodChannels Example: `Pod 2[3,1],Pod 3[10,5:3]` – this bus consists of 3 single channel *Numbers* and 1 *NumberRange* for a total of 6 channels Pod 2[3], Pod 2[1], Pod 3[10], Pod 3[5], Pod 3[4], Pod 3[3].
Polarity	A **String** that is either "+" or "-". This parameter is optional and is "+" if not specified.

Add Method (Markers Object)

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 96)]

Applies To

- Markers (see page 95) object

Description

Adds a Marker (see page 95) object to the Markers (see page 95) collection using specific values.

VB Syntax

object.Add Name, TextColor, BackgroundColor, Position

Parameters	Definition
object	An expression that evaluates to a **Markers (see page 95)** object.
Name	A **String** containing the name of the marker to be added.
TextColor	A **Long** representing the color of the text.
BackgroundColor	A **Long** representing the color of the background.
Position	A **Double** that specifies the position of the marker (in seconds) relative to the trigger.

AddXML Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 96)]

Applies To

- Markers (see page 95) object

Description

Adds multiple Marker (see page 95) objects to the Markers (see page 95) collection using an XML string.

VB Syntax

object.AddXML XMLMarkers

Parameters	Definition
object	An expression that evaluates to a **Markers (see page 95)** object.
XMLMarkers	An "XML format" (in the online help)**String** containing the markers to add (see "<Markers> element" (in the online help)).

Close Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- Instrument (see page 93) object

Description

Closes the current configuration.

VB Syntax

object.Close [SaveChanges=False] [SaveFileName=""] [SetupOnly=False]

Parameters	Definition
object	An expression that evaluates to an Instrument (see page 93) object.

Parameters	Definition
SaveChanges	A **Boolean** that specifies whether changes to the configuration should be saved.
SaveFileName	A **String** that is the name of the file to which the configuration information should be saved.
SetupOnly	A **Boolean** that specifies whether the configuration file contains captured data or just setup information: **True** – save only setup information. **False** – save data and setup information.

Connect Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- ConnectSystem (see page 91) object

Description

Connects to the remote logic analyzer system.

VB Syntax

object.Connect [HostNameOrIpAddress=""]

Parameters	Definition
object	An expression that evaluates to a ConnectSystem (see page 91) object.
HostNameOrIpAddress	A **String** that contains the hostname or IP address of the logic analyzer instrument or computer on which the *Keysight Logic Analyzer* application will run. This parameter is optional.

See Also

- RecvFile (see page 174) method
- SendFile (see page 177) method

CopyFile Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- Connect (see page 90) object

Description

Copies a file to the instrument file system.

The Instrument (see page 202) property must be called first to establish a connection to the logic analyzer to which the file will be copied.

VB Syntax

object.CopyFile SrcFileName, DestFileName

Parameters	Definition
object	An expression that evaluates to a Connect (see page 90) object.
SrcFileName	A **String** that is the name of the file on the local file system.
DestFileName	A **String** that is the name of the file on the instrument file system.

See Also

- Instrument (see page 202) property

DeleteFile Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- Instrument (see page 93) object

Description Deletes a file on the instrument file system.

VB Syntax object.DeleteFile FileName

Parameters	Definition
object	An expression that evaluates to an Instrument (see page 93) object.
FileName	A **String** that is the name of the file on the instrument file system.

DoAction Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- Instrument (see page 93) object
- Module (see page 99) object
- Probe (see page 100) object
- Tool (see page 118) object
- Window (see page 122) object

Description Executes a specific command action.

VB Syntax object.DoAction Action, Parameters

Parameters	Definition
object	An expression that evaluates to one of the objects in the "Applies to" list above.
Action	Name of the command to execute. For information about the XML-based actions supported by a tool, see the "Tool Control, COM Automation" topic in the tool's online help.
Parameters	Command parameters.

Return Value A Boolean indicating whether the command was successful.

DoCommands Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 131)]

Applies To

- Instrument (see page 93) object
- Module (see page 99) object
- Probe (see page 100) object
- Tool (see page 118) object
- Window (see page 122) object

Description Executes a particular XML-based command.

VB Syntax object.DoCommands XMLCommand

Parameters	Definition
object	An expression that evaluates to one of the objects in the "Applies to" list above.
XMLCommand	An XML-format string that configures a module, tool, or window. See "Remarks" below.

Return Value A Boolean indicating whether the command was successful.

Remarks The XMLCommand format is based on the object type:

Object	XMLCommand Format
Instrument (see page 93)	A string containing the XML format "<Configuration> element" (in the online help).
Module (see page 99)	A string containing the XML format "<Module> element" (in the online help) (under Configuration Setup).
Probe (see page 100)	A string containing the XML format "<Probe> element" (in the online help) (under Configuration Setup).
Tool (see page 118)	A string containing the XML format "<Tool> element" (in the online help) (under Configuration Setup).
Window (see page 122)	A string containing the XML format "<Window> element" (in the online help) (under Configuration Setup).

DoCommands Example

Visual Basic

```
'You must create the Connect object (see page 90) and use it
' to access the Instrument object.  In this example, "myInst"
' represents the Instrument object.
'
' Get the MPC8XX Inverse Assembler tool.
Dim myTool As AgtLA.Tool
Set myTool = myInst.GetToolByName("Motorola PowerQUICC (MPC8XX) " + _
      "Inverse Assembler-1")

' Query for XML-based command.
Dim XMLCommand As String
If myTool.QueryCommand("GetProperties", XMLCommand) Then
   MsgBox "MPC8XX IA Properties: " + XMLCommand
End If

' Execute XML-based command.
If myTool.DoCommands(XMLCommand) Then
   MsgBox "MPC8XX IA DoCommands successful."
End If
```

Visual C++

```
//
//  This simple Visual C++ Console application demonstrates how to use
//  the Keysight 168x/169x/169xx COM interface to execute XML based
//  commands.
//
//  This project was created in Visual C++ Developer.  To create a
//  similar project:
//
//    - Execute File -> New
//    - Select the Projects tab
//    - Select "Win32 Console Application"
//    - Select A "hello,World!" application (Visual Studio 6.0)
//
//  To make this buildable, you need to specify your "import" path
//  in stdafx.h (search for "TODO" in that file).  For example, add:
//  #import "C:/Program Files/Keysight Technologies/Logic Analyzer/LA \
```

```
//  COM Automation/agClientSvr.dll"
//
//  To run, you need to specify the host logic analyzer to connect
//  to (search for "TODO" below).
//

#include "stdafx.h"


//////////////////////////////////////////////////////////////////////
//
//  Forward declarations.
//
void DisplayError(_com_error& err);


//////////////////////////////////////////////////////////////////////
//
//  main() entry point.
//
int main(int argc, char* argv[])
{
   printf("*** Main()\n");

   //
   //  Initialize the Microsoft COM/ActiveX library.
   //
   HRESULT hr = CoInitialize(0);

   if (SUCCEEDED(hr))
   {
      try {   // Catch any unexpected run-time errors.
         _bstr_t hostname = "mtx33";   // TODO, use your logic
                                       // analysis system hostname.
         printf("Connecting to instrument '%s'\n", (char*) hostname);

         // Create the connect object and get the instrument object.
         AgtLA::IConnectPtr pConnect =
               AgtLA::IConnectPtr(__uuidof(AgtLA::Connect));
         AgtLA::IInstrumentPtr pInst =
               pConnect->GetInstrument(hostname);

         // Load the configuration file.
         _bstr_t configFile = "C:\\LA\\Configs\\config.ala";
         printf("Loading the config file '%s'\n", (char*) configFile);
         pInst->Open(configFile, FALSE, "", TRUE);

         // Run the measurement, wait for it to complete.
         pInst->Run(FALSE);
         pInst->WaitComplete(20);

         // Get the MPC8XX Inverse Assembler tool.
         _bstr_t myTool =
               "Motorola PowerQUICC (MPC8XX) Inverse Assembler-1";
         AgtLA::IToolPtr pTool = pInst->GetToolByName(myTool);

         // Query for XML-based command.
```

```
            BSTR commandXML;
            if (pTool->QueryCommand("GetProperties", &commandXML)) {
               printf("MPC8XX IA Properties '%S'\n", (char*) commandXML);
            }

            // Execute XML based command.
            if (pTool->DoCommands((_bstr_t) commandXML)) {
               printf("MPC8XX IA DoCommands successful.\n");
            }

            SysFreeString(commandXML);
         }
         catch (_com_error& e) {
            DisplayError(e);
         }

         // Uninitialize the Microsoft COM/ActiveX library.
         CoUninitialize();
      }
      else
      {
         printf("CoInitialize failed\n");
      }

      return 0;
   }


   ////////////////////////////////////////////////////////////////////
   //
   //  Displays the last error -- used to show the last exception
   //  information.
   //
   void DisplayError(_com_error& error)
   {
      printf("*** DisplayError()\n");

      printf("Fatal Unexpected Error:\n");
      printf("  Error Number = %08lx\n", error.Error());

      static char errorStr[1024];
      _bstr_t desc = error.Description();

      if (desc.length() == 0)
      {
         // Don't have a description string.
         strcpy(errorStr, error.ErrorMessage());
         int nLen = lstrlen(errorStr);

         // Remove funny carriage return ctrl<M>.
         if (nLen > 2 && (errorStr[nLen - 2] == 0xd))
         {
            errorStr[nLen - 2] = '\0';
         }
      }
      else
      {
```

```
        strcpy(errorStr, desc);
    }

    printf("  Error Message = %s\n", (char*) errorStr);
}
```

Execute Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 82)]

Applies To

- CompareWindow (see page 90) object

Description

Executes the compare using the current options.

VB Syntax

object.Execute

Parameters	Definition
object	An expression that evaluates to a CompareWindow (see page 90) object.

See Also

- Differences (see page 200) property
- SampleDifferences (see page 210) property

Export Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- Instrument (see page 93) object

Description

Exports data to a file on the instrument file system.

VB Syntax

object.Export ExportFileName SourceName [ExportRange=False] [StartRange] [EndRange]

Parameters	Definition
object	An expression that evaluates to an Instrument (see page 93) object.
ExportFileName	A **String** that contains the name of the file (on the instrument file system) to which data is exported.
SourceName	A **String** that contains the name of the Module, Tool, or Window whose data will be exported.
ExportRange	A **Boolean** that specifies whether a data range is used: **True** – StartRange and EndRange contain the data range to export. **False** – exports all the data.
StartRange EndRange	**Variant**s specifying the data range (see page 135).

Remarks

The file is stored onto a drive that is directly accessible by the instrument.

The file format is determined by the ExportFileName suffix. File names with the .csv suffix are "Standard CSV text file" format. File names with the .alb suffix are "Module binary file" format (and the SourceName must be a logic analyzer or import module).

The Export method does not support the "Module CSV text file" format. To export this file type, use the ExportEx (see page 135) method.

See Also

- ExportEx (see page 135) method

- Import (see page 165) method
- ImportEx (see page 166) method

Data Ranges

Data ranges are specified by Variant start and end parameters.

For:	Use the Variant Type:
sample numbers	**Integer** or **Long**
times	**Double**

In other words:

- To specify a range by sample numbers, use Integer or Long start and end parameters.
- To specify a range by time, use Double start and end parameters.

See Also

- Export (see page 134) method
- GetNumSamples (see page 152) method
- GetTime (see page 158) method

ExportEx Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- Instrument (see page 93) object

Description

Exports data to a file on the instrument file system.

VB Syntax

object.ExportEx ExportFileName SourceName [ExportRange=False] [StartRange] [EndRange] [FileType=""] [FileOptions=""]

Parameters	Definition
object	An expression that evaluates to an Instrument (see page 93) object.
ExportFileName	A **String** that contains the name of the file (on the instrument file system) to which data is exported.
SourceName	A **String** that contains the name of the Module, Tool, or Window whose data will be exported. When the "Module CSV text file" or "Module binary file" FileType is specified, the SourceName must be a logic analyzer or import module. Timing zoom data can be exported separately using the SourceName syntax of "<module_name>:TimingZoom", for example, "My 16950A-1:TimingZoom".
ExportRange	A **Boolean** that specifies whether a data range is used: **True** – StartRange and EndRange contain the data range to export. **False** – exports all the data.

Parameters	Definition
StartRange EndRange	**Variant**s specifying the data range (see page 135).
FileType	A **String** that identifies the type of data you want to export. This is the same string that you see in the *Keysight Logic Analyzer* application's Export dialog: ▪ "Standard CSV text file" ▪ "Module CSV text file" ▪ "Module binary file"

FileOptions	An XML format **String** that lets you specify export options (as you can with the *Keysight Logic Analyzer* application's Export dialog **Options...** button). For example:
	
	`<Options`
	`SeparationCharacters=','`
	`WriteLineNumberColumn='T'`
	`LineNumberColumnName='Line Number'`
	`IncludeHeader='T'`
	`WriteFixedWidthColumns='F'`
	`/>`
	The `<Options>` element attribute values can be:
	▪ `SeparationCharacters - 'string'`
	▪ `WriteLineNumberColumn - 'F'` (false) or `'T'` (true)
	▪ `LineNumberColumnName - 'string'`
	▪ `IncludeHeader - 'F'` (false) or `'T'` (true)
	▪ `WriteFixedWidthColumns - 'F'` (false) or `'T'` (true)
	Certain file types support certain file options (as in the *Keysight Logic Analyzer* application):
	▪ "Standard CSV text file" – `SeparationCharacters`, `WriteLineNumberColumns`, and `LineNumberColumnName`.
	▪ "Module CSV text file" – `SeparationCharacters`, `WriteLineNumberColumns`, `LineNumberColumnName`, and `IncludeHeader`.
	▪ "Module binary file" – `IncludeHeader`.

Remarks The file is stored onto a drive that is directly accessible by the instrument.

See Also
- Export (see page 134) method
- Import (see page 165) method
- ImportEx (see page 166) method

Find Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 137)]

Applies To
- Window (see page 122) object

Description Finds a specified data event with optional occurrence and time duration.

VB Syntax object.Find Event, [Occurrence=1], [Direction="F"], [From="Display Center"], [When="Present"], [Duration=""]

Parameters	Definition
object	An expression that evaluates to a Window (see page 122) object.
Event	A **String** containing the event to find. The event can be a simple string (see Event (see page 178)) or, for more complex searches, an XML format string (see "<Event> element (under Find)" (in the online help)).
Occurrence	A **Long** containing the number of occurrences of the Event parameter.
Direction	A **String** containing the direction to search. "F" searches forward, "B" searches backward.
From	A **String** containing the position to start searching. This can be any of the following: ▪ "Display Center" ▪ "Beginning Of Data" ▪ "End Of Data" ▪ "Trigger" ▪ Name of a currently defined marker. Note that these strings are case-sensitive.
When	A **String** specifying a time duration or other operator. ▪ "Present" ▪ "Not Present" ▪ "Present>" (Duration must contain only one time value) ▪ "Present>=" (Duration must contain only one time value) ▪ "Present<" (Duration must contain only one time value) ▪ "Present<=" (Duration must contain only one time value) ▪ "Present for Range" (Duration must contain a time range) ▪ "Not In Range" (Duration must contain a time range) ▪ "Entering" ▪ "Exiting" ▪ "Transitioning" Note that these strings are case-sensitive.
Duration	A **String** containing a time duration which is only valid for certain 'When' qualifiers above. The format of this string can be either a time value or time range value. A time range contains a time value followed by the string ".." followed by another time value. A time value contains a number followed by a time unit. A time unit can be any of the strings: ▪ "ps" - picoseconds ▪ "ns" - nanoseconds ▪ "us" - microseconds ▪ "ms" - milliseconds ▪ "s" - seconds ▪ "Gs" - gigaseconds Examples: ▪ "1 ps" ▪ "1ns..30ns"

Return Value A FindResult (see page 92) object containing the results of the find.

See Also

- FindNext (see page 143) method
- FindPrev (see page 143) method
- FindResult (see page 92) object

Find Example

Visual Basic

```
Dim myWindow As Window
Set myWindow = myInst.Windows("Listing-1")

' Find using a simple event.
Dim myResult As FindResult
Set myResult = myWindow.Find("My Bus 1 = h55", 1, "F", _
```

```
    "Beginning Of Data")
If myResult.Found Then
  ' The event was found...
End If

' Find the same event using XML.
Dim myXMLEvent As String
myXMLEvent = "<Event>" + _
                "<BusSignal Name='My Bus 1' Operator='=' " + _
                      "Value='h55' />" + _
             "</Event>"
Set myResult = myWindow.Find(myXMLEvent, 1, "F", "Beginning Of Data")
If myResult.Found Then
  ' The event was found...
End If
```

Visual C++

```
//
//  This simple Visual C++ Console application demonstrates how to use
//  the Keysight 168x/169x/169xx COM interface to find a specified data
//  event.
//
//  This project was created in Visual C++ Developer.  To create a
//  similar project:
//
//    - Execute File -> New
//    - Select the Projects tab
//    - Select "Win32 Console Application"
//    - Select A "hello,World!" application (Visual Studio 6.0)
//
//  To make this buildable, you need to specify your "import" path
//  in stdafx.h (search for "TODO" in that file).  For example, add:
//  #import "C:/Program Files/Keysight Technologies/Logic Analyzer/LA \
//  COM Automation/agClientSvr.dll"
//
//  To run, you need to specify the host logic analyzer to connect
//  to (search for "TODO" below).
//

#include "stdafx.h"


////////////////////////////////////////////////////////////////////////
//
//  Forward declarations.
//
void DisplayError(_com_error& err);


////////////////////////////////////////////////////////////////////////
//
//  main() entry point.
//
int main(int argc, char* argv[])
{
   printf("*** Main()\n");

   //
```

```
//  Initialize the Microsoft COM/ActiveX library.
//
HRESULT hr = CoInitialize(0);

if (SUCCEEDED(hr))
{
   try {   // Catch any unexpected run-time errors.
      _bstr_t hostname = "mtx33";   // TODO, use your logic
                                    // analysis system hostname.
      printf("Connecting to instrument '%s'\n", (char*) hostname);

      // Create the connect object and get the instrument object.
      AgtLA::IConnectPtr pConnect =
            AgtLA::IConnectPtr(__uuidof(AgtLA::Connect));
      AgtLA::IInstrumentPtr pInst =
            pConnect->GetInstrument(hostname);

      // Run the measurement, wait for it to complete.
      pInst->Run(FALSE);
      pInst->WaitComplete(20);

      // Get a specific window.
      _bstr_t myListing = "Listing-1";
      AgtLA::IWindowPtr pWindow = pInst->GetWindowByName(myListing);

      // Find using a simple event.
      _bstr_t myEvent = "My Bus 1 = h55";
      AgtLA::IFindResultPtr pFindResult = pWindow->Find(myEvent, 1,
            "F", "Beginning Of Data", "Present", "");
      if (pFindResult->GetFound()) {
         _bstr_t myTimeFound = pFindResult->GetTimeFoundString();
         printf("Event '%s' was found at '%s'.\n", (char*) myEvent,
               (char *) myTimeFound);
      }
      else {
         printf("Event '%s' was not found.\n", (char*) myEvent);
      }

      // Find the same event using XML.
      _bstr_t myXMLEvent = "<Event><BusSignal Name='My Bus 1' \
            Operator='=' Value='h55' /></Event>";
      pFindResult = pWindow->Find(myXMLEvent, 1, "F", \
            "Beginning Of Data", "Present", "");
      if (pFindResult->GetFound()) {
         _bstr_t myTimeFound = pFindResult->GetTimeFoundString();
         printf("XML event '%s' was found at '%s'.\n",
               (char*) myXMLEvent, (char *) myTimeFound);
      }
      else {
         printf("XML event '%s' was not found.\n",
               (char*) myXMLEvent);
      }
   }
   catch (_com_error& e) {
      DisplayError(e);
   }
```

```
      // Uninitialize the Microsoft COM/ActiveX library.
      CoUninitialize();
   }
   else
   {
      printf("CoInitialize failed\n");
   }

   return 0;
}


////////////////////////////////////////////////////////////////////////
//
//  Displays the last error -- used to show the last exception
//  information.
//
void DisplayError(_com_error& error)
{
   printf("*** DisplayError()\n");

   printf("Fatal Unexpected Error:\n");
   printf("  Error Number = %08lx\n", error.Error());

   static char errorStr[1024];
   _bstr_t desc = error.Description();

   if (desc.length() == 0)
   {
      // Don't have a description string.
      strcpy(errorStr, error.ErrorMessage());
      int nLen = lstrlen(errorStr);

      // Remove funny carriage return ctrl<M>.
      if (nLen > 2 && (errorStr[nLen - 2] == 0xd))
      {
         errorStr[nLen - 2] = '\0';
      }
   }
   else
   {
      strcpy(errorStr, desc);
   }

   printf("  Error Message = %s\n", (char*) errorStr);
}
```

FindImages Method

[Automation Home (see page 3)] [Objects (see page 79)]

NOTE: This method is applicable only for those Logic and Protocol Analyzer modules that support the Image View.

Applies To ProtocolWindow (see page 104) object

Description Searches acquisition data for frames containing images and puts them into the Protocol Viewer's list of images.

VB Syntax object.**FindImages** start end errorMessage success

Parameters	Definition
object	An expression that evaluates a ProtocolWindow object.
start	A **VARIANT** which specifies the starting point in the search of the acquisition data. If start is a floating point number, then start is interpreted as a packet time. If start is a string, then start must be one of these case-sensitive strings. ▪ "Beginning Of Data" ▪ "End Of Data" ▪ "Trigger" ▪ "M1" ▪ "M2" or any other marker name
end	A **VARIANT** which specifies the ending point in the search of the acquisition data. If end is a floating point number, then end is interpreted as a packet time. If end is a string, then end must be one of these case-sensitive strings. ▪ "Beginning Of Data" ▪ "End Of Data" ▪ "Trigger" ▪ "M1" ▪ "M2" or any other marker name
errorMessage	A string output parameter. If success returns VARIANT_FALSE, then errorMessage contains information about the error that occurred. If success returns VARIANT_TRUE, then errorMessage is the empty string.
success	A **VARIANT_BOOL** output parameter which returns VARIANT_TRUE or VARIANT_FALSE. VARIANT_FALSE indicates an error occurred and errorMessage contains an explanation of the error.

Remarks FindImages is one of the several methods that enables the COM user to identify and extract image frame data. It performs the same function as the 'Find' button on the 'Image View' tab of the Protocol viewer.

The Protocol Viewer maintains an internal list of image frames that have been found in the acquisition data. Both FindImages and the 'Find' button update the same internal list. All the other COM commands which access image data use this internal list. Thus, a call to FindImages is the first step to access image data.

FindImages Example

Visual C++ **NOTE:** The example below is incomplete. The example does not contain code which acquires data. In order to work, the code in the example must have packet data in acquisition memory. It is also required that the packet data must contain some image frames.

```
// Create the Connect object.

AgtLA::IConnectPtr pConnect =
AgtLA::IConnectPtr(__uuidof(AgtLA::Connect));…

// Using the Connect object, obtain the IInstrument interface.

AgtLA::IInstrumentPtr pInst = pConnect->GetInstrument(hostName);…

// Using the IInstrument interface, obtain the IWindows interface

AgtLA::IWindowsPtr windows = pInst->GetWindows();…
```

```
// Loop through all the windows in the instrument looking for the first
window
// that presents an IProtocolWindow interface.
AgtLA::IprotoWindowPtr protocolWindow = NULL;
for (long i=0; i < windows->GetCount(); i++)
{
  // Using the IWindows interface, obtain a IWindow interface.
  AgtLA::IWindowPtr window = windows->GetItem(_variant_t(i));

  // Check if this window is a ProtocolWindow by type-casting the
  // module pointer.
  protocolWindow= windows->GetItem(_variant_t(i));
  if(protocolWindow != NULL)
  {
    // Found a ProtocolWindow
    break;
  }
}
if (protocolWindow == NULL)
{
  QUIT - There are no Protocol Windows in this Logic Analzyer
}
// Find all the images in the acquisition data.
//
// Create start and end variant parameters.
VARIANT startVar;
VARIANT endVar;
startVar.vt = VT_BSTR;
startVar.bstrVal = _T("Beginning Of Data");
endVar.vt = VY_BSTR;
endVar.bstrVal = _T("End Of Data");

// Return parameters
```

```
VARIANT_BOOL bSuccess;

BSTR bstrErrorMessage;


// Call the FindImages COM method

bSuccess = protocolWindow->FindImages(startVar, endVar,
&bstrErrorMessage);

// Check result

if (bSuccess == VARIANT_FALSE)

{

  Display bstrErrorMessage and Quit

}
```

FindNext Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 137)]

Applies To

- Window (see page 122) object

Description

Finds the next event by searching forward from the last event found using the event specified by the last call to Find (see page 136).

VB Syntax

object.FindNext

Parameters	Definition
object	An expression that evaluates to a Window (see page 122) object.

Return Value

A FindResult (see page 92) object containing the results of the find.

See Also

- FindResult (see page 92) object
- Find (see page 136) method
- FindPrev (see page 143) method

FindPrev Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 137)]

Applies To

- Window (see page 122) object

Description

Finds the previous event by searching backward from the last event found using the event specified by the last call to Find (see page 136).

VB Syntax

object.FindPrev

Parameters	Definition
object	An expression that evaluates to a Window (see page 122) object.

Return Value

A FindResult (see page 92) object containing the results of the find.

See Also

- FindResult (see page 92) object
- Find (see page 136) method
- FindNext (see page 143) method

GetDataBySample Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 105)]

Applies To

- SampleBusSignalData (see page 104) object

Description

Given a range, returns an array of acquired data. GetDataBySample returns the data within a trigger relative sample range.

NOTE

The data can only be returned when the hardware is stopped. Before calling this method, call the Instrument object's **WaitComplete** (see page 180) method to make sure the hardware is stopped.

VB Syntax

object.GetDataBySample StartSample, EndSample, DataType, NumRowsRet

Parameters	Definition
object	An expression that evaluates to an SampleBusSignalData (see page 104) object.
StartSample	A **Long** containing the first sample to upload.
EndSample	A **Long** containing the last sample to upload. EndSample must be greater than or equal to StartSample.
DataType	Specifies the type of data to return. See DataTypes and Return Values (see page 144).

Returns	Definition
NumRowsRet	A **Long** initialized by this method to the number of rows being returned in the array.

Return Values

See DataTypes and Return Values (see page 144).

See Also

- GetDataByTime (see page 146) method
- StartSample (see page 212) property
- EndSample (see page 201) property
- GetTime (see page 158) method

DataTypes and Return Values

AgtDataType	Enum Value	Return Value
AgtDataRaw	&H0001 (1)	Returns an array of **Byte**s. The total size of the array is NumRowsRet multiplied by the value in the BusSignal (see page 81) object's ByteSize (see page 194) property. This can hold the maximum BusSignal size of 128 bits of unsigned data. This type is the most efficient in terms of bytes transferred. Using this type, only the smallest number of bytes needed to represent every channel in a bus/signal will be transferred. The bus/signal values are stored in the array from MSB to LSB.
AgtDataDecimal	&H0002 (2)	Returns an array of **Variant**s. The **Variant** contains a decimal data type that holds 96 bits of unsigned and signed integer data. Decimals are stored as 96-bit unsigned integers scaled by a variable power of 10. The power of 10 scaling factor specifies the number of digits to the right of the decimal point and ranges from 0 to 28. This data type can be used when the BusSignalType (see page 193) property is **AgtBusSignalSampleNum**, when the GetTime (see page 158) method is called, when the bus/signal is oscilloscope voltage data, or when the bus/signal you are getting data for is less than 96 bits wide, unsigned. No error is returned if the data is truncated.
AgtDataLong	&H0003 (3)	Returns an array of **Long**s. This holds 31 bits of unsigned integer data and 32 bits of signed integer data. This data type can be used when the BusSignalType (see page 193) property is **AgtBusSignalSampleNum** or when the bus/signal you are getting data for is less than 32 bits wide, unsigned. No error is returned if the data is truncated.
AgtDataTime	&H0004 (4)	Returns an array of **Double**s. This data type is only valid when the GetTime (see page 158) method is called. You can also use **AgtDataDouble** and **AgtDataDecimal** to access time values as well.
AgtDataStringDec	&H0005 (5)	Returns an array of **String**s, formatted in decimal.
AgtDataStringHex	&H0006 (6)	Returns an array of **String**s, formatted in hexadecimal. When the GetTime (see page 158) method is called, the value is formatted as a hex string in units of 10**-24 seconds. When the bus/signal is oscilloscope voltage data, the value is formatted as a hex string in units of 10**-12 volts. In both cases, the string is an exact representation of the internal value; therefore, no information is lost.
AgtDataString	&H0007 (7)	Returns an array of **String**s using the default format.
AgtDataDouble	&H0008 (8)	Returns an array of **Double**s. This holds 52 bits of unsigned integer data and 53 bits of signed integer data. This data type can be used when the BusSignalType (see page 193) property is **AgtBusSignalSampleNum**, when the GetTime (see page 158) method is called for any bus/signal, when the bus/signal is oscilloscope voltage data, or when the bus/signal you are getting data for is less than 52 bits wide, unsigned. No error is returned if the data is truncated.

OR'ed in AgtDataType	Enum Value	Return Value

AgtDataSubrows	&H0100 (256)	Returns an array of arrays of type specified by the lower 8 bits of **EnumValue**.
		For example, if the **EnumValue** is a bitwise OR of AgtDataString and AgtDataSubrows (see code examples below), an array of string arrays will be returned. This is used to return multiple rows of data per sample, for example, when an inverse assembler returns multiple rows of decoded strings per sample. If the sample does not contain subrows, the array for that sample will typically contain one value; however, when the BusSignalType (see page 193) property is **AgtBusSignalGenerated**, there are cases when the array may be empty.
		`sArray() = GetDataBySample(0,10, AgtDataString Or AgtDataSubrows, nNumRowsRet)  ' Visual Basic`
		`sArray = GetDataBySample(0,10, AgtDataString | AgtDataSubrows, nNumRowsRet);  /* C/C++ */`

AgtDataSigned	&H0200 (512)	Returns signed values of type specified by the lower 8 bits of **EnumValue**. This value is ignored for **EnumValueAgtDataRaw**. For example, if the **EnumValue** is a bitwise OR of AgtDataDouble and AgtDataSigned, each sample will be converted to a signed value and returned as an array of doubles.

GetDataByTime Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 105)]

Applies To

- SampleBusSignalData (see page 104) object

Description

Given a range, returns an array of acquired data. GetDataByTime returns the data within a trigger relative time range.

NOTE

The data can only be returned when the hardware is stopped. Before calling this method, call the Instrument object's **WaitComplete** (see page 180) method to make sure the hardware is stopped.

VB Syntax

object.GetDataByTime StartTime, EndTime, DataType, NumRowsRet

Parameters	Definition
object	An expression that evaluates to an SampleBusSignalData (see page 104) object.
StartTime	A **Double** containing the starting time (in seconds) to upload. **Double** values can be expressed as mmm**E**eee or mmm**D**eee, in which mmm is the mantissa and eee is the exponent (a power of 10); for example, a StartTime value of 450E-9 is 450 nanoseconds.
EndTime	A **Double** containing the ending time (in seconds) to upload. **EndTime** must be greater than or equal to **StartTime**.
DataType	Specifies the type of data to return. See DataTypes and Return Values (see page 144).

Returns	Definition
NumRowsRet	A **Long** initialized by this method to the number of rows being returned in the array.

Return Values See DataTypes and Return Values (see page 144).

See Also

- GetDataBySample (see page 144) method
- StartTime (see page 212) property
- EndTime (see page 201) property
- GetTime (see page 158) method

GetImageList Method

[Automation Home (see page 3)] [Objects (see page 79)]

NOTE: This method is applicable only for those Logic and Protocol Analyzer modules that support the Image View.

Applies To ProtocolWindow (see page 104) object

Description Returns a string containing the Protocol Viewer's list of images. Optionally, it writes this list to a text file.

VB Syntax object.**GetImageList** doWriteToFile csvFilePath success

Parameters	Definition
object	An expression that evaluates a ProtocolWindow object.
doWriteToFile	A **VARIANT_BOOL** that specifies whether to write the image list to a file. VARIANT_TRUE causes a file to be written using the path in csvFilePath.
csvFilePath	A string that specifies the pathname of the optional CSV file. We ignore csvFilePath if doWriteToFile is VARIANT_FALSE.
success	A **VARIANT_BOOL** output parameter which returns VARIANT_TRUE or VARIANT_FALSE. VARIANT_TRUE means the operation was successful and the return string contains the image list. VARIANT_FALSE indicates an error occurred and the return string contains an error message.

Return Value A string containing the Protocol Viewer's list of images.

Remark **GetImageList** performs the same function as 'Export list to CSV...' button in the 'Image View' tab of the Protocol Viewer.

GetImageList returns a string. The string contains one or more lines. The first line in the string is a heading. Subsequent lines contain data. There is one data line for each image in the list. The string written to the optional file is the same as the returned string.

Each data line contains four data fields separated by commas. Here is an example of an image list containing two images.

```
Frame Number, Sample Number, Time (ns), Module
0-0, 451, 0, CSI-101
0-1, 1177, 200444481, CSI-101
```

If there are no images in the Protocol Viewer's list, then **GetImageList** method returns an error.

GetImageList Example

Visual C++

NOTE: The example below is incomplete. The example does not contain code which acquires data. In order to work, the code in the example must have packet data in acquisition memory. It is also required that the packet data must contain some image frames.

```
// Create the Connect object.

AgtLA::IConnectPtr pConnect =
AgtLA::IConnectPtr(__uuidof(AgtLA::Connect));…

// Using the Connect object, obtain the IInstrument interface.

AgtLA::IInstrumentPtr pInst = pConnect->GetInstrument(hostName);…

// Using the IInstrument interface, obtain the IWindows interface

AgtLA::IWindowsPtr windows = pInst->GetWindows();…

// Loop through all the windows in the instrument looking for the first
window

// that presents an IProtocolWindow interface.

AgtLA::IprotoWindowPtr protocolWindow = NULL;

for (long i=0; i < windows->GetCount(); i++)

{

  // Using the IWindows interface, obtain a IWindow interface.

  AgtLA::IWindowPtr window = windows->GetItem(_variant_t(i));


  // Check if this window is a ProtocolWindow by type-casting the

  // module pointer.

  protocolWindow= windows->GetItem(_variant_t(i));

  if(protocolWindow != NULL)

  {

    // Found a ProtocolWindow

    break;

  }

}

if (protocolWindow == NULL)

{

  QUIT - There are no Protocol Windows in this Logic Analzyer

}

// Obtain the list of images.

//
```

```
// Create parameters for optional write to text file.

VARIANT_BOOL bWriteToFile = VARIANT_FALSE;

VARIANT_BOOL bSuccess;

_bstr_t bstrPath = "";


// Return variables.

BSTR bstrReturn;

// Call the GetImageList COM method

bstrReturn = protocolWindow->GetImageList(bWriteToFile, bstrPath,
&bSuccess);

// Check result

If (bSuccess == VARIANT_FALSE)

{

  Display error message in bstrReturn and Quit

}
```

GetModuleByName Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 149)]

Applies To

- Instrument (see page 93) object

Description

Given a module name, returns its corresponding hardware module object.

VB Syntax

object.GetModuleByName Name

Parameters	Definition
object	An expression that evaluates to an Instrument (see page 93) object.
Name	A **String** containing the module's name as defined by the Name (see page 205) property.

Return Value

A hardware Module (see page 99) object with the module name given.

See Also

- Modules (see page 205) property
- Name (see page 205) property

GetModuleByName Example

Visual Basic

```
'You must create the Connect object (see page 90) and use it
' to access the Instrument object.  In this example, "myInst"
' represents the Instrument object.
'
' Get the module named "My 16910A-1".
Dim myModule As AgtLA.Module
Set myModule = myInst.GetModuleByName("My 16910A-1")
```

```
' Display the module status.
MsgBox "Module: " + myModule.Name + ", status: " + myModule.Status
```

Visual C++

```
//
//  This simple Visual C++ Console application demonstrates how to use
//  the Keysight 168x/169x/169xx COM interface to get a module by name
//  and display its status.
//
//  This project was created in Visual C++ Developer.  To create a
//  similar project:
//
//    - Execute File -> New
//    - Select the Projects tab
//    - Select "Win32 Console Application"
//    - Select A "hello,World!" application (Visual Studio 6.0)
//
//  To make this buildable, you need to specify your "import" path
//  in stdafx.h (search for "TODO" in that file).  For example, add:
//  #import "C:/Program Files/Keysight Technologies/Logic Analyzer/LA \
//  COM Automation/agClientSvr.dll"
//
//  To run, you need to specify the host logic analyzer to connect
//  to (search for "TODO" below).
//

#include "stdafx.h"


////////////////////////////////////////////////////////////////////////
//
//  Forward declarations.
//
void DisplayError(_com_error& err);


////////////////////////////////////////////////////////////////////////
//
//  main() entry point.
//
int main(int argc, char* argv[])
{
   printf("*** Main()\n");

   //
   //  Initialize the Microsoft COM/ActiveX library.
   //
   HRESULT hr = CoInitialize(0);

   if (SUCCEEDED(hr))
   {
      try {   // Catch any unexpected run-time errors.
         _bstr_t hostname = "mtx33";   // TODO, use your logic
                                       // analysis system hostname.
         printf("Connecting to instrument '%s'\n", (char*) hostname);

         // Create the connect object and get the instrument object.
         AgtLA::IConnectPtr pConnect =
```

```
            AgtLA::IConnectPtr(__uuidof(AgtLA::Connect));
         AgtLA::IInstrumentPtr pInst =
            pConnect->GetInstrument(hostname);

         // Get a specific analyzer module.
         _bstr_t moduleName = "My 16910A-1";
         AgtLA::IAnalyzerModulePtr pAnalyzer =
            pInst->GetModuleByName(moduleName);

         // Display the module status.
         _bstr_t name = pAnalyzer->GetName();
         _bstr_t status = pAnalyzer->GetStatus();
         printf("Module '%s', status '%s'.\n", (char*) name,
            (char*) status);
      }
      catch (_com_error& e) {
         DisplayError(e);
      }

      // Uninitialize the Microsoft COM/ActiveX library.
      CoUninitialize();
   }
   else
   {
      printf("CoInitialize failed\n");
   }

   return 0;
}


//////////////////////////////////////////////////////////////////////
//
//  Displays the last error -- used to show the last exception
//  information.
//
void DisplayError(_com_error& error)
{
   printf("*** DisplayError()\n");

   printf("Fatal Unexpected Error:\n");
   printf("  Error Number = %08lx\n", error.Error());

   static char errorStr[1024];
   _bstr_t desc = error.Description();

   if (desc.length() == 0)
   {
      // Don't have a description string.
      strcpy(errorStr, error.ErrorMessage());
      int nLen = lstrlen(errorStr);

      // Remove funny carriage return ctrl<M>.
      if (nLen > 2 && (errorStr[nLen - 2] == 0xd))
      {
         errorStr[nLen - 2] = '\0';
      }
```

```
    }
    else
    {
       strcpy(errorStr, desc);
    }

    printf("  Error Message = %s\n", (char*) errorStr);
}
```

GetNumSamples Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 105)]

Applies To

- SampleBusSignalData (see page 104) object

Description

Given a range, returns the number of samples stored.

NOTE

The data can only be returned when the hardware is stopped. Before calling this method, call the Instrument object's **WaitComplete** (see page 180) method to make sure the hardware is stopped.

VB Syntax

object.GetNumSamples [StartPosition], [EndPosition]

Parameters	Definition
object	An expression that evaluates to an SampleBusSignalData (see page 104) object.
StartPosition EndPosition	**Variant**s specifying the data range (see page 135). **EndPosition** must be greater than or equal to **StartPosition**.

Return Values

A Long containing the number of samples in the range.

GetPacketCount Method

[Automation Home (see page 3)] [Objects (see page 79)]

Applies To

ProtocolWindow (see page 104) object

Description

Given a channel name, returns the number of packets for that particular channel.

VB Syntax

object.**get_PacketCount** samplechannel packetCount success

Parameters	Definition
object	An expression that evaluates a ProtocolWindow (see page 104) object.
samplechannel	A sample channel must be a channel name.
packetCount	An **INTEGER** output parameter which returns the number of packets.
success	A **VARIANT_BOOL** output parameter which returns VARIANT_TRUE or VARIANT_FALSE.

Return Value

Number of packets for a channel

Remarks

PacketCount is one of the several methods that enables the COM user to count the number of packets for a given channel.

GetPacketCount Example

Visual C++

NOTE: The example below is incomplete. The example does not contain code which acquires data. In order to work, the code in the example must have packet data in acquisition memory. It is also required that the packet data must contain some image frames.

```
// Create the Connect object.

AgtLA::IConnectPtr pConnect =
AgtLA::IConnectPtr(__uuidof(AgtLA::Connect));…

// Using the Connect object, obtain the IInstrument interface.

AgtLA::IInstrumentPtr pInst = pConnect->GetInstrument(hostName);…

// Using the IInstrument interface, obtain the IWindows interface

AgtLA::IWindowsPtr windows = pInst->GetWindows();…

// Loop through all the windows in the instrument looking for the first
window

// that presents an IProtocolWindow interface.

AgtLA::IprotoWindowPtr protocolWindow = NULL;

for (long i=0; i < windows->GetCount(); i++)

{

  // Using the IWindows interface, obtain a IWindow interface.

  AgtLA::IWindowPtr window = windows->GetItem(_variant_t(i));


  // Check if this window is a ProtocolWindow by type-casting the

  // module pointer.

  protocolWindow= windows->GetItem(_variant_t(i));

  if(protocolWindow != NULL)

  {

    // Found a ProtocolWindow

    break;

  }

}

if (protocolWindow == NULL)

{

  QUIT - There are no Protocol Windows in this Logic Analzyer

}

// call the packet event.

// Parameter to specify a folder
```

```
_bstr_t  bstrChannel= "";  // Channel name
Int packetCount   // No of packet for given channel name
VARIANT_BOOL bSuccess; //

// Call the packetcount COM method
bSuccess = protocolWindow->GetPacketCount(bstrChannel, &packetCount);
// Check result
If (bSuccess == VARIANT_FALSE)
{
  Display error message in bstrErrorMessage and Quit
}
```

GetProbeByName Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- Instrument (see page 93) object

Description

Given a probe name, returns the corresponding probe object.

VB Syntax

object.GetProbeByName Name

Parameters	Definition
object	An expression that evaluates to an Instrument (see page 93) object.
Name	A **String** containing the probe's name as defined by the Name (see page 205) property.

Return Value

A Probe (see page 100) object with the name given.

See Also

- Probe (see page 100) object
- Probes (see page 101) object
- Instrument (see page 93) object
- Name (see page 205) property

GetProtocolDataFields Method

[Automation Home (see page 3)] [Objects (see page 79)]

Applies To

- ProtocolWindow object (see page 104)

Description

Gets the acquisition data for the fields displayed in the Protocol Viewer.

VB Syntax

`object.GetProtocolDataFields start end channel success`

Parameters	Definition
object	An expression that evaluates to a ProtocolWindow (see page 104) object.
start	A **VARIANT** which specifies the first packet in a range of packets for which the acquisition data is fetched. If *start* is an integer, then *start* is interpreted as the packet number for the first packet in the range. If *start* is a floating point number, then *start* is interpreted as the packet time of the first packet in the range.
end	A **VARIANT** which specifies the last packet in a range of packets for which the acquisition data is fetched. If *end* is an integer, then end is interpreted as the packet number for the last packet in the range. If *end* is a floating point number, then *end* is interpreted as the packet time for the last packet in the range. ***NOTE***: "start" and "end" must either be both integers (i.e. packets numbers) or be both floating point numbers (i.e. packet times).
channel	A **string** which specifies the channel (or direction) of the sample numbers specified as the start and end parameters. channel is used only if start and end are integers representing packet numbers. channel is ignored if start and end are floating point numbers representing packet times.
success	A **VARIANT_BOOL** output parameter which returns VARIANT_TRUE or VARIANT_FALSE. VARIANT_TRUE means the operation was successful and the return string contains data. VARIANT_FALSE means an error occurred and the return string contains an error message.

Remarka GetProtocolDataFields returns a string that contains one or more lines. The first line in the string is a heading. Subsequent lines contain acquisition data. For each packet in the specified range of packets, there is one acquisition data line. Each data line contains a series of values separated by commas. This series of values is as per the fields displayed in the Packets pane of the Protocol Viewer. If you want to choose the fields that the method returns in a data line, you must insert, delete, or re-arrange the fields according to your requirements in the Packets pane of the Protocol Viewer.

GetRawData Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- AnalyzerModule (see page 80) object

Description Given a range, returns the raw analyzer data.

NOTE The data can only be returned when the hardware is stopped. Before calling this method, call the Instrument object's **WaitComplete** (see page 180) method to make sure the hardware is stopped.

VB Syntax object.GetRawData StartPosition, EndPosition, NumBytesPerRow, NumRowsRet

Parameters	Definition
object	An expression that evaluates to an AnalyzerModule (see page 80) object.
StartPosition EndPosition	**Variant**s specifying the data range (see page 135). **EndPosition** must be greater than or equal to **StartPosition**.

Returns	Definition
NumBytesPerRow	A **Long** initialized by this method to the width of each sample row being returned in the array.
NumRowsRet	A **Long** initialized by this method to the number of rows being returned in the array.

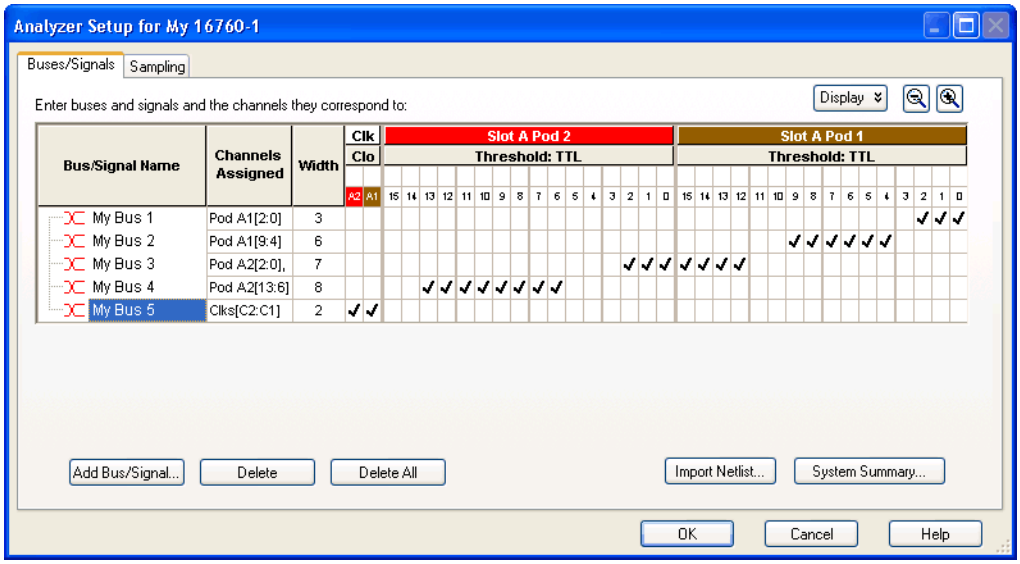
Return Values

Returns a Variant array of Bytes. The total size of the array is NumRowsRet multiplied by NumBytesPerRow.

The data is returned in the following format (which is reversed from the byte order that appears in the Buses/Signals Setup dialog):

```
Pod 1, Pod 2, Pod 3, ..., Clock
```

For example, if the logic analyzer has two pods:



The data would be stored in array like:

```
 Pod 1       Pod 1       Pod 2       Pod 2       Clocks
  7..0        15..8        7..0        15..8        1..0
--------    --------    --------    --------    --------
array[0]    array[1]    array[2]    array[3]    array[4]
```

The Clock bytes are rounded up to the nearest byte; for example, if there are 10 clock channels, the data is stored in two bytes.

See Also

- GetRawTimingZoomData (see page 156) method

GetRawTimingZoomData Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- AnalyzerModule (see page 80) object

Description

Given a range, returns the raw analyzer timing zoom data.

NOTE The data can only be returned when the hardware is stopped. Before calling this method, call the Instrument object's **WaitComplete** (see page 180) method to make sure the hardware is stopped.

VB Syntax object.GetRawTimingZoomData StartPosition, EndPosition, NumBytesPerRow, NumRowsRet

Parameters	Definition
object	An expression that evaluates to an AnalyzerModule (see page 80) object.
StartPosition EndPosition	**Variant**s specifying the data range (see page 135). **EndPosition** must be greater than or equal to **StartPosition**.

Returns	Definition
NumBytesPerRow	A **Long** initialized by this method to the width of each sample row being returned in the array.
NumRowsRet	A **Long** initialized by this method to the number of rows being returned in the array.

Return Values Returns an array of Bytes. The total size of the array is NumRowsRet multiplied by NumBytesPerRow.

The data is returned in the following format (which is reversed from the bit order that appears in the Buses/Signals Setup dialog):

```
Pod 1, Pod 2, Pod 3, ..., Clock
```

The Clock bytes are rounded up to the nearest byte; for example, if there are 10 clock channels, the data is stored in two bytes.

See Also

- GetRawData (see page 155) method

GetRemoteInfo Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- Connect (see page 90) object

Description Gets the logic analyzer's remote user login and computer name.

This method gives you information about who is remotely connected via COM to the logic analyzer. This is a passive interrogation and does not modify the current values returned by RemoteComputerName (see page 209) and RemoteUserName (see page 209).

VB Syntax object.GetRemoteInfo HostNameOrIPAddress, RemoteUserName, RemoteComputerName

Parameters	Definition
object	An expression that evaluates to a Connect (see page 90) object.
HostNameOrIPAddress	A **String** that is the hostname or IP address of the logic analyzer.

Returns	Definition
RemoteUserName	A **String** that is the remote user login name.
RemoteComputerName	A **String** that is the remote computer name.

See Also

- RemoteComputerName (see page 209) property
- RemoteUserName (see page 209) property

GetSampleNumByTime Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 105)]

Applies To

- SampleBusSignalData (see page 104) object

Description

Gets the closest sample number corresponding to the time given.

NOTE

The sample number can only be returned when the hardware is stopped. Before calling this method, call the Instrument object's **WaitComplete** (see page 180) method to make sure the hardware is stopped.

VB Syntax

object.GetSampleNumByTime Time

Parameters	Definition
object	An expression that evaluates to an SampleBusSignalData (see page 104) object.
Time	A **Double** containing the time that you want to get the sample number for. **Double** values can be expressed as mmm**E**eee or mmm**D**eee, in which mmm is the mantissa and eee is the exponent (a power of 10); for example, a Time value of 450E-9 is 450 nanoseconds.

Return Values

A Long containing the sample number.

GetTime Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 105)]

Applies To

- SampleBusSignalData (see page 104) object

Description

Given a range, returns the time for this bus/signal in the format specified by the data type given. GetTime gets the time values for the specific bus/signal.

NOTE

The time can only be returned when the hardware is stopped. Before calling this method, call the Instrument object's **WaitComplete** (see page 180) method to make sure the hardware is stopped.

VB Syntax

object.GetTime StartPosition, EndPosition, DataType, NumRowsRet

Parameters	Definition
object	An expression that evaluates to an SampleBusSignalData (see page 104) object.
StartPosition EndPosition	**Variant**s specifying the data range (see page 135). **EndPosition** must be greater than or equal to **StartPosition**.
DataType	Specifies the type of data to return. See DataTypes and Return Values (see page 144).

Returns	Definition
NumRowsRet	A **Variant** initialized by this method to the number of rows being returned in the time array.

Return Values See DataTypes and Return Values (see page 144).

See Also
- GetDataByTime (see page 146) method
- GetDataBySample (see page 144) method

GetToolByName Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To
- Instrument (see page 93) object

Description Given a tool name, returns its corresponding tool object.

VB Syntax object.GetToolByName Name

Parameters	Definition
object	An expression that evaluates to an Instrument (see page 93) object.
Name	A **String** containing the tool's name as defined by the Name (see page 205) property.

Return Value A Tool (see page 118) object with the tool name given.

See Also
- Tools (see page 216) property
- Name (see page 205) property

GetTriggerSampleNumber Method

[Automation Home (see page 3)] [Objects (see page 79)]

Applies To
- ProtocolWindow (see page 104) object

Description Gets the packet number and channel of the trigger packet.

VB Syntax object.GetTriggerSampleNumber success PacketNumber

Parameters	Definition
object	An expression that evaluates to a ProtocolWindow (see page 104) object.
success	A **VARIANT_BOOL** output parameter which returns VARIANT_TRUE or VARIANT_FALSE. VARIANT_TRUE means the operation was successful and the return string contains the channel name. VARIANT_FALSE means an error occurred and the return string contains an error message.
PacketNumber	An **integer** output parameter which returns the packet number of the trigger packet.

See Also

- GetProtocolDataFields (see page 154) method

GetWindowByName Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- Instrument (see page 93) object

Description

Given a window name, returns the corresponding window object.

VB Syntax

object.GetWindowByName Name

Parameters	Definition
object	An expression that evaluates to an Instrument (see page 93) object.
Name	A **String** containing the window's name as defined by the Name (see page 205) property.

Return Value

A Window (see page 122) object with the tool name given.

See Also

- Window (see page 122) object
- Windows (see page 122) object
- Instrument (see page 93) object
- Name (see page 205) property

GoOffline Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 160)]

Applies To

- Instrument (see page 93) object

Description

Disconnects the user interface from the logic analyzer frame.

VB Syntax

object.GoOffline

Parameters	Definition
object	An expression that evaluates to an Instrument (see page 93) object.

See Also

- GoOnline (see page 164) method
- IsOnline (see page 166) method

GoOffline Example

Visual Basic

```
Option Explicit   ' Must define all variables.

Sub Main()

' Define the logic analysis systems being used.
Dim myFirstLAS As String
Dim mySecondLAS As String
myFirstLAS = "mtx33"
mySecondLAS = "col-mil20"

'You must create the Connect object (see page 90) and use it
' to access the Instrument object.  In this example, "myInst"
' represents the Instrument object.

Dim myConnect As AgtLA.Connect
Dim myInst As AgtLA.Instrument
Set myConnect = CreateObject("AgtLA.Connect")
Set myInst = myConnect.Instrument(myFirstLAS)

' Display whether the instrument is offline or online.
DisplayConnected myInst, myFirstLAS

' Go offline.
myInst.GoOffline
DisplayConnected myInst, myFirstLAS

' Go online to second logic analysis system.
myInst.GoOnline (mySecondLAS)
DisplayConnected myInst, myFirstLAS

' Go offline.
myInst.GoOffline
DisplayConnected myInst, myFirstLAS

' Go online to first logic analysis system.
myInst.GoOnline (myFirstLAS)
DisplayConnected myInst, myFirstLAS

End Sub

Private Sub DisplayConnected(inst As AgtLA.Instrument, system As String)

  Dim connectedTo As String
  If inst.IsOnline(connectedTo) Then
    MsgBox "LA system: " + system + ", is online, connected to: " + _
        connectedTo
  Else
    MsgBox "LA system: " + system + ", is offline."
  End If

End Sub
```

Visual C++

```
//
//  This simple Visual C++ Console application demonstrates how to use
//  the Keysight 168x/169x/169xx COM interface to take a logic analysis
//  system offline and then go back online again.
//
```

```
//  This project was created in Visual C++ Developer.  To create a
//  similar project:
//
//    - Execute File -> New
//    - Select the Projects tab
//    - Select "Win32 Console Application"
//    - Select A "hello,World!" application (Visual Studio 6.0)
//
//  To make this buildable, you need to specify your "import" path
//  in stdafx.h (search for "TODO" in that file).  For example, add:
//  #import "C:/Program Files/Keysight Technologies/Logic Analyzer/LA \
//  COM Automation/agClientSvr.dll"
//
//  To run, you need to specify the logic analysis systems to connect
//  to (search for "TODO" below).
//

#include "stdafx.h"


////////////////////////////////////////////////////////////////////////
//
//  Forward declarations.
//
void DisplayConnected(
  AgtLA::IInstrumentPtr  pInst,
  _bstr_t                system);

void DisplayError(_com_error& err);


////////////////////////////////////////////////////////////////////////
//
//  main() entry point.
//
int main(int argc, char* argv[])
{
   printf("*** Main()\n");

   //
   //  Initialize the Microsoft COM/ActiveX library.
   //
   HRESULT hr = CoInitialize(0);

   if (SUCCEEDED(hr))
   {
      try {   // Catch any unexpected run-time errors.
         _bstr_t myFirstLAS = "mtx33";   // TODO, use your first logic
                                         // analysis system hostname.
         _bstr_t mySecondLAS = "col-mil20";   // TODO, use your second
                                              // logic analysis system
                                              // hostname.
         printf("Connecting to instrument '%s'\n", (char*) myFirstLAS);

         // Create the connect object and get the instrument object.
         AgtLA::IConnectPtr pConnect =
               AgtLA::IConnectPtr(__uuidof(AgtLA::Connect));
```

```
         AgtLA::IInstrumentPtr pInst =
               pConnect->GetInstrument(myFirstLAS);

         // Display whether the instrument is offline or online.
         DisplayConnected(pInst, myFirstLAS);

         // Go offline.
         pInst->GoOffline();
         DisplayConnected(pInst, myFirstLAS);

         // Go online to second logic analysis system.
         pInst->GoOnline(mySecondLAS, 1, FALSE, "", FALSE);
         DisplayConnected(pInst, myFirstLAS);

         // Go offline.
         pInst->GoOffline();
         DisplayConnected(pInst, myFirstLAS);

         // Go online to first logic analysis system.
         pInst->GoOnline(myFirstLAS, 1, FALSE, "", FALSE);
         DisplayConnected(pInst, myFirstLAS);

      }
      catch (_com_error& e) {
         DisplayError(e);
      }

      // Uninitialize the Microsoft COM/ActiveX library.
      CoUninitialize();
   }
   else
   {
      printf("CoInitialize failed\n");
   }

   return 0;
}


////////////////////////////////////////////////////////////////////
//
//  Displays whether a logic analysis system if offline or online,
//  and if online the system it it connected to.
//
void DisplayConnected(
  AgtLA::IInstrumentPtr  pInst,
  _bstr_t                system)
{
   printf("*** DisplayConnected()\n");

   BSTR connectedTo;
   if (pInst->IsOnline(&connectedTo)) {
      printf("LA system '%s', is online, connected to '%S'.\n",
            (char*) system, (char*) connectedTo);
   }
   else {
      printf("LA system '%s', is offline.\n", (char*) system);
```

```
   }
   SysFreeString(connectedTo);

}


//////////////////////////////////////////////////////////////////////
//
//  Displays the last error -- used to show the last exception
//  information.
//
void DisplayError(_com_error& error)
{
   printf("*** DisplayError()\n");

   printf("Fatal Unexpected Error:\n");
   printf("  Error Number = %08lx\n", error.Error());

   static char errorStr[1024];
   _bstr_t desc = error.Description();

   if (desc.length() == 0)
   {
      // Don't have a description string.
      strcpy(errorStr, error.ErrorMessage());
      int nLen = lstrlen(errorStr);

      // Remove funny carriage return ctrl<M>.
      if (nLen > 2 && (errorStr[nLen - 2] == 0xd))
      {
         errorStr[nLen - 2] = '\0';
      }
   }
   else
   {
      strcpy(errorStr, desc);
   }

   printf("  Error Message = %s\n", (char*) errorStr);
}
```

GoOnline Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 160)]

Applies To

- Instrument (see page 93) object

Description

Connects the user interface to a specific logic analyzer frame. The frame is locked to this user interface until it is released by calling the GoOffline (see page 160) method.

VB Syntax

object.GoOnline [ComputerNameOrIPAddress=""] [FrameNumber=1] [SaveChanges=False] [SaveFileName=""] [SetupOnly=False]

Parameters	Definition
object	An expression that evaluates to an Instrument (see page 93) object.
ComputerNameOrIPAddress	A **String** containing the frame's computer host name as defined by the ComputerName (see page 196) property or IP address as defined by the IPAddress (see page 202) property.
FrameNumber	A **Long** that specifies the number of the frame to connect to in a multiframe configuration.
SaveChanges	A **Boolean** that specifies whether changes to the current configuration should be saved.
SaveFileName	A **String** containing the name of the file to which current configuration information should be saved.
SetupOnly	A **Boolean** that specifies whether the file to which the current configuration will be saved contains captured data or just setup information: **True** – save only setup information. **False** – save data and setup information.

See Also

- GoOffline (see page 160) method
- IsOnline (see page 166) method

GoToPosition Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- Window (see page 122) object

Description

Moves the center of the window to a new position.

VB Syntax

object.GoToPosition Position

Parameters	Definition
object	An expression that evaluates to a Window (see page 122) object.
Position	A **Variant** that specifies the time of the sample to be placed at the center of the window. For time values, use the variant type **Double**.

Import Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- Instrument (see page 93) object

Description

Imports data from a file located on the instrument file system.

VB Syntax

object.Import ImportFileName [SaveChanges=False] [SaveFileName=""] [SetupOnly=False]

Parameters	Definition
object	An expression that evaluates to an Instrument (see page 93) object.
ImportFileName	A **String** that contains the name of the file (on the instrument file system) to be imported.

Parameters	Definition
SaveChanges	A **Boolean** that specifies whether changes to the current configuration should be saved.
SaveFileName	A **String** containing the name of the file to which current configuration information should be saved.
SetupOnly	A **Boolean** that specifies whether the file to which the current configuration will be saved contains captured data or just setup information: **True** – save only setup information. **False** – save data and setup information.

Remarks The import file must be directly accessible by the instrument.

See Also
- ImportEx (see page 166) method
- Export (see page 134) method
- ExportEx (see page 135) method

ImportEx Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To
- Instrument (see page 93) object

Description Imports data from a file located on the instrument file system into a particular module.

VB Syntax object.ImportEx ImportFileName DestinationName [SaveChanges=False] [SaveFileName=""] [SetupOnly=False]

Parameters	Definition
object	An expression that evaluates to an Instrument (see page 93) object.
ImportFileName	A **String** that contains the name of the file (on the instrument file system) to be imported.
DestinationName	A **String** that is the name of the module into which the data is imported.
SaveChanges	A **Boolean** that specifies whether changes to the current configuration should be saved.
SaveFileName	A **String** containing the name of the file to which current configuration information should be saved.
SetupOnly	A **Boolean** that specifies whether the file to which the current configuration will be saved contains captured data or just setup information: **True** – save only setup information. **False** – save data and setup information.

Remarks The import file must be directly accessible by the instrument.

The supported file type is:
- "Module CSV text file" (*.csv) — If DestinationName is not found, an import module is created with that name.

See Also
- Import (see page 165) method
- Export (see page 134) method
- ExportEx (see page 135) method

IsOnline Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 160)]

Applies To
- Instrument (see page 93) object

Description Tells whether the user interface is connected to a logic analyzer frame.

VB Syntax object.IsOnline ComputerName

Parameters	Definition
object	An expression that evaluates to an Instrument (see page 93) object.
ComputerName	A **String** that is initialized to the frame's name (as defined by the ComputerName (see page 196) property) if the user interface is online (connected to a frame).

Return Value A Boolean indicating whether the user interface is connected to a frame. If True is returned, the ComputerName will be initialized.

See Also
- GoOffline (see page 160) method
- GoOnline (see page 164) method

IsTimingZoom Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To
- BusSignal (see page 81) object

Description Is this a timing zoom bus/signal?

VB Syntax object.IsTimingZoom

Parameters	Definition
object	An expression that evaluates to an BusSignal (see page 81) object.

Return Value A Boolean indicating whether it is a timing zoom bus/signal.

New Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To
- Instrument (see page 93) object

Description Creates a new instrument Overview.

VB Syntax object.New [SaveChanges=False] [SaveFileName=""] [SetupOnly=False]

Parameters	Definition
object	An expression that evaluates to an Instrument (see page 93) object.
SaveChanges	A **Boolean** that specifies whether changes to the current configuration should be saved.
SaveFileName	A **String** that is the name of the file to which current configuration information should be saved.
SetupOnly	A **Boolean** that specifies whether the file to which the current configuration will be saved contains captured data or just setup information: **True** – save only setup information. **False** – save data and setup information.

Open Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- Instrument (see page 93) object

Description

Loads a previously saved configuration file located on the instrument file system. This will restore the instrument's settings and contents of the acquisition memory (if available). You can open either ALA or XML format configuration files.

VB Syntax

object.Open OpenFileName [SaveChanges=False] [SaveFileName=""] [SetupOnly=False]

Parameters	Definition
object	An expression that evaluates to an Instrument (see page 93) object.
OpenFileName	A **String** that contains the name of the file located on the Instrument.
SaveChanges	A **Boolean** that specifies whether changes to the current configuration should be saved.
SaveFileName	A **String** containing the name of the file to which current configuration information should be saved.
SetupOnly	A **Boolean** that specifies whether the file to which the current configuration will be saved contains captured data or just setup information: **True** – save only setup information. **False** – save data and setup information.

Remarks

The configuration file must be directly accessible by the instrument.

See Also

- Save (see page 176) method

PanelLock Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 168)]

Applies To

- Instrument (see page 93) object

Description

Coordinates user access to the instrument front panel or remote display with other users. When locked, a full screen message is displayed indicating the instrument is currently in use by a remote COM automation program. If desired, a custom message can be shown on the local display instead of a default message. As an example, a custom message might give information as to who has the unit locked. The instrument can then be unlocked when desired.

VB Syntax

object.PanelLock [Message = ""]

Parameters	Definition
object	An expression that evaluates to an Instrument (see page 93) object.
Message	A **String** containing a custom message that will be shown on the display. If the message is empty, then a default message will be used.

See Also

- PanelUnlock (see page 171) method

PanelLock Example

Visual Basic

```
'You must create the Connect object (see page 90) and use it
' to access the Instrument object.  In this example, "myInst"
' represents the Instrument object.
```

```
'
' Lock the instrument's front panel.
myInst.PanelLock ("Locked by Name, Phone")

' If locked, display the message.
Dim myMessage As String
If myInst.PanelLocked(myMessage) Then
   MsgBox "Remote user message: " + myMessage
End If

' Unlock the instrument's front panel.
myInst.PanelUnlock
```

Visual C++

```
//
//  This simple Visual C++ Console application demonstrates how to use
//  the Keysight 168x/169x/169xx COM interface to lock and unlock the
//  instrument's front panel.
//
//  This project was created in Visual C++ Developer.  To create a
//  similar project:
//
//    - Execute File -> New
//    - Select the Projects tab
//    - Select "Win32 Console Application"
//    - Select A "hello,World!" application (Visual Studio 6.0)
//
//  To make this buildable, you need to specify your "import" path
//  in stdafx.h (search for "TODO" in that file).  For example, add:
//  #import "C:/Program Files/Keysight Technologies/Logic Analyzer/LA \
//  COM Automation/agClientSvr.dll"
//
//  To run, you need to specify the host logic analyzer to connect
//  to (search for "TODO" below).
//

#include "stdafx.h"


////////////////////////////////////////////////////////////////////////
//
//  Forward declarations.
//
void DisplayError(_com_error& err);


////////////////////////////////////////////////////////////////////////
//
//  main() entry point.
//
int main(int argc, char* argv[])
{
   printf("*** Main()\n");

   //
   //  Initialize the Microsoft COM/ActiveX library.
   //
   HRESULT hr = CoInitialize(0);
```

```
   if (SUCCEEDED(hr))
   {
      try {   // Catch any unexpected run-time errors.
         _bstr_t hostname = "mtx33";   // TODO, use your logic
                                       // analysis system hostname.
         printf("Connecting to instrument '%s'\n", (char*) hostname);

         // Create the connect object and get the instrument object.
         AgtLA::IConnectPtr pConnect =
               AgtLA::IConnectPtr(__uuidof(AgtLA::Connect));
         AgtLA::IInstrumentPtr pInst =
               pConnect->GetInstrument(hostname);

         // Lock the instrument's front panel.
         pInst->PanelLock("Locked by Name, Phone");

         // If locked, display the message.
         BSTR myMessage;
         if (pInst->GetPanelLocked(&myMessage)) {
            printf("Remote user message '%S'\n", (char*) myMessage);
         }
         SysFreeString(myMessage);

         // Unlock the instrument's front panel.
         pInst->PanelUnlock();
      }
      catch (_com_error& e) {
         DisplayError(e);
      }

      // Uninitialize the Microsoft COM/ActiveX library.
      CoUninitialize();
   }
   else
   {
      printf("CoInitialize failed\n");
   }

   return 0;
}


////////////////////////////////////////////////////////////////////////
//
//  Displays the last error -- used to show the last exception
//  information.
//
void DisplayError(_com_error& error)
{
   printf("*** DisplayError()\n");

   printf("Fatal Unexpected Error:\n");
   printf("  Error Number = %08lx\n", error.Error());

   static char errorStr[1024];
   _bstr_t desc = error.Description();
```

```
   if (desc.length() == 0)
   {
      // Don't have a description string.
      strcpy(errorStr, error.ErrorMessage());
      int nLen = lstrlen(errorStr);

      // Remove funny carriage return ctrl<M>.
      if (nLen > 2 && (errorStr[nLen - 2] == 0xd))
      {
         errorStr[nLen - 2] = '\0';
      }
   }
   else
   {
      strcpy(errorStr, desc);
   }

   printf("  Error Message = %s\n", (char*) errorStr);
}
```

PanelUnlock Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 168)]

Applies To

- Instrument (see page 93) object

Description

Re-enables user access to the instrument front panel or remote display.

VB Syntax

object.PanelUnlock

Parameters	Definition
object	An expression that evaluates to an Instrument (see page 93) object.

See Also

- PanelLock (see page 168) method

QueryCommand Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 131)]

Applies To

- Instrument (see page 93) object
- Module (see page 99) object
- Probe (see page 100) object
- Tool (see page 118) object
- Window (see page 122) object

Description

Query for XML-based commands.

VB Syntax

object.QueryCommand Query, XMLCommand

Parameters	Definition
object	An expression that evaluates to one of the objects in the "Applies to" list above.
Query	A **String** that contains either an XML-based query command or an XML-based filter. If a filter is specified, the query command will be set to "GetAllSetup", and the output of the command will be filtered so that it only contains the attributes or tags of interest as specified by the 'Query' filter string. When a filter is used, the string returned may not be valid XML format. If the 'Query' starts with the character '<', it is treated as an XML-based filter (see page 172); otherwise, it is treated as an XML-based query (see page 172) command.

Returns	Definition
XMLCommand	A **String** that contains the XML-based command.

Return Values

A Boolean indicating whether the command was successful.

XML-Based Query

An XML-based query command is specific to the Module, Tool, or Window. If queries are supported, the query "GetAllSetup" returns all of the setup commands.

For information about the XML-based query commands supported by a tool, see the "Tool Control, COM Automation" topic in the tool's online help.

XML-Based Filter

A filter can be either a fully qualified XML string or a shortened XML string. A shortened XML string does not contain end tag notation and has the following format:

```
<tag [attribute[=value]] ...> ...
```

Example of a fully qualified XML string filter:

```
'You must create the Connect object (see page 90) and use it
' to access the Instrument object.  In this example, "myInst"
' represents the Instrument object.
'
'   Dim queryOutput As String

myInst.QueryCommand _
  "<Setup>" & _
    "<Module Name=''>" & _
      "<BusSignalSetup>" & _
        "<BusSignal Name=''/>" & _
      "</BusSignalSetup>" & _
    "</Module>" & _
  "</Setup>", queryOutput
```

`queryOutput` contains:

```
Name="MyLA-1"
Name="My Bus 1"
```

Example of a shortened XML string filter:

```
myInst.QueryCommand "<Setup><Module Name><BusSignalSetup>" + _
      "<BusSignal Name>", queryOutput
```

`queryOutput` contains:

```
Name="MyLA-1"
Name="My Bus 1"
```

Both examples are equivalent and, when called, return the first module's name and its first bus/signal name.

1 If an attribute is not present at the end, the entire tag is returned. For example:

```
myInst.QueryCommand "<Setup><Module><BusSignalSetup><BusSignal>",
_
        queryOutput
```

`queryOutput` contains:

```
<BusSignal Name='My Bus 1' Polarity='Positive' DefaultBase='Hex'
      Comment=''>
<Channels>Pod 1[7:0]</Channels>
</BusSignal>
```

2 More than one attribute can be returned. For example:

```
myInst.QueryCommand "<Setup><Module><BusSignalSetup>" + _
      "<BusSignal Name Polarity>", queryOutput
```

`queryOutput` contains:

```
Name="My Bus 1"
Polarity="Positive"
```

3 Tags are used to disambiguate; you can skip beginning and intermediate tag levels. Use with caution because the first tag found is used. For example:

```
myInst.QueryCommand "<Setup><BusSignal Name>", queryOutput
```

`queryOutput` contains:

```
Name="Sample Number"
```

NOTE You might have expected Name to be 'My Bus 1' instead of 'Sample Number'. Because a breadth-first search is done, the BusSignal tag for 'Listing-1' is at a higher level than 'My 1690A-1'. This is why care should be taken when skipping tag levels.

4 If an '=' sign and a non-empty value are used, the tag with the given attribute value is used. This is useful when there is more than one tag with the same name at the same level, like BusSignal. For example:

```
myInst.QueryCommand "<Setup><Module><BusSignalSetup>" + _
      "<BusSignal Name='My Bus 2' Polarity>", queryOutput
```

`queryOutput` contains:

```
Polarity="Positive"
```

5 White space is ignored. For example:

```
myInst.QueryCommand "< Setup > < BusSignal Name >", queryOutput
```

`queryOutput` contains:

```
Name="Sample Number"
```

See Also

- "XML Format" (in the online help)

RecallTriggerByFile Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 25)]

Applies To
- AnalyzerModule (see page 80) object
- SerialModule object (see page 117)

Description Loads a previously saved trigger file located on the instrument's file system.

VB Syntax object.RecallTriggerByFile TriggerFileName

Parameters	Definition
object	An expression that evaluates to an AnalyzerModule (see page 80) or a SerialModule (see page 117) object.
TriggerFileName	A **String** that contains the name of the XML-format trigger specification file located on the instrument's file system.

Remarks The RecallTriggerByFile method is a shortcut that reads an XML-format trigger specification file from the instrument's file system and sets the Trigger (see page 217) property of the applicable object (AnalyzerModule or SerialModule in this case).

See Also
- AnalyzerModule (see page 80) object
- SerialModule object (see page 117)

RecallTriggerByName Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To
- AnalyzerModule (see page 80) object

Description Loads a named trigger from the recall buffer.

VB Syntax object.RecallTriggerByName TriggerName

Parameters	Definition
object	An expression that evaluates to an AnalyzerModule (see page 80) object.
TriggerName	A **String** that represents the recall buffer title name.

See Also
- AnalyzerModule (see page 80) object

RecvFile Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To
- ConnectSystem (see page 91) object

Description Copies a file from the remote logic analyzer system to your local system.

The Connect (see page 129) method must be called first to establish a connection to the logic analyzer from which the file will be copied.

VB Syntax object.RecvFile SrcFileName, DestFileName

Parameters	Definition
object	An expression that evaluates to a ConnectSystem (see page 91) object.
SrcFileName	A **String** that is the name of the file on the remote logic analyzer system.
DestFileName	A **String** that is the name of the file on the local file system.

See Also

- Connect (see page 129) method

Remove Method (BusSignals Object)

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- BusSignals (see page 86) object

Description

Removes a bus/signal from the BusSignals collection.

VB Syntax

BusSignals (see page 86).Remove Name

Parameters	Definition
Name	A **String** containing the name of the bus/signal to be removed.

Remove Method (Markers Object)

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 96)]

Applies To

- Markers (see page 95) object

Description

Removes a marker from the Markers collection.

VB Syntax

object.Remove Name

Parameters	Definition
object	An expression that evaluates to a **Markers (see page 95)** object.
Name	A **String** containing the name of the marker to be removed.

RemoveAll Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 96)]

Applies To

- Markers (see page 95) object

Description

Removes all markers from the Markers collection.

VB Syntax

object.RemoveAll

Parameters	Definition
object	An expression that evaluates to a **Markers (see page 95)** object.

RemoveXML Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 96)]

Applies To

- Markers (see page 95) object

Description

Removes multiple Marker (see page 95) objects from the Markers (see page 95) collection using an XML string.

VB Syntax

object.RemoveXML XMLMarkers

Parameters	Definition
object	An expression that evaluates to a **Markers (see page 95)** object.
XMLMarkers	An "XML format" (in the online help)**String** containing the markers to remove (see "<Markers> element" (in the online help)).

Run Method (Instrument Object)

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- Instrument (see page 93) object

Description

Starts running all modules.

VB Syntax

object.Run [Repetitive=False]

Parameters	Definition
object	An expression that evaluates to an Instrument (see page 93) object.
Repetitive	A Boolean indicating the number of times the module(s) are started. **True** – run continuously until the Stop (see page 179) method is called. **False** – run only once.

See Also

- Stop (see page 179) method
- Status (see page 213) property
- WaitComplete (see page 180) method

Save Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- Instrument (see page 93) object

Description

Saves the current configuration to a file on the instrument file system.

VB Syntax

object.Save SaveFileName [SetupOnly=False]

Parameters	Definition
object	An expression that evaluates to an Instrument (see page 93) object.
SaveFileName	A **String** that contains the name of the file to be saved on the Instrument. If the filename string has a suffix of ".xml", the configuration is saved to an XML format file; if the filename string has a suffix of ".ala" or has no suffix at all, the configuration is saved to an ALA format file.
SetupOnly	A **Boolean** that specifies whether the file to which the current configuration will be saved contains captured data or just setup information: **True** – save only setup information. **False** – save data and setup information.

Remarks The file is stored onto a drive that is directly accessible by the instrument.

See Also
- Open (see page 168) method

SendFile Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To
- ConnectSystem (see page 91) object

Description Copies a file from your local system to the remote logic analyzer system.

The Connect (see page 129) method must be called first to establish a connection to the logic analyzer to which the file will be copied.

VB Syntax object.SendFile SrcFileName, DestFileName

Parameters	Definition
object	An expression that evaluates to a ConnectSystem (see page 91) object.
SrcFileName	A **String** that is the name of the file on the local file system.
DestFileName	A **String** that is the name of the file on the remote logic analyzer system.

See Also
- Connect (see page 129) method

SimpleTrigger Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 29)]

Applies To
- AnalyzerModule (see page 80) object

Description Trigger on a simple condition with optional occurrence and storage qualification. Default triggers on anything and stores everything.

VB Syntax object.SimpleTrigger [OnEvent = "Anything"], [Occurs=1], [StoreEvent="Anything"]

Parameters	Definition
object	An expression that evaluates to an AnalyzerModule (see page 80) object.

Parameters	Definition
OnEvent	A **String** containing the Event (see page 178) to trigger on.
Occurs	A **Long** containing the number of occurrences of **OnEvent** before triggering.
StoreEvent	A **String** containing the storage qualification Event (see page 178) while waiting for **OnEvent**.

Remarks If the analyzer is in timing mode, the only valid value for StoreEvent is "Anything".

See Also
- RecallTriggerByFile (see page 173) method
- RecallTriggerByName (see page 174) method

Simple Trigger/Find Event

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 29)]

Applies To
- SimpleTrigger (see page 177) method
- Find (see page 136) method

Description An Event contains either a ComboValue, the String "Anything", or the String "Nothing". For the Find (see page 136) method, an Event can only contain a ComboValue.

A ComboValue contains a BusSignal optionally followed by the String "And" or "Or" followed by another BusSignal.

A BusSignal contains a bus/signal name followed by a Relational followed by a Range, Value, or Edge.

A Relational string can be:
- "=", "!=", ">", "<", ">=", or "<="

Additional Relational strings used with the Find (see page 136) method are:
- "Entering" – the first sample of one or more consecutive samples that match the pattern. (By comparison, the "=" equals operator considers every sample that matches the pattern as an occurrence.)
- "Exiting" – the sample after one or more consecutive samples that match the pattern.
- "Transitioning" – entering or exiting one or more consecutive samples that match the pattern.

A Range contains a Value followed by the String ".." followed by another Value. Range values cannot contain don't care digits. A range can only be used with the Relational strings "=" and "!=".

A Value string contains a number with optional don't care digits or an edge. A number base is required; use the following prefixes:
- h – hexadecimal
- o – octal
- b – binary
- d – decimal

An Edge string begins with "e" followed by any combination of the following upper-case characters:
- X – don't care
- R – rising edge
- F – falling edge
- E – either edge

Remarks Value strings are case insensitive.

When an Edge string is used, the BusSignal's Relational string must be "=". When used with the Find (see page 136) method, an Edge specification can only be one bit wide. When used with the SimpleTrigger (see page 177) method, an Edge is only valid when the analyzer is in timing mode.

Bus/signal names that contain " And " (with spaces before and after), " Or " (with spaces before and after), or end in a non-alphanumeric character will confuse the parser and produce unexpected results.

Example

Value examples:

`hFFXX0022`	Hex number with 2 don't care digits (8 don't care bits).
`o7777xxxx`	Octal number with 4 don't care digits (12 don't care bits).
`b10110110xxxx0000`	Binary number with 4 don't care bits.

Range examples:

`hff00..hffff`	Range from hex ff00 to ffff.

Edge examples:

`eXXXXRFEG`	Edge specification with 4 don't care bits, then rising, falling, and either bits.
`eR`	A rising edge on a signal (one bit).

Event examples:

`ADDR=hffffxxxx`	Specifies a simple bus/signal value.
`ADDR=hffff0000..hffffffff`	Specifies the same simple bus/signal value as a range.
`ADDR=hffff0000..hffffffff And DATA=eXXXRXXXX`	Specifies a rising edge in bit 5 of DATA while ADDR is within the hex range ffff0000 to ffffffff.
`ADDR=hffxx And DATA=h055`	Specifies AND'ing two bus/signal values.

Trigger Event example:

`Anything`	Specifies any value.

Find Event example:

`ADDR entering hff00`	Specifies the first sample of one or more consecutive samples whose ADDR is hFF00.

See Also

- SimpleTrigger (see page 177) method
- Find (see page 136) method

Stop Method (Instrument Object)

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- Instrument (see page 93) object

Description

Stops all currently running data acquisition modules.

NOTE If self tests are running, this will also stop and close the Self Test dialog.

VB Syntax object.Stop

Parameters	Definition
object	An expression that evaluates to an Instrument (see page 93) object.

See Also
- Run (see page 176) method
- Status (see page 213) property

TestAll Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 115)]

Applies To
- SelfTest (see page 115) object

Description Runs an instrument's self-tests.

VB Syntax object.TestAll

Parameters	Definition
object	An expression that evaluates to a SelfTest (see page 115) object.

Return Values The TestAll method returns one of the following run result values:
- "RUN_RESULT_INIT_FAILED"
- "RUN_RESULT_FAILED"
- "RUN_RESULT_INCOMPLETE"
- "RUN_RESULT_PASSED"

WaitComplete Method

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 181)]

Applies To
- AnalyzerModule (see page 80) object
- Instrument (see page 93) object
- Module (see page 99) object

Description Waits until a measurement completes or a timeout (in seconds) occurs.

Executing the Instrument (see page 93) object's WaitComplete method waits for all data acquisition modules, tools, and viewers to complete their measurements.

Executing a Module (see page 99) object's WaitComplete method waits for the module to complete its measurement.

VB Syntax object.WaitComplete [Seconds="-1"]

Parameters	Definition
object	An expression that evaluates to one of the objects in the "Applies to" list above.
Seconds	A **Long** containing the maximum number of seconds to wait for the measurement to complete. **Note:** If set to -1, this method does not return until the specified measurement completes; therefore, we strongly recommend you make this value >= 0.

WaitComplete Example

Visual Basic

```
Private Sub Command1_Click()

' If WaitComplete times out, a run-time error occurs.
On Error GoTo ErrorHandler

'You must create the Connect object (see page 90) and use it
' to access the Instrument object.  In this example, "myInst"
' represents the Instrument object.
'
' Load the configuration file.
myInst.Open ("c:\LA\Configs\Test1.ala")

' Load the logic analyzer trigger file.
Dim myAnalyzer As AgtLA.AnalyzerModule
Set myAnalyzer = myInst.GetModuleByName("My 16756A-1")
myAnalyzer.RecallTriggerByFile ("c:\LA\Triggers\Test1_TrigSpec.xml")

' Run the measurement, wait for it to complete.
myInst.Run
myInst.WaitComplete (20)

' Notify when measurement is complete.
MsgBox "Measurement complete."

Exit Sub
ErrorHandler:
   ' Handle the error that occurs if WaitComplete times out.
   Select Case Err.Number
      Case -2147352567
         myInst.Stop
         MsgBox "WaitComplete timed out, measurement stopped."
         Resume Next
      Case Else
         Err.Raise Number:=Err.Number
   End Select

End Sub
```

Visual C++

```
//
//  This simple Visual C++ Console application demonstrates how to
//  use the Keysight 168x/169x/169xx COM interface to wait until a
//  measurement is complete.
//
//  This project was created in Visual C++ Developer.  To create a
//  similar project:
//
```

```
//    - Execute File -> New
//    - Select the Projects tab
//    - Select "Win32 Console Application"
//    - Select A "hello,World!" application (Visual Studio 6.0)
//
//  To make this buildable, you need to specify your "import" path
//  in stdafx.h (search for "TODO" in that file).  For example, add:
//  #import "C:/Program Files/Keysight Technologies/Logic Analyzer/LA \
//  COM Automation/agClientSvr.dll"
//
//  To run, you need to specify the host logic analyzer to connect
//  to (search for "TODO" below).
//

#include "stdafx.h"


////////////////////////////////////////////////////////////////////////
//
//  Forward declarations.
//
void DisplayError(_com_error& err);


////////////////////////////////////////////////////////////////////////
//
//  main() entry point.
//
int main(int argc, char* argv[])
{
   printf("*** Main()\n");

   //
   //  Initialize the Microsoft COM/ActiveX library.
   //
   HRESULT hr = CoInitialize(0);

   if (SUCCEEDED(hr))
   {
      try {   // Catch any unexpected run-time errors.
         _bstr_t hostname = "mtx33";   // TODO, use your logic
                                       // analysis system hostname.
         printf("Connecting to instrument '%s'\n", (char*) hostname);

         // Create the connect object and get the instrument object.
         AgtLA::IConnectPtr pConnect =
               AgtLA::IConnectPtr(__uuidof(AgtLA::Connect));
         AgtLA::IInstrumentPtr pInst =
               pConnect->GetInstrument(hostname);

         // Load the configuration file.
         _bstr_t configFile = "C:\\LA\\Configs\\config.ala";
         printf("Loading the config file '%s'\n", (char*) configFile);
         pInst->Open(configFile, FALSE, "", TRUE);

         // Set up the trigger.
         _bstr_t moduleName = "MPC860 Demo Board";
```

```
            AgtLA::IAnalyzerModulePtr pAnalyzer =
                  pInst->GetModuleByName(moduleName);
            pAnalyzer->SimpleTrigger("ADDR=0", 1, "Anything");

            // Run the measurement, wait for it to complete.
            pInst->Run(FALSE);
            try {
               pInst->WaitComplete(20);
               printf("Measurement complete.\n");
            }
            catch (_com_error& e) {
               switch (e.Error()) {
               case 0x80020009:
                  pInst->Stop();
                  printf("Inner WaitComplete timed out, ");
                  printf("measurement stopped.\n");
                  break;
               default:
                  throw;
                  break;
               };
            }
            printf("End of program.\n");
         }
         catch (_com_error& e) {
            DisplayError(e);
         }

         // Uninitialize the Microsoft COM/ActiveX library.
         CoUninitialize();
      }
      else
      {
         printf("CoInitialize failed\n");
      }

      return 0;
   }


   ////////////////////////////////////////////////////////////////////
   //
   //  Displays the last error -- used to show the last exception
   //  information.
   //
   void DisplayError(_com_error& error)
   {
      printf("*** DisplayError()\n");

      printf("Fatal Unexpected Error:\n");
      printf("  Error Number = %08lx\n", error.Error());

      static char errorStr[1024];
      _bstr_t desc = error.Description();

      if (desc.length() == 0)
      {
```

```
        // Don't have a description string.
        strcpy(errorStr, error.ErrorMessage());
        int nLen = lstrlen(errorStr);

        // Remove funny carriage return ctrl<M>.
        if (nLen > 2 && (errorStr[nLen - 2] == 0xd))
        {
            errorStr[nLen - 2] = '\0';
        }
    }
    else
    {
        strcpy(errorStr, desc);
    }

    printf("  Error Message = %s\n", (char*) errorStr);
}
```

WriteAllImagesToFiles Method

[Automation Home (see page 3)] [Objects (see page 79)]

NOTE: This method is applicable only for those Logic and Protocol Analyzer modules that support the Image View.

Applies To ProtocolWindow (see page 104) object

Description Writes all the images in the Protocol Viewer's list to separate bitmap files in a specified folder.

VB Syntax Object.**WriteAllImagesToFiles** imageFolderPath errorMessage success

Parameters	Definition
object	An expression that evaluates a ProtocolWindow object.
imageFolderPath	A string that specifies the pathname of a folder where all the bitmap files are written.
errorMessage	A string output parameter. If success returns VARIANT_FALSE, then errorMessage contains information about the error that occurred. If success returns VARIANT_TRUE, then errorMessage is the empty string.
success	A **VARIANT_BOOL** output parameter which returns VARIANT_TRUE or VARIANT_FALSE. VARIANT_FALSE indicates an error occurred and errorMessage contains an explanation of this error.

Remarks **WriteAllImagesToFiles** performs the same function as the 'Save All Images...' button in the 'Image View' tab.

Each image in the Protocol Viewer's list is identified with a Frame Number string, e.g. "0-2". **WriteAllImagesToFiles** constructs a filename for each image using the Frame Number string with the suffix ".bmp". For example, image "0-2" would be stored in a file named "0-2.bmp".

When writing files and constructing filenames, if a file with the same name already exists in the specified directory, **WriteAllImagesToFiles** overwrites the existing file with a new file.

WriteAllImagesToFiles Example

Visual C++

NOTE: The example below is incomplete. The example does not contain code which acquires data. In order to work, the code in the example must have packet data in acquisition memory. It is also required that the packet data must contain some image frames.

```
// Create the Connect object.

AgtLA::IConnectPtr pConnect =
AgtLA::IConnectPtr(__uuidof(AgtLA::Connect));…

// Using the Connect object, obtain the IInstrument interface.

AgtLA::IInstrumentPtr pInst = pConnect->GetInstrument(hostName);…

// Using the IInstrument interface, obtain the IWindows interface

AgtLA::IWindowsPtr windows = pInst->GetWindows();…

// Loop through all the windows in the instrument looking for the first
window

// that presents an IProtocolWindow interface.

AgtLA::IprotoWindowPtr protocolWindow = NULL;

for (long i=0; i < windows->GetCount(); i++)

{

  // Using the IWindows interface, obtain a IWindow interface.

  AgtLA::IWindowPtr window = windows->GetItem(_variant_t(i));


  // Check if this window is a ProtocolWindow by type-casting the

  // module pointer.

  protocolWindow= windows->GetItem(_variant_t(i));

  if(protocolWindow != NULL)

  {

    // Found a ProtocolWindow

    break;

  }

}

if (protocolWindow == NULL)

{

  QUIT - There are no Protocol Windows in this Logic Analzyer

}


// Write All images to bitmap files in a folder

//
```

```
// Parameter to specify a folder

_bstr_t  bstrPath = ".";  // current working directory

BSTR bstrErrorMessage;


// Call the WriteAllImagesToFiles COM method

bSuccess = protocolWindow->WriteAllImagesToFiles(bstrPath,
&bstrErrorMessage);

// Check result

If (bSuccess == VARIANT_FALSE)

{

  Display error message in bstrErrorMessage and Quit

}
```

WriteImageToFile Method

[Automation Home (see page 3)] [Objects (see page 79)]

NOTE: This method is applicable only for those Logic and Protocol Analyzer modules that support the Image View.

Applies To ProtocolWindow (see page 104) object

Description Writes a single image to a bitmap file. The image is selected using the 'Frame Number' string from the image list.

VB Syntax object.**WriteImageToFile** frameNumber bitmapFilePath errorMessage success

Parameters	Definition
object	An expression that evaluates a ProtocolWindow object.
frameNumber	A string that specifies the image to be written. A Frame Number string consists of a virtual channel number and an image number separated by a dash (-). For example, a Frame Number string might be “0-1”. The frame number string must exactly match the Frame Number string for one of the images in the Protocol Viewer's list of images.
bitmapFilePath	A string that specifies the pathname for the bitmap file.
errorMessage	A string output parameter. If success returns VARIANT_FALSE, then errorMessage contains information about the error that occurred. If success returns VARIANT_TRUE, then errorMessage is the empty string.
success	A **VARIANT_BOOL** output parameter which returns VARIANT_TRUE or VARIANT_FALSE. VARIANT_FALSE indicates an error occurred and errorMessage contains an explanation of this error.

Remarks **WriteImageToFile** performs the same function as the 'Show Image...' button in the 'Image View' tab followed by the 'Save Image' button in the 'Extracted Image' window.

If the file specified by bitmapFilePath already exists, **WriteImageToFile** overwrites the existing file with a new file.

WriteImageToFile Example

Visual C++

NOTE: The example below is incomplete. The example does not contain code which acquires data. In order to work, the code in the example must have packet data in acquisition memory. It is also required that the packet data must contain some image frames.

```
// Create the Connect object.

AgtLA::IConnectPtr pConnect =
AgtLA::IConnectPtr(__uuidof(AgtLA::Connect));…

// Using the Connect object, obtain the IInstrument interface.

AgtLA::IInstrumentPtr pInst = pConnect->GetInstrument(hostName);…

// Using the IInstrument interface, obtain the IWindows interface

AgtLA::IWindowsPtr windows = pInst->GetWindows();…

// Loop through all the windows in the instrument looking for the first
window

// that presents an IProtocolWindow interface.

AgtLA::IprotoWindowPtr protocolWindow = NULL;

for (long i=0; i < windows->GetCount(); i++)

{

  // Using the IWindows interface, obtain a IWindow interface.

  AgtLA::IWindowPtr window = windows->GetItem(_variant_t(i));


  // Check if this window is a ProtocolWindow by type-casting the

  // module pointer.

  protocolWindow= windows->GetItem(_variant_t(i));

  if(protocolWindow != NULL)

  {

    // Found a ProtocolWindow

    break;

  }

}

if (protocolWindow == NULL)

{

  QUIT - There are no Protocol Windows in this Logic Analzyer

}

// Write a single image to a bitmap file

//
```

```
// Parameters to specify a Frame Number string and a bitmap file path

_bstr_t bstrFrameNumber = "0-0";

_bstr_t  bstrPath = "Image.bmp";

BSTR bstrErrorMessage;

// Call the WriteImageToFile COM method

bSuccess = protocolWindow->WriteImageToFile(bstrFrameNumber, bstrPath,
&bstrErrorMessage);

// Check result

If (bSuccess == VARIANT_FALSE)

{

  Display error message in bstrErrorMessage and Quit

}
```

WriteProtocolDataFieldsToFile Method

[Automation Home (see page 3)] [Objects (see page 79)]

Applies To

- ProtocolWindow object (see page 104)

Description

Writes the acquisition data displayed in various fields in the Protocol Viewer to a specified CSV file.

VB Syntax

object.WriteProtocolDataFieldsToFile start end channel csvfile success

Parameters	Definition
object	An expression that evaluates to a ProtocolWindow (see page 104) object.
start	A **VARIANT** which specifies the first packet in a range of packets for which the acquisition data is to be written to the specified file. If start is an integer, then start is interpreted as a packet number. If start is a floating point number, then start is interpreted as a packet time.
end	A **VARIANT** which specifies the last packet in a range of packets for which the acquisition data is to be written to the specified file. If end is an integer, then end is interpreted as a packet number. If end is a floating point number, then end is interpreted as a packet time. ***NOTE***: "start" and "end" must either be both integers (i.e. packets numbers) or be both floating point numbers (i.e. packet times).

Parameters	Definition
channel	If the value for this string is specified as *all*, then all the acquired packets data is written to the specified CSV file. *start* and *end* parameters are then ignored. Else, this string specifies the channel (or direction) of the sample numbers in the *start* and *end* parameters. *channel* is used only if *start* and *end* are integers representing packet numbers. *channel* is ignored if *start* and *end* are floating point numbers representing packet times.
csvfile	A string that specifies the path and name of the CSV file to which the acquired data is to be written.
success	A **VARIANT_BOOL** output parameter which returns VARIANT_TRUE or VARIANT_FALSE. VARIANT_TRUE means the operation was successful and the return string contains data. VARIANT_FALSE means an error occurred and the return string contains an error message.

Remarks

WriteProtocolDataFieldsToFile writes the acquired data to a CSV file in one or more lines. The first line in the file is a heading. Subsequent lines contain acquisition data. For each packet in the specified range of packets, there is one acquisition data line. Each data line contains a series of values separated by commas. This series of values is as per the fields displayed in the Packets pane of the Protocol Viewer. If you want to choose the fields that the method writes in a data line, you must insert, delete, or re-arrange the fields according to your requirements in the Packets pane of the Protocol Viewer.

Properties

- SelfTest Property (see page 211)
- Setup Property (see page 211)
- Slot Property (see page 212)
- StartSample Property (see page 212)
- StartTime Property (see page 212)
- Status Property (see page 213)
- StatusMsg Property (see page 213)
- SubrowFound Property (see page 214)
- Symbols Property (see page 214)
- TargetControlPort Property (see page 215)
- TextColor Property (see page 215)
- TimeFound Property (see page 216)
- TimeFoundString Property (see page 216)
- Tools Property (see page 216)
- Trigger Property (see page 217)
- Type Property (see page 217)
- Value Property (see page 218)
- VBE Property (see page 218)
- Version Property (see page 219)
- Windows Property (see page 219)
- _NewEnum Property (see page 219)

Activity Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- BusSignal (see page 81) object

Description

Gets the activity indicators of the bus/signal.

VB Syntax

object.Activity

Parameters	Definition
object	An expression that evaluates to a **BusSignal (see page 81)** object.

Remarks

The Activity property has the String type.

There is an activity indicator character for each signal, which can be:

- - (dash) – no activity
- H – activity level high
- L – activity level low
- T – activity level transition
- B – activity level is both high and low

BackgroundColor Property

[Automation Home (see page 3)] [Objects (see page 79)]

Applies To

- Marker (see page 95) object

Description Gets or sets the marker background color.

```
' Display the marker background color.
Dim myBackgroundColor As Long
myBackgroundColor = myMarker.BackgroundColor
MsgBox Str(myBackgroundColor)

' Set the marker background color to green.
myBackgroundColor = &H0000FF00
myMarker.BackgroundColor = myBackgroundColor
```

VB Syntax object.BackgroundColor [=Color]

Parameters	Definition
object	An expression that evaluates to a Marker (see page 95) object.
Color	A **Long** value that is the marker background color.

Remarks The BackgroundColor property has the Long type.

Color values have the following hexadecimal form: 0x00BBGGRR. The low-order byte (RR) contains a value for the relative intensity of red; the second byte (GG) contains a value for green; and the third byte (BB) contains a value for blue. The high-order byte must be zero. The maximum value for a single byte is &HFF. The color white is &H00FFFFFF, black is &H00000000, and red is &H000000FF.

BitSize Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- BusSignal (see page 81) object

Description Gets the number of channels in the bus/signal.

VB Syntax object.BitSize

Parameters	Definition
object	An expression that evaluates to an **BusSignal (see page 81)** object.

Remarks The BitSize property has the Long type.

See Also

- ByteSize (see page 194) property

BusSignalData Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 105)]

Applies To

- BusSignal (see page 81) object

Description Gets the data associated with a bus/signal.

VB Syntax object.BusSignalData

Parameters	Definition
object	An expression that evaluates to a BusSignal (see page 81) object.

Remarks The BusSignalData property has the BusSignalData (see page 81) object type.

BusSignalType Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 105)]

Applies To

- BusSignal (see page 81) object

Description Gets the type of bus/signal.

VB Syntax object.BusSignalType

Parameters	Definition
object	An expression that evaluates to a BusSignal (see page 81) object.

Remarks The BusSignalType property can have the following values:

AgtBusSignalType	Enum Value	Description
AgtBusSignalProbed	1	The bus/signal is associated with a physically probed connection.
AgtBusSignalGenerated	2	The bus/signal is generated by a tool (like an inverse assembler) and therefore does not have a physically probed connection. This type is also used for an external oscilloscope because its bus/signals are not physically probed directly by the logic analyzer.
AgtBusSignalSampleNum	3	Represents the sample number.
AgtBusSignalTime	4	Represents the sample's absolute time. This is obsolete. The "Time" data is no longer returned as a bus/signal. To get the time associated with bus/signal data, use the GetTime (see page 158) method of the SampleBusSignalData (see page 104) object.

The Activity (see page 191), Channels (see page 195), and Polarity (see page 207) properties are only valid when the BusSignalType property is AgtBusSignalProbed.

BusSignalDifferences Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 82)]

Applies To

- SampleDifference (see page 114) object

Description Gets a collection of all the buses/signals with differences for this sample.

VB Syntax object.BusSignalDifferences

Parameters	Definition
object	An expression that evaluates to a SampleDifference (see page 114) object.

Remarks The BusSignalDifferences property has the BusSignalDifferences (see page 82) collection object type. Each item in the collection is a BusSignalDifference (see page 81) object.

BusSignals Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 105)]

Applies To
- Module (see page 99) object
- Tool (see page 118) object
- Window (see page 122) object

Description Gets a collection of the module's defined buses/signals.

VB Syntax object.BusSignals

Parameters	Definition
object	An expression that evaluates to one of the objects in the **Applies to** line.

Remarks The BusSignals property has the BusSignals (see page 86) collection object type. Each item in the collection is a BusSignal (see page 81) object.

ByteSize Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 105)]

Applies To
- BusSignal (see page 81) object

Description Gets the size of the bus/signal in bytes.

VB Syntax object.ByteSize

Parameters	Definition
object	An expression that evaluates to an BusSignal (see page 81) object.

Remarks The ByteSize property has the Long type.

If the size of the bus is not a multiple of 8, it will be rounded to the next byte. For example, if a bus is 17 bits wide, 3 bytes will be returned. This property is useful when attempting to extract data from a byte array (raw). See the GetDataBySample (see page 144) method.

See Also
- BitSize (see page 192) property

CardModels Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To
- Module (see page 99) object

Description Gets the card model numbers.

VB Syntax object.CardModels

Parameters	Definition
object	An expression that evaluates to a Module (see page 99) object.

Remarks The CardModels property has the String type.

Channels Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- BusSignal (see page 81) object

Description Gets the channels defined in the bus/signal.

VB Syntax object.Channels

Parameters	Definition
Object	An expression that evaluates to a BusSignal (see page 81) object.
Channels	A **String** containing *MultiplePodChannels* or the **String** "*None*" if no channels are assigned. **MultiplePodChannels** contains a comma separated list of *PodChannels*. **PodChannels** contains a *PodNumber* followed by the **String** "*[*" followed by a comma separated list of individual channel *Number(s)* and *NumberRange(s)* followed by the **String** "*]*". ***Note:****Pod channels normally are in MSB to LSB notation unless you are trying to reorder channels.* **NumberRange** contains a *Number* followed by the **String** "**:**" followed by a *Number*. PodChannels Example: `Pod 1[9:7,5,3:1]` – this bus consists of 1 single channel *Number* and 2 *NumberRange*'s for a total of 7 channels. They are Pod 1[9], Pod 1[8], Pod 1[7], Pod 1[5], Pod 1[3], Pod 1[2], Pod 1[1]. MultiplePodChannels Example: `Pod 2[3,1],Pod 3[10,5:3]` – this bus consists of 3 single channel *Numbers* and 1 *NumberRange* for a total of 6 channels Pod 2[3], Pod 2[1], Pod 3[10], Pod 3[5], Pod 3[4], Pod 3[3].

Remarks The Channels property has the String type.

Comments Property

[Automation Home (see page 3)] [Objects (see page 79)]

Applies To

- Marker (see page 95) object

Description Gets or sets the marker comments.

VB Syntax object.Comments [=Comments]

Parameters	Definition
object	An expression that evaluates to a Marker (see page 95) object.
Comments	A **String** containing the comments for the marker.

Remarks The Comments property has the String type.

ComputerName Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- Frame (see page 92) object

Description

Gets a frame's computer name.

VB Syntax

object.ComputerName

-or-

object

Parameters	Definition
Object	An expression that evaluates to a **Frame** (see page 92) object. The ComputerName property is the default property of the **Frame** (see page 92) object. Accordingly, you do not have to reference ComputerName explicitly, as shown in the syntax.

Remarks

The ComputerName property has the String type.

Count Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 196)]

Applies To

- BusSignalDifferences (see page 82) object
- BusSignals (see page 86) object
- Frames (see page 93) object
- Markers (see page 95) object
- Modules (see page 100) object
- Probes (see page 101) object
- SampleDifferences (see page 115) object
- Tools (see page 118) object
- Windows (see page 122) object

Description

Gets the number of items in a collection.

VB Syntax

object.Count

Parameters	Definition
object	An expression that evaluates to one of the objects in the **Applies to** list above.

Remarks

The Count property has the Long type.

See Also

- Item (see page 203) property

Item and Count Example

The following example displays the channels for each bus/signal in the BusSignals collection:

Visual Basic

```
Dim myBusSignalChannels As String
Dim myBusSignal As AgtLA.BusSignal
For i = 0 To myInst.GetModuleByName("My 1690A-1").BusSignals.Count - 1
   Set myBusSignal = myInst.GetModuleByName("My 1690A-1").BusSignals(i)
```

```
   ' Add the bus/signal name and channels to the string.
   myBusSignalChannels = myBusSignalChannels + vbNewLine
   myBusSignalChannels = myBusSignalChannels + "Bus/signal: " + _
         myBusSignal.Name
   myBusSignalChannels = myBusSignalChannels + ", Channels: " + _
         myBusSignal.Channels
Next
MsgBox "Bus/signal names and channels: " + vbNewLine + _
      myBusSignalChannels
```

Visual C++

```
//
//  This simple Visual C++ Console application demonstrates how to
//  use the Keysight 168x/169x/169xx COM interface to display the
//  channels for all buses/signals.
//
//  This project was created in Visual C++ Developer.  To create a
//  similar project:
//
//    - Execute File -> New
//    - Select the Projects tab
//    - Select "Win32 Console Application"
//    - Select A "hello,World!" application (Visual Studio 6.0)
//
//  To make this buildable, you need to specify your "import" path
//  in stdafx.h (search for "TODO" in that file).  For example, add:
//  #import "C:/Program Files/Keysight Technologies/Logic Analyzer/LA \
//  COM Automation/agClientSvr.dll"
//
//  To run, you need to specify the host logic analyzer to connect
//  to (search for "TODO" below).
//

#include "stdafx.h"


////////////////////////////////////////////////////////////////////////
//
//  Forward declarations.
//
void DisplayError(_com_error& err);


////////////////////////////////////////////////////////////////////////
//
//  main() entry point.
//
int main(int argc, char* argv[])
{
   printf("*** Main()\n");

   //
   //  Initialize the Microsoft COM/ActiveX library.
   //
   HRESULT hr = CoInitialize(0);

   if (SUCCEEDED(hr))
   {
```

```
      try {   // Catch any unexpected run-time errors.
         _bstr_t hostname = "mtx33";   // TODO, use your logic
                                       // analysis system hostname.
         printf("Connecting to instrument '%s'\n", (char*) hostname);

         // Create the connect object and get the instrument object.
         AgtLA::IConnectPtr pConnect =
               AgtLA::IConnectPtr(__uuidof(AgtLA::Connect));
         AgtLA::IInstrumentPtr pInst =
               pConnect->GetInstrument(hostname);

         // Load the configuration file.
         _bstr_t configFile = "C:\\LA\\Configs\\config.ala";
         printf("Loading the config file '%s'\n", (char*) configFile);
         pInst->Open(configFile, FALSE, "", TRUE);

         // Display the channels for all probed bus/signal.
         _bstr_t moduleName = "MPC860 Demo Board";
         _bstr_t busSignal;
         _bstr_t channels;

         AgtLA::IAnalyzerModulePtr pAnalyzer =
               pInst->GetModuleByName(moduleName);
         AgtLA::IBusSignalsPtr pBusSignals = pAnalyzer->GetBusSignals();

         for (long i = 0; i < pBusSignals->GetCount(); i++)
         {
            AgtLA::IBusSignalPtr pBusSignal = pBusSignals->GetItem(i);
            AgtLA::AgtBusSignalType busSignalType =
                  pBusSignal->GetBusSignalType();

            if (busSignalType == AgtLA::AgtBusSignalProbed) {

               busSignal = pBusSignal->GetName();
               channels = pBusSignal->GetChannels();
               printf("Bus/signal '%s', channels '%s'.\n",
                     (char*) busSignal, (char*) channels);
            }
         }
      }
      catch (_com_error& e) {
         DisplayError(e);
      }

      // Uninitialize the Microsoft COM/ActiveX library.
      CoUninitialize();
   }
   else
   {
      printf("CoInitialize failed\n");
   }

   return 0;
}


//////////////////////////////////////////////////////////////////////
```

```
//
//  Displays the last error -- used to show the last exception
//  information.
//
void DisplayError(_com_error& error)
{
   printf("*** DisplayError()\n");

   printf("Fatal Unexpected Error:\n");
   printf("  Error Number = %08lx\n", error.Error());

   static char errorStr[1024];
   _bstr_t desc = error.Description();

   if (desc.length() == 0)
   {
      // Don't have a description string.
      strcpy(errorStr, error.ErrorMessage());
      int nLen = lstrlen(errorStr);

      // Remove funny carriage return ctrl<M>.
      if (nLen > 2 && (errorStr[nLen - 2] == 0xd))
      {
         errorStr[nLen - 2] = '\0';
      }
   }
   else
   {
      strcpy(errorStr, desc);
   }

   printf("  Error Message = %s\n", (char*) errorStr);
}
```

CreatorName Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- BusSignal (see page 81) object

Description

Gets the name of the module, tool, or viewer that created this bus/signal.

VB Syntax

object.CreatorName

Parameters	Definition
object	An expression that evaluates to a BusSignal (see page 81) object.

Remarks

The CreatorName property has the String type.

DataType Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- SampleBusSignalData (see page 104) object

Description Gets the recommended bus/signal data type.

Note that, while this is the recommended data type, other data types can be used for uploading the data.

VB Syntax object.DataType

Parameters	Definition
object	An expression that evaluates to a SampleBusSignalData (see page 104) object.

Remarks The DataType property can have values defined by the AgtDataType enumerated type. See DataTypes and Return Values (see page 144).

See Also
- BusSignalType (see page 193) property

Description Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To
- Frame (see page 92) object
- Module (see page 99) object

Description Gets a description of the logic analyzer frame or module.

VB Syntax object.Description

Parameters	Definition
object	An expression that evaluates to a Frame (see page 92) or Module (see page 99) object.

Remarks The Description property has the String type.

There is no defined format for the description string; therefore, do not parse this string to extract specific information.

Differences Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To
- CompareWindow (see page 90) object

Description Gets the number of differences found on the last comparison.

VB Syntax object.Differences

Parameters	Definition
object	An expression that evaluates to a CompareWindow (see page 90) object.

Remarks The Differences property has the Long type.

A comparison can be done directly by calling the Execute (see page 134) method or indirectly when its input data changes and comparisons are enabled in the user interface.

See Also
- Execute (see page 134) method

EndSample Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- SampleBusSignalData (see page 104) object

Description

Gets the data's ending sample number relative to trigger.

VB Syntax

object.EndSample

Parameters	Definition
object	An expression that evaluates to an SampleBusSignalData (see page 104) object.

Remarks

The EndSample property has the Long type.

See Also

- StartSample (see page 212) property

EndTime Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- SampleBusSignalData (see page 104) object

Description

Gets the data's ending time (in seconds) relative to trigger.

VB Syntax

object.EndTime

Parameters	Definition
object	An expression that evaluates to an SampleBusSignalData (see page 104) object.

Remarks

The EndTime property has the Double type.

See Also

- StartTime (see page 212) property

Found Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- FindResult (see page 92) object

Description

Gets a Boolean value indicating whether the event was found.

VB Syntax

object.Found

Parameters	Definition
object	An expression that evaluates to a FindResult (see page 92) object.

Remarks

The Found property has the Boolean type.

Frame Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To
- Module (see page 99) object

Description Gets the frame in which the module resides.

VB Syntax object.Frame

Parameters	Definition
object	An expression that evaluates to a Module (see page 99) object.

Remarks The Frame property has the Frame (see page 92) object type.

See Also
- Frame (see page 92) object
- Frames (see page 93) object

Frames Property

[Automation Home (see page 3)] [Objects (see page 79)]

Applies To
- Instrument (see page 93) object

Description Gets a collection of all logic analyzer frames connected via the multiframe connector.

VB Syntax object.Frames

Parameters	Definition
object	An expression that evaluates to an Instrument (see page 93) object.

Remarks The Frames property has the Frames (see page 93) collection object type. Each item in the collection is a Frame (see page 92) object.

Instrument Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To
- Connect (see page 90) object

Description Gets the logic analyzer instrument object.

VB Syntax object.Instrument [HostNameOrIpAddress=""]

Parameters	Definition
object	An expression that evaluates to a Connect (see page 90) object.
HostNameOrIpAddress	A **String** that contains the hostname or IP address of the logic analyzer instrument or computer on which the *Keysight Logic Analyzer* application runs. This parameter is optional–if it is not specified, the computer name specified in the Distributed COM "Location" tab is used (see To verify remote computer Distributed COM properties (see page 19)).

Remarks The Instrument property has the Instrument (see page 93) object type.

IPAddress Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- Frame (see page 92) object

Description

Gets a frame's IP address(es). If the frame has more than one LAN card, a comma separated list of IP addresses will be returned.

VB Syntax

object.IPAddress

Parameters	Definition
Object	An expression that evaluates to a **Frame** (see page 92) object.

Remarks

The IPAddress property has the String type.

Item Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 196)]

Applies To

- BusSignalDifferences (see page 82) object
- BusSignals (see page 86) object
- Frames (see page 93) object
- Markers (see page 95) object
- Modules (see page 100) object
- Probes (see page 101) object
- SampleDifferences (see page 115) object
- Tools (see page 118) object
- Windows (see page 122) object

Description

Gets one of the objects in a collection given either an index or a name.

VB Syntax

object.Item [IndexOrName]

-or-

object IndexOrName

Parameters	Definition
Object	An expression that evaluates to one of the objects in the Applies To list above.
IndexOrName	A **Variant** that is a **Long** or **String** representing the zero-based index in the collection or the name of object.

Object	String Index
BusSignalDifferences (see page 82)	The BusSignalDifference's index.
BusSignals (see page 86)	The BusSignal's name (as defined by the Name (see page 205) property).
Frames (see page 93)	The Frame's computer name or IP address (as defined by the ComputerName (see page 196) or IPAddress (see page 202) properties, respectively).
Markers (see page 95)	The Marker's name (as defined by the Name (see page 205) property).
Modules (see page 100)	The Module's name (as defined by the Name (see page 205) property).

Object	String Index
Probes (see page 101)	The Probe's name (as defined by the Name (see page 205) property).
SampleDifferences (see page 115)	The SampleDifference's index.
Tools (see page 118)	The Tool's name (as defined by the Name (see page 205) property).
Windows (see page 122)	The Window's name (as defined by the Name (see page 205) property).

Remarks The Item property has the appropriate type of the object in the collection (BusSignalDifference (see page 81), BusSignal (see page 81), Frame (see page 92), Marker (see page 95), Module (see page 99), Probe (see page 100), SampleDifference (see page 114), Tool (see page 118), or Window (see page 122)).

If you specify numbers for *index*, do not store these for later use because the indices might change as items are added or removed.

The Item property is the default. Accordingly, you don't have to reference Item explicitly, as shown in the syntax.

Markers Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 96)]

Applies To

- Instrument (see page 93) object

Description Gets a collection of all markers.

VB Syntax object.Markers

Parameters	Definition
object	An expression that evaluates to an Instrument (see page 93) object.

Remarks The Markers property has the Markers (see page 95) collection object type. Each item in the collection is a Marker (see page 95) object.

When this property is called, a snapshot of the current markers are returned. If markers are subsequently added or deleted, this property must be called again to get an updated snapshot.

Model Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- Instrument (see page 93) object
- Module (see page 99) object

Description Gets the model number of an object.

VB Syntax object.Model

Parameters	Definition
object	An expression that evaluates to a Module (see page 99) object.

Remarks The Model property has the String type.

The following table summarizes the results of using the Model property with the objects in the Applies To list:

Object	Results
Instrument (see page 93)	Gets the instrument's model number.
Module (see page 99)	Gets the module's model number. If the module consists of different card model numbers, they are separated with spaces. For example, an analyzer model would look like "16756A"; a two-card analyzer module containing two different card types would look like "16750A 16750B".

Modules Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- Instrument (see page 93) object

Description

Gets a collection of all the enabled hardware modules in the instrument.

VB Syntax

object.Modules

Parameters	Definition
object	An expression that evaluates to an Instrument (see page 93) object.

Remarks

The Modules property has the Modules (see page 100) collection object type. Each item in the collection is a Module (see page 99) object.

Name Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- BusSignal (see page 81) object
- BusSignalDifference (see page 81) object
- Marker (see page 95) object
- Module (see page 99) object
- Probe (see page 100) object
- Tool (see page 118) object
- Window (see page 122) object

Description

Gets or sets (where appropriate) the name of an object.

VB Syntax

object.Name [=NewName]

-or-

object

Parameters	Definition
Object	An expression that evaluates to one of the objects in the Applies To list above. The Name property is the default property of all objects in the list. Accordingly, you do not have to reference Name explicitly, as shown in the syntax.
NewName	A **String** containing the new name of the object.

Remarks The Name property has the String type.

The following table summarizes the results of using the Name property with some of the objects in the Applies To list:

Object	Results
BusSignal (see page 81)	Gets or sets the name of the bus/signal.
BusSignalDifference (see page 81)	Gets the bus/signal name associated with the bus/signal difference.
Marker (see page 95)	Gets or sets the name of the marker.
Module (see page 99)	Gets or sets the name of the module. This is the name displayed in the instrument's Overview window.
Probe (see page 100)	Gets or sets the name of the probe. This is the name displayed in the instrument's Overview window.
Tool (see page 118)	Gets or sets the name of the tool. This is the name displayed in the instrument's Overview window.
Window (see page 122)	Gets or sets the name of the window. This is the name displayed in the instrument's Overview window.

OccurrencesFound Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- FindResult (see page 92) object

Description Gets a Long containing the number of times the event was found.

VB Syntax object.OccurrencesFound

Parameters	Definition
object	An expression that evaluates to a FindResult (see page 92) object.

Remarks The OccurrencesFound property has the Long type.

Options Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 82)]

Applies To

- CompareWindow (see page 90) object

Description Gets or sets the Compare window's "XML-format" (in the online help) options specification.

VB Syntax object.Options [=Options]

Parameters	Definition
object	An expression that evaluates to a CompareWindow (see page 90) object.
Options	A **String** containing the "XML format" (in the online help)"<Options> element" (in the online help) information.

Remarks The Options property has the String type.

Overview Property

[Automation Home (see page 3)] [Objects (see page 79)]

Applies To

- Instrument (see page 93) object

Description

Gets an XML version of the Overview (see the "XML format" (in the online help)"<Overview> element" (in the online help)).

```
' Display the instrument's XML-format overview specification.
Dim myOverview As String
myOverview = myInst.Overview
MsgBox myOverview
```

VB Syntax

object.Overview

Parameters	Definition
object	An expression that evaluates to an Instrument (see page 93) object.

Remarks

The Overview property has the String type.

PanelLocked Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 168)]

Applies To

- Instrument (see page 93) object

Description

Indicates whether the instrument's front panel is locked. If locked, it returns the displayed message.

VB Syntax

object.PanelLocked Message

Parameters	Definition
object	An expression that evaluates to an Instrument (see page 93) object.
Message	If the font panel is locked, a **String** containing the displayed message is returned.

Remarks

The PanelLocked property has the Boolean type.

Polarity Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- BusSignal (see page 81) object

Description

Gets the polarity of the bus/signal.

VB Syntax

object.Polarity

Parameters	Definition
object	An expression that evaluates to a BusSignal (see page 81) object.

Remarks

The Polarity property has the String type.

The polarity is either "+" or "-".

Position Property

[Automation Home (see page 3)] [Objects (see page 79)]

Applies To

- Marker (see page 95) object

Description

Gets or sets the marker's time position, in seconds, relative to the trigger.

```
' Display the marker time position.
Dim myPosition As Double
myPosition = myMarker.Position
MsgBox Str(myPosition)

' Set the marker time position.
myPosition = -30e-9
myMarker.Position = myPosition
```

VB Syntax

object.Position [=Time]

Parameters	Definition
object	An expression that evaluates to a Marker (see page 95) object.
Time	A **Double** value that is the marker's time position, in seconds, relative to the trigger.

Remarks

The Position property has the Double type.

Probes Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 101)]

Applies To

- Instrument (see page 93) object

Description

Gets a collection of all currently defined probes.

VB Syntax

object.Probes

Parameters	Definition
object	An expression that evaluates to an Instrument (see page 93) object.

Remarks

The Probes property has the Probes (see page 101) collection object type. Each item in the collection is a Probe (see page 100) object.

When this property is called, a snapshot of the currently defined probes are returned. If probes are subsequently added or deleted, this property must be called again to get an updated snapshot.

Reference Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 82)]

Applies To

- BusSignalDifference (see page 81) object

Description

Gets the reference buffer value associated with the bus/signal difference.

VB Syntax

object.Reference

Parameters	Definition
object	An expression that evaluates to a BusSignalDifference (see page 81) object.

Remarks The Reference property has the String type.

The value is in base hex, for example: "FF".

RemoteComputerName Property

[Automation Home (see page 3)] [Objects (see page 79)]

Applies To

- Instrument (see page 93) object

Description Gets or sets the remote computer name.

```
' Display the remote user computer name.
Dim myRemoteComputerName As String
myRemoteComputerName = myInst.RemoteComputerName
MsgBox Str(myRemoteComputerName)

' Set the remote user computer name.
myRemoteComputerName = "myComputer"
myInst.RemoteComputerName = myRemoteComputerName
```

VB Syntax object.RemoteComputerName [=RemoteComputerName]

Parameters	Definition
object	An expression that evaluates to a Instrument (see page 93) object.
RemoteComputerName	A **String** value that is the name of the remote computer.

Remarks The RemoteComputerName property has the String type.

RemoteUserName Property

[Automation Home (see page 3)] [Objects (see page 79)]

Applies To

- Instrument (see page 93) object

Description Gets or sets the remote user login name.

```
' Display the remote user login name.
Dim myRemoteUserName As String
myRemoteUserName = myInst.RemoteUserName
MsgBox Str(myRemoteUserName)

' Set the remote user login name.
myRemoteUserName = "myLogin"
myInst.RemoteUserName = myRemoteUserName
```

VB Syntax object.RemoteUserName [=RemoteUserName]

Parameters	Definition
object	An expression that evaluates to a Instrument (see page 93) object.
RemoteUserName	A **String** value that is the login name of the remote user.

Remarks The RemoteUserName property has the String type.

RunningStatus Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- Module (see page 99) object

Description Gets the detailed running status of the module. Call this method when the Status (see page 213) property returns "Running" to get a more detailed running status.

VB Syntax object.RunningStatus

Parameters	Definition
object	An expression that evaluates to a Module (see page 99) object.

Remarks The RunningStatus property has the String type. The string returned is based on the object types defined below.

Object	Description
Module (see page 99)	**Running** - the module is running. **Stopped** - the module has stopped running. **Initializing** - the module is initializing or calibrating. **Waiting** - the module is waiting for an event. **SelfTest** - the instrument is running SelfTest. **Filling Memory** - the module is filling up memory.

See Also

- Status (see page 213) property

SampleDifferences Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 82)]

Applies To

- CompareWindow (see page 90) object

Description Gets a collection of all the samples with differences found in the last comparison.

VB Syntax object.SampleDifferences

Parameters	Definition
object	An expression that evaluates to a CompareWindow (see page 90) object.

Remarks The SampleDifferences property has the SampleDifferences (see page 115) collection object type. Each item in the collection is a SampleDifference (see page 114) object.

When this property is called, a snapshot of the current differences are returned. If another compare is executed, this property must be called again to get an updated snapshot.

SampleNum Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- SampleDifference (see page 114) object

Description Gets the sample number at which differences occurred.

VB Syntax object.SampleNum

Parameters	Definition
object	An expression that evaluates to a SampleDifferences (see page 115) object.

Remarks The SampleNum property has the Long type.

SelfTest Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 115)]

Applies To

- Instrument (see page 93) object

Description Gets the SelfTest object.

VB Syntax object.SelfTest

Parameters	Definition
object	An expression that evaluates to a Instrument (see page 93) object.

Remarks The SelfTest property has the SelfTest (see page 115) object type.

See Also

- SelfTest (see page 115) object

Setup Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 36)]

Applies To

- AnalyzerModule (see page 80) object

Description Gets or sets the logic analyzer's "XML-format" (in the online help) setup specification.

```
' Display the logic analyzer setup specification.
Dim mySetup As String
mySetup = myInst.GetModuleByName("My 1690A-1").Setup
MsgBox mySetup

' Set the logic analyzer setup specification.
myInst.GetModuleByName("My 1690A-1").Setup = mySetup
```

VB Syntax object.Setup [=XMLSetupSpec]

Parameters	Definition
object	An expression that evaluates to an AnalyzerModule (see page 80) object.
XMLSetupSpec	A **String** containing the "XML format" (in the online help)"<Module> element" (in the online help) information.

Remarks The Setup property has the String type.

Slot Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- Module (see page 99) object

Description Gets the module's slot location in the frame (see the Frame (see page 201) property). If the module is comprised of a single card, the letter corresponding to the slot is returned. Possible values are "A" through "F". If the module is comprised of a multi-card set, a string identifying all slots occupied by the module, highlighting the master card slot, is returned. Format: "<starting slot>-<ending slot>[m=<master slot>]". Example: "A-C[m=B]".

VB Syntax object.Slot

Parameters	Definition
object	An expression that evaluates to a Module (see page 99) object.

Remarks The Slot property has the String type.

See Also

- Frame (see page 92) object
- Frames (see page 93) object

StartSample Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- SampleBusSignalData (see page 104) object

Description Gets the data's starting sample number relative to trigger.

VB Syntax object.StartSample

Parameters	Definition
object	An expression that evaluates to an SampleBusSignalData (see page 104) object.

Remarks The StartSample property has the Long type.

The starting sample number is relative to trigger; therefore, a starting sample number equal to -1024 means the trigger is at sample 0, 1024 samples after the starting sample.

See Also

- EndSample (see page 201) property

StartTime Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- SampleBusSignalData (see page 104) object

Description

Gets the data's starting time (in seconds) relative to trigger.

VB Syntax

object.StartTime

Parameters	Definition
object	An expression that evaluates to an SampleBusSignalData (see page 104) object.

Remarks

The StartTime property has the Double type.

The starting time is relative to trigger and can therefore be a negative number.

See Also

- EndTime (see page 201) property

Status Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- Instrument (see page 93) object
- Module (see page 99) object

Description

Gets the run status. The Instrument object's Status property returns the run status of all data acquisition modules, tools, and windows. The Module object's Status property only returns the run status of that specific module.

NOTE

Instead of polling the **Status** property in a loop, use the object's **WaitComplete** (see page 180) method to wait for a measurement to complete.

When a module is running repetitively, either "Stopped" or "Running" can be returned based on the current state of the module. Use the Instrument object's Status property which will always return "Running" during a repetitive run.

VB Syntax

object.Status

Parameters	Definition
object	An expression that evaluates to one of the objects in the "Applies to" list above.

Remarks

The Status property has the String type. The string returned is based on the object types defined below.

Object	Description
Instrument (see page 93)	Running - at least one of the data acquisition modules, tools, or windows is running Stopped - data acquisition modules, tools, or windows have stopped running
Module (see page 99)	Running - the module is running Stopped - the module has stopped running

See Also

- RunningStatus (see page 210) property

StatusMsg Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- Module (see page 99) object

Description

Gets the module's verbose status message.

VB Syntax

object.StatusMsg

Parameters	Definition
object	An expression that evaluates to a Module (see page 99) object.

Return Value

A String containing the verbose status message.

NOTE

This message is not guaranteed to be static and, therefore, should not be parsed. Use only for display purposes.

If you want to know the specific status of a module, see the RunningStatus (see page 210) or Status (see page 213) properties.

See Also

- RunningStatus (see page 210) property
- Status (see page 213) property

SubrowFound Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- FindResult (see page 92) object

Description

Gets a Long containing the subrow number if the data sample contains subrows.

VB Syntax

object.SubrowFound

Parameters	Definition
object	An expression that evaluates to a FindResult (see page 92) object.

Remarks

The SubrowFound property has the Long type.

Symbols Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- BusSignal (see page 81) object

Description

Gets or sets the symbols associated with a bus/signal.

```
' Display a bus/signal's symbols.
Dim mySymbols As String
mySymbols = _
   myInst.GetModuleByName("My 1690A-1").BusSignals("My Bus 1").Symbols
MsgBox mySymbols
```

```
' Set a bus/signal's symbols.
mySymbols = "<Symbols><Symbol Name='My Symbol' Operator='Equals' " + _
   "Value='hFF'/></Symbols>"
myInst.GetModuleByName("My 1690A-1").BusSignals("My Bus 1").Symbols = _
   mySymbols
```

VB Syntax object.Symbols [=XMLSymbols]

Parameters	Definition
object	An expression that evaluates to a BusSignal (see page 81) object.
XMLSymbols	A **String** containing the "XML format" (in the online help)"<Symbols> element" (in the online help) information.

Remarks The Symbols property has the String type.

TargetControlPort Property

[Automation Home (see page 3)] [Objects (see page 79)]

Applies To

- Frame (see page 92) object

Description Gets or sets the target control port value. The port is 8-bit TTL and can be used to remotely control switches on your device under test.

VB Syntax object.TargetControlPort [=Value]

Parameters	Definition
object	An expression that evaluates to a Frame (see page 92) object.
Value	A **String** containing a port value. The value returned is the value present on the physical pins. The value can be set to a decimal, hexadecimal, octal, or binary number with optional don't care digits "x". The don't care digit is used to indicate that the value will stay the same (don't care). To specify a number base, use the following prefixes: ▪ h - for hexadecimal ▪ b - binary ▪ o - octal ▪ d - decimal For example: "hf", "b11110000", "bxxxx1xxx".

Remarks The TargetControlPort property has the String type.

TextColor Property

[Automation Home (see page 3)] [Objects (see page 79)]

Applies To

- Marker (see page 95) object

Description Gets or sets the marker text color.

```
' Display the marker text color.
Dim myTextColor As Long
myTextColor = myMarker.TextColor
MsgBox Str(myTextColor)

' Set the marker text color to red.
```

```
myTextColor = &H000000FF
myMarker.TextColor = myTextColor
```

VB Syntax object.TextColor [=Color]

Parameters	Definition
object	An expression that evaluates to a Marker (see page 95) object.
Color	A **Long** value that represents the marker text color.

Remarks The TextColor property has the Long type.

Color values have the following hexadecimal form: 0x00BBGGRR. The low-order byte (RR) contains a value for the relative intensity of red; the second byte (GG) contains a value for green; and the third byte (BB) contains a value for blue. The high-order byte must be zero. The maximum value for a single byte is &HFF. The color white is &H00FFFFFF, black is &H00000000, and red is &H000000FF.

TimeFound Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- FindResult (see page 92) object

Description Gets a Double containing the time (in seconds) the event occurred in the data. You can call the GetDataByTime (see page 146) method with this value to get more details.

VB Syntax object.TimeFound

Parameters	Definition
object	An expression that evaluates to a FindResult (see page 92) object.

Remarks The TimeFound property has the Double type.

TimeFoundString Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- FindResult (see page 92) object

Description Gets the time found as a string.

VB Syntax object.TimeFoundString

Parameters	Definition
object	An expression that evaluates to a FindResult (see page 92) object.

Remarks The TimeFoundString property has the String type.

Tools Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 119)]

Applies To

- Instrument (see page 93) object

Description Gets a collection of all active software tools.

VB Syntax object.Tools

Parameters	Definition
object	An expression that evaluates to an Instrument (see page 93) object.

Remarks The Tools property has the Tools (see page 118) collection object type. Each item in the collection is a Tool (see page 118) object.

When this property is called, a snapshot of the currently active tools are returned. If tools are subsequently added or deleted from the Overview, this property must be called again to get an updated snapshot.

Trigger Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 32)]

Applies To

- AnalyzerModule (see page 80) object

Description Gets or sets the logic analyzer's "XML-format" (in the online help) trigger specification.

```
' Display the logic analyzer trigger specification.
Dim myTrigger As String
myTrigger = myInst.GetModuleByName("My 1690A-1").Trigger
MsgBox myTrigger

' Set the logic analyzer trigger specification.
myInst.GetModuleByName("My 1690A-1").Trigger = myTrigger
```

VB Syntax object.Trigger [=XMLTriggerSpec]

Parameters	Definition
object	An expression that evaluates to an AnalyzerModule (see page 80) object.
XMLTriggerSpec	A **String** containing the "XML format" (in the online help)"<Trigger> element" (in the online help) information.

Remarks The Trigger property has the String type.

Type Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- BusSignalData (see page 81) object
- Module (see page 99) object
- Tool (see page 118) object
- Window (see page 122) object

Description Gets the BusSignalData (see page 81), Module (see page 99), Tool (see page 118), or Window (see page 122) object's type. The type can be used to identify sub-objects and their specific methods and properties.

VB Syntax object.Type

Parameters	Definition
object	An expression that evaluates to one of the objects in the "Applies to" list above.

Remarks

The Type property has the String type.

The following tables show how types correspond to objects.

BusSignalData Type	BusSignalData Object
Sample	SampleBusSignalData (see page 104)

Module Type	Module Object
Analyzer	AnalyzerModule (see page 80)
ExternalScope	Module (see page 99)
Import	Module (see page 99)

Window Type	Window Object
Compare	CompareWindow (see page 90)

Value Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 82)]

Applies To

- BusSignalDifference (see page 81) object

Description

Gets the data value associated with the bus/signal difference.

VB Syntax

object.Value

Parameters	Definition
object	An expression that evaluates to a BusSignalDifference (see page 81) object.

Remarks

The Value property has the String type.

The value is in base hex, for example: "FF".

VBE Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- Instrument (see page 93) object

Description

Gets the VBE extensibility object.

VB Syntax

object.VBE

Parameters	Definition
object	An expression that evaluates to an Instrument (see page 93) object.

Remarks The VBE property has the VBE object type.

Version Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example]

Applies To

- Instrument (see page 93) object

Description Gets the version number of the system software.

VB Syntax object.Version

Parameters	Definition
object	An expression that evaluates to an Instrument (see page 93) object.

Remarks The Version property has the String type.

The format of the version string is ##.##.#### (for example, "02.00.0000")

Windows Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 123)]

Applies To

- Instrument (see page 93) object

Description Gets a collection of all windows/viewers.

VB Syntax object.Windows

Parameters	Definition
object	An expression that evaluates to an Instrument (see page 93) object.

Remarks The Windows property has the Windows (see page 122) collection object type. Each item in the collection is a Window (see page 122) object.

When this property is called, a snapshot of the currently active windows are returned. If windows are subsequently added or deleted from the Overview, this property must be called again to get an updated snapshot.

_NewEnum Property

[Automation Home (see page 3)] [Objects (see page 79)] [Example (see page 220)]

Applies To

- BusSignalDifferences (see page 82) object
- BusSignals (see page 86) object
- Frames (see page 93) object
- Markers (see page 95) object
- Modules (see page 100) object
- Probes (see page 101) object
- SampleDifferences (see page 115) object
- Tools (see page 118) object
- Windows (see page 122) object

Description References objects in a collection.

VB Syntax *object*._NewEnum

Parameters	Definition
Object	With Visual C++, you can browse a collection to find a particular item by using the **_NewEnum** property or the Item (see page 203) property. In Visual Basic, you do not need to use the **_NewEnum** property because it is automatically used in the implementation of **For Each ... Next**.

Remarks The _NewEnum property has the appropriate type of the object in the collection (BusSignalDifference (see page 81), BusSignal (see page 81), Frame (see page 92), Marker (see page 95), Module (see page 99), SampleDifference (see page 114), Tool (see page 118), or Window (see page 122)).

With Visual C++, you can browse a collection to find a particular item by using the _NewEnum property or the Item (see page 203) property. In Visual Basic, you do not need to use the _NewEnum property because it is automatically used in the implementation of For Each ... Next.

_NewEnum Example

This Visual Basic example uses the _NewEnum property to iterate through all buses/signals and display's their names.

Visual Basic

```
' Display all of the bus/signal names.
Dim myBusSignals As AgtLA.BusSignals
Set myBusSignals = myInst.GetModuleByName("My 1690A-1").BusSignals

Dim myBusSignalNames As String
Dim myBusSignal As AgtLA.BusSignal

For Each myBusSignal in myBusSignals
   ' Add the bus/signal name to the string.
   myBusSignalNames = myBusSignalNames + vbNewLine + myBusSignal.Name
Next
MsgBox "Bus/signal names: " + myBusSignalNames
```

Visual C++

```
//
//  This simple Visual C++ Console application demonstrates how to
//  use the Keysight 168x/169x/169xx COM interface to iterate through
//  all buses/signals and display their names.
//
//  This project was created in Visual C++ Developer.  To create a
//  similar project:
//
//    - Execute File -> New
//    - Select the Projects tab
//    - Select "Win32 Console Application"
//    - Select A "hello,World!" application (Visual Studio 6.0)
//
//  To make this buildable, you need to specify your "import" path
//  in stdafx.h (search for "TODO" in that file).  For example, add:
//  #import "C:/Program Files/Keysight Technologies/Logic Analyzer/LA \
//  COM Automation/agClientSvr.dll"
//
//  To run, you need to specify the host logic analyzer to connect
```

```
//  to (search for "TODO" below).
//

#include "stdafx.h"


////////////////////////////////////////////////////////////////////////
//
//  Forward declarations.
//
void DisplayError(_com_error& err);


////////////////////////////////////////////////////////////////////////
//
//  main() entry point.
//
int main(int argc, char* argv[])
{
   printf("*** Main()\n");

   //
   //  Initialize the Microsoft COM/ActiveX library.
   //
   HRESULT hr = CoInitialize(0);

   if (SUCCEEDED(hr))
   {
      try {   // Catch any unexpected run-time errors.
         _bstr_t hostname = "mtx33";   // TODO, use your logic
                                       // analysis system hostname.
         printf("Connecting to instrument '%s'\n", (char*) hostname);

         // Create the connect object and get the instrument object.
         AgtLA::IConnectPtr pConnect =
               AgtLA::IConnectPtr(__uuidof(AgtLA::Connect));
         AgtLA::IInstrumentPtr pInst =
               pConnect->GetInstrument(hostname);

         // Load the configuration file.
         _bstr_t configFile = "C:\\LA\\Configs\\config.ala";
         printf("Loading the config file '%s'\n", (char*) configFile);
         pInst->Open(configFile, FALSE, "", TRUE);

         // Display all of the bus/signal names.
         _bstr_t moduleName = "MPC860 Demo Board";
         _bstr_t busSignal;

         AgtLA::IAnalyzerModulePtr pAnalyzer =
               pInst->GetModuleByName(moduleName);
         AgtLA::IBusSignalsPtr pBusSignals = pAnalyzer->GetBusSignals();

         for (long i = 0; i < pBusSignals->GetCount(); i++)
         {
            busSignal = pBusSignals->GetItem(i)->GetName();
            printf("Bus/signal name '%s'.\n", (char*) busSignal);
         }
```

```
      }
      catch (_com_error& e) {
         DisplayError(e);
      }

      // Uninitialize the Microsoft COM/ActiveX library.
      CoUninitialize();
   }
   else
   {
      printf("CoInitialize failed\n");
   }

   return 0;
}


//////////////////////////////////////////////////////////////////////
//
//  Displays the last error -- used to show the last exception
//  information.
//
void DisplayError(_com_error& error)
{
   printf("*** DisplayError()\n");

   printf("Fatal Unexpected Error:\n");
   printf("  Error Number = %08lx\n", error.Error());

   static char errorStr[1024];
   _bstr_t desc = error.Description();

   if (desc.length() == 0)
   {
      // Don't have a description string.
      strcpy(errorStr, error.ErrorMessage());
      int nLen = lstrlen(errorStr);

      // Remove funny carriage return ctrl<M>.
      if (nLen > 2 && (errorStr[nLen - 2] == 0xd))
      {
         errorStr[nLen - 2] = '\0';
      }
   }
   else
   {
      strcpy(errorStr, desc);
   }

   printf("  Error Message = %s\n", (char*) errorStr);
}
```

5 What's Changed

- ·The RunningStatus property now returns the status "Filling Memory" instead of the status "Unknown" when the module is filling up memory.

Index

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

Q

R

S

T

U

V

W