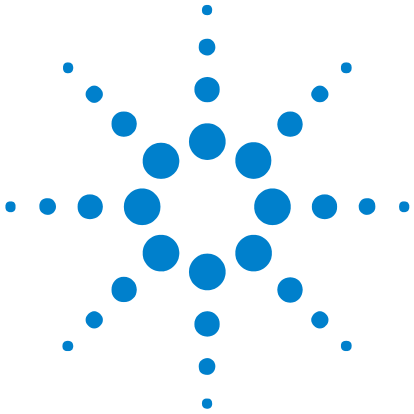


# Advanced Customization Environment (ACE)

## Online Help

Agilent Technologies

# Notices



## Trademarks

Microsoft®, MS-DOS®, Windows®, Windows 2000®, and Windows XP® are U.S. registered trademarks of Microsoft Corporation.

Adobe®, Acrobat®, and the Acrobat Logo® are trademarks of Adobe Systems Incorporated.

## Manual Part Number

Version 03.67.0000

## Edition

July 15, 2007

Available in electronic format only

Agilent Technologies, Inc.
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, Agilent 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. Agilent shall not be liable for errors or for incidental or consequential damages in connection with the furnishing, use, or performance of this document or of any information contained herein. Should Agilent 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 shall 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.

## Restricted Rights Legend

If software is for use in the performance of a U.S. Government prime contract or subcontract, Software is delivered and licensed as "Commercial computer software" as defined in DFAR 252.227-7014 (June 1995), or as a "commercial item" as defined in FAR 2.101(a) or as "Restricted computer software" as defined in FAR 52.227-19 (June 1987) or any equivalent agency regulation or contract clause. Use, duplication or disclosure of Software is subject to Agilent Technologies' standard commercial license terms, and non-DOD Departments and Agencies of the U.S. Government will receive no greater than Restricted Rights as defined in FAR 52.227-19(c)(1-2) (June 1987). U.S. Government users will receive no greater than Limited Rights as defined in FAR 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.**

# Using the Advanced Customization Environment (ACE)

- To distribute VBA project code via .zip files (see page 79)

- **Visual Basic Programming Tips** (see page 87)
  - Visual Basic Syntax (see page 88)
  - Guidelines for C++ Programmers (see page 90)
  - Common VBA Error Messages (see page 91)

**See Also**

- "COM Automation Reference" (in the online help)
- Help inside the VBA Integrated Development Environment (IDE) (for syntax or method calls)
- Web sites:
  - "http://groups.google.com" (get answers to VBA questions)
  - "http://www.msdn.com" (Microsoft knowledge base)
- Books:
  - *VBA Developer's Handbook* by Ken Getz & Mike Gilbert, 2nd Edition
  - *VBA for Dummies* by John Paul Mueller (Good reference for Forms)

**TIP** When you get a reference book, be sure the book is for Visual Basic for Applications (VBA). VBA is not the same as VB or VB.NET. Also, don't forget about the "F1" help.

# Contents

## 4 Analyzing Data in ACE



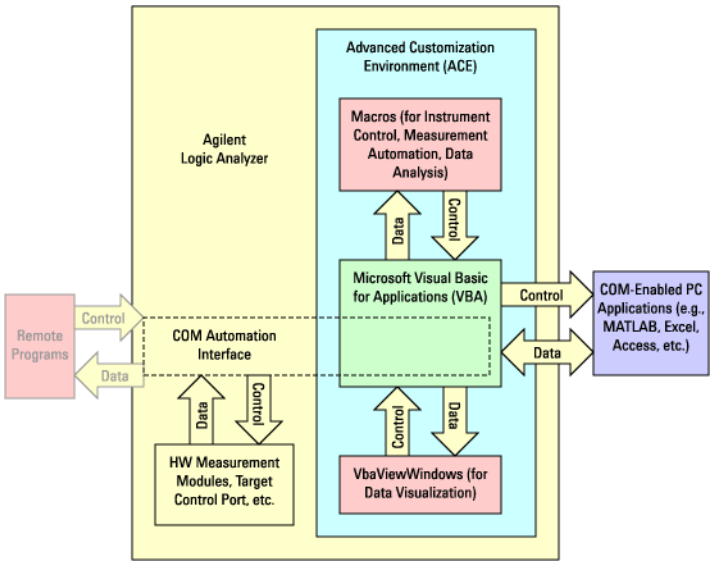
## 5 Displaying Data in VbaView Windows



## 6 Distributing VBA Code

# 1 Advanced Customization Environment (ACE)—At a Glance

The Advanced Customization Environment (ACE) is a seamless integration of Microsoft Visual Basic for Applications (VBA) into the *Agilent Logic Analyzer* application.




**Integrated Visual Basic for Applications (VBA)**

Visual Basic for Applications (VBA) is a fully established Microsoft application that provides an Integrated Development Environment (IDE) for *automating tasks* and *customizing an application*. Just as VBA is integrated into Microsoft Excel, Word, and Access (for example), it is now integrated into the *Agilent Logic Analyzer* application. VBA's seamless integration into the *Agilent Logic Analyzer* application shortens the learning curve and lets you perform automation and customization tasks in the same environment as you perform other logic analysis tasks.

**TIP** VBA is not the same as VB or VB.NET. It is the same VBA that is part of Excel.

**With the Advanced Customization Environment (ACE), You Can:**

- Control the instrument and automate measurements (see page 11).
- Add data analysis customization (see page 12).
- Add data visualization windows (see page 13).
- Link to other COM-enabled PC applications (see page 14).

**See Also**

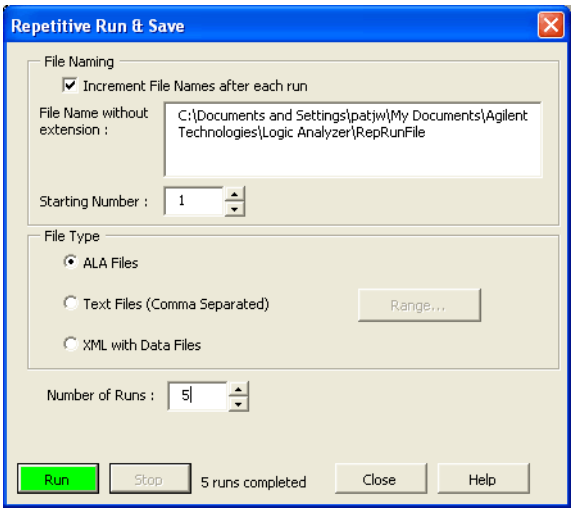
- VBA Macros and VbaView Windows (see page 15)

# Instrument Control and Measurement Automation

With the Advanced Customization Environment (ACE), you can create macros that control the instrument and automate measurements. For a quick example of a macro that runs the logic analyzer, saves the results, and runs the logic analyzer again:

1 In the *Agilent Logic Analyzer* application, choose **Tools>Run Macro>RepetitiveSaveToFileSample**.

2 In the Repetitive Run & Save dialog, select the desired options, and click **Run**.



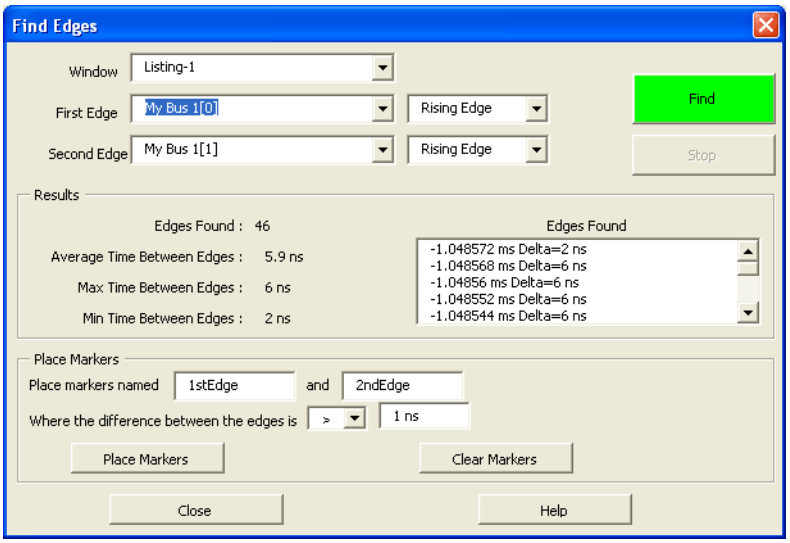

3 Click **Close** to close the macro dialog.

# Data Analysis

With the Advanced Customization Environment (ACE), you can create macros that analyze captured data. For example, a macro can run the logic analyzer and search the captured data for the occurrence of an event too complicated to be triggered on. For a quick example of a data analysis macro that finds the time between two edges and places markers on particular difference values:

1 Choose **Tools>Run Macro>FindEdgesSample**.

2 In the Find Edges dialog, select the desired options, and click **Find**.



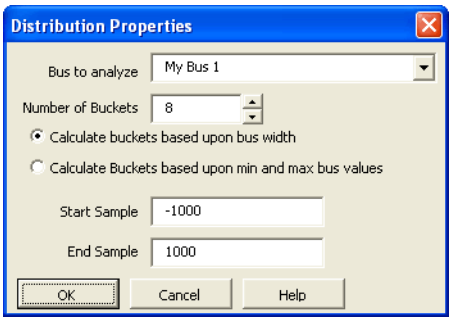

3 Click **Close** to close the macro dialog.

# Data Visualization

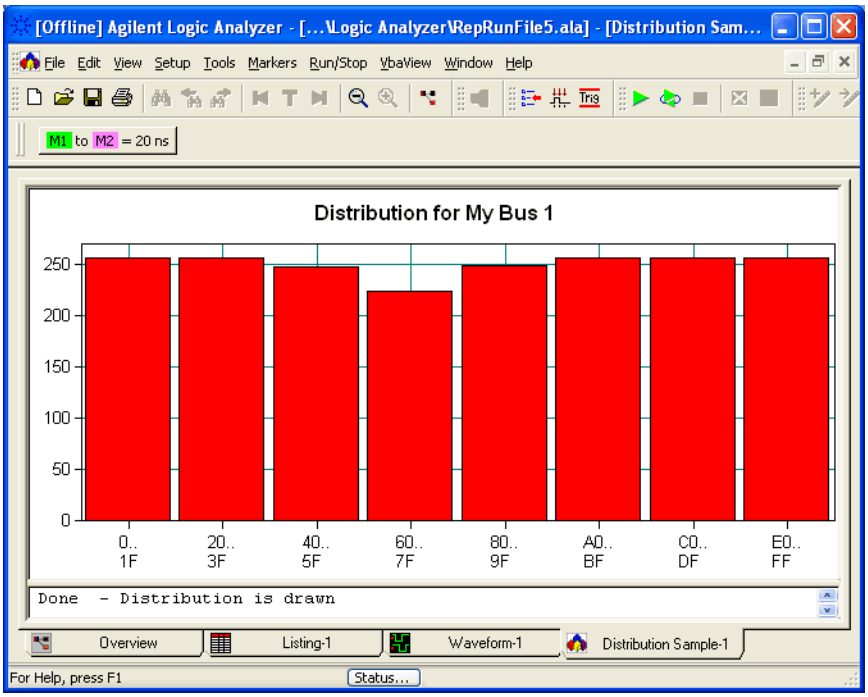
With the Advanced Customization Environment (ACE), you can add VbaView windows that help visualize captured data in XY scattergrams, bar charts, line charts, and pie charts. For a quick example of a VbaView window that creates a bar chart of the distribution of values on a bus:

1 Choose **Window>New VbaView>Distribution Sample...**.

2 In the Distribution Properties dialog, select the desired options, and click **OK**.




The distribution of values on the specified bus are shown in the VbaView window.

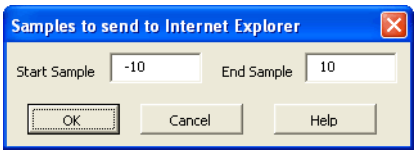


# Linking to Other COM-Enabled PC Applications

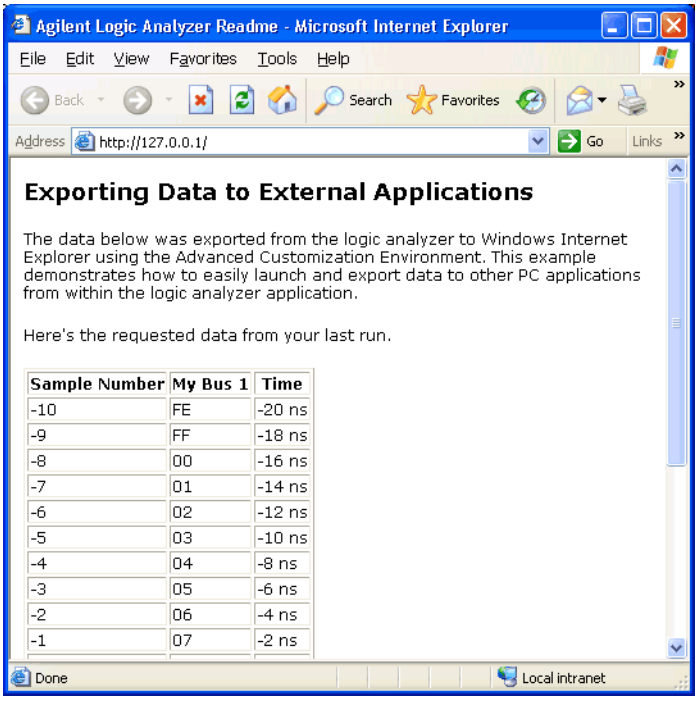
With the Advanced Customization Environment (ACE), you can create macros or add VbaView windows that interact with other COM-enabled PC applications. For a quick example of a VbaView window that sends captured data to an Internet Explorer window:

1 Choose **Window>New VbaView>Export to IE Sample...**.

2 In the "Samples to send to Internet Explorer" dialog, select the desired options, and click **OK**.




Logic analyzer data is sent to an Internet Explorer window.



# VBA Macros and VbaView Windows

**Macro = Script Used to Automate Tasks and Customize Analysis**

A macro:

- Is written in Microsoft Visual Basic.
- Can optionally create a custom dialog for user input.
- Is manually run from the menu or tool bar.
- Has no data visualization capabilities.

See the samples included with the *Agilent Logic Analyzer* application by choosing **Tools>Run Macro>**:

- **FindEdgesSample** (simple find time between edges and place markers example).
- **RepetitiveSaveToFileSample** (simple repetitive run and save example).

**VbaView = Macro + Visualization**

A VbaView is a window that:

- Has charting capabilities.
- Is integrated into the Overview window.
- Responds to logic analyzer events.

See the samples included with the *Agilent Logic Analyzer* application by choosing **Window>New VbaView>**:

- **Bus vs Bus Sample...** (simple XY scattergram chart example).
- **Distribution Sample...** (simple data distribution bar chart example).
- **Export to IE Sample...** (simple export data to another application example).
- **Hello World Sample...** (simple text output example).

**More on Macros**

A macro is a generic function that is run by: choosing **Tools>Macro>Macros...**, selecting the macro (in the Macros dialog), and clicking **Run**. (This Run button does *not* run the logic analyzer.) Macros are generally used to start a one-time data analysis or data export. A macro should be used when there is no need to graph data or respond to events in the logic analysis system (like when a measurement is run, for example). It is common for a macro to open a dialog (also known as a "user form") to prompt for user-selectable options or parameters before running the analysis or export.

A macro must contain at least one module with at least one public subroutine with no parameters. The name of the module does not have to be any specific name. All subroutines with no parameters appear in the Macros dialog (which appears when you choose **Tools>Macro>Macros...**). Subroutines that are private or have parameters do not appear in the Macros dialog. In general, you are not able to access VbaView window

functionality from the Macros dialog. (A public function with no parameters can be put in the AgtVbaView module if desired, but there generally isn't a need to do this).

**More on VbaView Windows**

A VbaView window has graphing and text display capabilities. It also has the ability to respond to events in the logic analysis system (like when a measurement is run or when the **VbaView>Properties...** command is chosen, for example). A VbaView window essentially provides a superset of the macro capabilities plus graphing and the ability to respond to events. Use a VbaView window whenever data is to be graphed or when text is to be displayed. VbaView windows generally have a properties dialog that lets you choose window options.

VbaView windows must contain a Notify function in the AgtVbaView module (see page 61). The name "Notify" *must* be used for the function and "AgtVbaView" *must* be used for the module. The Notify function is how the logic analysis system tells the VbaView window about events such as a run or when the **VbaView>Properties...** command is chosen. It is common for a VbaView window to have a user form called agtProperties although any name can be used.

# 2 Creating and Running Macros

In Microsoft Visual Basic for Applications (VBA), "*macros*" (in the online help) are public **Sub** procedures (with no parameters) that can be run from the user interface to automate an application.

Macros extend the functionality of an application. Macros can be used to perform functions within an application or even across applications. For example, a macro in the logic analyzer might cause an instance of Microsoft Excel to open and copy data from the logic analyzer.

In the logic analyzer, macros generally fit into one of the categories:

- Control — For example, a macro can run the logic analyzer, save the results, run the logic analyzer again, etc.
- Analyze — For example, a macro can run the logic analyzer and search the captured data for the occurrence of an event too complicated to be triggered on. If the event is not found, the logic analyzer can be run again until the event is found.
- Export — For example, macros can export logic analyzer data to Excel, Access, MathWorks MATLAB, the Agilent Vector Signal Analyzer, National Instruments LabVIEW, Agilent VEE, and SysStat. A macro can copy data from a logic analyzer directly into a pattern generator module.

There is another category of VBA program that is not created or run like macros:

- Graph — As a part of the added VBA functionality, graphing functions such as XY scattergrams, bar charts, line charts, and pie charts can be used in the VbaView window. See Displaying Data in VbaView Windows (see page 57).

To create and run macros, see:

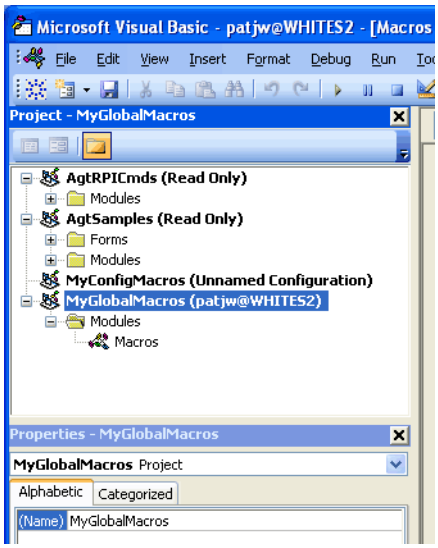
**See Also**

# Considerations When Creating a Macro

**TIP**

A macro needs to be stored into a Project.

A standalone macro can ONLY by stored in the MyConfigMacros or MyGlobalMacros projects. The other projects are either read only or are not appropriate.




**Config Macros**

- The MyConfigMarcos project is saved with the ALA or XML format configuration file (in other words, it is file specific).
- To access a macro in the MyConfigMacros project, you have to load the associated configuration file.

**Global Macros**

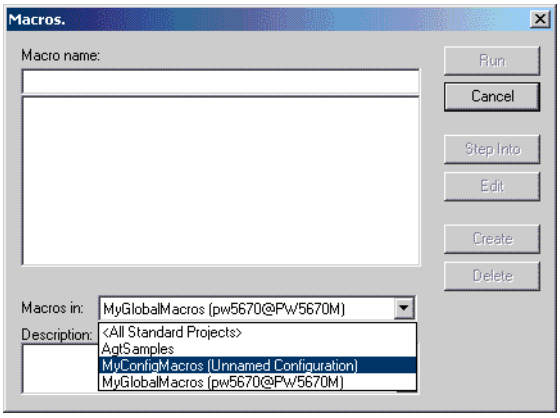
- The MyGlobalMacros project is saved on the logic analysis system or host PC, not in an ALA or XML format configuration file (in other words, it is specific to the system).
- Macros in the MyGlobalMacros project are available with any setup/configuration.

# Creating a New Macro

1 Run the *Agilent Logic Analyzer* application.

2 Choose **Tools>Macro>Macros...** or press the ALT+F8 keyboard shortcut.

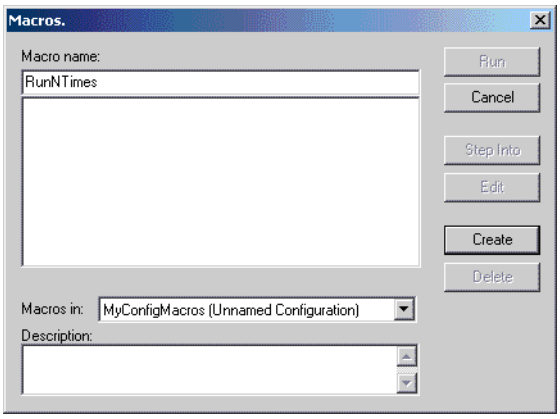
3 In the Macros dialog, use the **Macros in** drop-down to select the project in which the macro should be created.



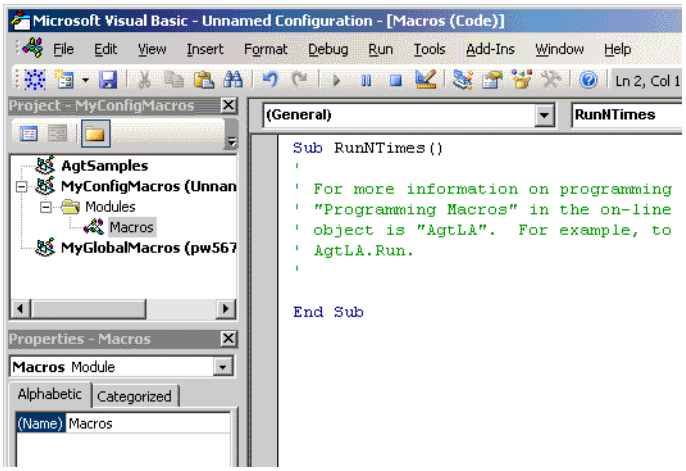

Choose between:

- MyConfigMacros — the macro will be saved with the ALA or XML format logic analyzer configuration file.
- MyGlobalMacros — the macro will be saved in "My Documents\Agilent Technologies\Logic Analyzer\Vba Files\MyGlobalMacros.alv" when you exit the *Agilent Logic Analyzer* application and loaded automatically the next time you start the application.

4 In the Macros dialog, enter the desired macro name, and click **Create**.




5 In the VBA IDE, enter the program (see Notes on Programming Macros (see page 30)).



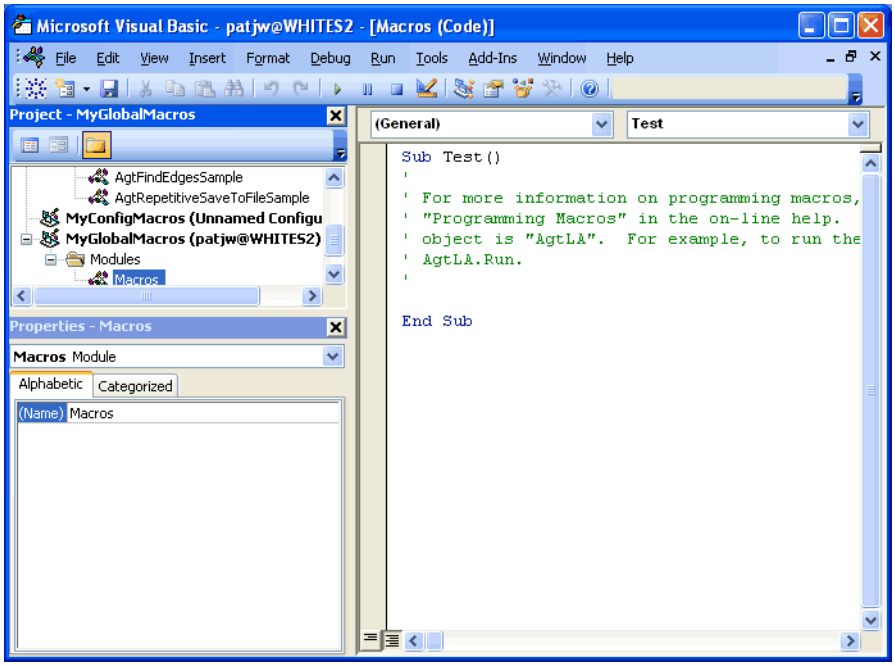

6 Close the VBA IDE window.

**See Also**

- VBA Macros and VbaView Windows (see page 15)

# Editing Macros in the VBA IDE

The VBA IDE looks like:




**TIP** Type "AgtLA" and the rest is magic.

```
For Each theModule In AgtLA.Modules
  MsgBox theModule.Name
Next

End Sub
```

Name
QueryCommand
RunningStatus
Slot
Status
StatusMsg
Type

---

In the VBA IDE, you can:

- Edit code. The text editor has Intellisense, which means that menus appear when typing.
- Debug macros:
  - Step through code.
  - Display the value of a variable.
  - Get a stack trace.

  See Debugging Macros in the VBA IDE (see page 29).

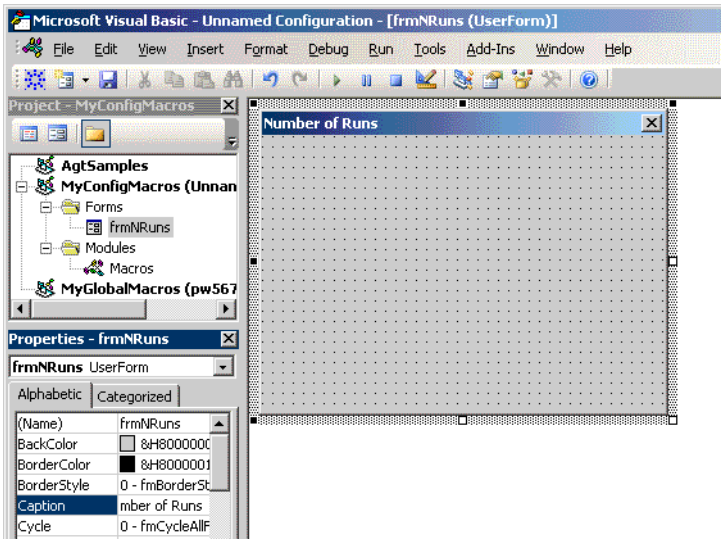
The VBA IDE is basically the same as the one used in Microsoft Excel, except it has knowledge of the logic analysis system COM model.

# Using Forms for Program Input/Output

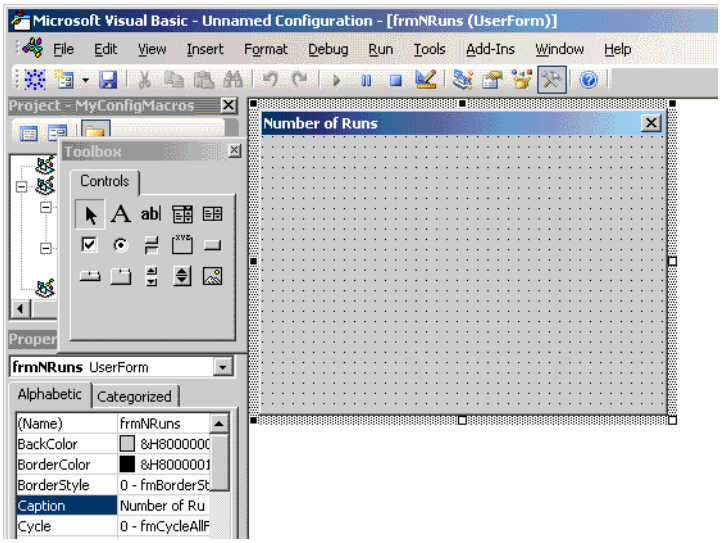
1 Create a macro as you would normally (see Creating a New Macro (see page 20)).

2 In the VBA IDE, choose **Insert>UserForm**.

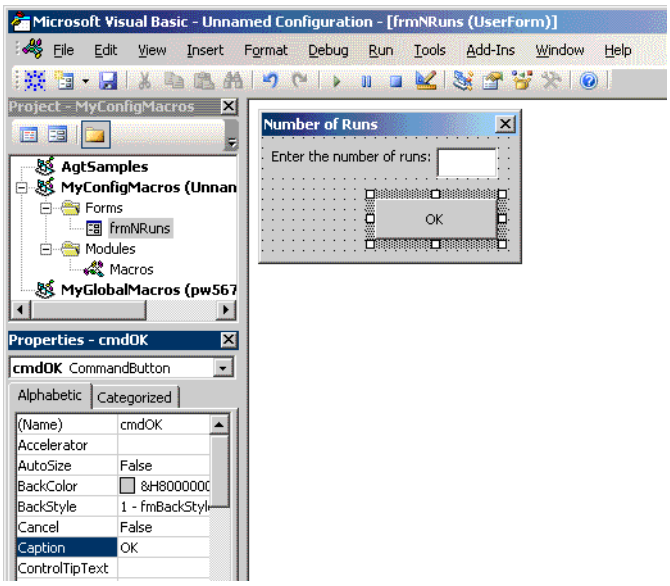
3 In the Properties window, rename the UserForm from "UserForm1" to the desired form name, and change the Caption property to the desired form dialog title.




4 Click in the form to open the Toolbox.

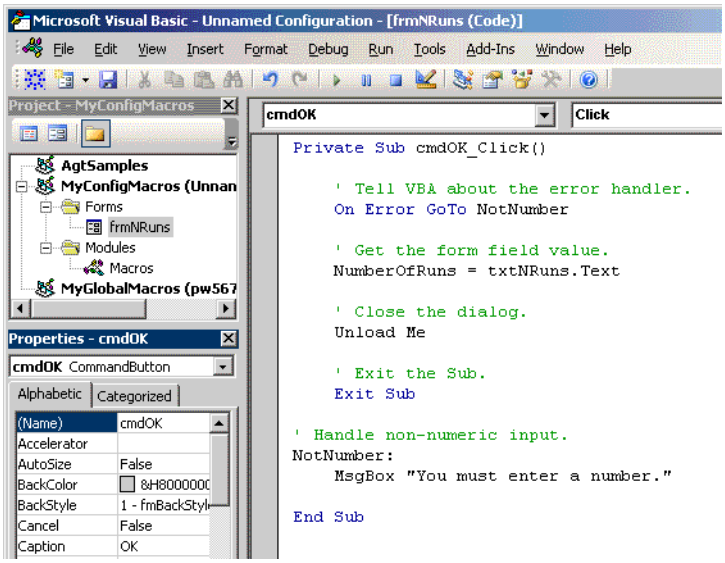


5 Drag controls from the Toolbox to the form. Position the controls, name them, and change the Caption and any other properties you desire.




6 In the Project window, right-click on the form and choose **View Code**.

7 In the Code window under "(General)", select the control you want to add code for, and enter the code for the form.



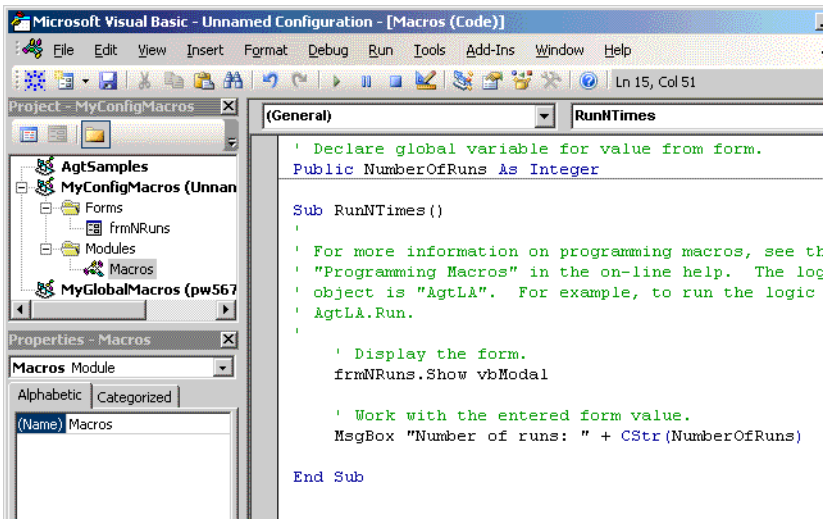

For more examples of code used in UserForms, see:

- Example: To populate a combo box with buses/signals (see page 26)
- Example: To tell if a bus/signal is valid (see page 26)

- Example: To get the selected string from a combo box (see page 27)
- Example: To select an item in a combo box based upon a string (see page 27)
- Example: To ensure that a text box allows only numeric input (see page 27)

8 In the Project window, right-click on the module and choose **View Code**.

9 Enter the module code that operates on the entered form values.




**See Also**

- Creating a New Macro (see page 20)
- Notes on Programming Macros (see page 30)
- The VBA IDE's online help.

## Example: To populate a combo box with buses/signals

```
Dim i As Integer
Me.cmbXAxisBusSignal.Clear

For i = 0 To myWindow.BusSignals.Count - 1
  Me.cmbXAxisBusSignal.AddItem myWindow.BusSignals(i).Name
Next i
Me.cmbYAxisBusSignal.ListIndex = 0
```

## Example: To tell if a bus/signal is valid

```
Private Function IsBusSignalValid(ByVal strBuSignal As String) _
    As Boolean
  Dim myData As AgtLA.SampleBusSignalData

  On Error GoTo busSignalNotValid
```

```
  IsBusSignalValid = True
  Set myData = myWindow.BusSignals(strXAxisBusSignal).BusSignalData
  Exit Function

busSignalNotValid:
  IsBusSignalValid = False
End Function
```

## Example: To get the selected string from a combo box

```
strXAxisBusSignal = _
    Me.cmbXAxisBusSignal.List(Me.cmbXAxisBusSignal.ListIndex)
```

## Example: To select an item in a combo box based upon a string

```
Private Sub SetComboBoxValue(ByRef cmbControl As ComboBox, _
    ByVal strValue As String)
  Dim i As Integer
  Dim bFound As Boolean

  i = 0
  Do While Not (cmbControl.List(i) <> strValue)
    i = i + 1
  Loop

  If (cmbControl.List(i) = strValue) Then
    cmbControl.ListIndex = i
  Else
    'A ListIndex of -1 means no item is selected
    cmbControl.ListIndex = -1
  End If
End Sub
```

## Example: To ensure that a text box allows only numeric input

```
Private Sub txtEndSample_KeyPress(ByVal KeyAscii As _
    MSForms.ReturnInteger)
  ' Cancel any non-numeric keys.
  If KeyAscii < vbKey0 Or KeyAscii > vbKey9 Then
    KeyAscii = 0
  End If
End Sub
```

# Running a Macro

- In the *Agilent Logic Analyzer* application, choose **Tools>Run Macro>(name of macro)**.

Or:

1. In the *Agilent Logic Analyzer* application, choose **Tools>Macro>Macros...** or press the ALT+F8 keyboard shortcut.
2. In the Macros dialog, select the macro you want to run, and click **Run**.

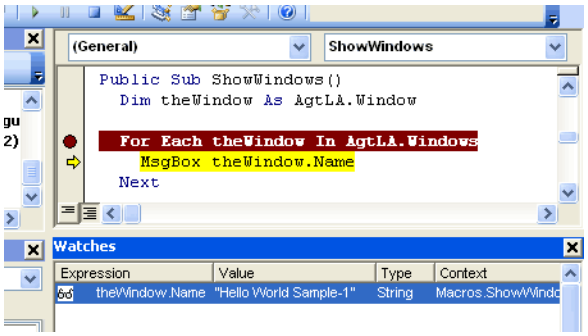
Or:

- In the VBA IDE, press F5.

**See Also**

- VBA Macros and VbaView Windows (see page 15)

# Debugging Macros in the VBA IDE




**To add a breakpoint**

- Click to the left of a line of code to set a breakpoint.

**To "watch" a variable**

1. When a macro is running, choose **Debug>Add Watch...** from the VBA IDE's main menu.
2. In the Add Watch dialog, enter the variable expression you want to watch, and click **OK**.

# Notes on Programming Macros

- To make a macro available via the **Tools>Macros>** menu, the macro must be public (not private) and have no parameters.
- The root object of the entire logic analyzer instrument is AgtLA. This object is used for accessing both logic analyzer sub-objects (like Modules and Windows) and also for logic analyzer specific data types.
- For example, to run the logic analyzer:

```
AgtLA.Run
AgtLA.WaitComplete(999)
```

**NOTE** This way of accessing the instrument object is different than when controlling the logic analyzer remotely using COM automation. Please keep this in mind when referring to the examples in the "COM Automation" (in the online help) online help.

- To extract data from the logic analyzer, see the "GetDataBySample Method" (in the online help).
- To create forms for user input or display of results, see Using Forms for Program Input/Output (see page 24).
- For a brief description of the differences between a VbaView and a Macro and when to use each, see VBA Macros and VbaView Windows (see page 15).
- For information about Visual Basic syntax, see Visual Basic Syntax (see page 88).
- For simple macro examples, see:
  - Example: Control Macro (see page 30)
  - Example: Analysis Macro (see page 31)
  - Example: Export Macro (see page 32)

  For more detailed macro examples, see the FindEdgesSample and the RepetitiveSaveToFileSample macros that are shipped with the *Agilent Logic Analyzer* application.

## Example: Control Macro

```
Option Explicit   ' Must define all variables.

Sub RunNTimes()

  Dim nCompletedRuns As Integer
  Dim i As Integer
  Dim strFile As String

  ' Use a loop to go through each of the runs.
```

```
  nCompletedRuns = 0
  For i = 1 To 5

    AgtLA.Run
    AgtLA.WaitComplete (999)

    'Given a file type, do the requested save
    strFile = "c:\LA\Data\RunNTimes" + VBA.LTrim(VBA.Str(i)) + ".ala"
    Call AgtLA.Save(strFile)   ' This tells the logic analyzer to save.
    AgtLA.Modules(0).WaitComplete (99)

    nCompletedRuns = nCompletedRuns + 1
    MsgBox (VBA.Str(nCompletedRuns) + " run(s) completed.")
  Next i

End Sub
```

## Example: Analysis Macro

```
Option Explicit   ' Must define all variables.

' This finds the first occurrence of a rising edge on bit 0
' of My Bus 1 from the beginning of the data.

Public Sub SimpleFind()

  Dim myWindow As AgtLA.Window
  Dim theFindResult As AgtLA.FindResult
  Dim strEvent As String
  Dim dTimeFound As Double

  ' Set the window object.
  Set myWindow = AgtLA.Windows("Waveform-1")

  ' Get the XML based string representing the edge.
  strEvent = "<Event>" + _
             "<BusSignal Name='My Bus 1' Bit='0' " + _
             "Operator='Rising Edge' />" + _
             "</Event>"

  ' Invoke the find command.
  Set theFindResult = myWindow.Find(strEvent, _
                                    1, _
                                    "F", _
                                    "Beginning Of Data", _
                                    "Present")

  ' Get the time where the result was found.
  If (theFindResult.Found) Then dTimeFound = theFindResult.TimeFound

End Sub
```

## Example: Export Macro

When exporting data to another application, you need to make sure the VBA project references that application's object library:

1. In the VBA IDE's Project Explorer window, select the VBA project your macro is in (or select a Form or Module in the project).
2. Choose **Tools>References...**.
3. In the References dialog, check the object library of the application you are exporting data to.

   For example, the object library for Internet Explorer is "Microsoft Internet Controls".
4. Click **OK**.

**Example**

```
Option Explicit   ' Must define all variables.

Public myIEApp As SHDocVw.InternetExplorer   ' This is IE application.
Public myModule As AgtLA.Module
Public nExportToIEStartSample As Long        ' Starting sample.
Public nExportToIEEndSample As Long          ' Ending sample.
Public ExportToIELogicAnalyzerData As Collection   ' Data collected
                                                   ' from LA.

Sub ExportDataToIE()

  Dim strHTML As String
  Dim nSample As Long
  Dim myData As AgtLA.SampleBusSignalData

  On Error GoTo noData

  nExportToIEStartSample = -10
  nExportToIEEndSample = 10
  Set myModule = AgtLA.Modules(0)
  Set myData = myModule.BusSignals(0).BusSignalData
  nSample = myData.StartSample   ' If error, there is no data.

  Call CreateIEObject         ' Create the new IE object if one isn't
                              ' already available.
  Call GetLogicAnalyzerData   ' Get the data from the logic analyzer.

  strHTML = "<HTML><TITLE>Exporting Data to IE</TITLE><H1>Export" + _
            "ing Data to External Applications</H1>"
  strHTML = strHTML + "<P>The data below was exported from the " + _
                      "logic analyzer to Windows Internet "
  strHTML = strHTML + "Explorer using the Advanced Customization " + _
                      "Environment.  This example "
  strHTML = strHTML + "demonstrates how to easily launch and " + _
                      "export data to other PC "
  strHTML = strHTML + "applications from within the logic " + _
                      "analyzer application."
  strHTML = strHTML + "<P>"
  strHTML = strHTML + "Here's the requested data from your last run."
```

```
  strHTML = strHTML + "<P>"
  strHTML = strHTML + "<TABLE BORDER>"
  strHTML = strHTML + GetHeaderString

  'Go through the data acquired from the logic analyzer and display it.
  'We obtain the data row by row.
  For nSample = nExportToIEStartSample To nExportToIEEndSample
    strHTML = strHTML + GetRowString(nSample)
  Next nSample
  strHTML = strHTML + "</TABLE>"
  strHTML = strHTML + "</HTML>"

  myIEApp.Document.Body.innerHTML = strHTML

  Exit Sub

noData:
  MsgBox ("There is no data.  Press Run button; then, run macro again.")

End Sub

' This subroutine creates a new IE object, but only if one is needed.
' The IE object is a new instance of the Internet Explorer application.
Private Sub CreateIEObject()
  If (IsIEAlreadyRunning() = False) Then
    Set myIEApp = CreateObject("InternetExplorer.Application")
    myIEApp.GoHome

    ' Wait until page loading complete before continuing.
    While myIEApp.ReadyState <> READYSTATE_COMPLETE
      DoEvents
    Wend
      myIEApp.Visible = True
  End If
End Sub

Private Function IsIEAlreadyRunning() As Boolean
  On Error GoTo invalidIE

  ' First, check to see if the IE app variable is empty.
  If (IsEmpty(myIEApp)) Then
    IsIEAlreadyRunning = False
    Exit Function
  End If

  ' Now, see if it is nothing.
  If (myIEApp Is Nothing) Then
    IsIEAlreadyRunning = False
    Exit Function
  End If

  If (myIEApp.Visible = False) Then
    IsIEAlreadyRunning = False
    Exit Function
  End If
  IsIEAlreadyRunning = True
  Exit Function
```

```
invalidIE:
  IsIEAlreadyRunning = False

End Function

' We need to keep all of the logic analyzer data in memory at one time,
' so we store it to a collection called ExportToIELogicAnalyzerData.
' This collection contains arrays of strings; one array for each Bus
' or Signal in this window.
Private Sub GetLogicAnalyzerData()
  Dim myBusSignal As AgtLA.BusSignal
  Dim strArray() As String

  ' Clear out the old data (if any).
  Set ExportToIELogicAnalyzerData = New Collection

  For Each myBusSignal In myModule.BusSignals
    ' This is where we get the data and save it to a collection.
    Call GetStringDataArray(myBusSignal, strArray)    ' Get the data.
    ExportToIELogicAnalyzerData.Add (strArray)    ' Save to collection
                                                  ' of strings.
  Next
End Sub

' This is where we get the data for a specific bus or signal and
' place it into a a string array so that we can add it to our web
' page later.
Private Sub GetStringDataArray(ByVal myBusSignal As AgtLA.BusSignal, _
                               ByRef strArray() As String)
   Dim i As Integer
   Dim myData As AgtLA.SampleBusSignalData
   Dim nArray() As Double
   Dim nDataRowCount As Long
   Dim nBitWidth As Integer

   nBitWidth = myBusSignal.BitSize

   Set myData = myBusSignal.BusSignalData

   ' Make sure that the nExportToIEStartSample and nExportToIEEndSample
   ' are within bounds.
   If (nExportToIEStartSample < myData.StartSample) Then
     nExportToIEStartSample = myData.StartSample
   End If
   If (nExportToIEEndSample > myData.EndSample) Then
     nExportToIEEndSample = myData.EndSample
   End If

   ' Extract the bus/signal data.
   ' In this case, we extract all data as a string.
   strArray = myData.GetDataBySample(nExportToIEStartSample, _
       nExportToIEEndSample, AgtDataStringHex, nDataRowCount)
End Sub

' This gets the list of bus and signal names formated as an HTML
' table header.
```

```
Private Function GetHeaderString() As String
  Dim strHeader As String
  Dim myBusSignal As AgtLA.BusSignal

  ' Get the names of each of the buses and signals.
  strHeader = "<TR>"
  For Each myBusSignal In myModule.BusSignals
    strHeader = strHeader + "<TH>" + myBusSignal.Name + "</TH>"
  Next
  strHeader = strHeader + "</TR>"
  GetHeaderString = strHeader
End Function

' Given a sample number, get the HTML formatted string corresponding
' to that row.
Private Function GetRowString(ByVal nSampleNumber As Long) As String
  Dim strArray() As String
  Dim nRow As Long
  Dim i As Integer
  Dim strRow As String

  nRow = nSampleNumber - nExportToIEStartSample

  strRow = "<TR>"
  For i = 1 To ExportToIELogicAnalyzerData.Count
    strArray = ExportToIELogicAnalyzerData(i)
    strRow = strRow + "<TD>" + strArray(nRow) + "</TD>"
  Next i
  strRow = strRow + "</TR>"
  GetRowString = strRow
End Function
```

# 3 Using Logic Analysis COM Objects in the ACE

# Start with AgtLA Namespace (Connect Object Not Needed)

- Start with the AgtLA namespace:

```
AgtLA.Run
AgtLA.Modules("My 1690D-1")
```

- No Connect object is needed in VBA because it is integrated into the *Agilent Logic Analyzer* application.

| Integrated VBA | External VB |
|---|---|
| `AgtLA.Run` | `Dim myConnect As AgtLA.Connect`<br>`Dim myInst As AgtLA.Instrument`<br><br>`Set myConnect = CreateObject("AgtLA.Connect")`<br>`Set myInst = myConnect.Instrument("myLAHostname")`<br>`myInst.Run` |

# Accessing Window and BusSignal Objects

Hierarchy of Window and BusSignal objects:

- "Windows" (in the online help) — Collection of all windows.
  - "*Window*" (in the online help) — Specific instance of a window such as Waveform-1.
    - "BusSignals" (in the online help) — Collection of all buses and signals in this window.
      - "BusSignal" (in the online help) — Specific instance of a bus or signal such as My Bus 1.

To access Window and BusSignal objects:

- Define variables using "Dim":

```
Dim theWindow As AgtLA.Window
Dim theBusSignal As AgtLA.BusSignal
```

- Using AgtLA as a starting point, obtain the objects:

```
Set theWindow = AgtLA.Windows("Waveform-1")
Set theBusSignal = theWindow.BusSignals("My Bus 1")
```

Result is two variables, one with Waveform-1 and one with My Bus 1.

Notice that we used "Set" because we are setting the value of an object.

### Other Ways to Access Objects

- Accessing by index:

```
Set theModule = AgtLA.Modules(0)
```

- Using a string variable instead of a string constant:

```
Dim strWindow As String
strWindow = "Waveform-1"
Set myWindow = AgtLA.Windows(strWindow)
```

# Generic and Specific Objects

Hierarchy of generic and specific objects:

- "Modules" (in the online help) — Collection of all modules in the system.
  - "*Module*" (in the online help) — Generic object that covers both logic analyzer and pattern generator modules.
  - "Analyzer*Module*" (in the online help) — Logic analyzer module specific object.
  - "Pattgen*Module*" (in the online help) — Pattern generator module specific object.

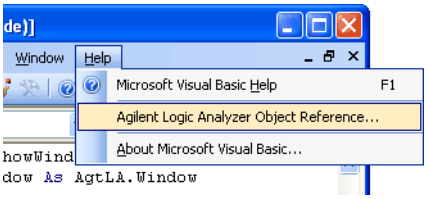
About generic and specific objects:

- Generic object Module contains the properties and methods that are common to both logic analyzer and pattern generator modules.
  - Properties such as "Name" apply to both.
- There are separate objects for analyzer modules and pattern generator modules:
  - AnalyzerModule contains logic analyzer specific properties and methods such as GetDataBySample, but it also has access to all of the generic properties and methods in Module.
  - PattgenModule contains pattern generator specific methods such as InsertLine, but it also has access to all of the generic properties and methods in Module.
- If you know what type of object you have, use the specific objects like AnalyzerModule and PattgenModule.
- If you don't know what type of object you have, use the generic object such as Module.
  - You can start by using a generic object, and, depending upon the type, use the more specific objects:

```
' Start generic:
Dim theModule As AgtLA.Module
Set theModule = AgtLA.Modules(0)

Dim theAnalyzerModule As AgtLA.AnalyzerModule
If (theModule.Type = "Analyzer") Then
  ' Once you know the type, use the more specific objects.
  Set theAnalyzerModule = theModule
End If

Dim thePattgenModule As AgtLA.PattgenModule
If (theModule.Type = "Pattgen") Then
  ' Once you know the type, use the more specific objects.
  Set thePattgenModule = theModule
End If
```

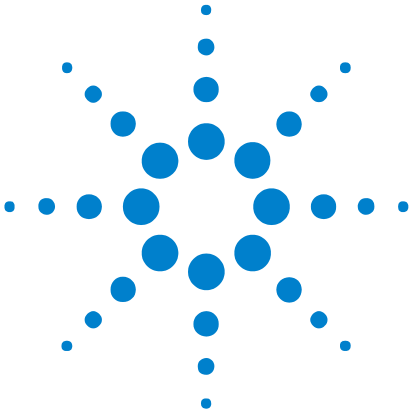
## Getting Help on COM Objects

- In the VBA IDE, choose **Help>Agilent Logic Analyzer Object Reference...**.




- Highlight the word you want to learn more about, and press the **F1** key.

```
Set theWindow = AgtLA.Windows("Waveform-1")
Set theBusSignal = theWindow.BusSignals("My Bus 1")
MsgBox theBusSignal.Name
```



# 4 Analyzing Data in ACE

When analyzing data in the Advanced Customization Environment (ACE), there are two ways to access the data you're interested in:

- Use the Find method to only find the events of interest.

  This is much faster if events of interest are sparse. Also, this is hardware accelerated if you are connected to logic analyzer hardware.

- Use the GetDataBySample or GetDataByTime methods to downloads chunks of data.

  This is much faster if almost all the captured data is needed. It is faster to download 1 M of data than to do 500,000 Finds.

Some examples of analyzing data:

- Use TimingZoom data to verify timing of a bus.
- Find incomplete transactions.
- Analyze bus utilization.

For more information on analyzing data, see:

# Finding Events (Using Logic Analyzer Hardware)

To find "My Bus 1 = h11":

```
Public Sub FindMethod()
  Dim strStartingTime As String
  Dim myWindow As AgtLA.Window
  Dim theFindResult As AgtLA.FindResult

  Set myWindow = AgtLA.Windows("Waveform-1")

  ' Find from beginning of data.
  ' The event string varies.
  Set theFindResult = myWindow.Find("My Bus 1 = h11", _
                                    1, _
                                    "F", _
                                    "Beginning Of Data", _
                                    "Present")
End Sub
```

For more information, see:

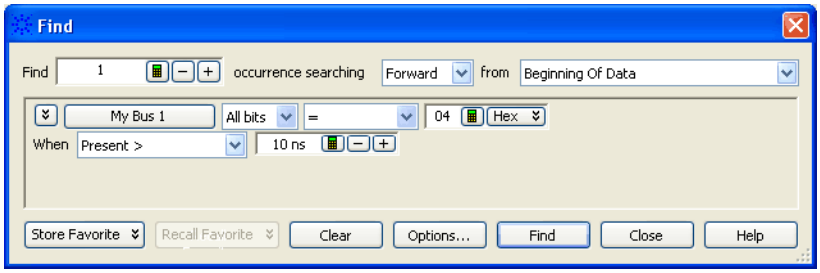
- Understanding the Find Method and FindResult Object (see page 44)
- Using Simple Event Strings (see page 46)
- Using XML Event Strings (see page 46)
  - Shortcut to Code Development: Secret to Creating XML Strings (see page 47)
- Finding a Sequence of Events (see page 50)
  - Creating a Sequential Search (see page 51)
  - Example: Sequential Find (see page 52)
  - Debugging a Sequential Find (see page 52)

## Understanding the Find Method and FindResult Object

**Find Method**

```
Set theFindResult = myWindow.Find("My Bus 1 = h04", _    ' Event string.
                                  1, _    ' Occurrence.
                                  "F", _    ' Forward or Backward.
                                  "Beginning Of Data", _
                                  "Present>", _
                                  "10 ns")
```

This Find method call is the same as the Find shown below in the *Agilent Logic Analyzer* interface.




**FindResult Object**

The FindResult object returned by the Find method has the following properties:

| Properties | Description |
|---|---|
| "Found" (in the online help) | Gets the found status. |
| "OccurrencesFound" (in the online help) | Gets the number of occurrences found. |
| "SubrowFound" (in the online help) | Gets the subrow number if found on a subrow. |
| "TimeFound" (in the online help) | Gets the time found as a double. |
| "TimeFoundString" (in the online help) | Gets the time found as a string. |

The following example shows how some of the FindResult properties are used:

```
Set theFindResult = myWindow.Find("My Bus 1 = h08", _
                                  1, _
                                  "F", _
                                  "Beginning Of Data", _
                                  "Present")
If (theFindResult.Found) Then
  MsgBox "Found at " + theFindResult.TimeFoundString
Else
  MsgBox "Not found"
End If
```

**Event Strings**

Can either be:

- Simple:

```
"My Bus 1=h04"
```

  These are best for very simple events (see Using Simple Event Strings (see page 46)).

- XML:

```
"<Event><BusSignal Name='My Bus 1' Bit='0' Operator='Rising Edge'/>
</Event>"
```

These provide more power at the cost of using XML tags (see Using XML Event Strings (see page 46)).

## Using Simple Event Strings

**TIP**

Simple event string programming rules:

- Precede each value with a base.
- Use "e" for edges.

Example:

```
"My Bus 1=h04 And Sig1=eR"
```

- You can use "And" and "Or". Note that these are case sensitive – "AND" won't work!

Precede each value with the base (such as h04):

| Prefix | Base | Example |
|---|---|---|
| h | hexadecimal | h04 |
| o | octal | o3 |
| b | binary | b10110 |
| d | decimal | d99 |

Use e for edges (such as eR):

| Value | Type | Example |
|---|---|---|
| X | don't care | eX |
| R | rising edge | eR |
| F | falling edge | eF |
| E | either edge | eE |

## Using XML Event Strings

```
<Event>
   <And>
      <BusSignal Name='Sig1' Bit='All' Operator='High'/>
      <BusSignal Name='ADDR' Bit='All' Operator='Equals' Value='hXX'/>
   </And>
</Event>
```

- XML event strings are more complicated but provide much greater power.
- Put the entire event within <Event>...</Event>.
- Each pair of bus/signal names needs an <And>...</And> or <Or>...</Or>.
- Operators include: Rising Edge, Falling Edge, Either Edge, Range, etc.

Example:

```
strEvent = "<Event><BusSignal Name='My Bus 1' Bit='All' " + _
           "Operator='Equals' Value='h01'/></Event>"
```

**TIP** Note the use of single quotes instead of double quotes because double quotes are used for strings in Visual Basic.

**See Also**

- Shortcut to Code Development: Secret to Creating XML Strings (see page 47)

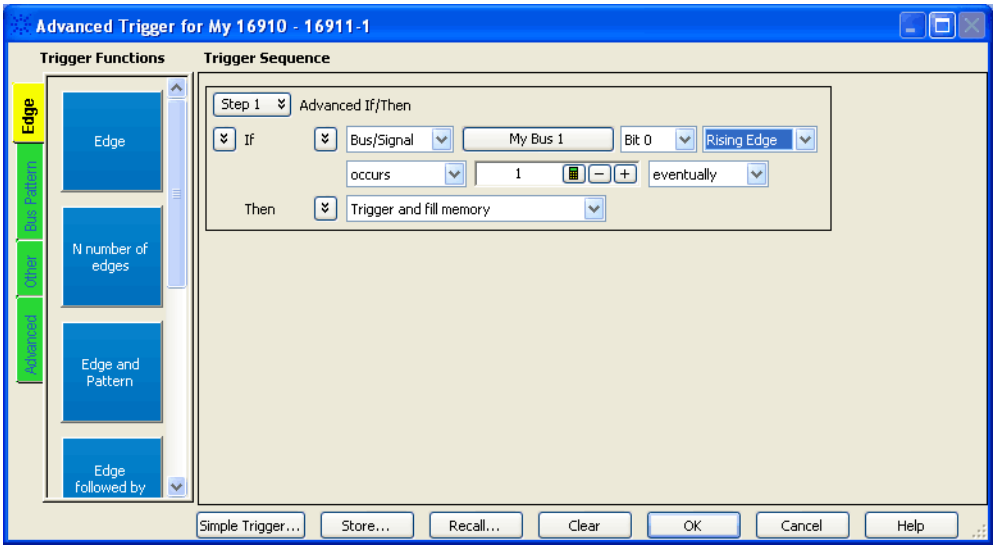
### Shortcut to Code Development: Secret to Creating XML Strings

**TIP** The easy way to specify an XML find event string is:

1. Use the Advanced Trigger dialog to create the equivalent trigger.
2. Select **Store...** to save it to an XML file.
3. Open up the XML file in Notepad and cut-and-paste it into your program.

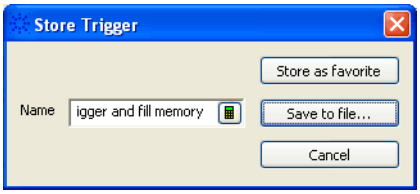
To create an XML event string:

1 In the *Agilent Logic Analyzer* application, create the equivalent trigger in the Advanced Trigger dialog.



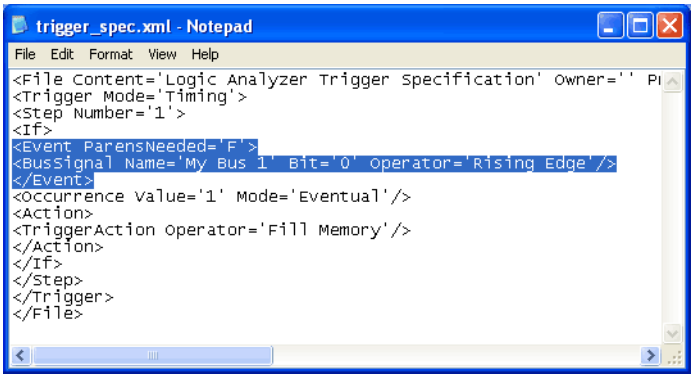

2 Click **Store...**.

3 In the Store Trigger dialog, click **Save to file...**, and save the trigger to a file.




4 Open the trigger XML file in Notepad (not Internet Explorer because of its double quoted attribute values).

5 Copy the XML Event string.




**TIP**

- Do not copy <Step>, <Action>, <Trigger>, or anything except <Event>.
- Only <Event> will work even though some of these tags are similar to parameters in the Find Method.
- Example: Don't use <Occurrence> even though the Find Method has an Occurrence parameter.

**6** In the VBA IDE, paste the XML event string into a string variable.

```
  Dim strEvent As String
  strEvent = <Event ParensNeeded='F'>
<BusSignal Name='My Bus 1' Bit='0' Operator='Rising Edge'/>
</Event>

  Set theFindResult = myWindow.Find("My Bus 1 = h08", _
                                    1, _
                                    "F", _
                                    "Beginning Of Data", _
                                    "Present")
```

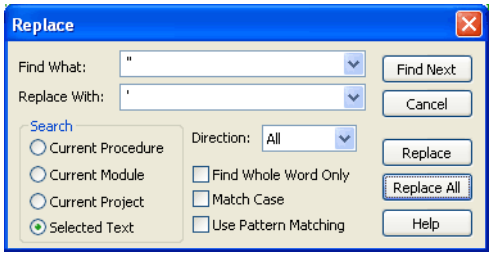
(Red means there is an error.)

**7** If you happen to have an XML event string with double quoted attribute values, convert double quotes to single quotes.

**a** Highlight the entire event string.

**b** Choose **Edit>Replace...**.

**c** Replace " with '.



8 Put double quotes around each line, and use the "+" operator to concatenate lines.

9 Use line extension " _" to extend the line.

10 Use the string variable in the Find method.

The resulting XML event string in code looks like:

```
Dim strEvent As String
strEvent = "<Event ParensNeeded='F'>" + _
           "<BusSignal Name='My Bus 1' Bit='0' Operator='Rising Edge'/>" + _
           "</Event>"

Set theFindResult = myWindow.Find(strEvent, _
                                  1, _
                                  "F", _
                                  "Beginning Of Data", _
                                  "Present")
```

Notice that the XML Event String is on three lines. Each line is within double quotes. All lines except the last end in "+ _" for concatenation and extension.

## Finding a Sequence of Events

Just as you can set up a trigger on a sequence of events (see the XML example below), you can also find a sequence of events in the Advanced Customization Environment (ACE).

```
<File Content='Logic Analyzer Trigger Specification'>
   <Trigger Mode='Timing'>
      <Step Number='1'>
         <If>
            <Event ParensNeeded='F'>
               <BusSignal Name='My Bus 1' Bit='0'
                     Operator='Rising Edge'/>
            </Event>

            <Occurrence Value='1' Mode='Eventual'/>
            <Action>
               <Goto Step='Next'/>
            </Action>
         </If>
      </Step>
      <Step Number='2'>
         <If>
            <Event ParensNeeded='F'>
               <BusSignal Name='Sig1' Bit='All'
                     Operator='Falling Edge'/>
            </Event>
            <Occurrence Value='1' Mode='Eventual'/>
            <Action>
               <TriggerAction Operator='Fill Memory'/>
            </Action>
         </If>
```

```
      </Step>
   </Trigger>
</File>
```

**TIP**

- Only use an Event string. Nothing else will work!
- Copy each event string individually. (Strings that begin with <Event> and end with </Event>.)
- Nothing else will work — not an occurrence, not an if, not a step, not a sequence, ...

To find a sequence of events, see:

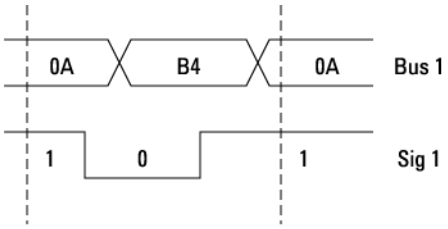
- Creating a Sequential Search (see page 51)
- Example: Sequential Find (see page 52)
- Debugging a Sequential Find (see page 52)

### Creating a Sequential Search

When finding A followed by B:

- 1st find is often started from "Beginning Of Data".
- 2nd find starts with the "Found" marker because it starts searching at the point that the 1st find ended.

Watch out for this:




Sample #10

- 1st find matches at sample #10.
- 2nd find has an event that is also matched by sample #10.

  2nd find stays on sample 10 instead of finding the next occurrence.

**TIP**

To avoid creating an infinite loop that continues to find the same sample:

1. In the 2nd find result, check that the found time is not the same as the found time for the 1st find result.
2. If the found times are the same, do a "Find Next".

### Example: Sequential Find

```
Public Sub SequentialFind()
  Dim strStartingTime As String
  Dim myWindow As AgtLA.Window
  Dim theFindResult As AgtLA.FindResult

  Set myWindow = AgtLA.Windows("Waveform-1")

  ' Find from beginning of data.
  Set theFindResult = myWindow.Find("MyBus1=h08", _
                                    1, _
                                    "F", _
                                    "Beginning Of Data", _
                                    "Present")

  If (theFindResult.Found) Then strStartingTime = _
      theFindResult.TimeFoundString

  ' Find starting at the Found marker.
  Set theFindResult = myWindow.Find("MyBus1=h15", _
                                    1, _
                                    "F", _
                                    "Found", _
                                    "Present")

  ' Verify if we haven't moved from the 2nd find.
  If (theFindResult.Found And strStartingTime = _
      theFindResult.TimeFoundString) Then
    Set theFindResult = myWindow.FindNext
  End If

  If (theFindResult.Found) Then
    MsgBox "Found them at " + strStartingTime + " and " + _
        theFindResult.TimeFoundString
  Else
    MsgBox "Found failed"
  End If

  FindEdge = theFindResult.Found
End Sub
```

### Debugging a Sequential Find

- Place a marker on each event after you find it.
- The last event always gets the Found marker, so you don't have to place a marker on it.

```
' AgtLA.Markers.Add markerName, textColor, backgroundColor,
' timePosition:
AgtLA.Markers.Add "My New Marker", vbWhite, vbBlue, nPos
```

# Getting Data from the Logic Analyzer

Hierarchy of objects with sample data:

- "Windows" (in the online help)
  - "Window" (in the online help)
    - "BusSignals" (in the online help)
      - "BusSignal" (in the online help)
        - "*BusSignalData*" (in the online help) — Generic object for all data.
        - "Sample*BusSignalData*" (in the online help) — Specific object for sample data.

The SampleBusSignalData object is currently the only type of bus/signal data available, so always use the SampleBusSignalData object instead of the BusSignalData object.

```
Dim myWindow As AgtLA.Window
Dim myBusSignalData As AgtLA.SampleBusSignalData

Set myWindow = AgtLA.Windows("Waveform-1")
Set myBusSignalData = myWindow.BusSignals("My Bus 1").BusSignalData
```

For more information on getting data from the logic analyzer, see:

- Understanding the GetDataBySample Method (see page 53)
- Data Types for GetDataBySample (see page 54)
- Getting the Entire Trace (from Beginning of Data to End of Data) (see page 55)
- Example: GetTenSamples (see page 55)

## Understanding the GetDataBySample Method

The GetDataBySample method has the following parameters and returns the number of rows:

| Parameters | Definition |
|---|---|
| object | An expression that evaluates to an "SampleBusSignalData" (in the online help) 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 Data Types for GetDataBySample (see page 54). |

| Returns | Definition |
|---|---|
| NumRowsRet | A **Long** initialized by this method to the number of rows being returned in the array. |

For example:

```
Dim myWindow As AgtLA.Window
Dim myBusData As AgtLA.SampleBusSignalData
Dim nNumDataRows As Long
Dim dArray() As Double

Set myWindow = AgtLA.Windows("Waveform-1")
Set myBusData = myWindow.BusSignals("My Bus 1").BusSignalData

dArray = myBusData.GetDataBySample(0, 10, AgtDataDouble, nNumDataRows)
'         ^                          ^   ^         ^               ^
'         |                          |   |         |               |
'      object            StartSample  EndSample DataType   NumRowsRet
```

## Data Types for GetDataBySample

Possible GetSampleByData method data types and the associated Visual Basic data types are shown below.

| AgtDataType | Max Channels | VB Data Type |
|---|---|---|
| AgtDataLong | 31 | Long |
| AgtDataDouble | 52 | Double |
| AgtDataDecimal | 96 | Variant |
| AgtDataTime | n/a | Double |
| AgtDataStringHex | 128 | String |
| AgtDataStringDec | 128 | String |
| AgtDataRaw | 128 | Variant (Not for VBA!) |

- Use the smallest data type you can to save space.
- AgtDataRaw cannot be manipulated by VBA and is essentially useless to VBA. (It is useful for Visual Studio C++ users.)
- For buses wider than 96 bits, break them into two buses or use String.

**See Also**

- "DataTypes and Return Values" (in the online help)

## Getting the Entire Trace (from Beginning of Data to End of Data)

To get the data from the beginning sample to the end sample, use the BusSignalData object's StartSample and EndSample properties as the StartSample and EndSample parameters in the GetDataBySample method.

```
Set myBusSignalData = myWindow.BusSignals("My Bus 1").BusSignalData

' Getting Entire Trace!
dArray = myBusSignalData.GetDataBySample(myBusSignalData.StartSample, _
    myBusSignalData.EndSample, AgtDataDouble, nNumDataRows)
```

**TIP**

To avoid memory overflow, get your data in chunks.

Memory overflow depends on the amount of data requested and the data type chosen:

- For the double data type, use less than 4 M samples.
- The 4 M samples limit is the total for all buses you keep in memory at once (for example, for 2 buses, use only 2 M samples).
- For the string data types, use less than 10,000 samples.

To process data, get the data a chunk at a time and process it.

- Generally, there is no reason to have 64 M of samples in memory at one time.

## Example: GetTenSamples

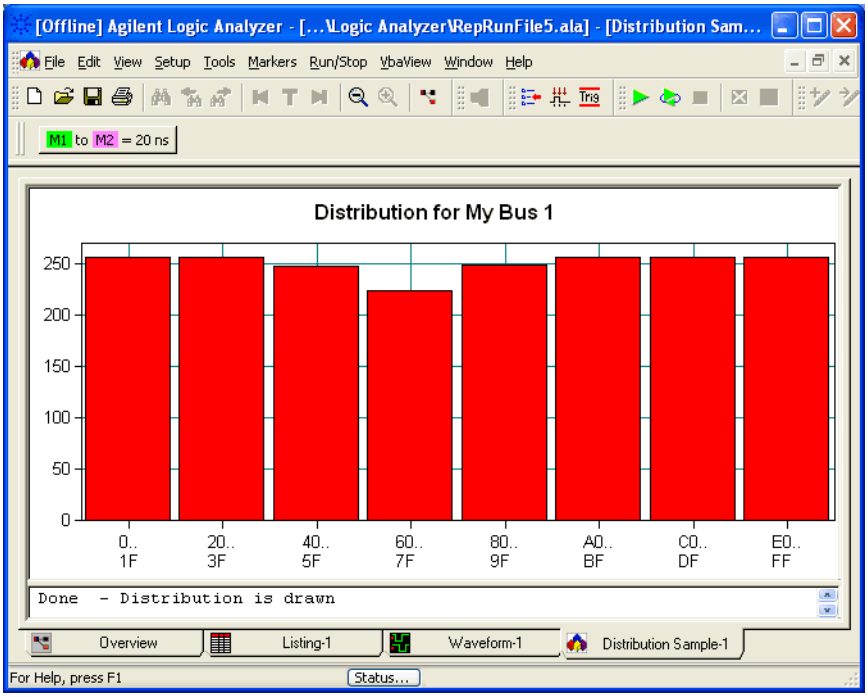
```
Public Sub GetTenSamples()
  Dim myWindow As AgtLA.Window
  Dim myBusSignalData As AgtLA.SampleBusSignalData
  Dim nNumDataRows As Long

  Dim dArray() As Double   ' Place to keep our data.

  ' Get the objects we need.
  Set myWindow = AgtLA.Windows("Waveform-1")
  Set myBusSignalData = _
      myWindow.BusSignals("My Bus 1").BusSignalData

  ' Get the data from the logic analyzer.
  dArray= myBusSignalData.GetDataBySample(0, 10, AgtDataDouble, _
      nNumDataRows)

  Dim i As Integer
  Dim nCount As Integer

  ' This loop iterates through all ten samples.
  For i = 0 To nNumDataRows -1
    ' dArray(i) is the value of the sample.
    ' We use Hex to convert it to a hex string, but dArray(i) is the
    ' value itself.
    MsgBox "Value of My Bus 1 at Sample " + VBA.Str(i) + " was " + _
```

```
            Hex(dArray(i))
    Next i
End Sub
```

# 5 Displaying Data in VbaView Windows

The VbaView window works with the integrated Microsoft Visual Basic for Applications (VBA) to provide custom data visualization charts. The VbaView window is a data window like Waveform or Listing, except that it requires VBA code to display data.

There are events in the *Agilent Logic Analyzer* user interface that (in many cases) the VbaView window should respond to. These events include a logic analyzer run, a screen update, and clicking on items in the VbaView menu. You program the VbaView window by writing VBA code that responds to these events; this code belongs in the "Notify" function within the "AgtVbaView" module. (It is not necessary to create macros when programming the VbaView window.)



# Adding a New VBA View "Hello World Sample" Window

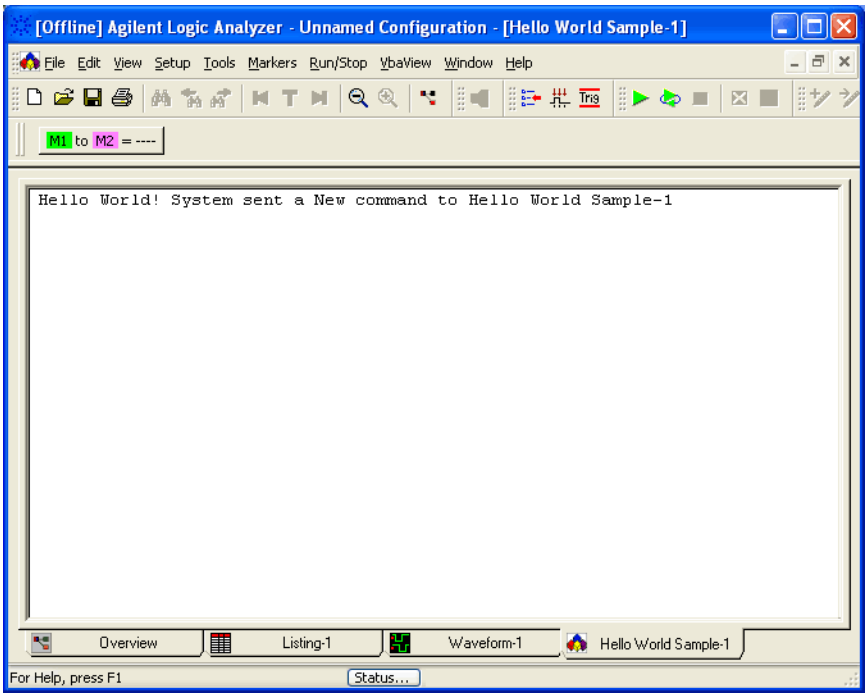
The purpose of the Hello World Sample VbaView window is to give you a short cut to creating a VbaView window. To add a new Hello World Sample VbaView window:

1 In the *Agilent Logic Analyzer* application, choose **Window>New VbaView>Hello World Sample...**.

**Next** • Using the Hello World Sample VbaView (Text) Window (see page 59)

**See Also** • VBA Macros and VbaView Windows (see page 15)

## Using the Hello World Sample VbaView (Text) Window




1 Choose **VbaView>Redraw**.

2 Choose **VbaView>Properties...**.

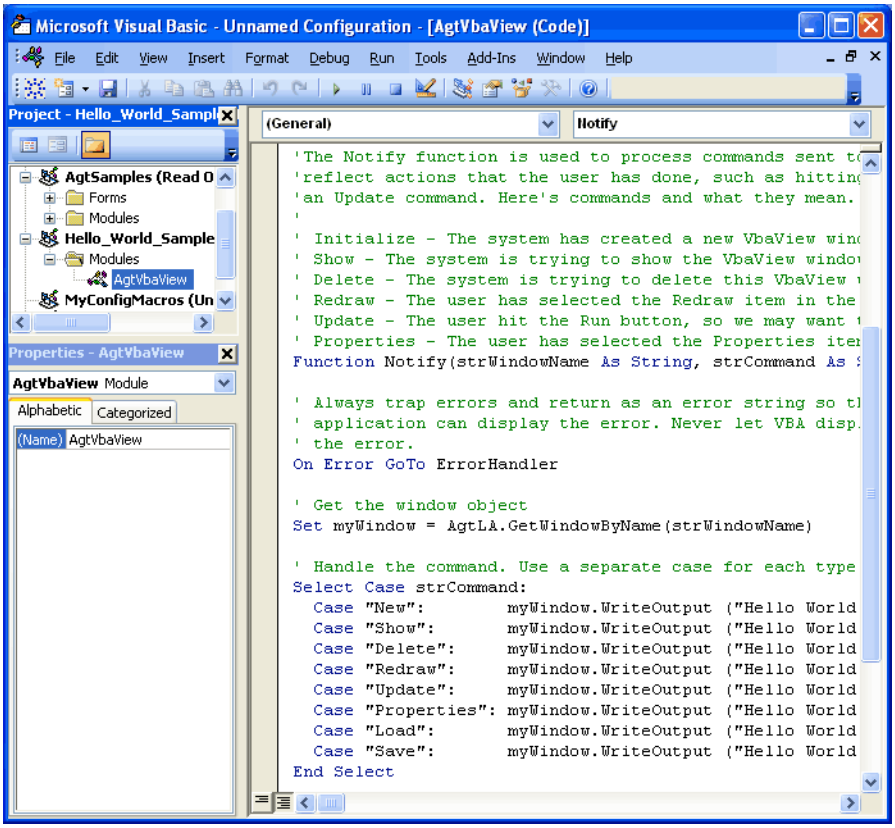
3 Choose **Run/Stop>Run** (or press the F5 keyboard shortcut).

Note that events in the logic analysis system cause text to be written to the VbaView window.

**Next** • Viewing the VbaView Code (see page 60)

## Viewing the VbaView Code

1 Choose **VbaView>View Code...** (or press the ALT+F11 keyboard shortcut).




**TIP** "AgtVbaView" is a reserved word for Module name. "Notify" is a reserved word for Function name. Nothing else will work, so don't change the names.

**See Also**

- Understanding the Notify Function (see page 61)

# Understanding the Notify Function

The Notify function that must be present in VbaView window code is called with the following parameters:

```
Function Notify(strWindowName As String, _
                strCommand As String, _
                ByRef varCommandParm As Variant) As Variant
```

- StrWindowName is the name of the VbaView, such as "Hello World Sample- 1".


- strCommand is what happened in the logic analysis system.
- varCommandParm is an optional parameter that is declared ByRef, which means a parameter value can be sent to and received from the Notify function. The definition of this parameter is strCommand specific.

**Notify Commands (strCommand)**

The Notify function receives the following commands from the logic analysis system and calls functions to respond to them if needed. For example, when an Update command is received, the VbaView should update itself based upon the current data.

| strCommand | varCommandParm | What makes it happen |
|---|---|---|
| New | — | Creating a new VbaView window. |
| Show | — | Making VbaView window visible, such as switching to a VbaView window from another window. This is only used when controlling an external application to make it visible. See the Export to IE Sample VbaView window. |
| Delete | — | Deleting a VbaView window. |
| Redraw | — | Choosing **VbaView>Redraw**. |
| Update | Boolean (output) | Anything that causes the window to update, such as a Run. The varCommandParm parameter is True if the update succeeded, False if the update was cancelled. |
| Properties | — | Choosing **VbaView>Properties...**. |
| Load | String (input) | Opening a configuration file. This command lets you load the state of a VbaView window. The varCommandParm parameter contains a previously saved XML string. |

| | | |
|---|---|---|
| Save | String (output) | Saving a configuration file. This command lets you save the state of a VbaView window. The varCommandParm parameter returns an XML string to be loaded later.<br>**Caution:** The Save command also occurs when the Instrument object's QueryCommand method is called. Agilent recommends that you do not call the Instrument object's QueryCommand method from within the Notify function because this can cause your software to be re-entrant. Instead, call the QueryCommand method for the specific subsystem you are interested in (for example, Instrument.Overview, Module.QueryCommand, Tool.QueryCommand, etc.). |
| QueryCommand | String (input) | When a QueryCommand is sent to the window, the varCommandParm parameter contains the command to be queried. |
| | Boolean (output) | The varCommandParm parameter returns the boolean False if the command is not valid or True if the command is valid and nothing needs to be returned. |
| | String (output) | The varCommandParm parameter returns an XML string if the command is valid and the query produced output. |

The "Notify" function is not used if there is no VbaView window.

# Using the VbaViewChart Object

Hierarchy of objects containing VbaViewChart object:

- "Windows" (in the online help) – Collection of all windows in the system.
  - "*Window*" (in the online help) – Generic object for any type of window.
  - "VbaView*Window*" (in the online help) – VbaView specific window object.
    - "VbaViewChart" (in the online help) – The chart object in a VbaView window.
      - "VbaViewChartAxis" (in the online help)
      - "VbaViewChartData" (in the online help) – The data in the chart.
      - "VbaViewChartLegend" (in the online help)
      - "VbaViewChartTitle" (in the online help)
        - "VbaViewChartFont" (in the online help)
    - "VbaViewWebBrowser" (in the online help)

To access data in a chart:

```
Dim myWindow As AgtLA.VbaViewWindow
Dim myChart As AgtLA.VbaViewChart
Dim myData As AgtLA.VbaViewChartData

Set myWindow = AgtLA.Windows(strWindow)
Set myChart = myWindow.Chart
Set myData = myChart.Data   ' This is the data you are going to chart.
```

For more information on using the VbaViewChart object, see:

- Setting the Chart Type (see page 64)
- Using the AddPointArrays Method (see page 64)
- Setting Titles in the Chart (see page 64)
- Updating the Chart Display (see page 64)
- Example: XY Scattergram (see page 65)
- Example: Line Chart (see page 66)
- Example: Bar Chart (see page 67)
  - Example: Horizontal Bar Chart (see page 68)
  - Example: Horizontal Stacked Bar Chart (see page 69)
  - Example: Vertical Bar Chart (see page 70)
  - Example: Vertical Stacked Bar Chart (see page 70)

- Example: Pie Chart (see page 71)

## Setting the Chart Type

**Chart Types** When creating a VbaViewChart, you can choose between the following types of charts:

- AgtChartTypeNone
- AgtChartTypeLine
- AgtChartTypeLineOnly
- AgtChartTypeXYScatter
- AgtChartTypeHorizontalBar
- AgtChartTypeVerticalBar
- AgtChartTypePie
- AgtChartTypeStackedVerticalBar
- AgtChartTypeStackedHorizontalBar

**To set the chart type** Set the chart's ChartType property to one of the values above. For example, to set up a line chart:

```
myChart.ChartType = AgtChartTypeLine
```

## Using the AddPointArrays Method

See "AddPointArrays Method" (in the online help).

## Setting Titles in the Chart

**TIP** Order is important when creating Chart titles or axis titles. You *must* do a "HasTitle=True" before each title that you set; otherwise, you will get a run-time error.

For example:

```
myChart.HasTitle = True
myChart.Title = "My Chart"
myChart.Axis(AgtChartAxisTypeX).HasTitle = True
myChart.Axis(AgtChartAxisTypeX).Title = "My Bus 1"
myChart.Axis(AgtChartAxisTypeY).HasTitle = True
myChart.Axis(AgtChartAxisTypeY).Title = "My Bus 2"
```

Notice the X and Y axis each have a title.

## Updating the Chart Display

- Start by changing the Notify function:

```
Select Case strCommand:
  Case "New":         myWindow.WriteOutput ("Hello World! System " + _
                          "sent a New command to " + strWindowName)
  Case "Show":        myWindow.WriteOutput ("Hello World! System " + _
                          "sent a Show command to " + strWindowName)
  Case "Delete":      myWindow.WriteOutput ("Hello World! System " + _
                          "sent a Delete command to " + strWindowName)
  Case "Redraw":      myWindow.WriteOutput ("Hello World! System " + _
                          "sent a Redraw command to " + strWindowName)
  Case "Update":      UpdateDisplay strWindowName
  Case "Properties": myWindow.WriteOutput ("Hello World! System " + _
                          "sent a Properties command to " + _
                          strWindowName)
End Select
```

- Then, create the UpdateDisplay subroutine in the AgtVbaView module. For example, see:
  - Example: XY Scattergram (see page 65)
  - Example: Line Chart (see page 66)
  - Example: Horizontal Bar Chart (see page 68)
  - Example: Horizontal Stacked Bar Chart (see page 69)
  - Example: Vertical Bar Chart (see page 70)
  - Example: Vertical Stacked Bar Chart (see page 70)
  - Example: Pie Chart (see page 71)

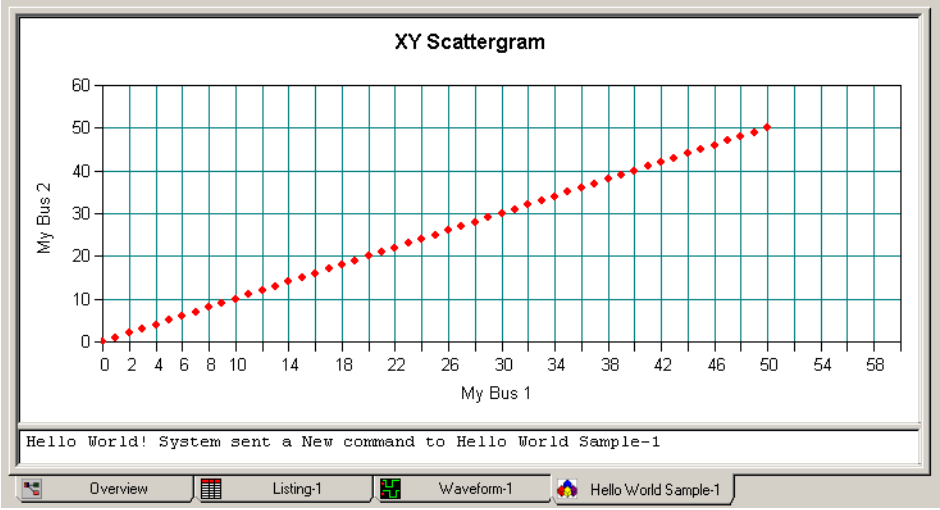
  The UpdateDisplay subroutine will not be accessible to the AgtVbaView module unless you create it within the module.

## Example: XY Scattergram

This example uses generated points, but you're more likely to use double arrays that you've obtained from the logic analyzer via GetDataBySample.

```
Private Sub UpdateDisplay(ByVal strWindow As String)
  Dim myWindow As AgtLA.VbaViewWindow
  Dim myChart As AgtLA.VbaViewChart
  Dim myData As AgtLA.VbaViewChartData

  Set myWindow = AgtLA.Windows(strWindow)
  Set myChart = myWindow.Chart
  Set myData = myChart.Data

  ' Set the ChartType:
  myChart.ChartType = AgtChartTypeXYScatter

  ' Set up the titles:
  myChart.HasTitle = True
  myChart.Title = "XY Scattergram"
  myChart.Axis(AgtChartAxisTypeX).HasTitle = True
  myChart.Axis(AgtChartAxisTypeX).Title = "My Bus 1"
  myChart.Axis(AgtChartAxisTypeY).HasTitle = True
```

```
  myChart.Axis(AgtChartAxisTypeY).Title = "My Bus 2"

  ' Populate the arrays:
  Dim xValueArray(50) As Double
  Dim yValueArray(50) As Double
  Dim i As Integer
  For i = 0 To 50
    xValueArray(i) = i
    yValueArray(i) = i
  Next i

  ' Clear the old data, add the new, and redraw:
  myData.Clear
  myData.AddPointArrays xValueArray, yValueArray, _
                        AgtDataPointTypeCircle, AgtDataPointSizeSmall
  myChart.Draw
End Sub
```

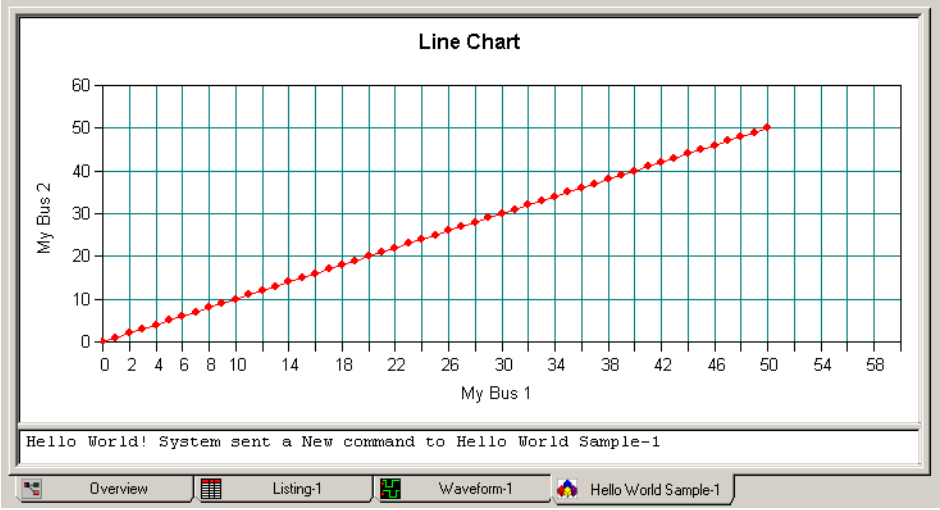



## Example: Line Chart

This example uses generated points, but you're more likely to use double arrays that you've obtained from the logic analyzer via GetDataBySample. Also, you can draw line charts with or without points. Use AgtChartTypeLineOnly for the ChartType if you don't want the points drawn.

```
Private Sub UpdateDisplay(ByVal strWindow As String)
  Dim myWindow As AgtLA.VbaViewWindow
  Dim myChart As AgtLA.VbaViewChart
  Dim myData As AgtLA.VbaViewChartData

  Set myWindow = AgtLA.Windows(strWindow)
  Set myChart = myWindow.Chart
  Set myData = myChart.Data
```

```
  ' Set the ChartType:
  myChart.ChartType = AgtChartTypeLine

  ' Set up the titles:
  myChart.HasTitle = True
  myChart.Title = "Line Chart"
  myChart.Axis(AgtChartAxisTypeX).HasTitle = True
  myChart.Axis(AgtChartAxisTypeX).Title = "My Bus 1"
  myChart.Axis(AgtChartAxisTypeY).HasTitle = True
  myChart.Axis(AgtChartAxisTypeY).Title = "My Bus 2"

  ' Populate the arrays:
  Dim xValueArray(50) As Double
  Dim yValueArray(50) As Double
  Dim i As Integer
  For i = 0 To 50
    xValueArray(i) = i
    yValueArray(i) = i
  Next i

  ' Clear the old data, add the new, and redraw:
  myData.Clear
  myData.AddPointArrays xValueArray, yValueArray, _
                        AgtDataPointTypeCircle, AgtDataPointSizeSmall
  myChart.Draw
End Sub
```




## Example: Bar Chart

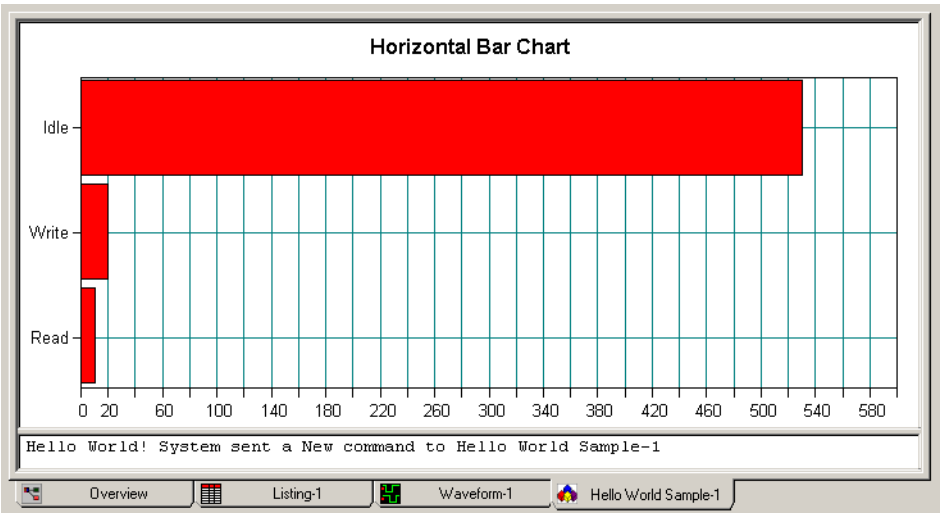
There are both vertical and horizontal bar charts. There are also regular bar charts and stacked bar charts. The VbaView window has the concept of "Values" and "Groups". Use both groups and values to deal with stacked bar charts. For regular bar charts, only values are needed.

- Example: Horizontal Bar Chart (see page 68)

- Example: Horizontal Stacked Bar Chart (see page 69)
- Example: Vertical Bar Chart (see page 70)
- Example: Vertical Stacked Bar Chart (see page 70)

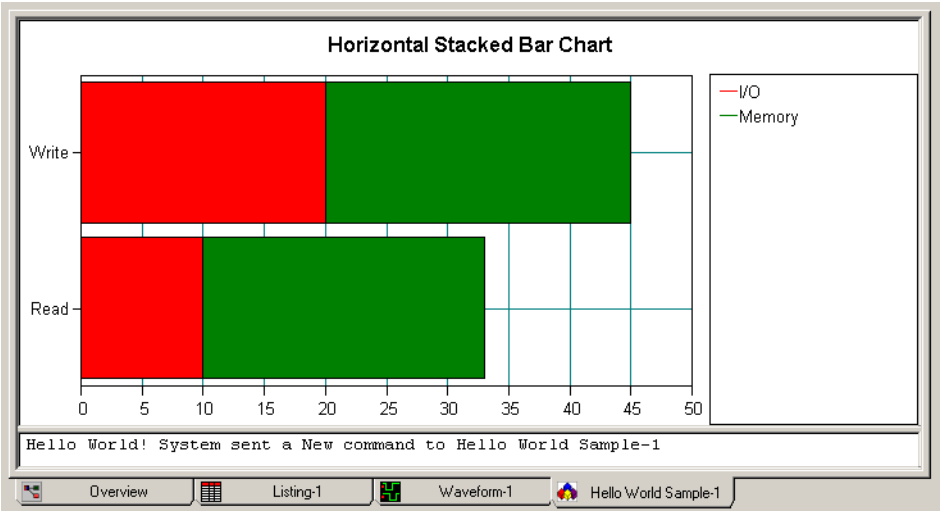
### Example: Horizontal Bar Chart

```
Private Sub UpdateDisplay(ByVal strWindow As String)
  Dim myWindow As AgtLA.VbaViewWindow
  Dim myChart As AgtLA.VbaViewChart
  Dim myData As AgtLA.VbaViewChartData

  Set myWindow = AgtLA.Windows(strWindow)
  Set myChart = myWindow.Chart
  Set myData = myChart.Data

  ' Set the ChartType:
  myChart.ChartType = AgtChartTypeHorizontalBar

  ' Set up the titles:
  myChart.HasTitle = True
  myChart.Title = "Horizontal Bar Chart"
  myChart.Axis(AgtChartAxisTypeX).HasTitle = False
  myChart.Axis(AgtChartAxisTypeY).HasTitle = False

  ' Clear the old data, add the new, and redraw:
  myData.Clear
  Call myChart.Data.SetValueCaption(0, "Read")
  Call myChart.Data.SetValueCaption(1, "Write")
  Call myChart.Data.SetValueCaption(2, "Idle")
  Call myChart.Data.SetValue(0, 0, 10)
  Call myChart.Data.SetValue(0, 1, 20)
  Call myChart.Data.SetValue(0, 2, 530)
  myChart.Draw
End Sub
```



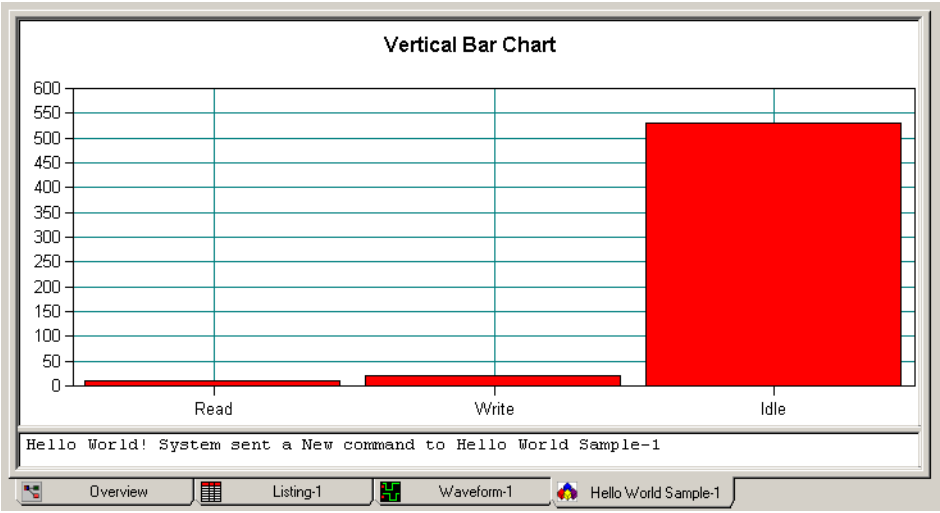
## Example: Horizontal Stacked Bar Chart

```
Private Sub UpdateDisplay(ByVal strWindow As String)
  Dim myWindow As AgtLA.VbaViewWindow
  Dim myChart As AgtLA.VbaViewChart
  Dim myData As AgtLA.VbaViewChartData

  Set myWindow = AgtLA.Windows(strWindow)
  Set myChart = myWindow.Chart
  Set myData = myChart.Data

  ' Set the ChartType:
  myChart.ChartType = AgtChartTypeStackedHorizontalBar

  ' Set up the titles:
  myChart.HasTitle = True
  myChart.Title = "Horizontal Stacked Bar Chart"
  myChart.HasLegend = True
  myChart.Axis(AgtChartAxisTypeX).HasTitle = False
  myChart.Axis(AgtChartAxisTypeY).HasTitle = False

  ' Clear the old data, add the new, and redraw:
  myData.Clear
  Call myChart.Data.SetValueCaption(0, "Read")
  Call myChart.Data.SetValueCaption(1, "Write")
  Call myChart.Data.SetGroupCaption(0, "I/O")
  Call myChart.Data.SetGroupCaption(1, "Memory")
  Call myChart.Data.SetValue(0, 0, 10)
  Call myChart.Data.SetValue(0, 1, 20)
  Call myChart.Data.SetValue(1, 0, 33)
  Call myChart.Data.SetValue(1, 1, 45)
  myChart.Draw
End Sub
```



## Example: Vertical Bar Chart

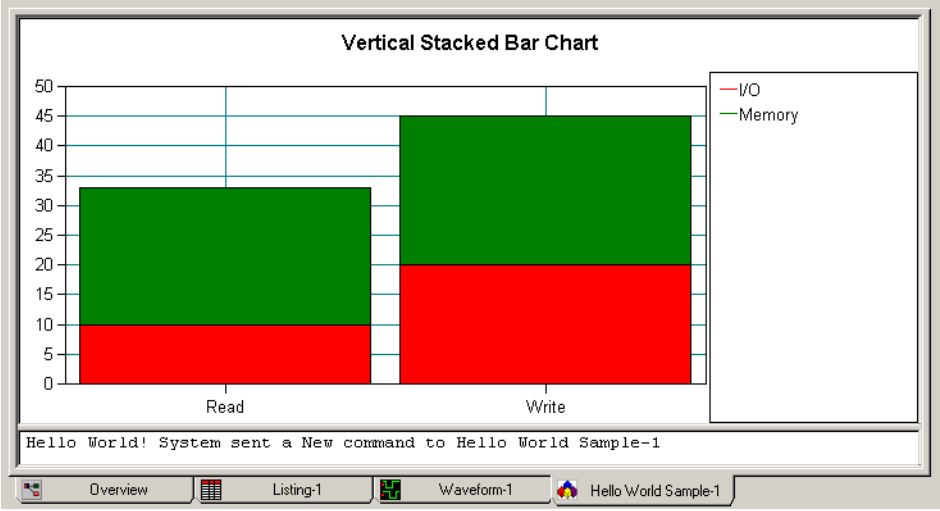
```
Private Sub UpdateDisplay(ByVal strWindow As String)
  Dim myWindow As AgtLA.VbaViewWindow
  Dim myChart As AgtLA.VbaViewChart
  Dim myData As AgtLA.VbaViewChartData

  Set myWindow = AgtLA.Windows(strWindow)
  Set myChart = myWindow.Chart
  Set myData = myChart.Data

  ' Set the ChartType:
  myChart.ChartType = AgtChartTypeVerticalBar

  ' Set up the titles:
  myChart.HasTitle = True
  myChart.Title = "Vertical Bar Chart"
  myChart.Axis(AgtChartAxisTypeX).HasTitle = False
  myChart.Axis(AgtChartAxisTypeY).HasTitle = False

  ' Clear the old data, add the new, and redraw:
  myData.Clear
  Call myChart.Data.SetValueCaption(0, "Read")
  Call myChart.Data.SetValueCaption(1, "Write")
  Call myChart.Data.SetValueCaption(2, "Idle")
  Call myChart.Data.SetValue(0, 0, 10)
  Call myChart.Data.SetValue(0, 1, 20)
  Call myChart.Data.SetValue(0, 2, 530)
  myChart.Draw
End Sub
```




## Example: Vertical Stacked Bar Chart

```
Private Sub UpdateDisplay(ByVal strWindow As String)
  Dim myWindow As AgtLA.VbaViewWindow
  Dim myChart As AgtLA.VbaViewChart
```

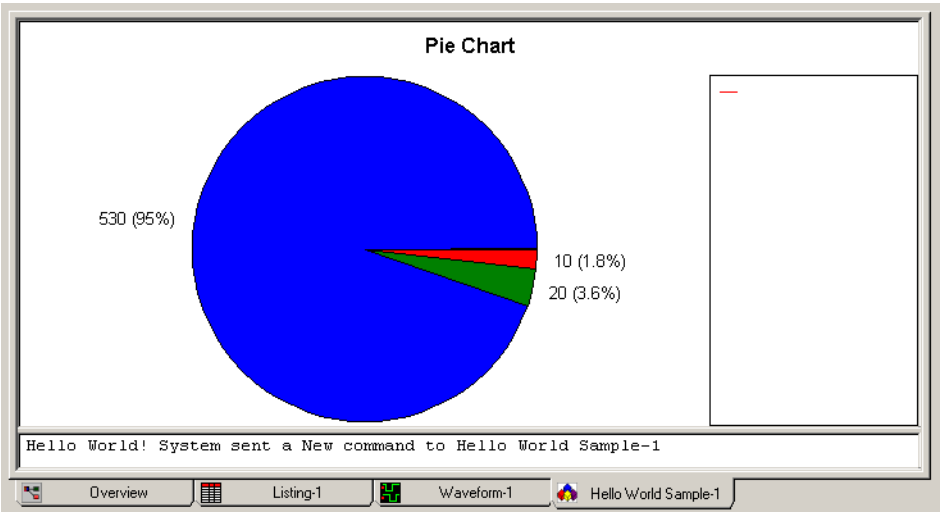
```
  Dim myData As AgtLA.VbaViewChartData

  Set myWindow = AgtLA.Windows(strWindow)
  Set myChart = myWindow.Chart
  Set myData = myChart.Data

  ' Set the ChartType:
  myChart.ChartType = AgtChartTypeStackedVerticalBar

  ' Set up the titles:
  myChart.HasTitle = True
  myChart.Title = "Vertical Stacked Bar Chart"
  myChart.HasLegend = True
  myChart.Axis(AgtChartAxisTypeX).HasTitle = False
  myChart.Axis(AgtChartAxisTypeY).HasTitle = False

  ' Clear the old data, add the new, and redraw:
  myData.Clear
  Call myChart.Data.SetValueCaption(0, "Read")
  Call myChart.Data.SetValueCaption(1, "Write")
  Call myChart.Data.SetGroupCaption(0, "I/O")
  Call myChart.Data.SetGroupCaption(1, "Memory")
  Call myChart.Data.SetValue(0, 0, 10)
  Call myChart.Data.SetValue(0, 1, 20)
  Call myChart.Data.SetValue(1, 0, 33)
  Call myChart.Data.SetValue(1, 1, 45)
  myChart.Draw
End Sub
```




## Example: Pie Chart

```
Private Sub UpdateDisplay(ByVal strWindow As String)
  Dim myWindow As AgtLA.VbaViewWindow
  Dim myChart As AgtLA.VbaViewChart
  Dim myData As AgtLA.VbaViewChartData
```

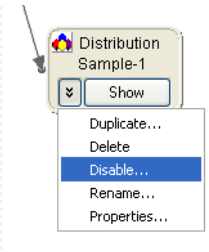
```
  Set myWindow = AgtLA.Windows(strWindow)
  Set myChart = myWindow.Chart
  Set myData = myChart.Data

  ' Set the ChartType:
  myChart.ChartType = AgtChartTypePie

  ' Set up the titles:
  myChart.HasTitle = True
  myChart.Title = "Pie Chart"
  myChart.HasLegend = True
  myChart.Axis(AgtChartAxisTypeX).HasTitle = False
  myChart.Axis(AgtChartAxisTypeY).HasTitle = False

  ' Clear the old data, add the new, and redraw:
  myData.Clear
  Call myChart.Data.SetValueCaption(0, "Read")
  Call myChart.Data.SetValueCaption(1, "Write")
  Call myChart.Data.SetValueCaption(2, "Idle")
  Call myChart.Data.SetValue(0, 0, 10)
  Call myChart.Data.SetValue(0, 1, 20)
  Call myChart.Data.SetValue(0, 2, 530)
  myChart.Draw
End Sub
```



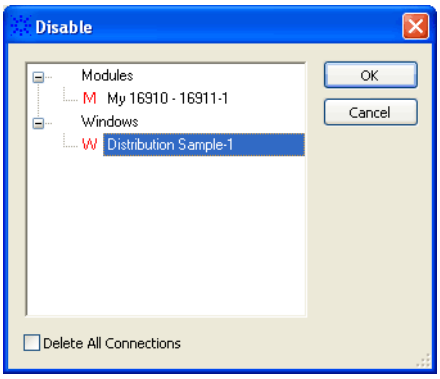
# Disabling VbaView Windows

Sometimes it can be useful to turn off VbaView window processing by disabling the window.

1 In the Overview window, select the drop-down menu for a VbaView window; then, choose **Disable...**.




2 In the Disable dialog, select the VbaView windows you wish to disable; then, click **OK**.




**To re-enable VbaView windows**

1 In the Overview window, select the drop-down menu for a VbaView window; then, choose **Enable...**.

2 In the Enable dialog, select the VbaView windows you wish to enable; then, click **OK**.

# 6 Distributing VBA Code

There are several ways to distribute VBA macro code:

- Through ALA format configuration files (if logic analyzers are compatible).
- Through XML format configuration files (if logic analyzers are incompatible).
- By exporting and importing individual Module or Form files.
- By exporting and importing VBA project .zip files. By placing VBA project .zip files in the "<Install>\VBA\" directory, you can cause the VBA code to load when the *Agilent Logic Analyzer* application starts up.

When distributing VbaView window code, it must load at application startup so the window appears in the **Window>New VbaView>** menu; therefore, VBA project .zip files are required.

# To distribute VBA code via ALA format configuration files

VBA code can be saved as part of ALA format configuration files.

If you want to share a VBA macro with another user who has a compatible logic analyzer, all you need to do is give them the ALA format configuration file.

However, if the other user has an incompatible logic analyzer,you must use one of the other methods for distributing VBA code.

**See Also**

- To distribute VBA code via XML format configuration files (see page 77)
- To distribute individual files (for VBA Modules/Forms) (see page 78)
- To distribute VBA project code via .zip files (see page 79)

# To distribute VBA code via XML format configuration files

VBA code can be saved as part of XML format configuration files.

If you want to share a VBA macro with another user who has an incompatible logic analyzer, all you need to do is give them the XML format configuration file.

**See Also**

- To distribute individual files (for VBA Modules/Forms) (see page 78)
- To distribute VBA project code via .zip files (see page 79)

## To distribute individual files (for VBA Modules/Forms)

This procedure describes how, using the VBA IDE, to export Module and Form VBA code to individual files from one ALA format configuration file and import those files into another configuration.

1. Open the ALA format configuration file that contains the VBA code to export.
2. In the Information dialog that tells you about incompatible modules, click **Load Offline**.
3. Choose **Tools>Macro>Visual Basic Editor** to open the VBA IDE.
4. In the VBA IDE, export each Module and Form from the VBA IDE to .bas and .frm files, respectively. To do this, right-click on the Module or Form in the Project Browser (upper left hand corner); then, choose **Export>File...**.
5. Now, if you are offline, choose **File>Go Online**; otherwise, create a new configuration by choosing **File>New**.
6. Choose **Tools>Macro>Visual Basic Editor** to open the VBA IDE.
7. In the VBA IDE's Project Browser, select the configuration into which the files should be imported.
8. Choose **File>Import...** to import the files that you previously exported.

**See Also**

- To distribute VBA code via ALA format configuration files (see page 76)
- To distribute VBA project code via .zip files (see page 79)

## To distribute VBA project code via .zip files

This procedure describes how to export all of a project's VBA code from one ALA format configuration file and import it into another configuration.

1 Open the ALA format configuration file that contains the VBA code to export.

2 In the Information dialog that tells you about incompatible modules, click **Load Offline**.

3 Choose **Tools>Macro>Export Zip File...**. (You can also choose **File>Export Zip File** from within the VBA IDE.)

4 In the Export Macros dialog, select the project whose VBA code should be exported; then, click **OK**.

5 In the next "Export macros in project" dialog, enter the name of the .zip file that will contain the project's VBA code; then, click **Save**.

If you want to transfer the logic analyzer configuration as well, save it to an XML format file.

6 Now, if you are offline, choose **File>Go Online**; otherwise, create a new configuration by choosing **File>New**.

If you saved the logic analyzer configuration to an XML format file in the previous step, open it now.

7 If you are importing code for a VbaView window project (and the VbaView window wasn't created by opening an XML format logic analyzer configuration file), choose **Window>New VbaView...**.

8 Choose **Tools>Macro>Import Zip File...**. (You can also choose **File>Import Zip File** from within the VBA IDE.)

9 In the Import Macros dialog, select the project into which the VBA code should be imported; then, click **OK**.

10 In the next "Select file to import" dialog, select the .zip file that contains the project's VBA code; then, click **Open**.

When importing VBA code from .zip files, only files with the following extensions are directly imported:

- .bas — Module
- .cls — Class Module
- .frm — Form

**See Also**

- To export VBA project code to .zip files (see page 80)
- To create VBA project code .zip files with agZip.exe (see page 80)
- To import VBA project code from .zip files (see page 84)
- To load VBA project code at application startup (see page 84)

## To export VBA project code to .zip files

This procedure describes how to export a VBA project's code to a .zip file.

1. Open the ALA format configuration file that contains the VBA project code to export.
2. Choose **Tools>Macro>Export Zip File...**. (You can also choose **File>Export Zip File** from within the VBA IDE.)
3. In the Export Macros dialog, select the project whose VBA code should be exported; then, click **OK**.
4. In the next "Export macros in project" dialog, enter the name of the .zip file that will contain the project's VBA code; then, click **Save**.

**See Also**

- To create VBA project code .zip files with agZip.exe (see page 80)
- To import VBA project code from .zip files (see page 84)
- To load VBA project code at application startup (see page 84)

## To create VBA project code .zip files with agZip.exe

This procedure describes how to create a VBA project .zip file using the **agZip.exe** program.

1. Place all VBA project source files in a directory.
2. Add a Project.xml file to the directory (see Project.xml File Format (see page 81)).
3. Open a Command Prompt window, and run the command:

```
agZip.exe <directory>
```

The **agZip.exe** program is located in the directory:

```
<Install directory>\
```

For example:

```
C:\Program Files\Agilent Technologies\Logic Analyzer\
```

**Password Protecting VBA Project Zip Files**

The **agZip.exe** executable generates an encrypted, password protected zip that ONLY the *Agilent Logic Analyzer* application can unencrypt. No standalone unzip executables exist.

The password protected zip is a one-way street so that access to the zip file can only be done via the application or the vendor directly sending sources. Think of a password protected VBA project zip file as a binary file.

**When VBA Projects are Password Protected**

If a project is password protected in the VBA IDE, or the zip file is password protected, the project cannot be exported to a zip file via **Tools>Macro>Export Zip File...**.

For the case of password protected zip files with no VBA IDE password (like AgtRPICmds, for example), you can only export files individually via the **Export File...** command in the VBA IDE.

**See Also**

- To export VBA project code to .zip files (see page 80)
- To import VBA project code from .zip files (see page 84)
- To load VBA project code at application startup (see page 84)

### Project.xml File Format

The Project.xml file identifies the type of VBA project a .zip file contains. XML format elements and attributes provide for password protection, licensing, and other complexities of distributing VBA intellectual property. XML elements for the Project.xml file have the following hierarchy:

```
<VbaProject> (see page 82)
   <References> (see page 81)
      <Reference> (see page 81)
```

**<Reference> Element** The <Reference> element contains type library information.

**Attributes**

| Name | Description |
|---|---|
| Guid | 'string'. The type library GUID. |
| Major | 'string'. Major type library version. |
| Minor | 'string'. Minor type library version. |
| Description | 'string'. The description is just to make the XML readable and is not required. |

**Parents** This element can have the following parents: <References> (see page 81).

**Example**

```
<Reference Description='Microsoft Internet Controls'
      Guid='{EAB22AC0-30C1-11CF-A7EB-0000C05BAE0B}'
      Major='1' Minor='1' />
```

**<References> Element** The <References> element contains references to type libraries. References are added when the project is loaded so that external references will not cause syntax errors. These references are added manually by choosing **Tools>References...** in the VBA IDE. The <References> element can contain multiple references.

**Children** This element can have the following children: <Reference> (see page 81).

**Parents** This element can have the following parents: <VbaProject> (see page 82).

**Example**

```
<References>
   <Reference Description='Microsoft Internet Controls'
         Guid='{EAB22AC0-30C1-11CF-A7EB-0000C05BAE0B}'
         Major='1' Minor='1' />
   <Reference Description='Microsoft Forms 2.0 Object Library'
         Guid='{0D452EE1-E08F-101A-852E-02608C4D0BB4}'
         Major='2' Minor='0' />
</References>
```

**<VbaProject> Element** The `<VbaProject>` element lets the *Agilent Logic Analyzer* application know that this is a valid Project.xml file. If this element is not present, all other information is ignored. The `<VbaProject>` element is the top element in the Project.xml file.

**Attributes**

| Name | Description |
|---|---|
| `Type` | `'VbaView'` or `''`. If `'VbaView'`, this project will not be loaded at application startup because it needs to be loaded into an existing "VbaView" project. All of "VbaView" projects are placed in the menu **Window>New VbaView>**. |
| `Name` | `'string'`. Typically, this attribute isn't set, and the name is the base name of the project zip file. This attribute overrides the default behavior. |
| `Action` | `'Recall'` (default) or `'Create'`. Contains the action to take with this project.<br>`'Recall'` — the zip contents are imported into an already existing project. If the project has the `Name` attribute set, it will highlight that project in the Import Zip File project selection dialog. This attribute is set when a zip file is created from the **Export Zip File...** menu.<br>`'Create'` — the zip contents are loaded into a newly created project. See the `Name` attribute for the new project name. This is only valid for non "VbaView" type projects because "VbaView" type projects can't be created directly and must be associated with a view. This attribute is usually used in zip files that are created by Agilent or third parties and installed into the <Install directory>/VBA directory so they can be automatically loaded at startup. This type of project can also be loaded directly via the **Import Zip File..** menu if you don't want it loaded at application startup. If the project has already been loaded, an error will be returned. |
| `Description` | `'string'`. The description is used when displaying error messages. For example, this will be used when the license can't be obtained. If this is not set, the full zip file name is used as the description. |

| | |
|---|---|
| UseDefaultPassword | 'F', False or 'T', True. If this is not set, or is set to 'False' or 'F', and the zip file contains a password, the project will be password protected with the password used in the zip. If the Password attribute is set, it overrides the UseDefaultPassword attribute which only works for the default password. The UseDefaultPassword attribute is used to support projects like "AgtRPICmds" where the zip is password protected but users need to be able to view the source. |
| Password | 'string'. This is the password string used to password protect the VBA project in the VBA IDE. If this isn't set, the password used to encrypt the zip file will be used if UseDefaultPassword isn't True. Because the encrypted password used in the zip is a one-way street (see Password Protecting VBA Project Zip Files (see page 80)), vendors can set a password that they control in case they need access to the VBA project for debugging the macro at a customer site. |
| LicenseName | 'string'. If a LicenseName is specified, the application will look for this license (which is different from the VBA Runtime license). If the license doesn't exist, the project isn't loaded, and no errors will be displayed. |
| LicenseVersion | 'string'. If the LicenseName attribute is specified, and this is not set, then "1.0" will be used. |
| LicenseVendor | 'string'. If the LicenseName attribute is specified, and this is not set, then "Agilent Technologies" will be used. |

**Children** This element can have the following children: <References> (see page 81).

**Parents** None.

**Example**

```
<VbaProject Type='VbaView' Name='Export to IE Sample'
      HelpFile='VBA_View_Export_to_IE.chm'>
   <References>
      <Reference Description='Microsoft Internet Controls'
            Guid='{EAB22AC0-30C1-11CF-A7EB-0000C05BAE0B}'
            Major='1' Minor='1' />
      <Reference Description='Microsoft Forms 2.0 Object Library'
            Guid='{0D452EE1-E08F-101A-852E-02608C4D0BB4}'
            Major='2' Minor='0' />
   </References>
</VbaProject>
```

## To import VBA project code from .zip files

This procedure describes how to import VBA project VBA code from a .zip file.

1 In the *Agilent Logic Analyzer* application, choose **Tools>Macro>Import Zip File...**. (You can also choose **File>Import Zip File** from within the VBA IDE.)

2 In the Import Macros dialog, select the project into which the VBA code should be imported; then, click **OK**.

3 In the next "Select file to import" dialog, select the .zip file that contains the project's VBA code; then, click **Open**.

When importing VBA code from .zip files, only files with the following extensions are directly imported:

- .bas — Module
- .cls — Class Module
- .frm — Form

**See Also**

- To export VBA project code to .zip files (see page 80)
- To create VBA project code .zip files with agZip.exe (see page 80)
- To load VBA project code at application startup (see page 84)

## To load VBA project code at application startup

1 Place the VBA project zip file in the directory:

```
<Install directory>\VBA\
```

For example:

```
C:\Program Files\Agilent Technologies\Logic Analyzer\VBA\
```

Projects are loaded automatically at startup if they exist in the VBA installation directory.

**At Application Startup**

If the *Agilent Logic Analyzer* application has at least a Runtime VBA license, an attempt will be made to automatically load the zip projects installed. The application looks for the Runtime VBA license but does not prompt for one if it is missing because it is confusing having the license dialog display at startup.

The following algorithm for handling VBA project zip files is also used in **Tools>Macro>Import Zip File...** because you may not want a project loaded at startup when you can just recall it an any time.

The *Agilent Logic Analyzer* application looks through all of the zip files in the VBA directory. For each zip file:

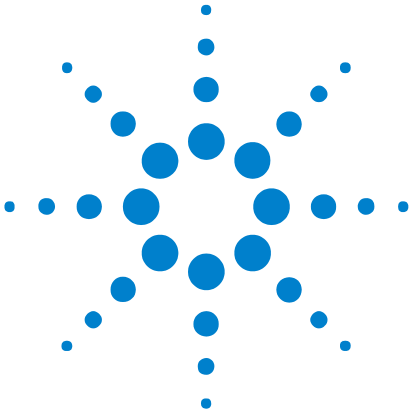
1 Look for a Project.xml file. If the file does not exist, the project will be loaded directly. See When a Project is Loaded (see page 85) below for details. The main objective of the Project.xml file is to handle the complexity of password protection, licensing, and distribution of VBA intellectual property.

2 If a Project.xml file exists, the application looks for valid XML elements (see Project.xml File Format (see page 81)).

It is strongly recommended if licensing information or the password is set in the Project.xml file, the zip file should be password protected. See Password Protecting VBA Project Zip Files (see page 80).

**When a Project is Loaded**

When a VBA project is loaded at application startup, it is placed in a project that is not saved with the ALA format configuration file.



# 7
# Visual Basic Programming Tips

# Visual Basic Syntax

- VBA divides the code into modules and forms. Modules are just code while forms are both GUI controls and code. In general, modules are used to contain code that is global to the project or that spans multiple forms.
- All code must be in a subroutine or a function. A subroutine does not return a value but a function does.

```
Public Sub MySubroutine(Byval i as integer, ByRef retVal as boolean)
```

  - This is public so it can be called outside of this module or form. The other alternative is Private.
  - ByVal means this parameter is passed by value. In other words, if you don't want MySubroutine to change the value of i, then make it "ByVal".
  - ByRef means this parameter is passed by reference. In this case, MySubroutine can change the value of retVal.
  - To call this subroutine, use Call MySubroutine(i, retVal)

```
Private Function MyFunction(ByVal i as integer) as boolean
```

  - Notice the "as boolean" at the end of this function declaration. This means this function returns a boolean value.
  - To set the return value, pretend that "MyFunction" is a variable and give it as value, as in "MyFunction = true". VBA does not use "Return true".
  - To call this function, use myBool = MyFunction(i)
- Variables are defined in the following manner:
  - Local variables within a subroutine or function. `Dim myVal as long`.
  - Private variables within a module or form. `Private myVal as integer`.
  - Public variables within a module or form. `Public myVal as string`.
- Common data types are:
  - Integer (no signed or unsigned; everything except boolean and string are signed)
  - Long
  - String
  - Boolean
  - Double (Use for real numbers; also used by the VbaView graphing tools).

- Defining and using arrays:
  - If you know the dimensions of the array. `Dim myArray(5, 5) as boolean`
  - If you or another function will re-dimension the array later. (GetDataBySample does this). `Dim myArray() as boolean`
  - To use the array, `myArray(x,y)`
- Common control structures:

```
If (bEntering) Then
  strEnteringExiting = "Entering"
Else
  strEnteringExiting = "Exiting"
End If

For i = 0 To myNumDataRows - 2
  If (OEArray(i) = 0 And WEArray(i) = 1) Then
    dReadTime = dReadTime + TimeArray(i + 1) - TimeArray(i)
  End If
Next i

Do While (bFound)
  bFound = FindTransition(bRead, True, True, dEnteringTime)
Loop

Select Case Command:
  Case "Show"
  Case "Delete"
  Case "Update" : Call DrawChart
End Select
```

## Guidelines for C++ Programmers

- Visual Basic has come a long way from the original BASIC, but it's not C++ either.
- VBA is based upon VB6, not VB.NET. VB.NET works quite a bit different from VB6, so if you refer to a book or website make sure that you are dealing with VB6 or VBA but not .NET.
- There is no scoping of variables within a subroutine.
- There are no pointers at all. (But for VBA, you probably won't miss them).
- There is no concept of resource files. There are forms are used to define dialogs, but forms contain code. They are not like Visual C++ resource files.
- In general, modules are code only and forms are both GUI and code.
- Variables do **not** need to be defined before they are used. This means that a misspelled variable will not be detected by the compiler. You can, however, use the "`Option Explicit`" compiler option to tell VBA that variables should be defined before they are used.
- The single "=" is used both as an assignment operator and as a comparison operator. VB doesn't support "==". So, there is "`if(myVal = 1)`" and "`myVal=1`". The system knows the difference because of the "If".
- VB does not use "!". In general, they use "not", although "not equals" is "<>".
- VB cares about line breaks. To make code wrap to the next line, you must end in a " _". Notice that there is a space before the underscore.
- VB likes you to say "Call" before calling a subroutine, as in "`Call UpdateWindow(true)`".
- VB is not case sensitive like C and C++.
- VB is not object oriented. There is no inheritance.
- VB does not use { or }. It uses "then" and "end if".

## Common VBA Error Messages

There are a couple of common VBA mistakes.

- One common mistake is a failure to put a "Set" in front of a function that returns an object. For example:

```
Dim myFindResult as AgtLA.FindResult
myFindResult = AgtLA.Windows(0).Find(strEvent)
```

  This code results in the error message "Object Variable or With block variable not set". Instead, this code should be used:

```
Dim myFindResult as AgtLA.FindResult
Set myFindResult = AgtLA.Windows(0).Find(strEvent)
```

- When using VBA with the *Agilent Logic Analyzer* application, it is unusual to create a new object using the "New" keyword. The problem with the example below is that we try to create a new FindResult object, but the Find function already returns a FindResult object.

```
Dim myFindResult as New AgtLA.FindResult
Set myFindResult = AgtLA.Windows(0).Find(strEvent)
```

  This code results in the error message "ActiveX Component is unable to create object". To fix this problem, you must remove the "New".

# 7 Visual Basic Programming Tips

# Index

## N



## O



## P



## R



## S



## T



## U



## V



## W



## X