N5990A Remote Test GUI and Remote Programming Description

Version 1.0

Contents

Remote Test GUI....3
Introduction....3
Initialization....4
Configuration....8
Execution of Single Test Procedures....12
Result Handling....14
Remote Interface Reference....19
Initialization and Configuration....20
Execution and Flow Control....22
Result Handling and Other Information....23
Events....25
Helper Classes....26

List of Figures

Figure 1: Typical ValiFrame flow diagram....3
Figure 2: ValiFrame Remote Interface – Test GUI start screen....4
Figure 3: Selection of the application via the application combo box....6
Figure 4: Test GUI after an application has been selected....7
Figure 5: Configure DUT dialog (example MIPI D-PHY application)....8
Figure 6: Procedure list after the application has been configured....9
Figure 7: Example of procedure properties after the MIPI D-PHY LP High Level Clock Calibration has been selected....10
Figure 8: Connection diagram example....12
Figure 9: Example MS Excel Result output and the xml-formatted output....14
Figure 10: View of the ValiFrameRemote class via the Visual Studio's Object Browser....19

Remote Test GUI

Introduction

The Agilent Technologies Test Automation Software Platform N5990A “ValiFrame” provides a remote programming interface (RPI). This RPI is part of the N5990A options 010 and 009. Option 009 is the Base Core Product. It is restricted to remote use and comprises

- a remote interface,
- a basic graphical user interface (GUI) for test purposes,
- drivers for the supported Agilent Technologies test instruments,
- MS Excel report generation, and
- multi-application support (i.e., multiple applications such as MIPI D-PHY, PCI Express, SATA or USB can be installed on the same controller PC and be run through this option).

The N5990A Core Product Option 010 provides a full GUI. The basic GUI of Option 009 provides reduced functionality. It is intended to be used for testing the functionality of automated tests. In the following, it is also referred to as “Test GUI”. The Test GUI uses the valiframeRemote.dll to control the test and calibration procedures.

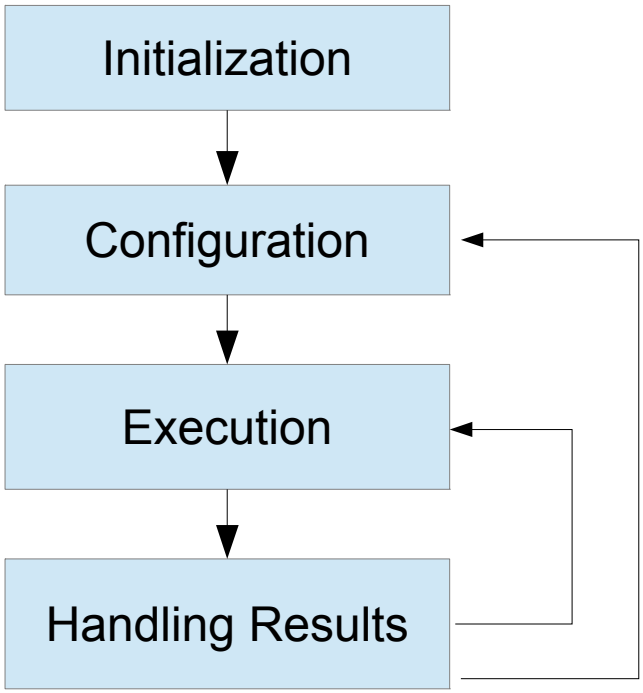

Figure 1: Typical ValiFrame flow diagram

This document describes the functions of the remote interface and shows how to use them for a typical application. The flow of usage is always the same. First, the application needs to be selected and initialized, then the system needs to be configured, then the tests and calibrations can be carried out, and finally, the results need to be processed, as shown in Figure 1. Usually the relevant Test GUI functions are explained first, then adjacent example code using the remote interface is given. Example code and explanations of use of the remote interface are highlighted by a gray background.

Initialization

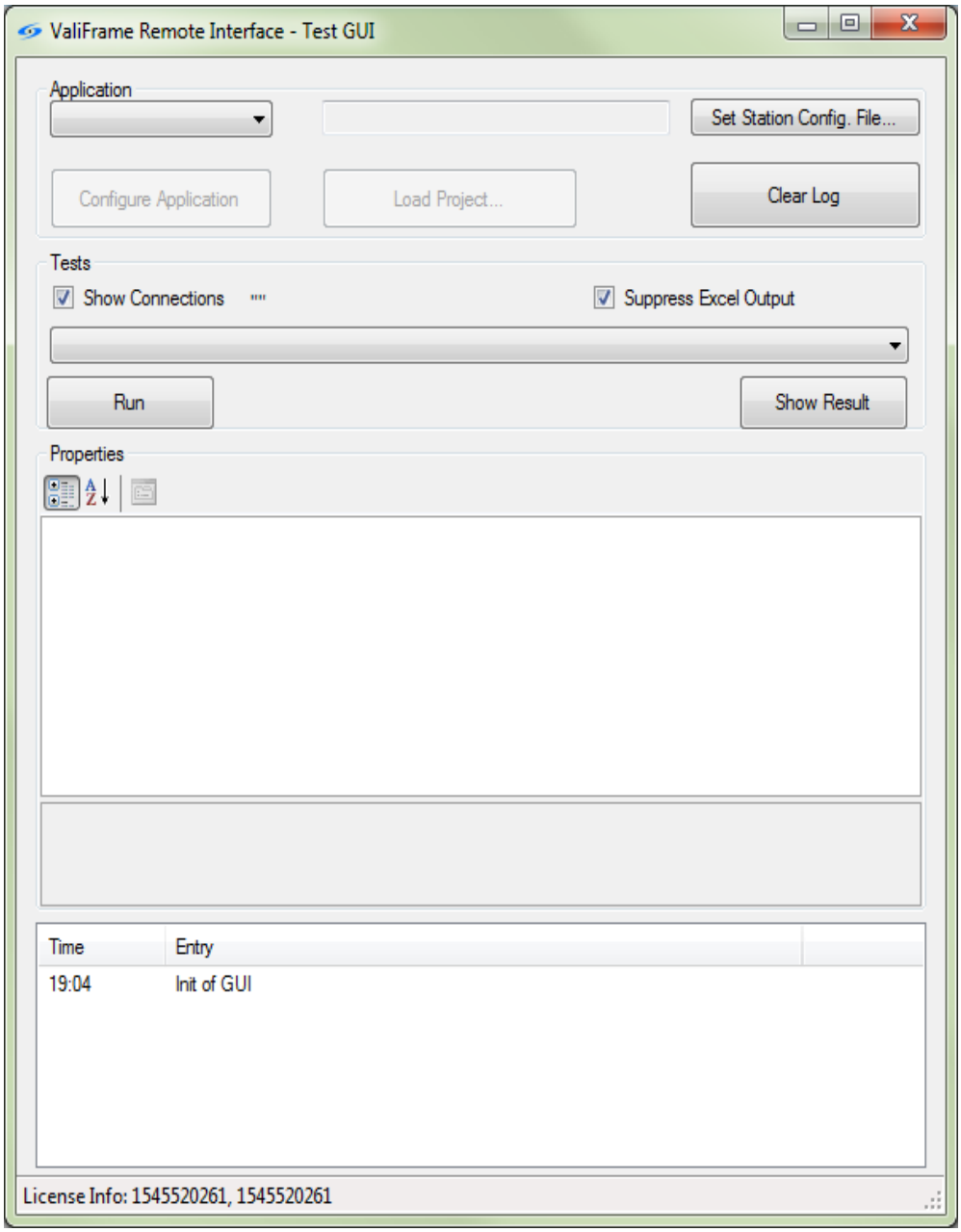

Figure 2: ValiFrame Remote Interface – Test GUI start screen

BitifEye.ValiFrame.ValiFrameRemote.ValiFrameRemote() //constructor

To get a reference to the ValiFrame remote control, call the ValiFrameRemote() constructor.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using BitifEye.ValiFrame.ValiFrameRemote;
using BitifEye.ValiFrame.Sequence;
using System.IO;
using BitifEye.ValiFrame.Base;
using System.Threading;

namespace MyNameSpace
{
        public class MyClass : Form
        {
                ValiFrameRemote m_VfRemote;
                public MyClass()
        {
                        m_VfRemote=new ValiFrameRemote();
        …
```

ValiframeRemote is the class that provides the entire functionality to control ValiFrame remotely. In the following the functions are explained in the order as they are called in a typical application.

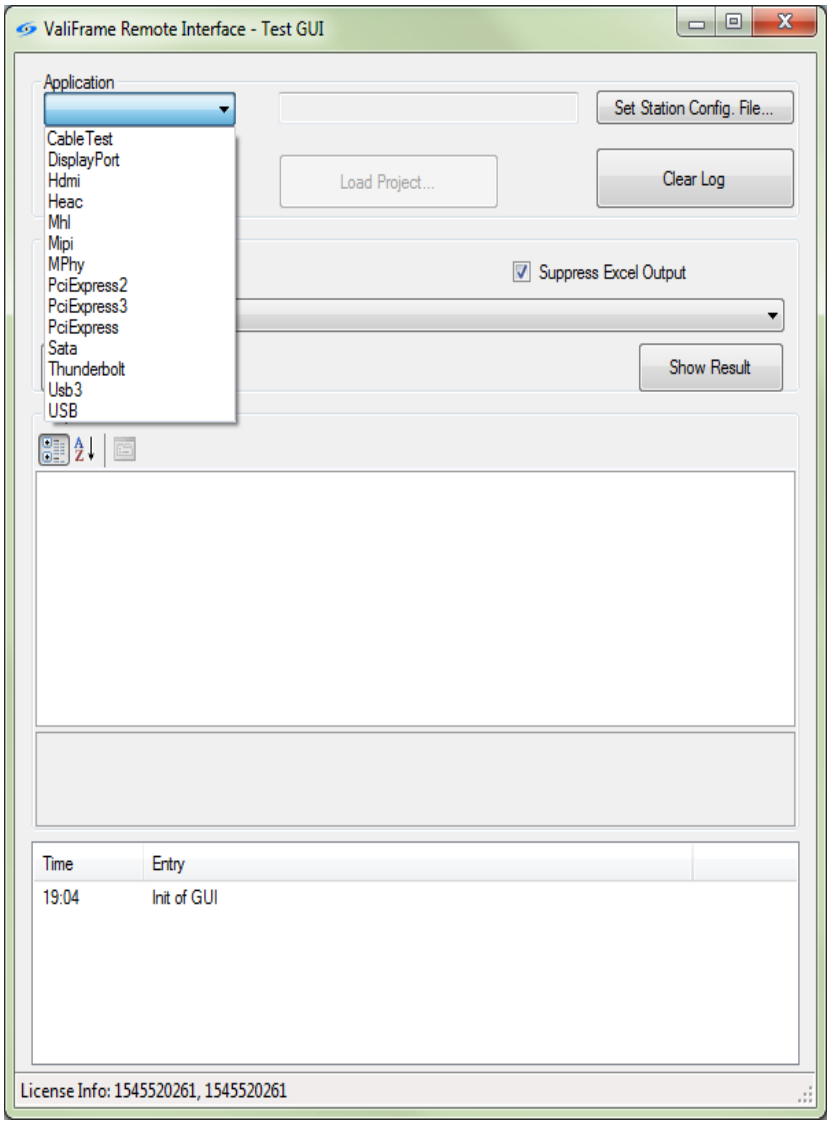

Figure 3: Selection of the application via the application combo box

Once the Test GUI has been started (Figure 2), the application needs to be selected. All applications installed in the same folder as the Test GUI are visible in the combo box at the top left of the GUI (Figure 3). After one of the entries has been selected, the corresponding test station is initialized. The required instruments are connected and the first initialization of these instruments is run.

GetApplications()

```
string[] applicationNames=m_VfRemote.GetApplications();
```

Get a list of available applications. It can be used in the InitApplication() function to select the desired DUT type.

```
m_VfRemote.InitApplication("Mipi");
```

Connect to the instruments and initialize them.

The configuration file of the instruments can be selected. If no configuration file is selected, the default setup generated by the ValiFrame Station Configurator is used. If the Station Configurator has not previously

been used, the default settings are used. The default settings can be found in the registry (Local Machine\Software\Bitifeye\ValiFrame\Stations\<Application Name>Station).

SetConfigurationFileName(string filename)

```
m_VfRemote.SetConfigurationFileName(filename);
```

Once an application is selected, ValiFrame checks the software license status. If no license is installed, request the license file from BitifEye (licensing@bitifeye.com). In order to generate a license file, the hardware ID of the test controller PC is required. It is displayed in the status bar.

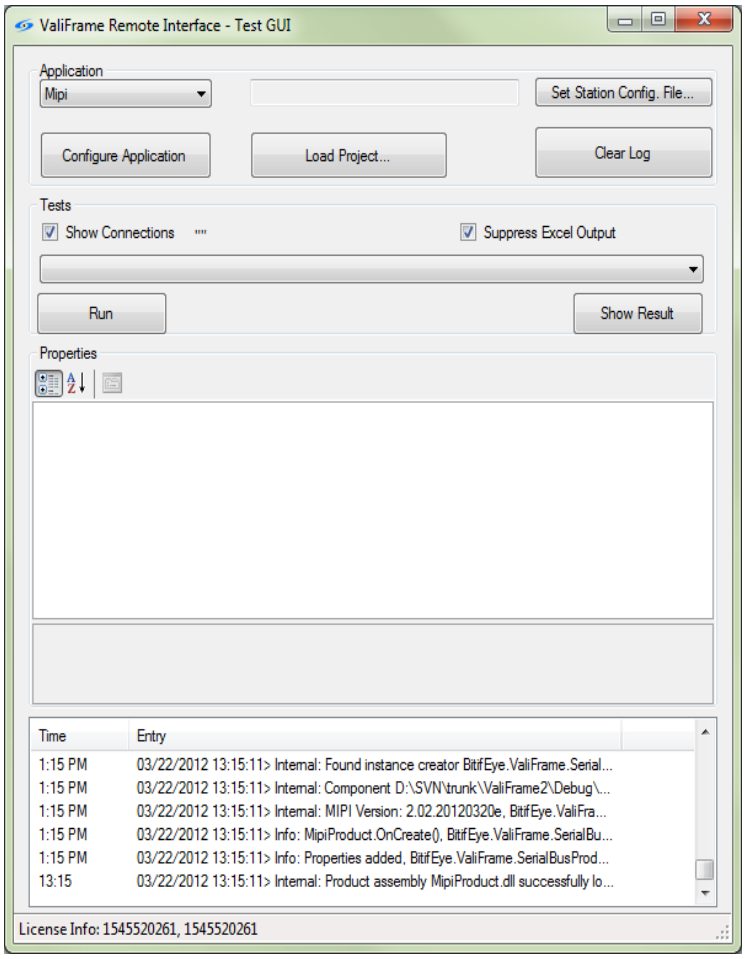

Figure 4: Test GUI after an application has been selected

GetLicenseInfo()

```
string licenseInfo = m_VfRemote.GetLicenseInfo();
```

Configuration

After the selection of an application, the buttons “Configure Application” and “Load Project...” are enabled (Figure 4).

- “Configure Application” opens the “Configure DUT” (Device Under Test) panel (Figure 5) with which the DUT and high-level application parameters are set.
- Alternately, the configuration can be achieved by loading a configuration file. This file can be generated by the full ValiFrame GUI (Option 010), for example.

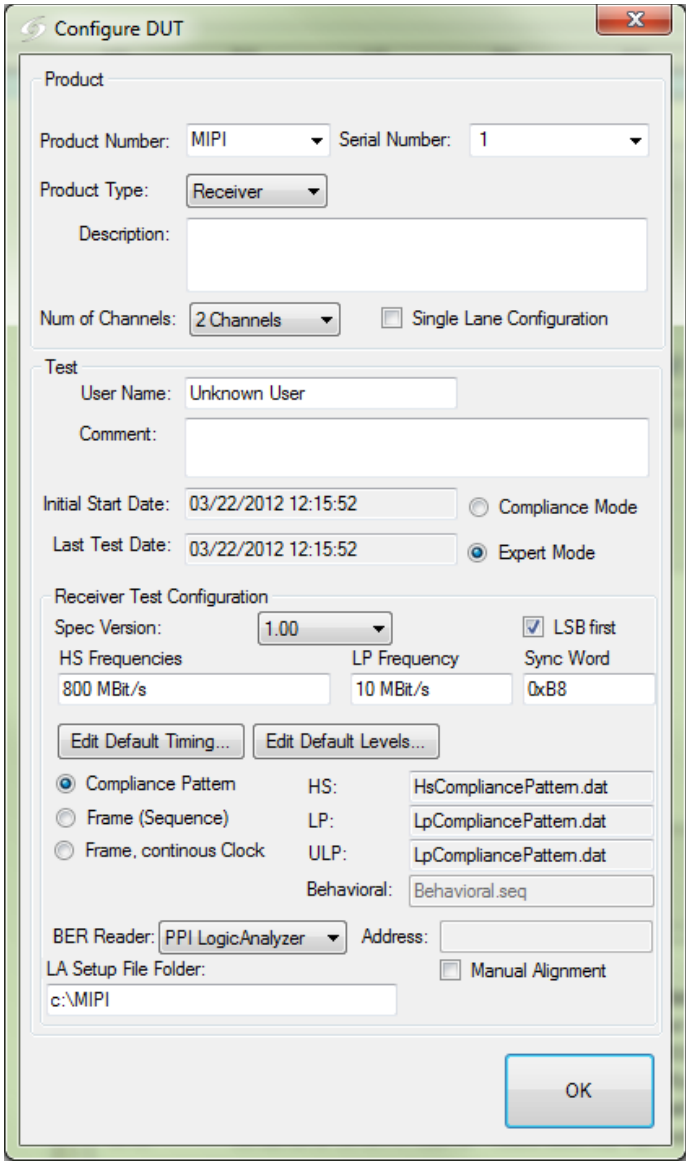

Figure 5: Configure DUT dialog (example MIPI D-PHY application)

ConfigureProduct()
LoadProject(string filename)

The functionality of the "Configure Application" button is available through ConfigureProduct()

```
m_VfRemote.ConfigureProduct();
```

For "Load Project..." LoadProduct(string settingsFile) can be used

```
m_VfRemote.LoadProject("myMipiProjectFile.vfc");
```

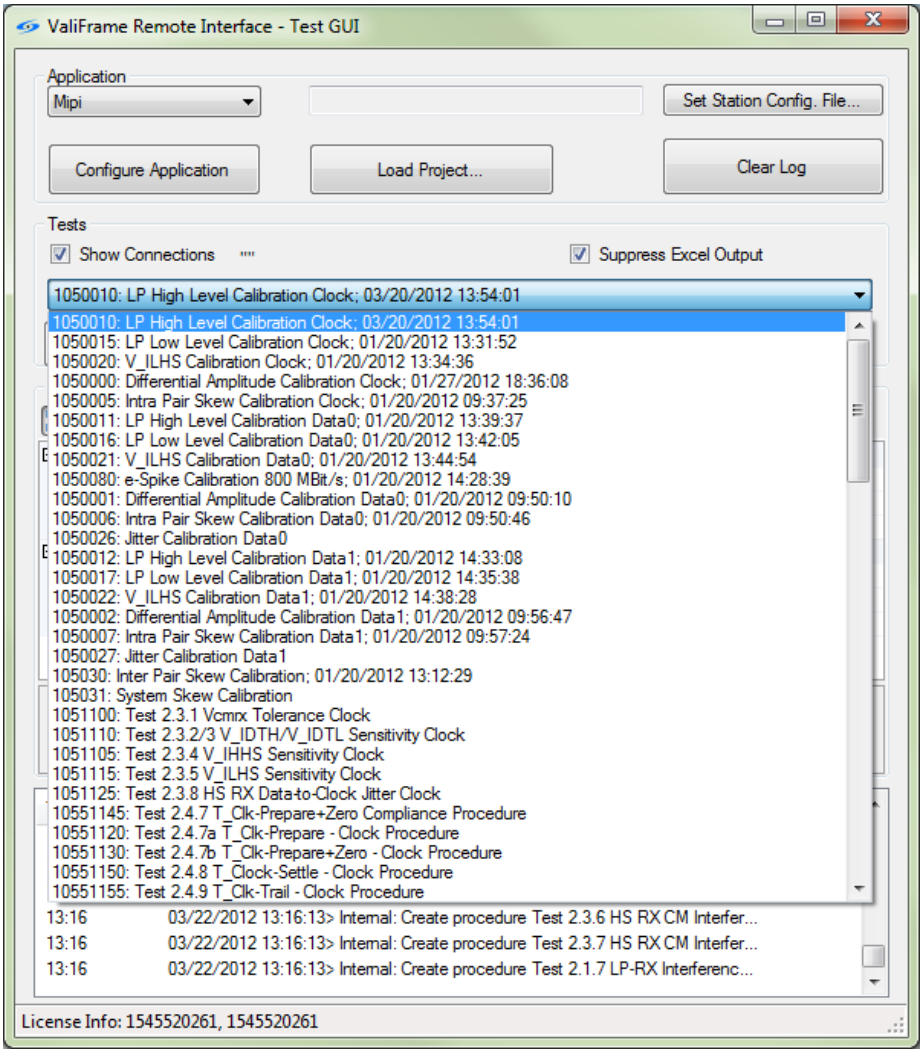

Figure 6: Procedure list after the application has been configured

After the configuration, the test and calibration procedures are available. All procedures are listed in the procedure combo box: Each procedure has a unique number – the procedure ID – and name (Figure 6).

GetProcedures(out int procedureIDs, out string procedureNames)

This function returns two arrays (procedure IDs and procedure names). The code shows how to access the procedure IDs and names:

```
int[] procedureIDs;
string[] procedureNames;
m_VfRemote.GetProcedures(out procedureIDs, out procedureNames);
proceduresComboBox.Items.Clear();
for(int i = 0; i < procedureIDs.Length; i++)
{
        proceduresComboBox.Items.Add(procedureIDs[i].ToString() + ": " +
        procedureNames[i]);
}
```

Access to all procedure parameters and the execution of the procedures require the procedure ID. If an individual procedure is selected, its properties are shown in the “Properties” grid in the middle of the Test GUI (Figure 7) and can be modified there.

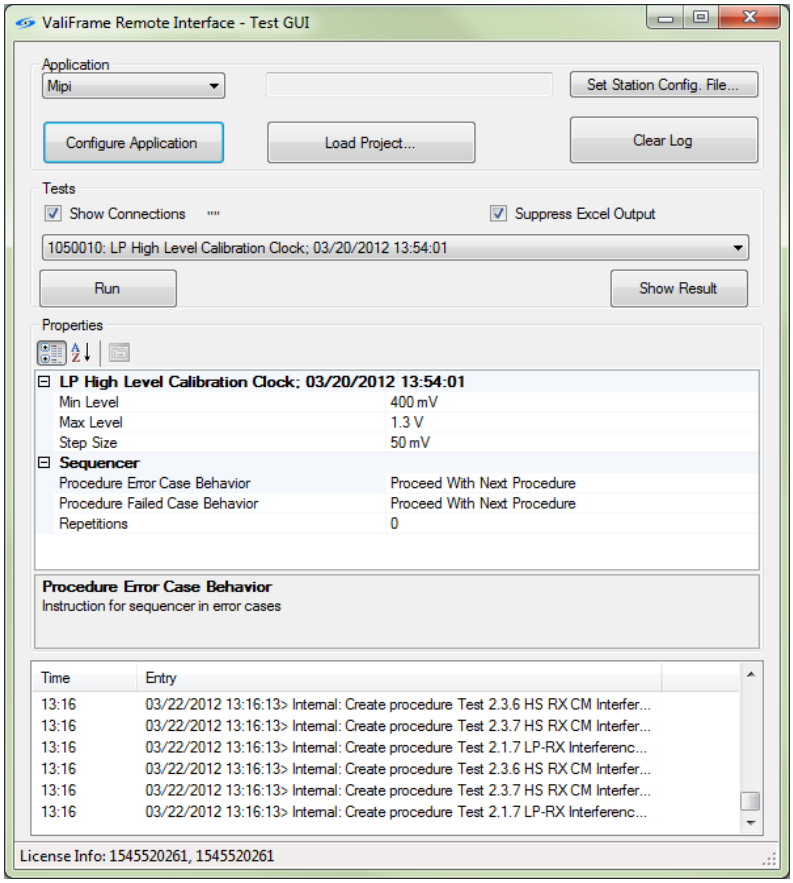

Figure 7: Example of procedure properties after the MIPI D-PHY LP High Level Clock Calibration has been selected

GetProcedureProperties(int procedureID)
SetProcedureProperty(int procedureID,string propertyName, string propertyValue)
GetProcedureRelatedProperties(int procedureID)
SetProcedurePropertiesToDefault(int procedureID)

Several functions are available to access the procedure properties remotely. The overloaded function GetProcedureProperties(string procedureName) and GetProcedureProperties(int procedureID) can be used to get a list of VFObjects. The VFObjects have names that can be used to access the properties' values. The names are the same as the names in the Test GUI "Properties" grid (Figure 7).

An alternate function to access the procedure properties is the GetProcedurePropertiesList(int procedureID). It returns a System.Collections.Generic.Dictionary. Via the properties' names the values can be changed by SetProcedureProperty(). This function is overloaded and allows the properties' values to be modified via several pathways.

```
SetProcedureProperty(1050010,"Min Level","500 mV");
```

The function SetProcedurePropertiesToDefault() is used to reset all values to its default (i.e., starting) values.

The procedure properties are structured hierarchically. In addition to the procedure properties, the higher level test group and the application (top level) can have properties. These properties can impact the test and calibration procedures. All properties can be modified via the remote interface. To access the properties in the higher hierarchy, use the method GetProcedureRelatedProperties(). The usage is the same as for the GetProcedureProperties() method.

Execution of Single Test Procedures

After a specific test or calibration procedure has been selected and, optionally, the procedure and/or higher level properties have been modified, the procedure is executed by pressing the “Run” button. If “Show Connection” (see Figure 7) is checked, typically a connection diagram pops up (Figure 8).

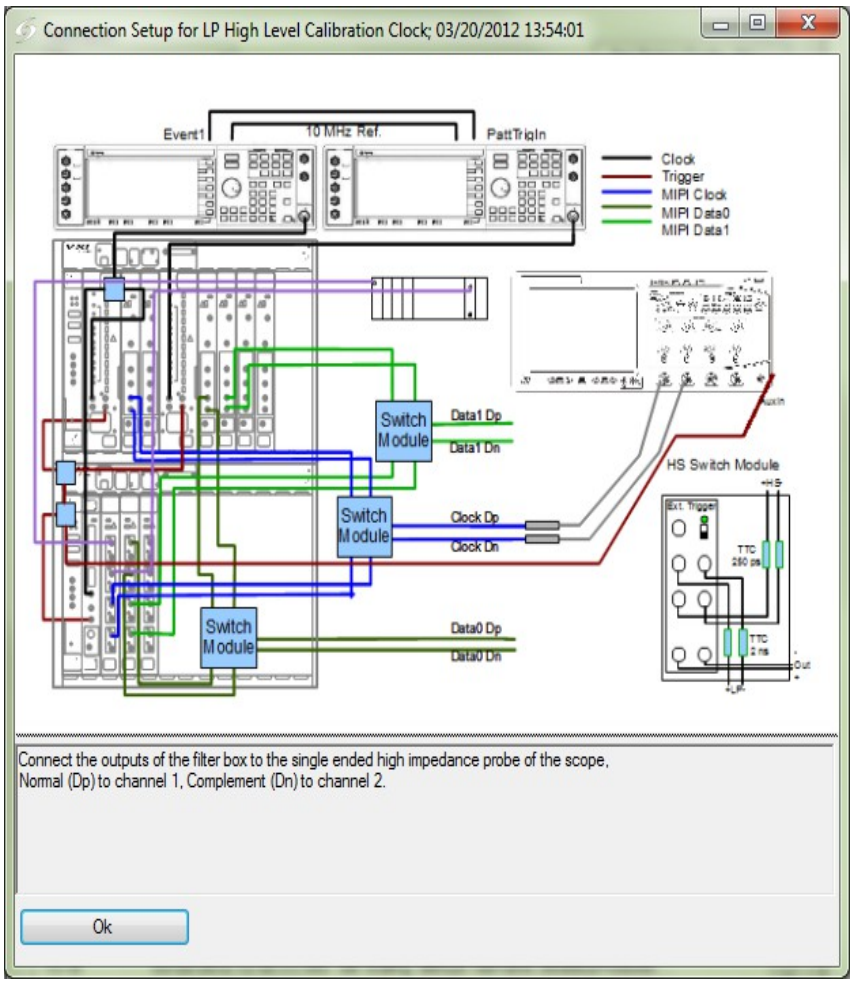

Figure 8: Connection diagram example

void RunProcedure(int procedureID, out string xmlResult)
void RunProcedures(int[] procedureIDs, out string[] xmlResults)
void SelectProcedures(int[] procedureIDs)
void StartRun()
event StatusChanged()
event ProcedureComplete

Remotely, test and calibration procedures can be run either synchronously or asynchronously. The synchronous method RunProcedure(int procedureID, out string xmlResult) is used to run a single procedure, RunProcedures(int[] procedureIDs, out string[] xmlResults) to run a list of procedures. These methods will only return after the procedure or the list of procedures has been fully executed. As it can take several hours to run an individual procedure (such as the MIPI D-PHY common mode interference tests) completely, it is not recommended that a windows form event function Is used to execute these functions. In this case the user interface would freeze until the procedure is finished.

A better way is to execute the procedure(s) in a separate thread, or to use the asynchronous method StartRun() in combination with the SelectProcedures(int[] procedureIDs) method and the events StatusChanged and ProcedureCompleted. The latter is fired as soon as one of the selected procedures is completed. The arguments of the event handler contain the procedure ID and the result string.

```
       private void StartButton_Click(object sender, EventArgs e)
              {
                     m_VfRemote.SelectProcedures(GetSelectedProcedureIds());
                     m_VfRemote.SetExcelSuppressState(suppressExcelOutputCheckBox.Checked);
                     if(m_DumpResultsFile!=null)
                            m_DumpResultsFile.Close();


m_DumpResultsFile=File.CreateText("c:\\Temp\\"+ApplicationComboBox.Text+"_ProcedureInfo.txt");
                     m_VfRemote.StartRun();
              }
```

Once the connection diagram is closed, the test is performed. If “Suppress Excel Output” is unchecked, a MS Excel workbook is opened to show the results of the procedure while it is running.

SetExcelSuppressState(bool suppressEnable)

Via the remote interface the Excel output at run-time can be suppressed by the SetExcelSuppressState(bool) method.

Result Handling

Once a procedure has been run, the results are available in an MS Excel workbook (Figure 9). When the procedure is completed, a dialog pops up, which shows an xml-formatted output.

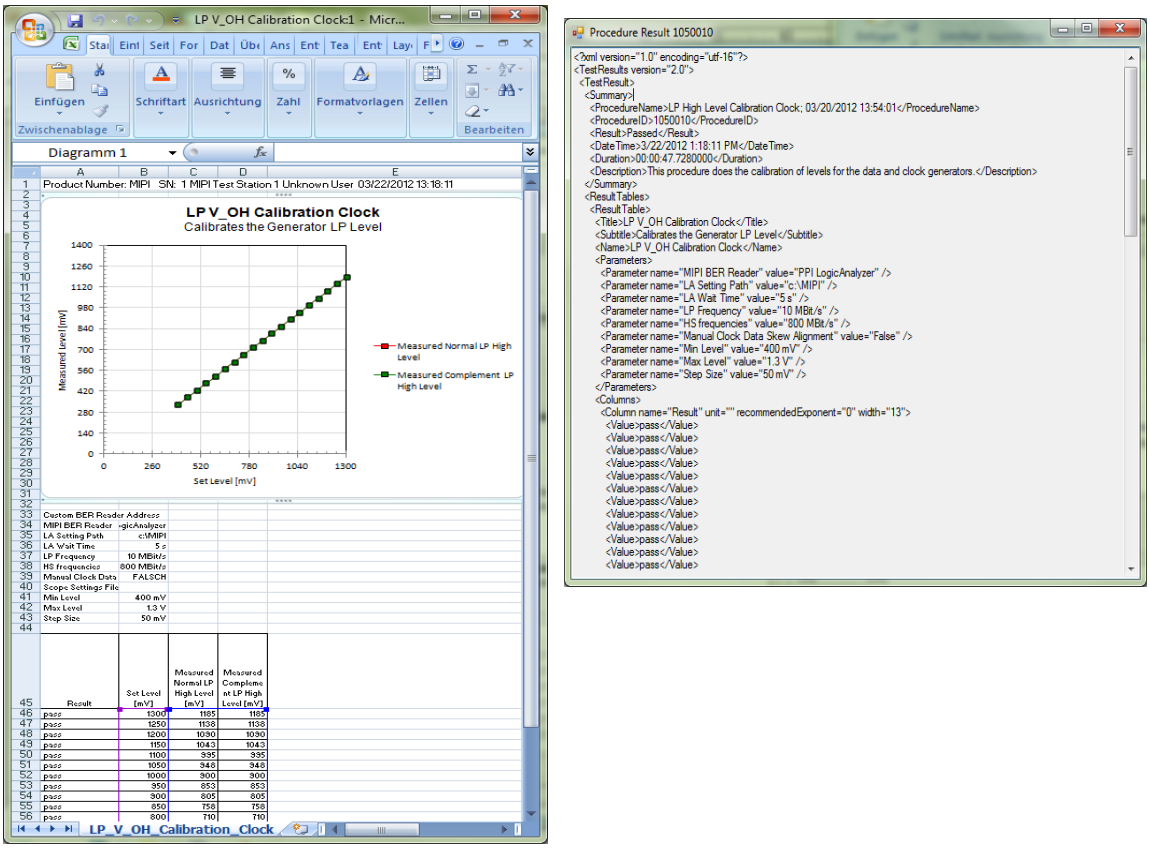

Figure 9: Example MS Excel Result output and the xml-formatted output

event ProcedureCompleted(int procedureID, string xmlResult)

The xml result string is the output parameter of the RunProcedure() and RunProcedures() methods. In the case of the asynchronous StartRun(), the xml result can be accessed via the ProcedureCompleted() method. The following example code accesses the xmlResult string and writes it into a text file:

```
void m_VfRemote_ProcedureCompleted(int procedureId, string xmlResult)
{

        StreamWriter sw = File.CreateText("c:\\Results\result" + procedureId +
".txt");

        sw.Write(xmlResult);
        sw.Close();
}
```

SaveResultsAsWorkbook(string filename)

The test result can be saved in an MS Excel Workbook by SaveResultsAsWorkbook(). If a list of tests is executed, the result sheet will contain one or more sheets per test. In the case of StartRun() and RunProcedures(), the result sheet contains the results of all selected tests.

void SetXmlResultFormat(ResultFormatE format)
string[] Results { get; }

Two choices are available for the result format in the xml result string, "Version 1" and "Version 2". The version can be selected with SetXmlResultFormat(). After the version has been set, the xml result string, which can be obtained by the Results property, which is part of the ValiFrameRemote class, is formatted according to this setting. Version 1 is more compact, but some information, e.g., the result unit and the column width, is missing.

Examples of Version 1 and 2 formats are given below, which contain the same results as given in the Excel output of Figure 10. By comparing the Excel output with the two xml result strings, all values can be identified and the structure can be used to extract the information that is needed.

xml Version 1 result example

```
<?xml version="1.0" encoding="utf-16"?>
<TestResults>
  <Summary>
    <ProcedureName>LP High Level Calibration Clock; 03/22/2012 13:18:59</ProcedureName>
    <ProcedureID>1050010</ProcedureID>
    <Result>Passed</Result>
    <DateTime>3/26/2012 3:07:40 PM</DateTime>
  </Summary>
  <DocumentElement>
    <Parameters>
      <Name>MIPI BER Reader</Name>
      <Value>PPI LogicAnalyzer</Value>
    </Parameters>
    <Parameters>
      <Name>LA Setting Path</Name>
      <Value>c:\MIPI</Value>
    </Parameters>
    <Parameters>
      <Name>LA Wait Time</Name>
      <Value>5 s</Value>
    </Parameters>
    <Parameters>
      <Name>LP Frequency</Name>
      <Value>10 MBit/s</Value>
    </Parameters>
    <Parameters>
      <Name>HS frequencies</Name>
      <Value>800 MBit/s</Value>
    </Parameters>
    <Parameters>
      <Name>Manual Clock Data Skew Alignment</Name>
      <Value>False</Value>
    </Parameters>
    <Parameters>
      <Name>Min Level</Name>
      <Value>400 mV</Value>
    </Parameters>
    <Parameters>
      <Name>Max Level</Name>
```

```
      <Value>1.3 V</Value>
    </Parameters>
    <Parameters>
      <Name>Step Size</Name>
      <Value>50 mV</Value>
    </Parameters>
  </DocumentElement>
  <Data>
    <ColumnHeader>|Result|Set Level|Measured Normal LP High Level|Measured Complement LP High Level|
</ColumnHeader>
    <Values>|pass|1.3|1.185|1.185|</Values>
    <Values>|pass|1.25|1.1375|1.1375|</Values>
    <Values>|pass|1.2|1.09|1.09|</Values>
    <Values>|pass|1.15|1.0425|1.0425|</Values>
    <Values>|pass|1.1|0.995|0.995|</Values>
    <Values>|pass|1.05|0.9475|0.9475|</Values>
    <Values>|pass|1|0.9|0.9|</Values>
    <Values>|pass|0.95|0.8525|0.8525|</Values>
    <Values>|pass|0.9|0.805|0.805|</Values>
    <Values>|pass|0.85|0.7575|0.7575|</Values>
    <Values>|pass|0.8|0.71|0.71|</Values>
    <Values>|pass|0.75|0.6625|0.6625|</Values>
    <Values>|pass|0.7|0.614999999999999|0.614999999999999|</Values>
    <Values>|pass|0.649999999999999|0.567499999999999|0.567499999999999|</Values>
    <Values>|pass|0.599999999999999|0.519999999999999|0.519999999999999|</Values>
    <Values>|pass|0.549999999999999|0.472499999999999|0.472499999999999|</Values>
    <Values>|pass|0.499999999999999|0.424999999999999|0.424999999999999|</Values>
    <Values>|pass|0.449999999999999|0.377499999999999|0.377499999999999|</Values>
    <Values>|pass|0.4|0.33|0.33|</Values>
  </Data>
</TestResults>
```

xml Version 2 result example

```
<?xml version="1.0" encoding="utf-16"?>
<TestResults version="2.0">
  <TestResult>
    <Summary>
      <ProcedureName>LP High Level Calibration Clock; 03/22/2012 13:18:59</ProcedureName>
      <ProcedureID>1050010</ProcedureID>
      <Result>Passed</Result>
      <DateTime>3/26/2012 3:07:40 PM</DateTime>
      <Duration>00:00:02.8945788</Duration>
      <Description>This procedure does the calibration of levels for the data and clock
generators.</Description>
    </Summary>
    <ResultTables>
      <ResultTable>
        <Title>LP V_OH Calibration Clock</Title>
        <Subtitle>Calibrates the Generator LP Level</Subtitle>
        <Name>LP V_OH Calibration Clock</Name>
        <Parameters>
          <Parameter name="MIPI BER Reader" value="PPI LogicAnalyzer" />
          <Parameter name="LA Setting Path" value="c:\MIPI" />
          <Parameter name="LA Wait Time" value="5 s" />
          <Parameter name="LP Frequency" value="10 MBit/s" />
          <Parameter name="HS frequencies" value="800 MBit/s" />
          <Parameter name="Manual Clock Data Skew Alignment" value="False" />
          <Parameter name="Min Level" value="400 mV" />
          <Parameter name="Max Level" value="1.3 V" />
          <Parameter name="Step Size" value="50 mV" />
        </Parameters>
        <Columns>
          <Column name="Result" unit="" recommendedExponent="0" width="13">
            <Value>pass</Value>
            <Value>pass</Value>
            <Value>pass</Value>
            <Value>pass</Value>
            <Value>pass</Value>
            <Value>pass</Value>
            <Value>pass</Value>
```

```
            <Value>pass</Value>
            <Value>pass</Value>
            <Value>pass</Value>
            <Value>pass</Value>
            <Value>pass</Value>
            <Value>pass</Value>
            <Value>pass</Value>
            <Value>pass</Value>
            <Value>pass</Value>
            <Value>pass</Value>
            <Value>pass</Value>
            <Value>pass</Value>
          </Column>
          <Column name="Set Level" unit="V" precision="0" recommendedExponent="-3" width="8">
            <Value>1.3</Value>
            <Value>1.25</Value>
            <Value>1.2</Value>
            <Value>1.15</Value>
            <Value>1.1</Value>
            <Value>1.05</Value>
            <Value>1</Value>
            <Value>0.95</Value>
            <Value>0.9</Value>
            <Value>0.85</Value>
            <Value>0.8</Value>
            <Value>0.75</Value>
            <Value>0.7</Value>
            <Value>0.649999999999999</Value>
            <Value>0.599999999999999</Value>
            <Value>0.549999999999999</Value>
            <Value>0.499999999999999</Value>
            <Value>0.449999999999999</Value>
            <Value>0.4</Value>
          </Column>
          <Column name="Measured Normal LP High Level" unit="V" precision="0"
recommendedExponent="-3" width="8">
            <Value>1.185</Value>
            <Value>1.1375</Value>
            <Value>1.09</Value>
            <Value>1.0425</Value>
            <Value>0.995</Value>
            <Value>0.9475</Value>
            <Value>0.9</Value>
            <Value>0.8525</Value>
            <Value>0.805</Value>
            <Value>0.7575</Value>
            <Value>0.71</Value>
            <Value>0.6625</Value>
            <Value>0.614999999999999</Value>
            <Value>0.567499999999999</Value>
            <Value>0.519999999999999</Value>
            <Value>0.472499999999999</Value>
            <Value>0.424999999999999</Value>
            <Value>0.377499999999999</Value>
            <Value>0.33</Value>
          </Column>
          <Column name="Measured Complement LP High Level" unit="V" precision="0"
recommendedExponent="-3" width="8">
            <Value>1.185</Value>
            <Value>1.1375</Value>
            <Value>1.09</Value>
            <Value>1.0425</Value>
            <Value>0.995</Value>
```

```
            <Value>0.9475</Value>
            <Value>0.9</Value>
            <Value>0.8525</Value>
            <Value>0.805</Value>
            <Value>0.7575</Value>
            <Value>0.71</Value>
            <Value>0.6625</Value>
            <Value>0.614999999999999</Value>
            <Value>0.567499999999999</Value>
            <Value>0.519999999999999</Value>
            <Value>0.472499999999999</Value>
            <Value>0.424999999999999</Value>
            <Value>0.377499999999999</Value>
            <Value>0.33</Value>
          </Column>
        </Columns>
        <Images />
      </ResultTable>
    </ResultTables>
  </TestResult>
</TestResults>
```

The Remote Interface Test GUI does not allow multiple tests to be run at once, so the Excel result workbook always only contains the result of a single procedure. After a test has been run, the Excel result and the xml result output can be viewed with the “Show Result” button, even if another test has already been selected or run.

Remote Interface Reference

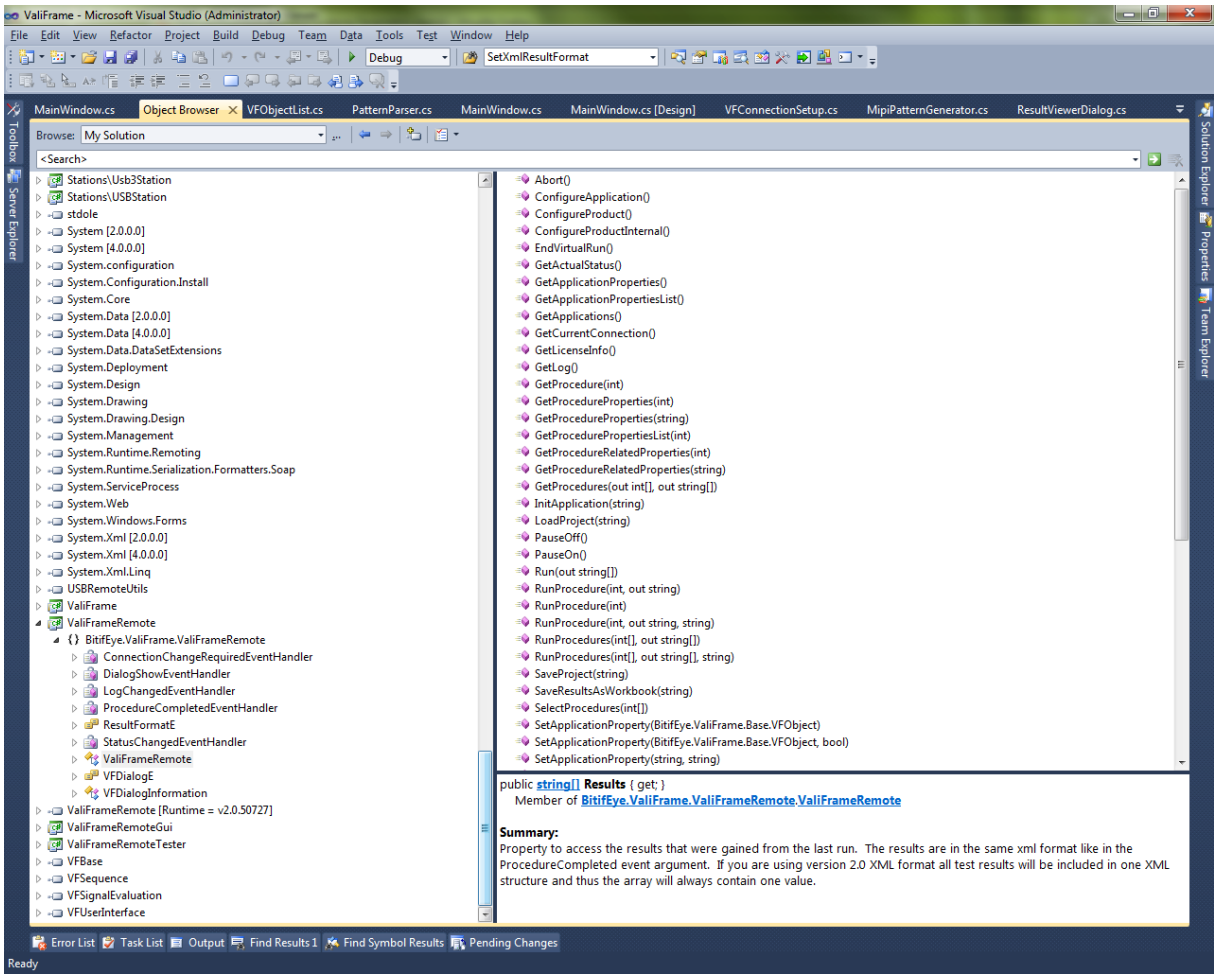

Figure 10: View of the ValiFrameRemote class via the Visual Studio's Object Browser

In the previous chapter the use of the Test GUI and remote interface for a typical application was shown. This chapter contains the list of methods, properties, and events of the ValiFrameRemote class. The easiest way to access the methods, properties, and events is the Visual Studio “auto complete” functionality and the “Object Browser” (Figure 10). To access the ValiFrameRemote class, ValiFrameRemote.dll needs to be added to the project as a reference. The ValiFrameRemote class needs some base classes in some other ValiFrame framework dlls. Therefore the dlls VFBase.dll, VFSequence.dll, and VFUserInterface.dll also need to be added to the references. This can be done by right-clicking on the “References” folder of a project and selecting “Add Reference...”, selecting the “Browse” tab in the “Add Reference” dialog, and choosing the “ValiFrameRemote.dll” and the others from the ValiFrame program files folder in the file browse dialog.

Method, Property, Event	Description
Constructor ValiFrameRemote()	Gets an instance of a ValiFrameRemote class
Initialization and Configuration	
void SetConfigurationFileName(string filename)	Sets the path to the valiFrame xml configuration file. It contains the instrument settings and the default values for all application properties. A default file can be found after using the StationConfigurator and setting the connections of all instruments.
void SetValiFrameDirectory(string path)	Sets the path to the ValiFrame directory, e.g. C:\Program Files\BitifEye\ValiFrame, to allow missing assemblies (dlls) to be loaded from this directory.
bool InitApplication(string application)	Initializes an application, i.e., connects it to the instruments and prepares the settings for testing a DUT with this application. If the initialization was successful, “true” is returned.
bool ConfigureApplication()	Opens the Configure DUT dialog. The call requires that the InitApplication() was already executed, and is needed before a test can be executed. If the configuration was successful, “true” is returned.
bool ConfigureProduct()	Same as ConfigureApplication()
bool ConfigureProductInternal()	Same as ConfigureApplication() except that this function has to be called via the Windows Thread. This is the case if this method is called by a method of a Windows GUI control event handler such as Button.Clicked().
void LoadProject(string filename)	Loads stored project files. The project files can be stored via the ValiFrame User Interface (N5990A option 010).
void SaveProject(string filename)	Saves the current settings. The settings file contains the procedure properties settings as well as the application properties settings.
bool SetApplicationProperty(VFObject property)	Method to set the value of a test property. VFObject contains a name that needs to be equal to an application property. The types of the VFObject and the application property also need to be equal. The type of most properties is VFValue, which contains a double as value. Application properties are higher-level properties that are common to all procedures. In the ValiFrame user interface they can be seen if the user clicks on the root of the test tree.
bool	Same as SetApplicationProperty(VFObject) with the additional

Method, Property, Event	Description
SetApplicationProperty(VFObject property, bool configureApplication)	parameter “configureApplication” to see the changes in the procedure tree (see ProcedureTreeView below). Some of the application properties impact the list of procedures, e.g., the number of channels.
bool SetApplicationProperty(string name, string value)	Same as SetApplcationProperty(VFObject), but instead of a VFObject just the name and the value as a string are needed.
bool SetApplicationProperty(string name, string value, bool configureApplcation)	Same as SetApplicationProperty(VFObject, bool), but instead of a VFObject just the name and the value as a string are needed.
bool SetProcedureProperty(int procedureID, string propertyName, string propertyValue)	Sets the procedure property “propertyName” to the string value “propertyValue”. The property names and IDs can be seen with the Test GUI (see previous chapter). The method returns “true” if the property was found and the value was set successfully.
bool SetProcedureProperty(string procedureName, string propertyName, string propertyValue)	Same as SetProcedureProperty(int, string, string), but instead of the procedure ID the procedure name can be given to select the test.
bool SetProcedureProperty(int procedureID, VFObject property)	Same as SetProcedureProperty(int, string, string), but instead of a name and a value a VFObject instance with the same name and type can be used (see GetProcedureProperties).
bool SetProcedureProperty(string procedureName, VFObject property)	Same as SetProcedureProperty(int, VFObject), but instead of the procedure ID the procedure name can be used.
void SetProcedurePropertiesToDefault(int procedureID)	Sets all properties of a procedure with “procedureID” back to default, i.e., to the state before they were changed with the methods SetProcedureProperty(...).
void SetProcedurePropertiesToDefault(string procedureName)	Same as SetProcedurePropertiesToDefault(int), but instead of the procedure ID the procedure name is given.
void SetExcelSuppressState(bool suppressExcelOutput)	This method allows you to suppress Excel from being opened to show the test results.

Method, Property, Event	Description
void SetXmlResultFormat(ResultFormatEresultFormat)	Sets the format of the xml result string (see description of the formats on page 15).
Execution and Flow Control	
void SelectProcedures(int[] procedureIDs)	Selects the procedures that will be run by the Run() and StartRun() methods
bool Run(out string[])	Executes the selected procedures (see SelectProcedures()) and returns an array of procedure results. The call returns after all procedures were executed, and the return value is true if no procedure returns a setup error.
void StartRun()	Starts an asynchronous execution of the selected procedures (see SelectProcedures()). This call will return immediately, and the execution can be monitored by the events StatusChanged, LogChanged, and ProcedureCompleted.
bool RunProcedure(int procedureID, out string xmlResult)	Executes a single procedure of the ID procedureID. The xmlResult string contains the xml result of this procedure. The method returns after the execution is completed, and the return value is true if the execution was successful.
string RunProcedure(int procedureID)	Same as RunProcedure(int procedureID, out string xmlResult), but having the xml result as return value.
bool RunProcedure(int procedureID, out string xmlResult, string worksheetPath)	Same as RunProcedure(int procedureID, out string xmlResult), but with an additional argument for the Excel workbook output path. In this folder a separate Excel workbook file will be created for each test run.
bool RunProcedures(int[] procedureIDs, out string[] xmlResults)	Same as RunProcedure(int procedureID, out string xmlResult), but running a bunch of procedures by just one call.
bool RunProcedures(int[] procedureIDs, out string[] xmlResults, string worksheetDirectory)	Same as bool RunProcedures(int[], out string[]), but with an additional argument for the Excel workbook output path. At this path each test result is stored in a separate Excel workbook file.
void StartVirtualRun()	If single procedures are run and it is desired to avoid a full initialization being done each time, you can call this method and only at the following first call will a complete initialization be done. This call speeds up the execution. At the end the EndVirtualRun() needs to be called to clean up.

Method, Property, Event	Description
void EndVirtualRun()	Some of the test and calibration procedures require extended configuration steps before they can be executed, and an extended clean-up procedure before they are completed. The extended configuration is required only once if a group of procedures is selected for execution, or the StartVirtualRun() function was executed before. In this case the extended configuration and clean-up will be run only once. This is important for some Tx test applications. By using StartVirtualRun() and EndVirtualRun(), the test time can be reduced tremendously.
void Abort()	During an asynchronous run the execution of the procedure can be aborted. As soon as this function is called, the execution of the procedure is aborted at an appropriate step. This may take some time, because all open instrument calls and subsequences of the procedure need to be finished before it can be aborted.
void PauseOn()	Pauses the execution. The test currently being executed will not proceed until PauseOff() is called, or it will be aborted, if Abort() is called.
void PauseOff()	Exits the pause state that was set by the PauseOn() call.
Result Handling and Other Information	
string[] Results { get; }	Property to access the results that were gained from the last run. The results are in the same xml format as in the ProcedureCompleted event argument. If you are using version 2.0 XML format all test results will be included in one XML structure and thus the array will always contain one value.
void SaveResultsAsWorkbook(string filename)	Saves the test results to a workbook, where each test result is stored in one sheet
string[] GetApplications()	Return strings such as HDMI,DisplayPort,..., will check the license information to see what is licensed. One of these strings will be needed to be used in the InitApplication() method
void GetProcedures(out int[]	Returns the lists of procedure IDs and names. This list is

Method, Property, Event	Description
procedureIDs, out string[] procedureNames)	available after calling the ConfigureApplication() or LoadProject() methods, and the IDs and/or names are needed for accessing the procedure properties and execution.
GetProcedure(int procedureID)	Gets a reference to the test procedure instance with the procedure ID procedureID. The resulting reference is atype of VFProcedure and allows direct access to the VFProcedure members (see VFProcedure members via the Visual Studio Object Browser)
VFObject[] GetProcedureProperties(int procedureID)	Gets a list of VFObjects that contains all references to the properties of the procedure with the procedureID.
VFObject[] GetProcedureProperties(string procedureName)	Same as GetProcedureProperties(int), but instead of the procedure ID the procedure name can be given.
System.Collections.Generic. Dictionary<string, string> GetProcedurePropertiesList(int procedureID)	Same as GetProcedure(int), but instead of a reference to the procedure it will return a dictionary with names and values.
VFObject[] GetProcedureRelatedProperties(int procedureID)	Same as GetProcedureProperties(int), but it contains the application and procedure group properties as well.
VFObject[] GetProcedureRelatedProperties(strin g procedureName)	Same as GetProcedureRelatedProperties(int), but instead of the procedure ID the procedure name can be given.
string GetActualStatus()	Current sequencer status. Will show which procedure is actually running, in which step, etc. The description is the same as in the StatusChanged event argument.
System.Windows.Forms.TreeView ProceduresTreeView { set; get;}	Gets or sets the procedure tree view. It is the same GUI component as is used in the ValiFrame main window to show and control the available procedures. This GUI component can be added to a customized GUI to show and select the test and calibration procedures. The selection in this control will be used in the Run() and StartRun() function for the selected procedure list.
string GetLog()	Returns the complete list of log entries in one string. Each entry is separated by \n.
string GetCurrentConnection()	Gets the current connection information, xml formatted.

Method, Property, Event	Description
Events	
ConnectionChangeRequiredEventHandler ConnectionChangeRequired	This event will be fired if the setup needs a connection change.
DialogShowEventHandler DialogPopUp	This event will be fired if a dialog should appear in ValiFrame. If an event handler is added to this event then the dialog will not appear and the dialog should be handled by the event handler. For UserDecisionForm and UserActionForm you can set the DialogResult to specify the button you want to click.
LogChangedEventHandler LogChanged	This event will be fired if a new entry was added to the log output.
ProcedureCompletedEventHandler ProcedureCompleted	Each completed execution of a test procedure causes this event to be fired. It is to be used if the StartRun() call is used to get the procedure results.
StatusChangedEventHandler StatusChanged	Each change of a status of the ValiFrame sequencer will cause this event to be fired. The argument contains the actual status as a string (see definition of ProcedureCompletedEventHander).

Helper Classes	
delegate **void ConnectionChangeRequiredEventHandler**(**BitifEye.ValiFrame.Base.VFConnectionSetup** *connection*)	Event Handler for the ConnectionChangeRequired event. The VFConnectionSetup contains the information for the setup of the next test procedure.
delegate **void DialogShowEventHandler**(**object** *sender*, **BitifEye.ValiFrame.ValiFrameRemote.VFDialogInformation** *dialogInformation*)	Event Handler for the DialogShow event. The VFDialogInformation contains the dialog information of the dialog that needs to be displayed to the user.
delegate **void LogChangedEventHandler**(**string** *logEntry*)	Event Handler for the LogChanged event. The log entry contains the string of the log entry.
delegate **void ProcedureCompletedEventHandler**(**int** *procedureId*, **string** *xmlResult*)	Event Handler for the ProcedureCompleted event. *procedureId*: ID of the completed procedure *xmlResult*: xml-formatted result string. Contains the information that can also be seen in the Excel result sheet.
delegate **void StatusChangedEventHandler**(**object** *sender*, **string** *description*)	Handler for the StatusChanged event. The description is the description of the actual status.
class VFDialogInformation	This class contains all information necessary to allow the programmer to handle dialogs. It is the argument of the DialogShowEvent. Besides the information about the buttons, the button texts, the dialog text, and image, it contains a reference to the dialog itself. Via this dialog reference it would be possible to show the dialog by the System.Windows.Forms.Form.ShowDialog() method.