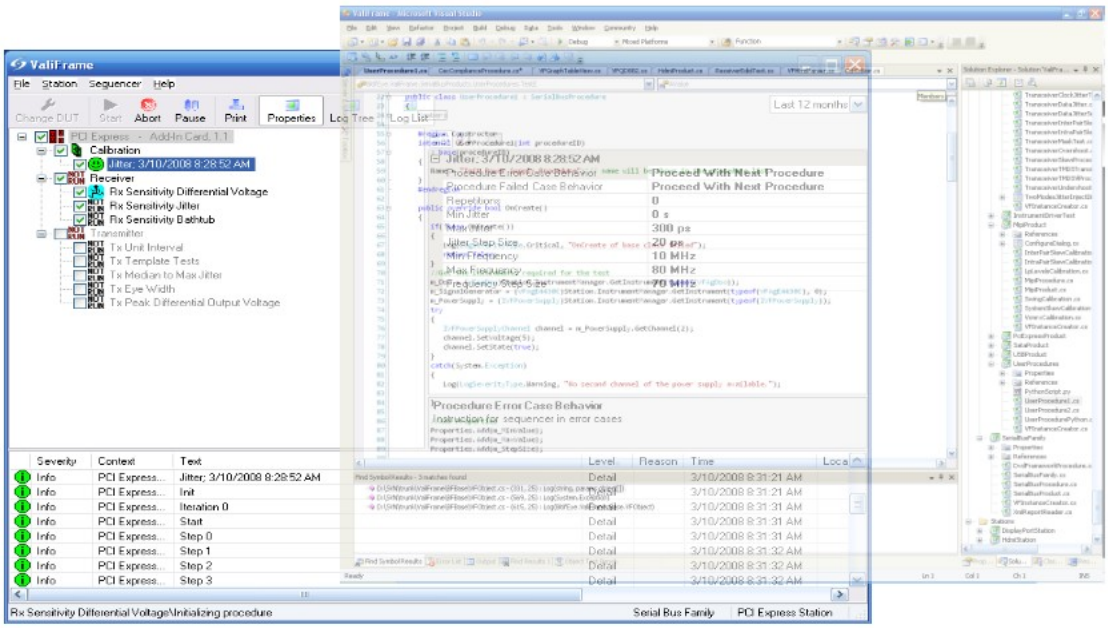
ValiFrame N5990A Test Automation Software
User Programming Option User Guide



Version 1.01 (2011-05-04)

BitifEye Digital Test Solutions GmbH

Table of Contents

Revision History

Version 0.9 (2008-07-21)

- Initial Release

Version 0.91 (2009-01-21)

- Added initialization examples for base types in chapter 2.10

Version 0.91 (2010-07-13)

- Updated code examples for ValiFrame 2.20

Version 1.00 (2010-07-15)

- Adapted to ValiFrame 2.20

Version 1.01 (2011-05-04)

- Added examples for connections to additional instruments

1 Introduction

The ValiFrame test automation software platform is the most powerful tool for serial and multi-lane gigabit testing. It is the unique universal platform for testing a wide range of digital buses such as PCI Express®, SuperSpeed USB or HDMI. ValiFrame can be tailored to your individual test needs with the flexible test sequencer and it controls all instruments needed for your tests. The configurable database interface of the ValiFrame test automation platform enables the convenient storage of all test results. A web interface allows an effective and easy operation.
The ValiFrame software takes test automation to the next level of performance and convenience. It addresses specific bus standards by individual software modules with the same look and feel. All modules share the same user interface which increases efficiency and productivity.

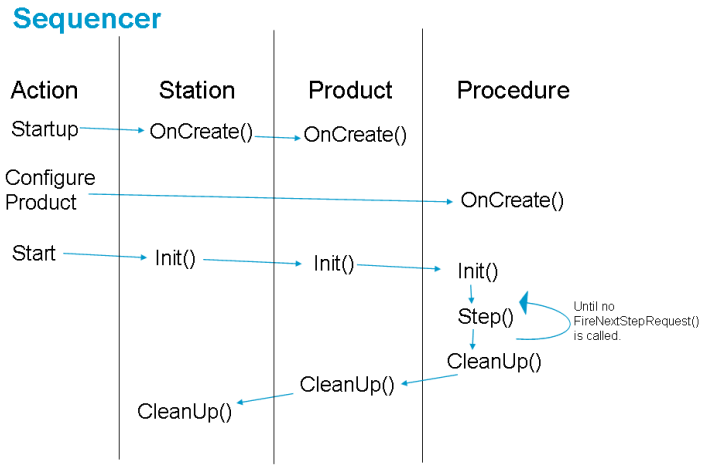
2 ValiFrame Internals and Terminology

2.1 Glossary

Terminology	Description
Procedure	A procedure is a class that defines a test sequence for one specific test.
Product	The product class represents the tested device.
Station	The station contains all required instruments and connections to test a specific standard.
Sequencer	The sequencer is responsible for the test flow, it starts the procedures and handle their return states.
UserInstrument	A ValiFrame compatible instrument driver, created by the user.
UserProcedure	A procedure defined by the user.
ValiFrame	Test automation software framework created by BitifEye
BitifEye	Company name, manufacturer of ValiFrame

2.2 The Sequencer

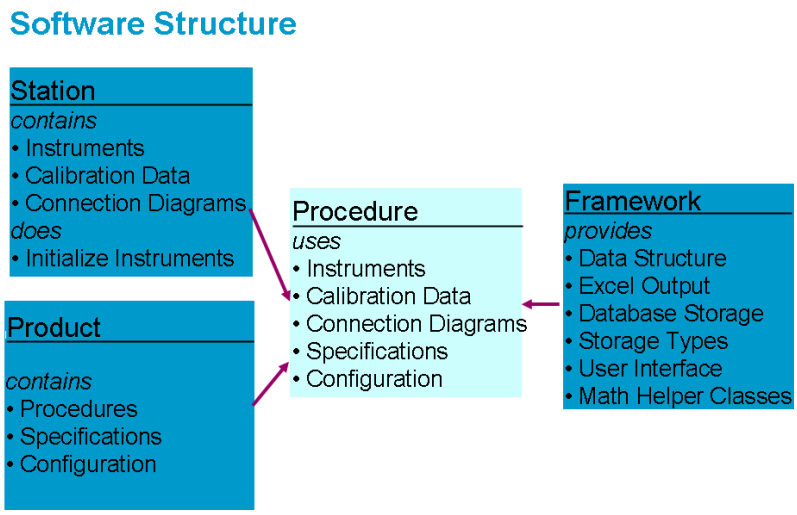
The sequencer is the software part of the ValiFrame framework that is responsible for the test execution flow. The sequencer is user controlled by selecting the test procedures it should execute and by the start, pause and abort buttons.
The sequencer runs through the different components of ValiFrame and through pre-defined methods and executes these statements. The image below shows the order the methods are called by the sequencer.



After the start-up of ValiFrame the sequencer calls the OnCreate methods of the Station as well as of the product.
In the second step, after the product is configured, it calls the OnCreate method of the procedures.

After the user has selected the test procedure that should be executed and hit the start button, the sequencer will first call the Init method of the Product and Station and then start with the first procedure in the list and calling its Init method as well as the Step method and CleanUp method. The Step methods can be called multiple times, to separate the test into several test steps. This is recommended as it has the advantage that the sequencer will check the Abort state after each step, and a status message will be displayed after each step to provide feedback for the user.

2.3 Station



The Station holds the instruments required for the test of a specific test module. The HdmiStation for example holds the list of instruments required to test HDMI. Every Procedure can get the instruments that are required for a test from the Station. The advantage is that you only need to open the connection to this instrument once and not for every test.
The Station also contains the Calibration Data that were created from the calibration procedures and which are required for all or some of the test procedures. Finally the Station holds the different connections. Connections are a text, describing the required connection and an image showing the connection. Every test tells the Station what connection is required for this test, and the station compares this to the last used connection. If they are identical no connection dialog will be shown, but if there is a change required you will see the dialog with the text and image.
The station also contains the initialization of the instruments to prepare them for the use during the tests.

2.4 Product

The product contains the procedures in its procedure set. In addition it saves the specifications for this test module. Finally the product remembers the configuration of the currently tested product. For example, the HDMI Product saves the information if the currently tested device is a Transmitter, Receiver or Cable, and if applicable the supported video modes. Depending on this a different procedure set will be shown and different specifications will be used.

2.5 Procedure

A procedure (=test) is a class containing the logic for one test. The procedures are using the Instruments, the Calibration Data and the connection diagrams from the Station. In addition they use the specifications and configurations from the product and the other functionalities provided by the ValiFrame framework. One example for the framework functionality is the Excel output interface explained in chapter 3.2.
The procedures are derived from the SerialBusProcedure and already contain the required methods for the sequencer.

2.6 Calibration Tables

Calibration Tables or short CalTables are used to save calibration data persistently, and to allow using them again during the next execution of ValiFrame.
To save the calibration data, the ValiFrame framework provides three different classes, VFCalTable0dim, VFCalTable1dim and VFCalTable2dim. Depending on what type of data you want to save, you should select the according cal table.
The 0 dimensional cal table provides storage for a series of points, but the only way you can reference them is the order they are saved in the cal file, there is no custom index available.
The 1 dimensional cal table can hold two series of points where two values (one from each series) build up a pair.
The 2 dimensional cal table saves an array of curves, for example if a calibration calibrates an amplitude dependent on the frequency, you want to save the measured amplitude for a specific set amplitude with a specific frequency.

2.6.1 Using CalTables inside your procedure

In the calibration procedure, first declare a member variable of the cal table type you want to use. In the example we will use a 1-dimensional cal table.

```
protected VFCalTable1dim m_MyCalibrationTable;
```

In the OnCreate method instantiate the cal table and give it a name. After that load the cal table into the station and add it to the list of cal tables.

```
m_MyCalibrationTable = new VFCalTable1dim("MyCalibration");
Station.LoadCalTable(m_MyCalibrationTable, true);
```

During the Step method add entries to the cal table.

```
m_MyCalibrationTable.Append(m_SetValue.Value, measuredValue.Value);
```

Once you have successfully completed the calibration procedure you want to save the cal table. To do this you have to first copy the new values to the initial cal table and then save them. The step with the initial cal table is used to keep the old values for the case that the new calibration fails, and if you want to keep working with the old values.

```
m_MyCalibrationTable.CopyToInitialCalTable();
Station.SaveCalTable(m_MyCalibrationTable);
```

2.7 Logging Functionality

ValiFrame provides a log functionality that shows log information in the user interface as well as in a text file. The log is used to inform the user about the actual task and progress as well as for saving information about errors that may have occurred during the tests. This information is helpful to debug errors.
The Log entries consist of the following parts:

- The Log Message
- The date and time of the entry (generated automatically)

- Entry severity (Info, Warning, Critical, Exception, Fatal)

There are two possibilities to add entries to the log, most of the classes you are using are derived from VFObject and thus provide the method Log(). This method is overloaded and thus is available for a wide variety of parameters.
If you are currently in a class derived from VFObject and want to create a log entry simply call the method and provide the parameters you want to add.
The second possibility is to use the static class VFLog, this class also provides the Log() methods. However, in this case the log entry does not use the right context automatically.
To explain this in more detail, usually the log entries can be separated in groups that were created in an instrument driver context or in the framework.
Below you'll find one example of a typical log entry:

```
Log(VFLogSeverityTypeE.Warning, "The pre test failed!");
```

2.8 Instrument Manager

The instrument manager is used to manage the station instruments. You can access it via your Station. The instrument manager connects to all instruments in the InstrumentsInUse list during startup. In case of connection problems it will provide a message to the user telling that the connection was not successful. The user has two possibilities in this case, either retry the connection attempt or cancel it and therefore closing ValiFrame.
The instrument manager is the right place to get the instruments from. There are multiple possibilities to get the required instruments, these are shown and explained below.
The easiest possibility is to ask for a specific instrument type:

```
Station.InstrumentManager.GetInstrument(typeof(IVFPowerSupply));
```

The example above requests an instrument that implements the interface IVFPowerSupply. You can request an instrument using a specific instrument driver class, too, but the approach with interfaces is more flexible and allows exchanging the power supplies easier.
If you have multiple instruments implementing the same interface and you want to have a specific one you need to specify an index for the instrument you request. The index is used as second parameter.

2.9 InstanceCreator

ValiFrame uses a technology named reflection to dynamically load the UserProcedures.dll and the instrument driver dlls. To use this technology ValiFrame needs a pre-defined interface class and method name to instantiate the user class.
The VFInstanceCreator class is this interface point between the ValiFrame Test Automation software and your own tests or instrument drivers.
ValiFrame tries to execute the CreateInstance or CreateOfflineInstance method of the VFInstanceCreator class during startup. The VFInstanceCreator must be inherited from the VFComponent class. The VFInstanceCreator functions CreateInstance or CreateOfflineInstance should be overwritten and return an object which implements an interface IVFComponentClass. It will be used by ValiFrame to create an instance of your driver, list of procedures or a even a complete station:

```
public interface IVFComponentClass
{
        string Name { get; }
        DateTime GetBuildDateTime();
        VFVersion GetVersion();
        bool OnCreate();
}
```

It is important that you don't change the name of the class and the method. If you change it ValiFrame will not be able to use your code. However, this applies only to the VFInstanceCreator class. For all other methods you are free to use whatever name you want to use!

2.10 VFObject and the other ValiFrame base types

ValiFrame provides a couple of base type in analogy to the base types provided from C# itself. The overall base type is VFObject. This class already provides the possibility to specify a name for the object that can be used to display it to the user. In addition a couple of methods provide the functionality to save and load objects to/from the registry or files. You can specify a description of the object and a visibility that is used in the property grid of ValiFrame, too. Further details about the property grid can be found in chapter 3.3.
The equivalent to the C# type integer is VFInteger. It provides the functionality described above, because it is derived from the VFObject, and in addition it offers the possibility to save a value of the type integer. The same applies for VFBoolean for Boolean values and VFNumber for double values.
A special type that exists in ValiFrame is VFValue. It is derived from VFNumber and thus has the ability to save a double number. However, this class adds a unit type to the number itself and allows to always provide an easy to read representation of the number and unit for the user. For example, if you have specified the unit "Voltage" and the number has the value 0.003, the user will see this as "3mV". The class automatically adds the 'm' prefix. The VFValue type also provides the functionality for the reverse way to parse a string into a VFValue. It takes care that the unit cannot be changed for a specific type, thus if you have for example a VFValue with unit type voltage and the user provides the value 20s - which is of the unit type seconds (time) - it wouldn't allow this value.
In addition you can specify minimum and maximum values. Once you have set them the user cannot specify a number outside of this range. If the user provides it anyway, a message will be shown that the value is not allowed, and the corresponding minimum or maximum value will be used. All these types are derived from VFParameter<base type> and they can be used in the properties grid.

The following examples show how to instantiate the ValiFrame base types:

```
protected VFNumber m_Number = new VFNumber("Number", 2.5);
```

The example shows a VFNumber that contains a double value together with a string for the name.

```
protected VFValue m_StepSize = new VFValue("Step Size", 0.2,
0.0,2.0,UnitType.Voltage);
```

Here a VFValue is specified as a VFNumber plus a unit type. The initalization defines the user readable name, an actual value (0.2), minimum (0.0) and maximum (2.0) values as well as the unit type (Voltage).

For enums a VFEnum<enum> is defined:

```
    public enum TestE
    {
       AValue,
       BValue,
       CValue,
    }
    protected VFEnum<TestE> m_Enum = new VFEnum<TestE>("Enum Value",
TestE.BValue);
```

The last example shows a VFEnum, a base type that is used for a specific enum type. You can specify the enum type in the <>. Once you have done this you can only change the actual value, not the type.

3 Creating a User Procedure

These classes must be derived directly or indirectly from the SerialBusProcedure class. If you want to write your own procedures your classes can have arbitrary names. However, the class must contain several methods with fixed interfaces. The following is an overview list of the required methods.

- `public override bool OnCreate()`
- `protected override bool Init()`
- `protected override bool Step(VFSequenceStepInfo sequenceStepInfo)`
- `protected override bool CleanUp(bool sequenceComplete)`

These methods are required, because the sequencer runs through the list of selected tests and executes these methods in a fixed order. Each of the methods has a different task. Section 2.2 explains this in detail.
Each user procedure has to implement a constructor with at least one parameter. This parameter - the procedure ID (type int) - is used to identify the procedure. This is important as the name of the procedure may change from time to time. For your own user procedures you can use the range 1 800 000-1 999 999.
The constructor passes the procedure ID to its base constructor using code similar to the one below

```
internal UserProcedure2(int procedureID)
        : base(procedureID)
        {
                Name = "My Measurement Procedure";
        }
```

This code specifies a name for the procedure, too.

3.1 Accessing existing Instruments

The Station class is responsible for holding the instrument information, the calibration tables and the connection diagrams. The instrument drivers are instantiated at startup, and those instruments which are set to “online” in the station configurator are connected. The drivers are organized in a list which holds a helper class VFInstrumentManager, which is a member of the Station class. The driver instances can be accessed by the test procedures via the member InstrumentManager. The method GetInstrument()in the VFInstrumentManager is used to access the instrument drivers. There are several overloads available to make this access as comfortable as possible:
`IVFInstrument GetInstrument(Type instrumentType)`: Access the instrument via the instrument type.

Example:

```
IVFPowerSupply powerSupply =
(IVFPowerSupply)Station.InstrumentManager.GetInstrument(typeof(IVFPowerSupply));
```

In this example an instrument which supports the interface IVFPowerSupply is accessed. If a specific model is needed, a specific driver can be used instead of an interface:

```
VFAgE364xA powerSupply2 =
(VFAgE364xA)Station.InstrumentManager.GetInstrument(typeof(VFAgE364xA));
```

However, this is not recommended, because as soon as the the power supply model is changed in the station, the specific driver cannot be found. The recommended method is to use interfaces instead of specific drivers in order to make the test procedure more flexible.

Note: The test procedure should define references to instrument interfaces as members which can be accessed in the Init, Step and Cleanup method. The references can be set by the GetInstrument method above in the OnCreate method.

If two or more instruments of the same type/model are part of the station configuration, a second method can be used which has a second parameter, the instrument number of the same type:

```
IVFPowerSupply powerSupply =
(IVFPowerSupply)Station.InstrumentManager.GetInstrument(typeof(IVFPowerSupply),1);
```

The numbering starts at 0 and the order is equal to the order in the station configurator's instrument list (last panel). Both methods throw an ApplicationException if the instrument is not found.

3.1.1 Using the Instrument Drivers

After accessing the driver via the station's instrument manager (see previous chapter), the instrument can be controlled via this driver. Every driver has several methods to set or measure parameters which are supported via the instrument's remote command interface. The parameters which can be set are accessed with methods named Set<parameter name>(<parameter type> <parameter value>). E.g. in the IVFSignalGenerator interface the method to set the frequency is SetFrequency(double frequency).

Some instrument drivers are organized in channels. This is the case for instruments which have several outputs/inputs with the same functionality, like power supplies. A channel can be gained via the function GetChannel(channelNo). The channel number starts at 1 and is a separate class to access the specific channel's parameters.
Example:

```
IVFPowerSupplyChannel powerSupplyChannel1 = powerSupply.GetChannel(1);
powerSupplyChannel1.SetState(true); //switch on the output
powerSupplyChannel1.SetVoltage(1.0); //set the output voltage to 1.0 V
```

3.2 Initializing the Excel Table

In the Init function you should first set up a table to be used for the test results. This table will be visible in Excel and show you the test results. Typically you want to specify the available columns, the title and subtitle and a Pass/Fail column:

1. Create a new VFStandardGraphTable instance
   ```
   m_GraphTable = new VFGraphTable("Name of the table");
   ```
   The table name must be unique, because this name is used to identify the table.
2. Specify a title, subtitle and caption for the table. This information will be used to create the Excel sheet.
   ```
   m_GraphTable.Caption = "This is the caption";
   m_GraphTable.Title = "This is the title of the Table";
   m_GraphTable.SubTitle = "This is the subtitle";
   ```
   This information will be used to show a sheet similar to the one below.

This is the title of the table				
This is the subtitle				
Result	Channel	Min Passed Diff. Swing [mV]	Min Spec Diff. Swing [mV]	Max Swing Test [V]
pass	All	70	150	1,200

3. The next step is to add columns to the table. For every column you can specify the name, the Unit (Voltage,BER,…) a width, an exponent and the precision. The following code snippet shows you an example that creates a column with the name "Min Passed Diff. Swing", unit is Voltage, exponent is -3 (milliVolt=mV), precision is 0 (0 extra digits after the decimal point) and a width of 8 (used to specify the column width in Excel).

   ```
   m_GraphTable.AddFurtherColumn("Min Passed Diff. Swing", UnitType.Voltage, -3, 0, 8, true);
   ```

4. A special column is the column with the name Result. You can specify such a column and specify whether it should offer to add a value for each row, or only one for the complete test. You can do this with the following command:

   ```
   m_GraphTable.PassFailColumn = PassFailColumnType.OneForAllY;
   ```

5. After this setup you have to call the `InitTableData(m_GraphTable)` method to complete the initialization of the table.

Add Entries during the Procedure Step

To add an entry for a column you must specify the table name and column name. The following example adds a value stored in the variable m_Value to Excel. You need to use a variable of the type VFValue to pass the value to Excel and to specify the unit as well as the value:

```
AddEntry(m_TableName, "Min Passed Diff. Swing", new VFValue(0.07, UnitType.Voltage));
```

To add an entry to the Result column, you can use the following code. True means "pass", false means "fail".

```
AddResult(m_TableName, true);
```

Using a chart in the Excel Test Result

In addition to only set up text based outputs, you can specify additional charts to visualize the test results.

If you want to use a chart in your test result, the setup of the table looks slightly different. You have to specify the chart by setting up the X and Y axis as well as the chart type.

The definition of the axis allows to specify a caption, a start and end value, an exponent and a scale type (linear or logarithmic). The following code defines the two axes starting from 0 to 3 with an exponent of e-3.

```
m_GraphTable.DefineXAxis("X-Axis Caption", new VFValue( 0,UnitType.NoUnit),  new
VFValue(3,UnitType.NoUnit), -3, ScaleType.Linear);
m_GraphTable.DefineY1Axis("Y-Axis Caption", new VFValue(0, UnitType.NoUnit), new VFValue(4,
UnitType.NoUnit), -3, ScaleType.Linear);
```

In addition you have to specify the columns with the values used to create the chart.

The following code defines the columns for the X and Y axis.

```
m_GraphTable.AddXColumn("Set Values", 0, 8);
m_GraphTable.AddY1Plot("Measured", 0, 8, false);
```

To add entries to the columns use the same methods that were described above. The name of the columns is the name you have specified in the `AddXColumn/AddY1Plot`. The AddY1Plot adds a column in the table and a plot in the diagram which is plotted relative to the Y1 (left axis) and the X Axis. If it is needed then AddY2Plot will add a plot to the right Y-Axis which needs to be defined beforehand by `DefineY2Axis, similar to the DefineY1Axis.`

3.3 Properties

ValiFrame provides a technology to add properties to almost all classes (all inherited from `VFPropertyOwner`). Especially the test procedures and the product properties are important as they are visible for the user in the user interface. On the right side of the main window you will find a property grid control. This control shows the properties of the tree item selected on the left side.
To add items to the property list you can use the following code:

```
Properties.Add(VFObject)
```

All items added to the property list must be derived from the VFObject base class. This class implements the required functionality to show the properties in the property grid. The important information held by each VFObject are:

- A string to hold the name of the property
- A description to explain the property
- A visibility enum to define if the property should be visible/editable in the property grid

The property grid provides editors for the following ValiFrame types:

- `VFString`
- `VFNumber`
- `VFEnum<>`
- `VFInteger`
- `VFBoolean`
- `VFRange`
- `VFValue`

Depending on the type you might be able to enter arbitrary text into the property grid control (VFString, VFNumber, ..) or only see a selection of possible alternatives (VFEnum<>, VFBoolean). If arbitrary text can be inserted, ValiFrame will parse this text and show an error message if the text cannot be converted to the target type.

3.4 Sequencing

The sequencer (see 2.2) is the responsible part for the sequencing in ValiFrame. As you can see in the image below there are 3 different levels (station, product and procedure). Every level has its own OnCreate, Init and CleanUp methods. The procedures have the additional Step method.

When you start a test run, first the Init of the station will be called, followed by the Init of the product. Prior each test procedure the Init of this procedure will be called. During the test the Step method will be called multiple times as long as the FireNextStepRequest() is called in the current step. Once a test is finished its CleanUp method is called. Once all (selected) tests are finished the CleanUp of the product and station is called.

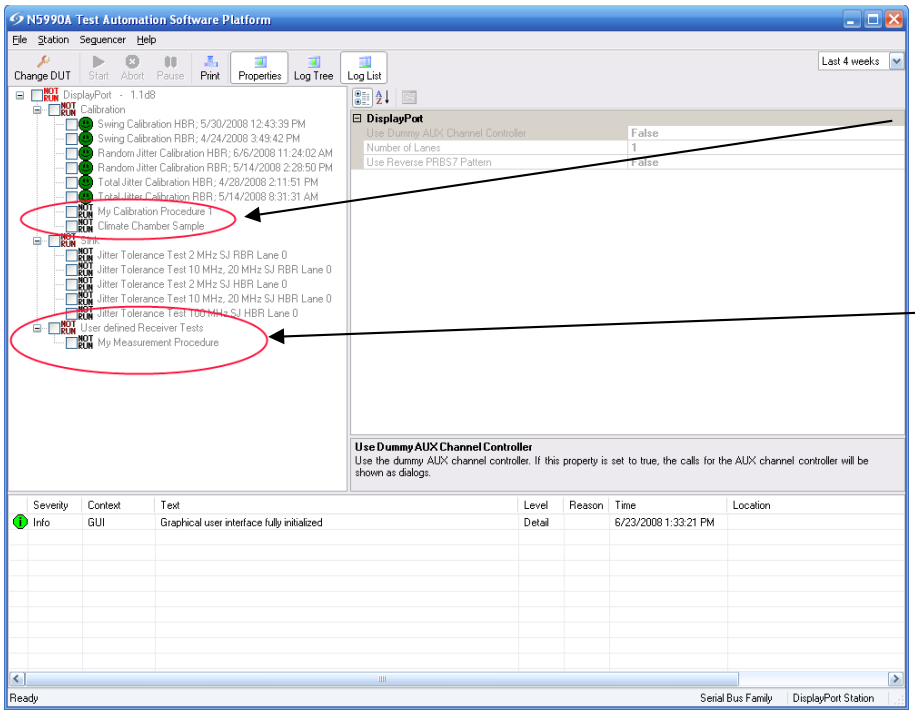
3.5 Create your own procedure sub set

To create your own procedure set you need to add the procedures in the method VFInstanceCreator.ConfigureProduct. An example solution containing such a definition is included in the UserProgramming option. See also chapter 7.
To create a new procedure set you have to call the CreateTree(string tree, int pos) method of the IVFProductConfigurator, which you get from casting the IVFComponentClass from the ConfigureProduct argument to IVFProductConfigurator. The Argument is just a string which defines the tree entries, separated by "->". For example, if you want to create a new subset "My Calibrations" in the Calibration tree, you need to set the string "Calibration->My Calibrations". The tree will be created and the position parameter allows to set the "My Calibrations" subset at a specific position in the list of Calibration procedures. In the following AddProcedure calls you can add your procedures to the new subset. It is not really necessary to call CreateTree, because if AddProcedure refers to a new tree location this tree is automatically created. The advantages of calling the CreateTree() is that the position inside the existing tree can be defined.
Using the call AddProcedure(VFProcedure procedure, string tree) or AddProcedure(VFProcedure procedure, string tree, int pos) you can add one instance of your own procedure to the procedure tree. The tree parameter is equal to the tree parameter in the CreateTree() method. The "branches" are separated by "->" as above. With the second call which contains the 'pos' parameter you can define the position inside the procedure list of this subset. If a pos=0 is set the procedure will be added at the first position. If pos is greater then the number of procedures in this subset, the procedure will be set at the end.

3.6 Creating your own Procedure base class

If you want to create multiple user procedures and/or use the technology described in section 5.1 you should consider creating a base procedure. All your user procedures can be derived from this procedure. The base class can connect to the required instruments, load calibration tables and show the connection diagrams.
If you wan to connect to an instrument you can implement this in the Init method and disconnect the instrument in the CleanUp. The user procedures derived from this class will call the base implementations in their Init and CleanUp and thus the connection will be made.

3.7 Using your own UserProcedure



Adding procedure to existing procedure subsets

Adding procedures in new procedure subsets

4 Creating a User Instrument

ValiFrame provides a couple of instrument drivers for a lot of instruments. However, ValiFrame cannot provide instrument drivers for all available instruments, and you might want to create your own instrument driver for one of your instruments.
This chapter will explain the details you need to know to be able to write a new instrument driver. Each instrument driver consists of one .Net dll. This dll contains the instrument driver class as well as the appropriate VFInstanceCreator class. More details about the VFInstanceCreator class can be found in section 2.9.
Typically the instance creator has two different methods that should be implemented:

```
public override bool CreateInstance(out IVFComponentClass instance);
public override bool CreateOfflineInstance(out IVFComponentClass instance)
```

These methods create an instance of the instrument driver class, call the OnCreate method of the driver and add their version to the VFVersions class.

4.1 The different base classes

ValiFrame provides a couple of base classes that help you writing an instrument driver. Depending on what type of remote connection your instrument provides you need to use a different base class for your implementation.

4.1.1 The VFBaseInstrument base class

The VFBaseInstrument is the base class for all instrument drivers. It provides the required virtual methods that have to be implemented in the derived classes. In addition to that the class provides functionality to load and save the instrument details in the registry.
If you want to implement your instrument driver using an existing plug&play driver you should use this class for your implementation.

4.1.2 The VFVisaInstrument base class

The VFVisaInstrument class is the most often used base class and is derived from the VFBaseInstrument. This class uses the Agilent Visa32.dll library to communicate with the instrument. Every instrument that provides such a remote interface with a SCPI command set can be controlled with this class. Several of the SCPI common commands like *IDN? and *RST are already implemented in this class.
For all other commands you can use the default methods to send commands and to query values. The SendCommand method is used to post commands and the methods QueryBool, QueryDouble, QueryInt, QueryString and a couple more are used to query values from the instrument. The parsing of the return values as well as the error handling and the connection handling are implemented in these methods to reduce the required implementation to a minimum. If the instrument is not connected, or if it is in offline mode, these methods will return their default values.
For multi-channel instruments the ValiFrame framework provides a companion for this class, the VFVisaInstrumentChannel class. Multi-channel instruments are typically implemented in a way that the instrument driver class holds an array of instrument channels for the instrument. Each instrument channel should be derived from the VFVisaInstrumentChannel class. This provides access to the parent instruments as well as access to the existing connection to the instrument.

4.1.3 The VFVisaLanInstrument base class

The VFVisaLanInstrument class provides access to instruments using a socket connection, it provides the same basic methods for posting and querying commands as the VFVisaInstrument. The class can also connect to instruments using a Visa connection (same as the VFVisaInstrument) This is implemented in this class, because some instrument families provide different remote interfaces depending on the age of the specific model, and you may want to only implement the instrument driver a single time for both Visa and socket connection.

4.2 Error Handling

Instrument drivers should implement error handling. For instrument drivers based on the VFVisa- or the VFVisaLanInstrument the error handling is already included in the query and post methods. After each call the ErrorQueryString is used to check for errors.
If you implement an instrument driver based on the VFBaseInstrument you have to care for the error handling by your own!

4.3 Implementing an instrument driver

Depending on what type of remote connection your instrument supports you should use the appropriate base class described in section 4.1. If you want to extend an existing ValiFrame driver you should derive it from this class.
The driver class should be named with the following naming convention: VF<Vendor short form (Ag for Agilent)><instrument name>. For example the Agilent signal generator E4438C would be named VFAgE4438C.dll.
In general you should also try to use interfaces for your Instruments which make it easier to exchange instruments later. For example, an instrument that can set a specific frequency with a specific amplitude should implement the IVFStimulus interface. With this approach you can use the interfaces in your procedures and exchange the instruments easily.
This and a couple of additional hints are listed below in the list of guidelines for a good instrument driver.

- Try to use existing interfaces for your implementation. If there is no interface available for the instrument group then try to define an interface first and then implement this interface in your driver. This will make it easier to exchange instruments if necessary.
- Try to avoid using properties in your class which communicate with the instrument. The Visual Studio debugger tries to get the value of the property each time you hover with the mouse over the property and therefore tries to internally execute the GetProperty function. It very often leads to an unpredictable behaviour of the debugger.
- Assign useful names for the methods to be used by the intellisense function of Visual Studio. Most ideally you would also add a /// comment above each method. This comment will be shown in the intellisense, too.
- Try to avoid “monster”-functions which do a lot of things at once. They are hard to understand and to debug ,and they may not be suitable for other methods. Try splitting it up into several methods and use them combined in one method.
- Implement error handling within the driver and check if the command was successful. Details about error handling are described in section 4.2

5 Integrating a UserInstrument

When writing your own test procedures, you might want to write your own instrument drivers as well. There are two scenarios for this, either one of the ValiFrame drivers does not provide all required functionality and you want to extend it with additional functionality, or you want to add a new instrument.
Independent of which of the two scenarios applies, there are two possibilities to integrate new instruments drivers into ValiFrame and to make them available in your test procedures. The following sections will describe these two integration possibilities as well as their advantages. For the first steps and the testing of your instrument driver it is recommended to use the method described in section 5.1.
To use your instrument driver dll in ValiFrame you need to copy the dll into the ValiFrame directory. To always use the latest dll, you can modify the output path of your project to copy the results into the ValiFrame directory automatically.

5.1 Integrate the UserInstrument in your Procedure

This method is the easier method to use your own instrument drivers in your test procedures. To use a new instrument during your tests, you have to follow some simple steps.

1. Add a reference to the instrument driver in your UserProcedures project.
2. Create a member variable of the instrument driver type.
   ```
   protected VFTmT2800 m_ClimateChamber = new VFTmT2800();
   ```
3. Connect to the instrument in the Init method of the test procedure, using code similar to the code below. This will connect to a climate chamber using the GPIB VISA address GPIB0::10::INSTR, set the timeout to 30 seconds and identify the instrument. It will perform a reset after the connection was established, too.
   ```
   m_ClimateChamber.Connect("GPIB0::10::INSTR", new TimeSpan(0, 0, 30), true, true);
   ```
4. Now you can perform the instrument calls.
5. In the CleanUp method you need to close the connection to the instrument by calling the Disconnect method. Closing the connection is important because some instruments only allow a limited number of connections.

5.2 Integrate the UserInstrument as a Station Instrument

Using this method, the ValiFrame Instrument Manager will take care of establishing a connection to the instrument. During the startup process of ValiFrame the instrument manager will try to connect to all instruments that are required for the actual station. If the connection is not successful an error message will show up and ask you to check the setup as well as allow you to retry the connection attempt.
ValiFrame uses the Registry as a persistent storage for its default settings. The path is HKEY_LOCAL_MACHINE\SOFTWARE\Bitifeye\ValiFrame. There is a sub-key named Stations and for each Station there is a sub-key named equal to the station name ValiFrame\Station\HDMI Station. This contains the Instrument key with the list of instrument settings. If an additional instrument should be handled by ValiFrame (connect/disconnect) then it needs to be added to the instrument list in the registry. In this case ValiFrame will connect to this instrument at startup and the user procedures can access them via the Instrument Manager (see above).

In future the storage of the setting will be moved into a xml file (especially for Windows 7 and its access restrictions), but this will be transparent for all classes and the structure will be the same as in the registry.

6 Modify existing procedures

There are scenarios where you might want to modify existing procedures provided by ValiFrame and to add your own logic to it. To do this you can create a new class that is derived from the one you want to modify. Once you have done this initial step you can override the existing OnCreate, Init, Step and CleanUp methods. Inside these methods you need to call the base implementation of the test procedure and add your own code as well. For example, you might want to use a climate chamber to run the test with a specific environment temperature and humidity. To do this you have to override the Init method and connect to your climate chamber in this method. In addition you have to set the desired temperature and wait until the temperature was set. Once this was achieved you can call the base implementation of the Init method. The CleanUp method should be overridden as well to disconnect from the climate chamber.
Another possibility would be to extend the Step method itself and to run the Step method more often than in the original test in order to run it with different climate profiles. These settings can also be added to the Excel output.
The temperature or temperature range can also be made accessible by adding it to the properties of the procedure. A good way to call base implementations is shown in the example below:

```
if(!base.Init())
{
	Log(LogSeverityType.Critical, "Init of base class failed");
	return false;
}
```

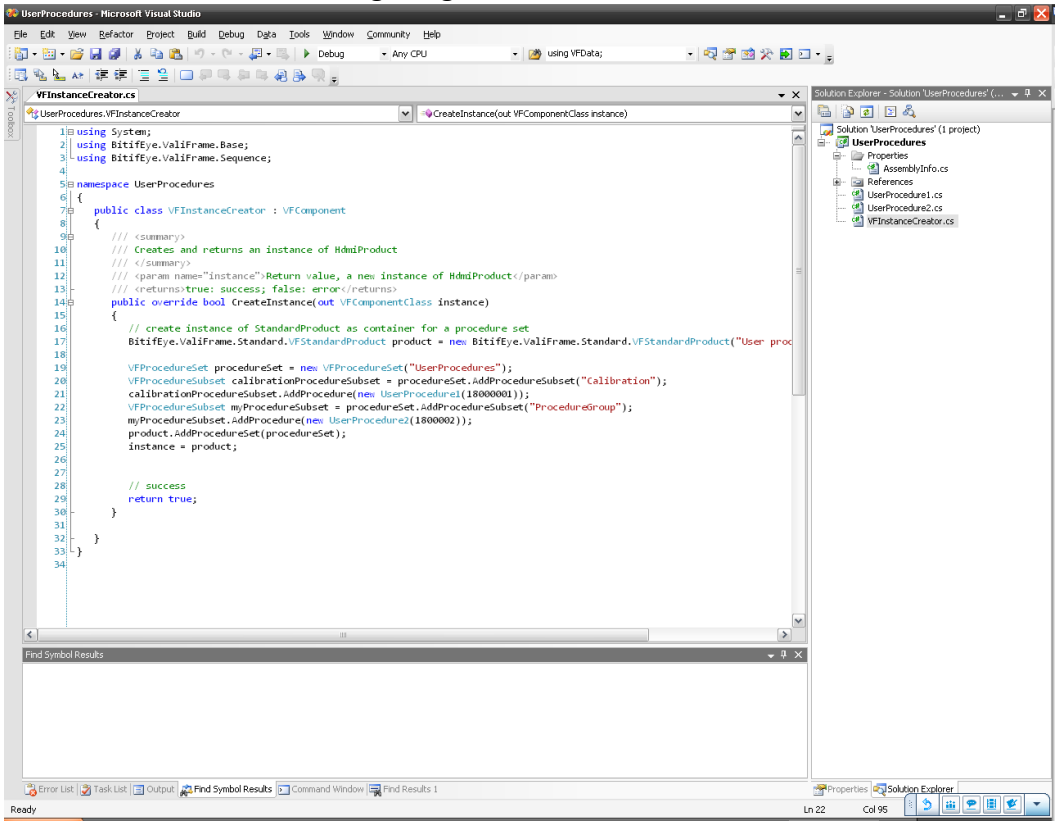
The base implementation is called in the if condition and if it returns false, indicating that it was not successful, a log entry is created telling exactly this problem and returning a false as well.

7 Using the example solution

The ValiFrame UserProgramming option is delivered together with an example solution file. This chapter will assist you with using the example solution in order to create your own procedures. A couple of things already mentioned in the chapters above are explained again in a short manner to allow working with the example solutions without having read the entire document.
You can use this solution to write your own procedures.

1. Start Microsoft Visual Studio by either double-clicking on the UserProcedures.sln file or start Visual Studio and then select File >Open Project/Solution.

You should see the following image:

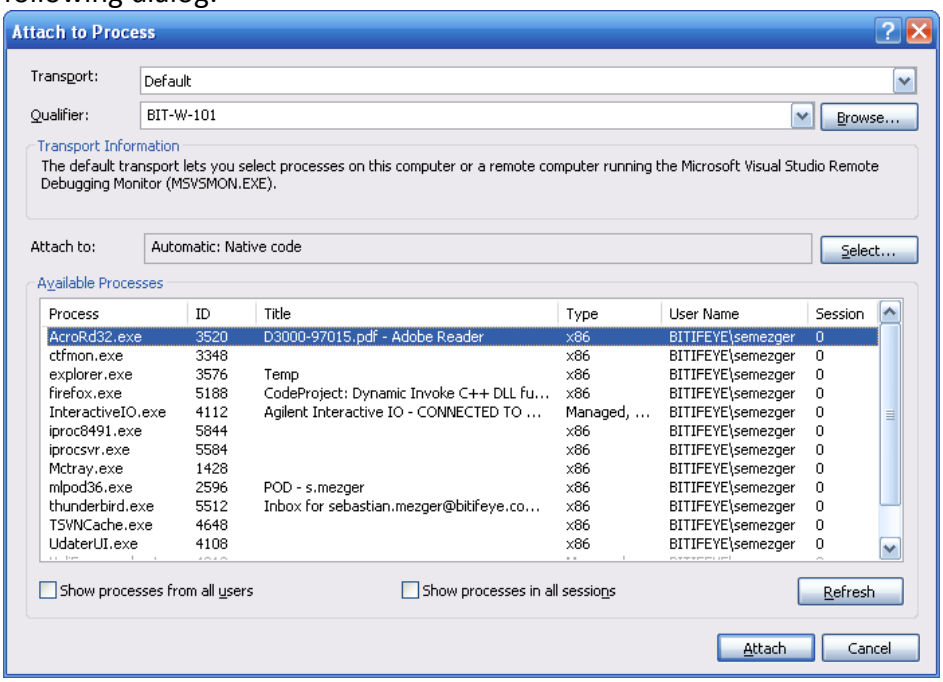


This project will create a file named UserProcedures.dll when you compile it.

2. The UserProcedures.dll must be placed in the ValiFrame program directory. During each start up ValiFrame checks if the User Programming option is installed and if a UserProcedures.dll file is in its directory.
   ValiFrame uses a .NET technology called reflection to load this assembly and tries to find a class named VFInstanceCreator in this dll. This class must have either the method
   `CreateInstance(out IVFComponentClass instance)`
   or
   `ConfigureProduct(IVFComponentClass productConfigurator)`
   implemented.

These methods will be called from ValiFrame after the Configure DUT dialog is closed. For this reason these two names must not be changed. For the other classes you are free to define your own names. The VFInstanceCreator must be inherited from VFComponent. The CreateInstance should provide a class which supports the IVFComponent interface.
This class is the right place to create instances of your own procedure classes and to add them to the procedure set of ValiFrame. The procedure set is the set of tests ordered in categories, that are available in ValiFrame.

3. The UserProgramming solution includes 2 classes named UserProcedure1.cs and UserProcedure2.cs. These two classes are examples of user defined procedures. In these classes you can implement your test logic and execute the tests in ValiFrame.
4. Once you have finished your work you can compile the solution by selecting the Build > Build Solution menu item. Copy the result (UserProcedures.dll) to your ValiFrame directory, either manually or automatically by Visual Studio by setting the project output directory to the ValiFrame program files directory. Now you can start ValiFrame. After you have configured your product you can see your procedures in the procedure tree.
5. To debug your code you can also attach the Visual Studio debugger to the ValiFrame process or you can select ValiFrame.exe as executable. For the first approach you need to select the Debug > Attach to process… menu item. You should see the following dialog:



Select the ValiFrame process from the list and click the attach button.

Appendix A. Migration from ValiFrame 2.00 to 2.20

With ValiFrame 2.20 the remote interfacing was slightly changed. The following modifications need to be done to the VFInstanceCreator class of a UserProcedure.dll code that was written for ValiFrame 2.00.

- In ValiFrame 2.20 the Instance Creator returns a class that needs to support the `IVFComponentClass` interface instead of a class of type `VFComponentClass`. In this case the CreateInstance functions of Products, User Procedures, Instrument Drivers, Stations etc. need to be modifed from

  ```
  CreateInstance(out VFComponentClass instance);
  ```

  to

  ```
  CreateInstance(out IVFComponentClass instance);
  ```
- All the other code can stay the same. It is more powerful to use the ConfigureProduct method to add your procedures, because here you can get the Product (DUT) properties and define which procedures are added and at which place. The IVFProductConfigurator will get more functionality in the future to allow more flexibility.
- The class VFProcedureSet was combined with VFProcedureSubset and therefore no VFProcedureSet class is needed and all entries VFProcedureSet need to be renamed to VFProcedureSubset.
- .vset Files are not used anymore for setting up the instruments:
  The vset files are not used anymore. Instead of this the registry is used directly. The entries can be found at „HKEY_LOCAL_MACHINE\SOFTWARE\BitifEye\ValiFrame\Stations\<Application name> Station“. The instrument list can be found in the sub key Intruments. If instruments need to be added a text file can be created by using the export function of regedit.exe. In this text file the list of instruments can be found and a new instrument can be added to copy one instrument node, rename the name (i.e. Instrument5->Instrument6), renaming the dll name and Address and then importing the modified text file back by regedit.exe into the registry.

  In the future a new settings file will be used which just contains the same entries as in the registry (everything below the ValiFrame key). After this change the new file (which is also a text file) needs to be modified if new instruments needs to be added. After this move from the registry back to a text file, ValiFrame will not be needed to be executed in Administrator mode.