GraphWorX64 Scripting – FrameWorX Client Wrapper Introduction

APPLICATION NOTE

July 2011

Description: This Application Note will introduce FrameWorX Client Wrapper, which allows interacting with data sources and more.
OS Requirement: Windows Server 2003 x64/Vista x64/ Server 2008 x64/Windows 7 x64/ Server 2008 R2 x64
General Requirement: GENESIS64 installed; basic knowledge of GraphWorX64, scripting in JScript.NET and .NET programming.

Introduction

Starting from V10.60, GENESIS64 introduces the FrameWorX Client Wrapper APIs (also known as FwxClientWrapper). You can use these APIs from GraphWorX64 Scripting to access a certain number of features:

- Read/write any data source (both synchronously and asynchronously);
- Login/logout the current user, in this way you can create your own security dialog;
- Log an event.

NOTE: In this document we will assume you have a basic understanding of GraphWorX64 Scripting. If you need any help with that, please refer to the GraphWorX64 Scripting – Quick Start Application Note.

The way you access these features is through the `FwxClientWrapper` class, an instance of which is readily available for use from `ThisWindow` object.

NOTE: All the code samples you'll find in each section are also summarized at the end of this Application Note for your convenience.

Reading and writing data source values synchronously using FwxClientWrapper

FrameWorX Client Wrapper allows us to read and write data source values both synchronously and asynchronously. The difference between the two is that a synchronous operation blocks the current thread until when the execution is completed, while an asynchronous operation lets the thread proceed with its execution, and will invoke a callback when the operation is completed.

In this first section we will focus on the synchronous way to read and write values to data sources. The two methods of the `FwxClientWrapper` class we will use for that are: `Read` and `Write`.

`Read` is used to synchronously get the value of a data source. It gets only one parameter as input, which is the data source, passed as a string. The returned value is of type `SampleValue`, which is an object exposing the following properties:

- `Status`: a `Status` object which reports the quality of the returned sample. The `Code` property of this object contains the quality value, or alternatively you can use `IsBad`, `IsGood`, `IsUncertain` methods to verify the sample quality.
- `Timestamp`: a `DateTime` object reporting the sample timestamp.
- `Value`: the actual sample value, its type is `object`.

Here is an example of a `Read` method call:

```
var dblVal : SampleValue =
ThisWindow.FwxClientWrapper.Read("@sim64:Double.S
tatic(\"Static1\").Value");
```

Because this is a synchronous read, this call will block the script execution until when a value is returned. Once that happens, `dblVal` will contain your sample data. If you're just interested in knowing the value, you will need to access the `Value` property using this syntax: `dblVal.Value`.

The `Write` method takes two input parameters and returns none. The first parameter, like in the `Read` method, is a string representation of the data source we want to write a value to. The second parameter is of type `object`.

Suppose we want to write the current time seconds to our data source, this is what we would use:

```
ThisWindow.FwxClientWrapper.Write("@sim64:Double.
Static(\"Static1\").Value", DateTime.Now.Second);
```

This call is blocking until the write operation completed. Please note that when passing strings to these methods we need to make sure we use the correct escape sequences.

Reading and writing data source values asynchronously using FwxClientWrapper

In the previous section we have seen that the read and write operation were blocking the execution of our thread. This is not always desirable, especially when the execution may take some time. In that case it is preferable to use an asynchronous method which does not wait for the operation to complete, so in the meanwhile we can perform some other tasks, eventually informing the user that his request is being processed.

Once the async operation is completed a "callback" method will be invoked. We can implement our custom logic into that method.

NOTE: Please refer to .NET documentation if you need additional information about event-based programming.

The asynchronous methods we will use are ReadAsync and WriteAsync. The calls are similar to sync calls, but we have to provide an extra parameter, which is the callback method to be called once the operation is completed. There is a third parameter (can be `null`) which is an `object`. You can pass anything you like and this will be passed along to the callback. For example it may represent a state information.

This is the async call, please note that we receive no output from it:

```
ThisWindow.FwxClientWrapper.ReadAsync("@sim64:Dou
ble.Static(\"Static1\").Value",
ThisDocument.CreateDelegate(ReadDoneDelegate,
this, "AsyncReadComplete"), null);
```

The `CreateDelegate` returns a delegate for the `AsyncReadComplete` method which is our callback. The read value will be processed by the callback, invoked automatically when the read operation is completed. This is how our callback looks like:

```
function AsyncReadComplete(result :
ReadDoneResult)
{
	var dblVal : SampleValue = result.Value;
	// Use dblVal here.
}
```

`ReadDoneResult` is an object containing two properties: `UserState` and `Value`. The first is the `object` you passed to `ReadAsync`, while `Value` is the actual value retrieved from the tag. As you can see from the code above, `Value` is not the raw value (e.g.: a number), it is a `SampleValue` object instead. Such object includes properties to get Timestamp, Value and Quality as mentioned in the previous section about sync read.

`WriteAsync` works in a very similar way. The only differences are related to the type of objects involved and the fact that callback will receive information about status of the operation (quality) rather than a `SampleValue` object.

NOTE: For your reference, sample code for WriteAsync is included in the Complete Code section below.

Log in and out using FwxClientWrapper

ICONICS provides a standard Login/Logout dialog which fits most of customers' needs. Some customers though, would like to customize that dialog to make it more consistent with the rest of their project graphics, or maybe to add some custom logic behind the authentication process.

FrameWorX Client Wrapper provides the script methods to easily log in and out from the system. You can then use regular GraphWorX64 graphic or JScript.NET forms to create your custom security dialog.

The method to log in takes username and password as parameters:

```
ThisWindow.FwxClientWrapper.Login("user",
"password");
```

The method to log out has 3 different overloads: the first takes no parameter and logs out the top-most user (i.e. the last one who logged in):

```
ThisWindow.FwxClientWrapper.Logout();
```

The second overload takes the user name as parameter, and will try to log that specific user out, even if he's not the top-most user.

```
ThisWindow.FwxClientWrapper.Logout("user");
```

The last overload is similar to the second one but also takes the password, if the security settings require password to be entered on log out as well.

```
ThisWindow.FwxClientWrapper.Logout("user",
"password");
```

All of the above methods return a `Status` object which contains information about the executed command.

Log events using FwxClientWrapper

GENESIS64 includes GenEvent event server which automatically send system-related events (e.g.: user login, set point writing, …). In some cases it is useful to be able to generate custom events based on your own logic. This is possible using two FwxClientWrapper methods: `LogSimpleEvent` and `LogTrackingEvent`.

A simple event is just a message you want to post to the event server. A tracking event is used to keep track of value changes and allows you to specify a data source identifier (a string) and a pair of old/new values.

This is an example of how you call `LogSimpleEvent`.

```
ThisWindow.FwxClientWrapper.LogSimpleEvent(EventS
everity.Error, "message", "comment",
"sourceName");
```

And this is how you may call `LogTrackingEvent`.

```
ThisWindow.FwxClientWrapper.LogTrackingEvent(EventSeverity.Error, "message", "comment", "dataTag", "oldValue", "newValue", "sourceName");
```

Of course you will replace these sample string values with your own values. Once this code is executed, these events will be fired by GenEvent. If you have an AlarmWorX64 Viewer control subscribed to GenEvent Server, you'll see these events coming in:

Time / Date	Tag	Priority	Type	Quality	Description
6/15/201...	sourceName	750		Good	message
6/15/201...	sourceName	750		Good	message

Figure 1 – Custom Events shown in AlarmWorX64 Viewer

Conclusion

FrameWorX Client Wrapper is a convenient way to access GENESIS64 features from your custom code.

All of the above actions can also be performed without the need to write code: you can read/write values using Process Points, you can log in/out using the Login Dialog which can be shown using a Pick Action, and the system automatically post the most important events through GenEvent. However, if the default way of performing these operations does not fit your needs exactly, you can take full control and write your own logic through FwxClientWrapper methods.

Complete Code

Here is the full code we created in this Application Note.

```
function SyncRead_CommandExecuted(sender : System.Object, cmdArgs : Ico.Gwx.CommandExecutionEventArgs)
{
    var dblVal : SampleValue =
ThisWindow.FwxClientWrapper.Read("@sim64:Double.Static(\"Static1\").Value");
    // Use dblVal here.
}

function AsyncRead_CommandExecuted(sender : System.Object, cmdArgs :
Ico.Gwx.CommandExecutionEventArgs)
{
    ThisWindow.FwxClientWrapper.ReadAsync("@sim64:Double.Static(\"Static1\").Value",
ThisDocument.CreateDelegate(ReadDoneDelegate, this, "AsyncReadComplete"), null);
}

function AsyncReadComplete(result : ReadDoneResult)
{
    var dblVal : SampleValue = result.Value;
    // Use dblVal here.
}

function SyncWrite_CommandExecuted(sender : System.Object, cmdArgs :
Ico.Gwx.CommandExecutionEventArgs)
{
    ThisWindow.FwxClientWrapper.Write("@sim64:Double.Static(\"Static1\").Value", DateTime.Now.Second);
}

function AsyncWrite_CommandExecuted(sender : System.Object, cmdArgs :
Ico.Gwx.CommandExecutionEventArgs)
{
    ThisWindow.FwxClientWrapper.WriteAsync("@sim64:Double.Static(\"Static1\").Value",
DateTime.Now.Second, ThisDocument.CreateDelegate(WriteDoneDelegate, this, "AsyncWriteComplete"),
null);
}

function AsyncWriteComplete(result : WriteDoneResult)
{
    MessageBox.Show(result.Status.Code.ToString())
}

function Login_CommandExecuted(sender : System.Object, cmdArgs : Ico.Gwx.CommandExecutionEventArgs)
{
    ThisWindow.FwxClientWrapper.Login("user", "password");
}

function Logout_CommandExecuted(sender : System.Object, cmdArgs : Ico.Gwx.CommandExecutionEventArgs)
{
    ThisWindow.FwxClientWrapper.Logout();
}
```

```
function LogoutUser_CommandExecuted(sender : System.Object, cmdArgs :
Ico.Gwx.CommandExecutionEventArgs)
{
    ThisWindow.FwxClientWrapper.Logout("user");
}

function LogoutUserPassword_CommandExecuted(sender : System.Object, cmdArgs :
Ico.Gwx.CommandExecutionEventArgs)
{
    ThisWindow.FwxClientWrapper.Logout("user", "password");
}

function LogEvent_CommandExecuted(sender : System.Object, cmdArgs : Ico.Gwx.CommandExecutionEventArgs)
{
    ThisWindow.FwxClientWrapper.LogSimpleEvent(EventSeverity.Error, "message", "comment",
"sourceName");
}

function LogTrackingEvent_CommandExecuted(sender : System.Object, cmdArgs :
Ico.Gwx.CommandExecutionEventArgs)
{
    ThisWindow.FwxClientWrapper.LogTrackingEvent(EventSeverity.Error, "message", "comment", "dataTag",
"oldValue", "newValue", "sourceName");
}
```