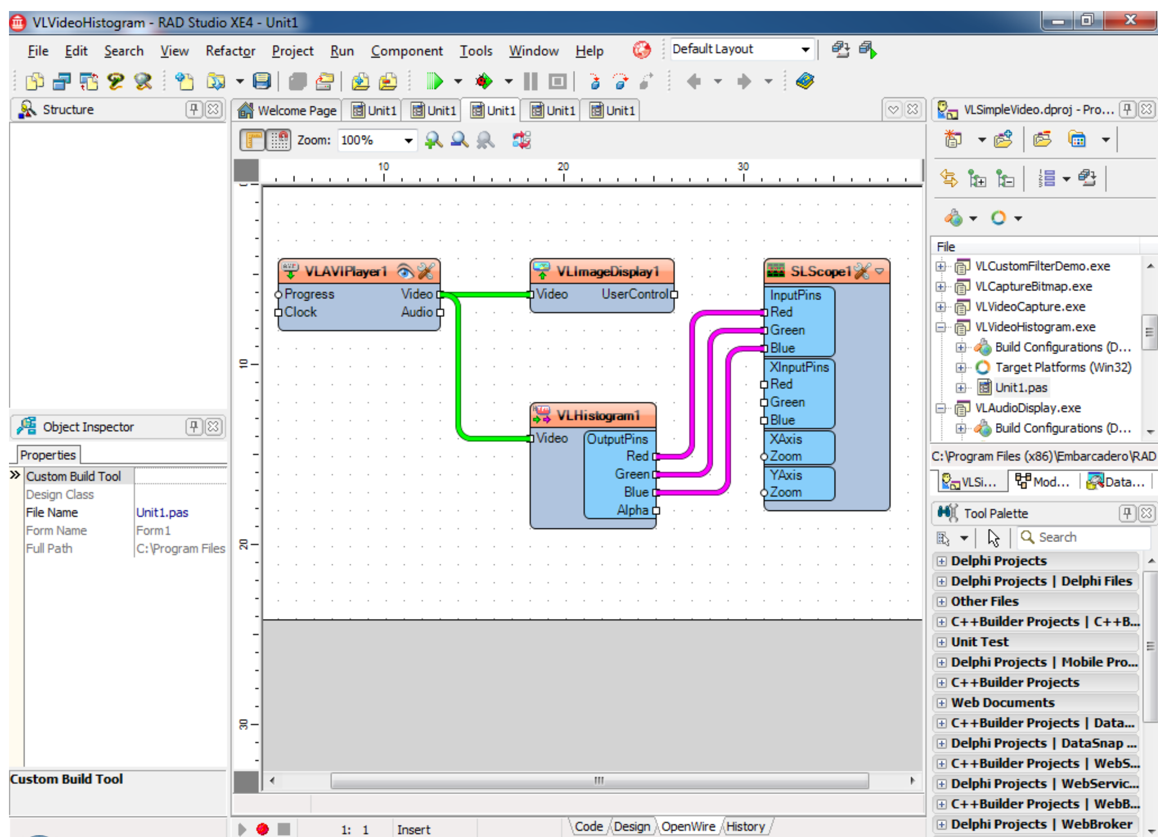
Open source project

OpenWire

Version 7.5



www.openwire.org
www.mitov.com

Open source project....1
Version 7.5....1
What's New In V7.5....4
What's New In V7.0....4
What's New In V6.0....4
What's New In V5.0....4
What's New In V4.5....4
What's New In V4.3....4
What's New In V4.0....4
What's New In V3.1....4
What's New In V3.0....4
What's New In V2.6....4
What's New In V2.5....5
What's New In V2.4....5
What's New In V2.3....5
What's New In V2.2....5
What's New In V2.1....5
What's New In V2.0....5
What's New In V1.8....5
OpenWire Web Sites....5
License....5
Introduction....6
Source, Sink Pins, and Multi Sink Pins....7
State Pins....12
Can I Use OpenWire for My Commercial Development?....13
OpenWire Installation....13
OpenWire Overview....16
Platforms....17
The OpenWire Graphical Editor....17
Naming Conventions....19
How to Use the Demo Package....19
Connecting Pins at Design Time (Using Property Editors)....24
Connecting Pins At Run Time (From Inside Your Code)....25
Understanding Basic OpenWire Pins....26
TOWObject....26
TOWBasicPin....27
TOWPin....29
TOWSinkPin....30
TOWMultiSinkPin....30
TOWSourcePin....30
TOWStatePin....31
TOWStateDispatcher....31
Downstreams and Upstreams....32
Change of State Broadcasting....32
OpenWire Stream Interfaces....32
Connecting and Handshaking....36
Standard (Well-known) Interfaces and Standard Pin Types....39
Clocking....39
Creating Components Using the Standard Interfaces and Pins....40
Using the Standard Source and Sink Pins In Your Components....40
Using the Standard State Pins In Your Components....43
How the Notify Really Works....45
Creating And Using Pins Implementing The Standard Interfaces....47
Defining Your Own Interfaces (Types Of Data)....53
OpenWire 7.5....2
Creating A Pin Implementing Your Interface....55
Creating A Pin Capable Of Sending Data Through your Interface....57
Registering Your Interface As a Standard One....59
Function Dependencies....59

Type Dependencies....60
Dynamic Streaming Order Balancing....60
Dynamic Pin Lists (Arrays)....62
Creating Components Using Pin Lists....63
Writing Threading Safe Components with OpenWire....66
Conclusion....67

What's New In V7.5

Added Delphi and C++ Builder XE7 support
Redesigned to use the new free Mitov.Runtime library.

What's New In V7.0

Added Delphi and C++ Builder XE5, and XE6 support.
Added MAC and Android support.
Added more standard pin types.
Added support for auto component suggestion in OpenWire Studio.
Simplified locking interface.
Improved integration with the OpenWire Editor.

What's New In V6.0

Added Delphi and C++ Builder XE3, and XE4 support.
Dropped support for Lazarus and versions older than XE2.
Complete redesign to utilize the latest Delphi features such as anonymous methods and attributes.
Improved integration with the OpenWire Editor.

What's New In V5.0

Added Delphi and C++ Builder XE and XE2 support.
Improved Lazarus support.
64 bit compatible.
Improved threading support.
Added expandable editors support.

What's New In V4.5

Added multi sink support.
Improved Lazarus support.

What's New In V4.3

Added Delphi and C++ Builder 2010 support.

What's New In V4.0

Added Format Converters.
Added Lazarus support for Windows, and Linux
New threading lock mechanism.
Added debug subscription support.

What's New In V3.1

Fixed loading from Frames.
Fixed support for languages other than English in PinLists.

What's New In V3.0

Introduces a brand new design for resolving pending pin connections.
Introduces a better .NET proxy support.
Added ConnectAfter and ConnectToStateAfter method and design time support for notification sequence control.

What's New In V2.6

Minor improvements.

What's New In V2.5

Improved multithreading support.
Added OperationID for the notify operations.
Added partial support for Delphi 10

What's New In V2.4

Multithreading support is added.
The FunctionSources and DataTyepeSources now are implemented as lists.
Improved data pumping support.
Delphi 2005 support is added.

What's New In V2.3

This version is mostly improved 2.2. No new features have been added, but there are some fixes.

What's New In V2.2

A new type of collection has been added, that supports expansion of the collection items inside the Object Inspector.

What's New In V2.1

Clock pins have been introduced.
The help has been expanded.

What's New In V2.0

Added support for State Pins.
The notification mechanism is changed to work trough operation objects.
Standard pins made simpler to use.

What's New In V1.8

Improved pin editor.
The TOWPinList has owner.
New mechanism of obtaining names.
TPinType.RemoveType and TPinType.ClearTypes methods have been added.
OWRegisterStream adds support for global dispatchers.
Restricting and renegotiating handshaking with different type of data (DataType restriction support) is finally implemented.

OpenWire Web Sites

The OpenWire web site: www.openwire.org
The Mitov Software web site: www.mitov.com

License

This software is provided 'as-is', without any express or implied warranty. In no event will the author be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented - you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

Introduction

OpenWire is a technology allowing VCL and CLX components to exchange data and event notifications among each other using unified basic framework.

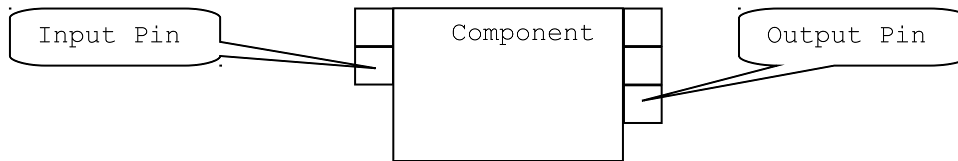
OpenWire is free open source library, available at www.openwire.org.

OpenWire defines the way the components can integrate and exchange data and state information, without having knowledge about each other.

OpenWire 2.0 defines 2 types of data exchange - Streaming and State. The Streaming is designed for sending a contiguous data stream from source to recipient (Sink). It is used typically in Data Acquisition and processing components. The State is used to synchronize the state of multiple components, or for synchronization with database components. The Streaming and the State connections are created via component properties named Pins. The Pins are one of the following 3 types – Source, Sink or State. Source and Sink are considered Streaming pins, however they can be used for State connections, and can be connected to State pins as well. The State pins are designed for exchanging state condition, but can be connected to Source of Sink pins, and thus be used for data streaming as well.

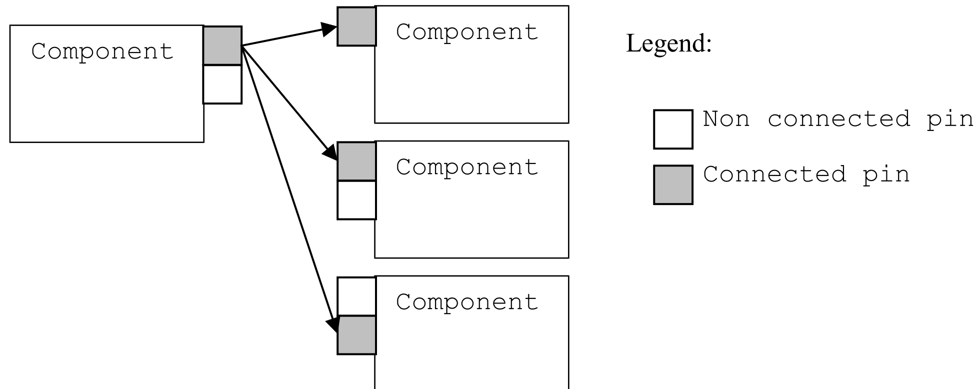
Source, Sink Pins, and Multi Sink Pins

Here is how a component, using OpenWire Source and Sink pins, looks like:

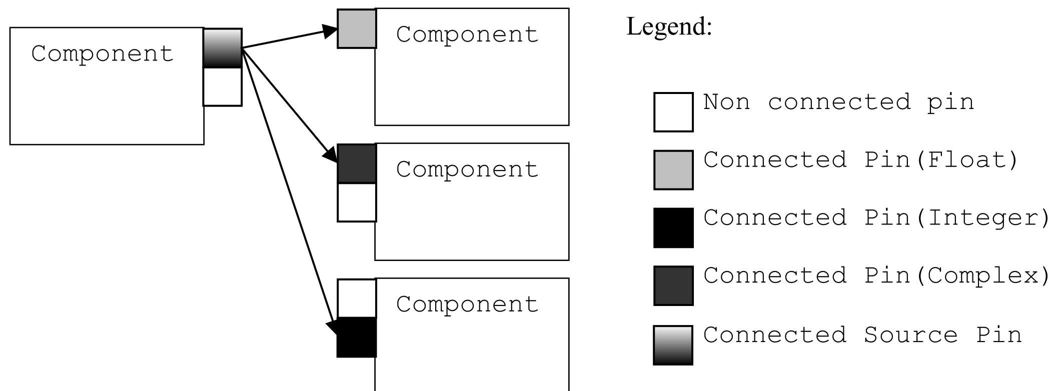


This component has 2 inputs and 3 outputs. The inputs and the outputs in OpenWire are called Pins. The input pins are called Sinks and the output pins are called Sources.

Every source pin can be connected and deliver data to multiple sinks:



One pin can stream different types of data, depend on what the sink will accept:



In this example the source pin delivers different type of data to each sink pin.

The sink pins can be made to accept multiple types of data as well.

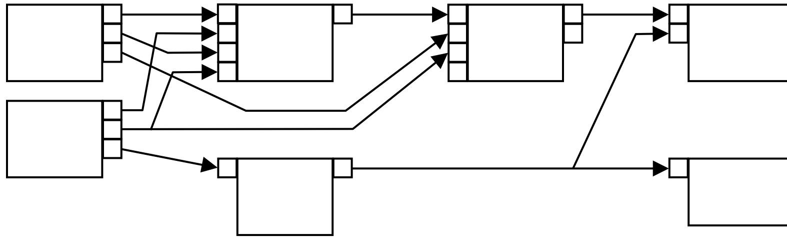
Every component may have as many source and sink pins as needed.

Version 4.5 also introduces a new type of multi sink pins. They can be connected to more than one source pin.

The pins are properties and are as easy to add as adding any other properties. They are even easier to add, then adding pointer to another component.

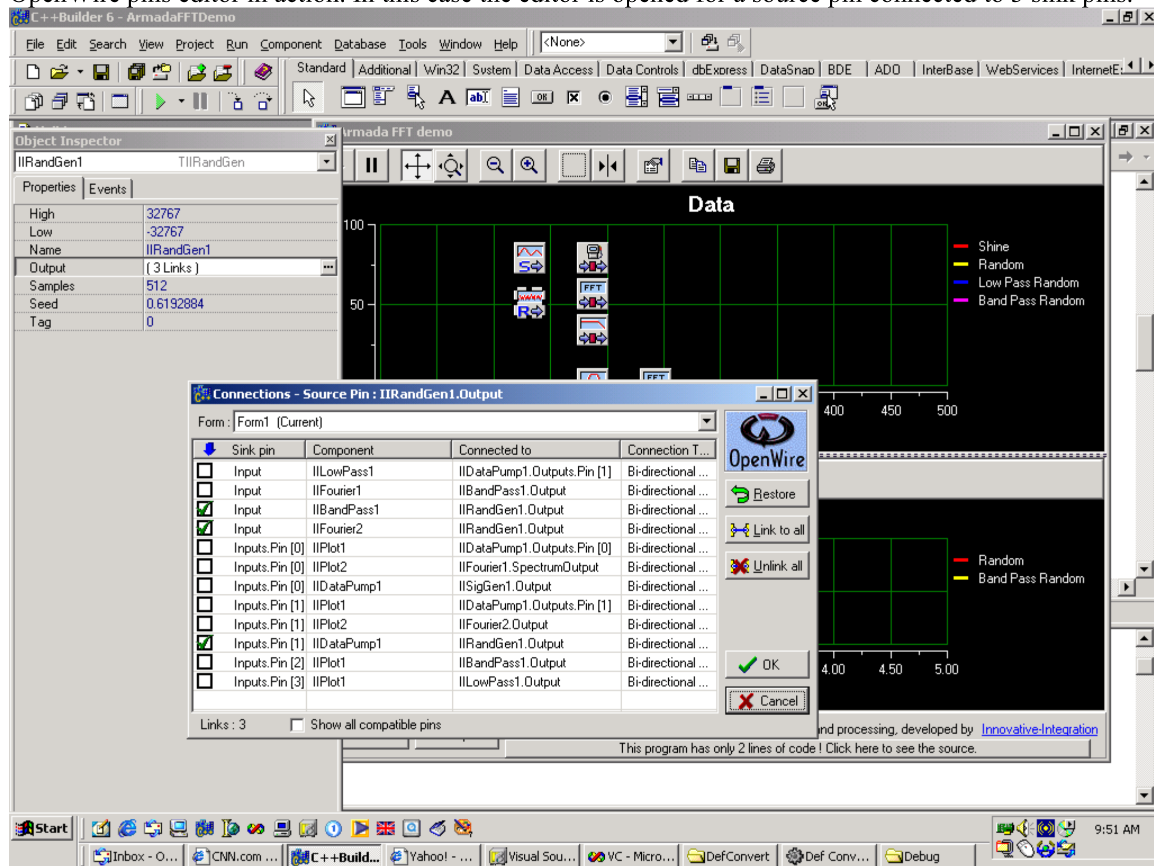
The pins are optimized for very high performance.

Here is a complex example of using pins:



All of the connections can be established at design time, without writing any code.
At run time you can connect, or disconnect 2 pins with only one line of code.

Here is an example of a very complex data processing application at design time. You can see the OpenWire pins editor in action. In this case the editor is opened for a source pin connected to 3 sink pins.



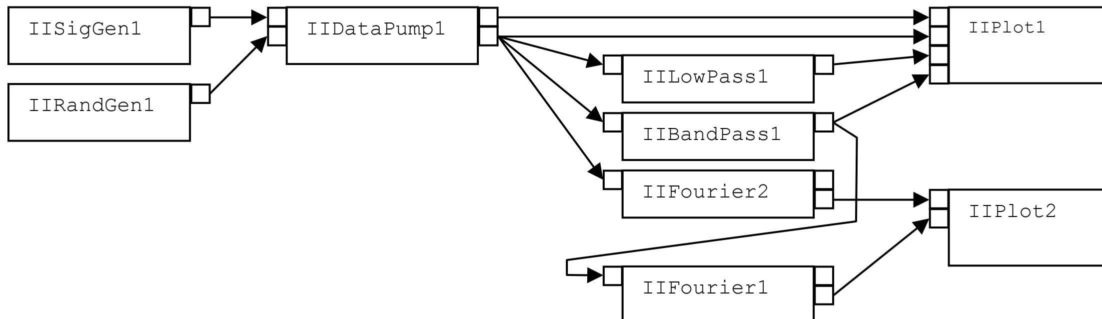
Here is the source of this sample:

```
#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
#include "Unit2.h"
#pragma package(smart_init)
#pragma link "IIDataPump"
#pragma link "IICommonFilter"
#pragma link "IISigGen"
#pragma link "iComponent"
#pragma link "IIPlot"
#pragma link "iPlot"
```

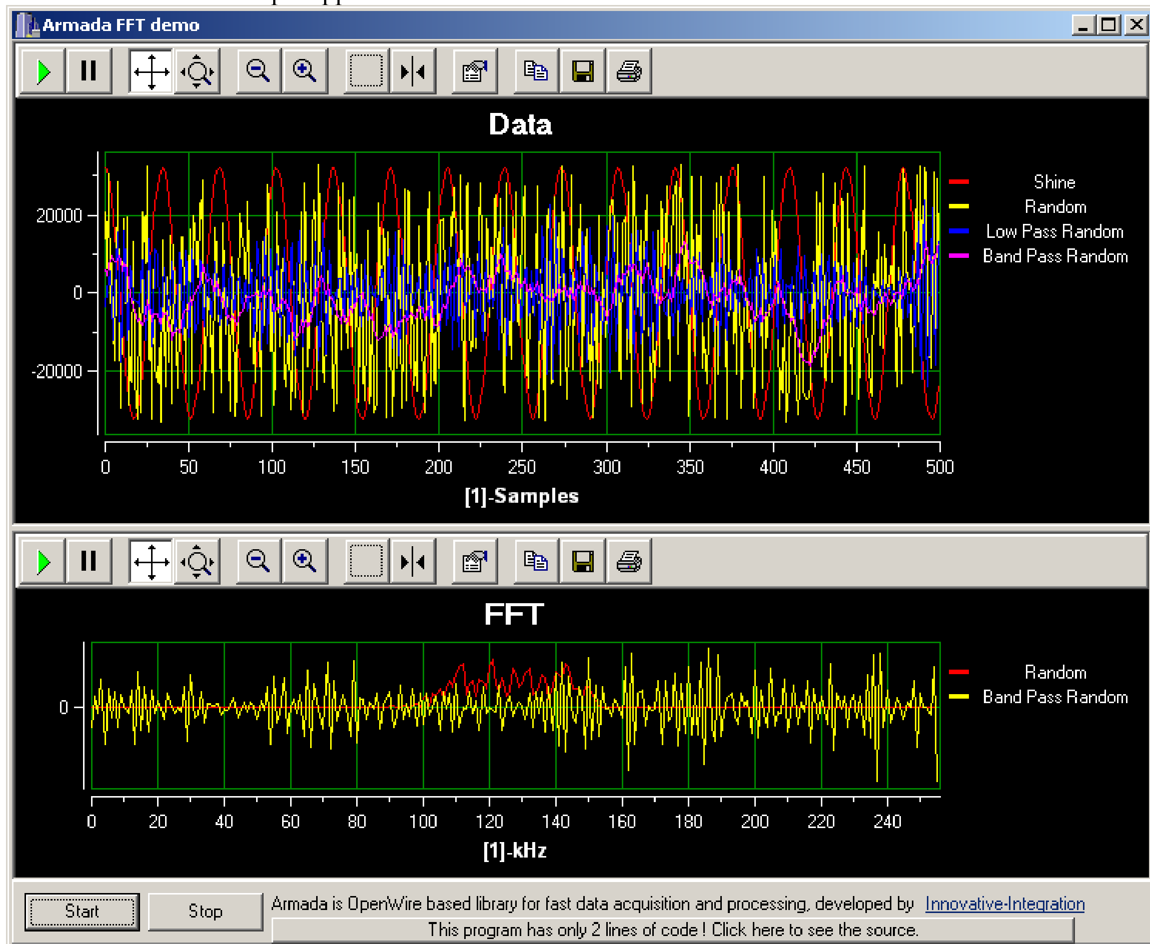
```
#pragma link "iPlotComponent"
#pragma link "IIRandGen"
#pragma link "IIHighPass"
#pragma link "IILowPass"
#pragma link "IIFourier"
#pragma link "IIBandPass"
#pragma resource "*.dfm"
TForm1 *Form1;

__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
}
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    IIDataPump1->Start();
}

void __fastcall TForm1::Button2Click(TObject *Sender)
{
    IIDataPump1->Stop();
}
```

As you can see the OpenWire is very powerful and yet easy to use technology. In this release of OpenWire, all of the connections at design time are established using property editors. A graphical editor allows you to connect the pins, by just dragging connections between them is under development.

Here is how all of the connections in this application look like:



Here is the same sample application at run time:



The application is doing a lot with only 2 lines of code:

```
IIDataPump1->Start();
```

and

```
IIDataPump1->Stop();
```

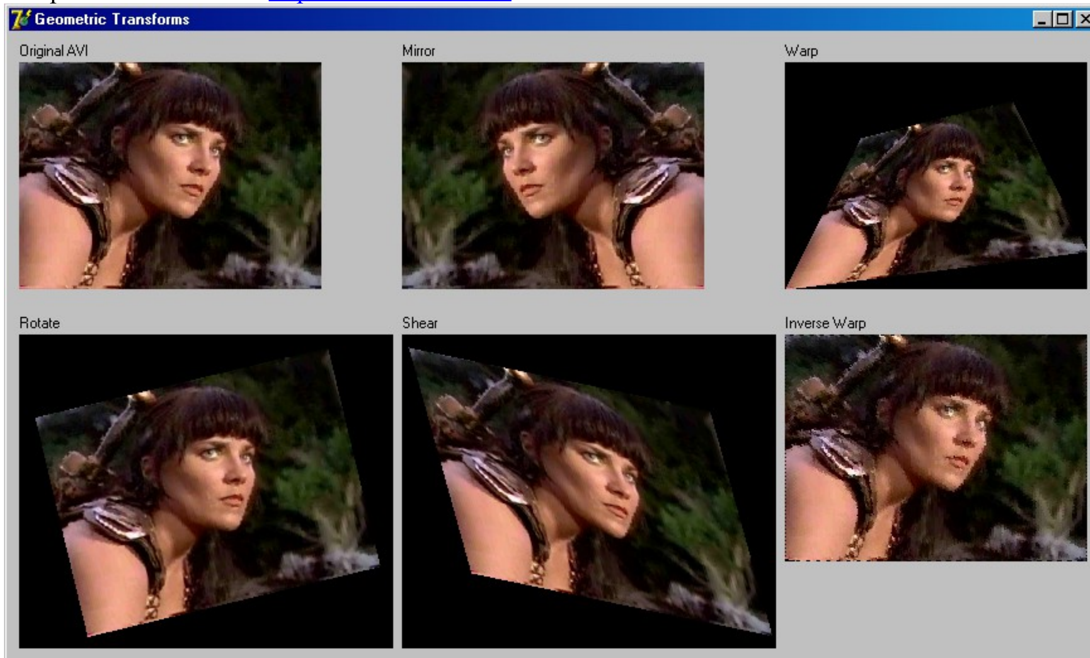
This application has been developed using the Armada VCL library, designed for very fast data acquisition and processing.

Armada is developed by Innovative-Integration and is sold with their data acquisition boards, allowing very fast and rapid development of complex real time data acquisition and processing applications.

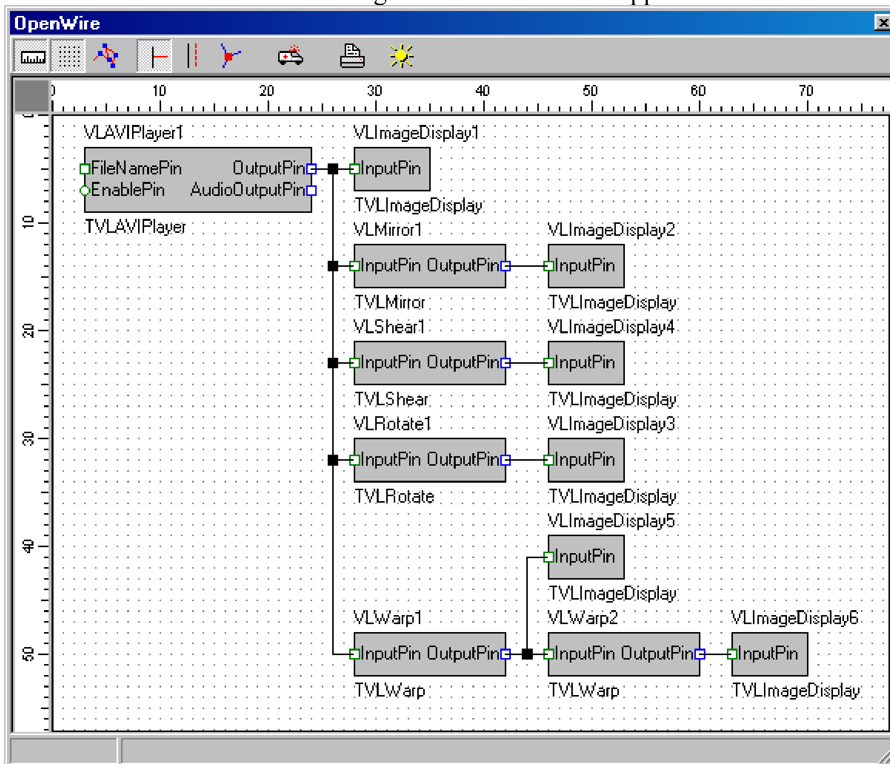
You can learn more about Armada, Chico, Vista and Toro data acquisition boards at: http://www.innovative-dsp.com.

OpenWire also allows many pins to be organized together in pin arrays. The pin arrays can be dynamic, allowing the component users to specify how many inputs and outputs they need. In the example above the first plot component has 4 input pins, the second only 2. They depend on the number of channels in the component, and are entered at design time by the developer.

Here is another example of OpenWire application, this time4 developed with the VideoLab set of components available at http://www.mitov.com.



Here is how the connection diagram looks like in this application:



And here is the source code in Delphi7:

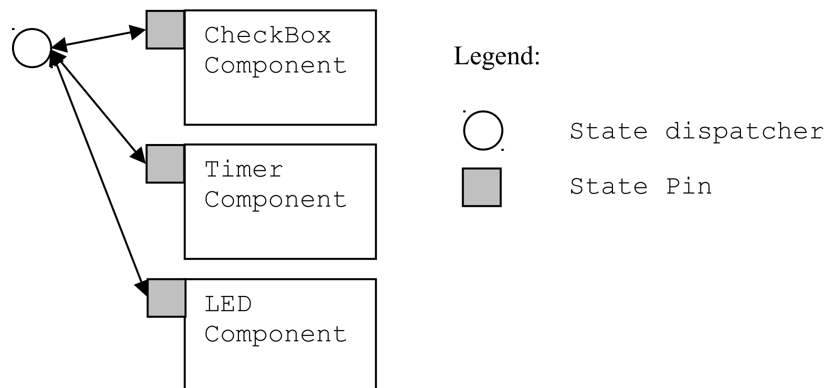
```
unit Unit1;

interface

uses
```

```
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms, Dialogs, VLAVIPlayer, VLImageDisplay, StdCtrls, VLCommonFilter,
VLMirror, VLShear, VLRotate, ExtCtrls, VLGenericFilter, VLWarp;

type
  TForm1 = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    VLImageDisplay1: TVLImageDisplay;
    VLImageDisplay2: TVLImageDisplay;
    VLImageDisplay3: TVLImageDisplay;
    VLImageDisplay4: TVLImageDisplay;
    VLAVIPlayer1: TVLAVIPlayer;
    VLMirror1: TVLMirror;
    VLRotate1: TVLRotate;
    VLShear1: TVLShear;
    VLImageDisplay5: TVLImageDisplay;
    VLImageDisplay6: TVLImageDisplay;
    Label5: TLabel;
    Label6: TLabel;
    VLWarp1: TVLWarp;
    VLWarp2: TVLWarp;
    procedure VLGenericFilter1FilterData(Sender: TObject;
      InBuffer: Variant; var OutBuffer: Variant;
      var SendOutputData: Boolean);

  end;

var Form1: TForm1;

implementation
{$R *.dfm}

end.
```

The application contains zero lines of code!

State Pins

State pins are designed primarily for exchange component state information. As example a CheckBox component can have StatePin connected to StatePin of a Timer, and to StatePin of a Led components. In this case if you click on the CheckBox you will enable/disable the timer as well, as you will turn the Led on/off. Here is how the connections will look like in this case:



Any time when you have State pin connected to one or more other pins, a hidden object named StateDispatcher is created. The object is responsible for holding the list of connected pins, and serves as event dispatcher. If you click on the CheckBox, the State pin will send the notification to the Dispatcher, and the dispatcher will send it to the rest of the pins. If you enable/disable the timer from within Delphi or C++ Builder code, the Timer itself will notify the CheckBox and the LED for the change of it's status.

You can connect One Source pin to a StateDispatcher, and thus use it as a state source. You can also connect any number of SinkPins to the StateDispatcher, and they will be notified for the change of state.

In order for a pin to be able to connect to a StateDispatcher, the pin should be able to be connected with any of the pins connected to the dispatcher. The pin can use a different interface for each of the pins connected to the Dispatcher.

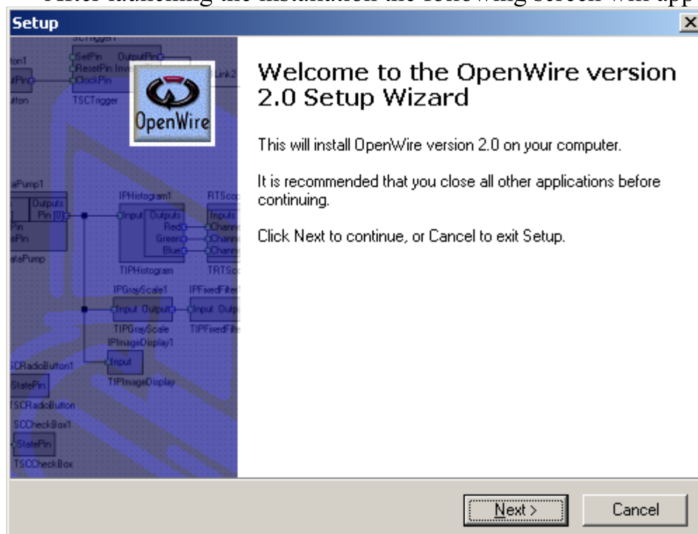
Can I Use OpenWire for My Commercial Development?

OpenWire is a free open source project. It can be used for any free or commercial project. You can define any type of proprietary pins, and you are not required to register them. However once you have created a well-defined data type, you may consider submitting it to the OpenWire web site, or using some other means of making it public knowledge. It will allow other developers to develop components capable of exchanging data with your set of components, and this way increasing the value of your product. Any changes to OpenWire as core done by you going into a public product, are required to be sent to the author of OpenWire, or the OpenWire committee in the future, in order to keep consistency of the standard. You are not required to provide any source code with any of your final products if they use OpenWire. Feel free to distribute them in any form you want. OpenWire is developed by Boian Mitov and as such you cannot claim it as your own product or development. Any references in your documentation to OpenWire should not refer to it as a technology developed by you or your company. You can charge any amount of money for any product developed using OpenWire. OpenWire is free and you don't have to pay any royalties or other fees in order to use it. However you can't charge for OpenWire itself other than shipping or packaging fees. The core technology is free and cannot be sold as standalone item.

OpenWire Installation

The following screenshots are taken from a real installation with a particular version of the installer. Though they will remain mostly the same among the different versions of the installation, some minor differences may exist.

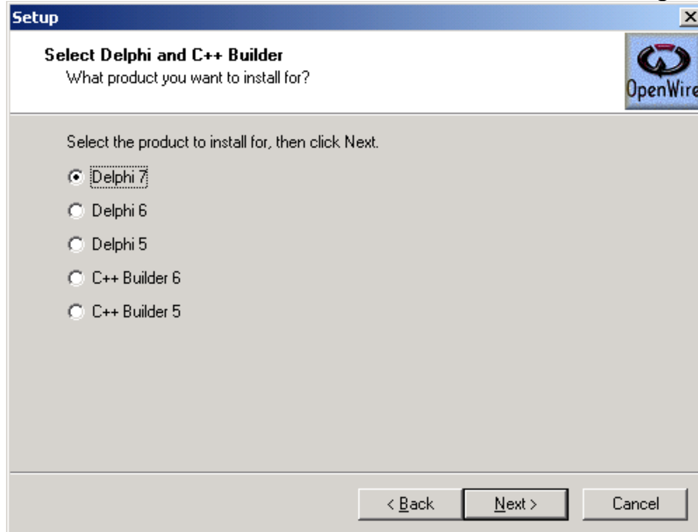
After launching the installation the following screen will appear:



Press Next. You will see the License agreement screen:

Read the text, and select "I accept the agreement" if you agree.

Press Next. You should see screen similar to the following one:

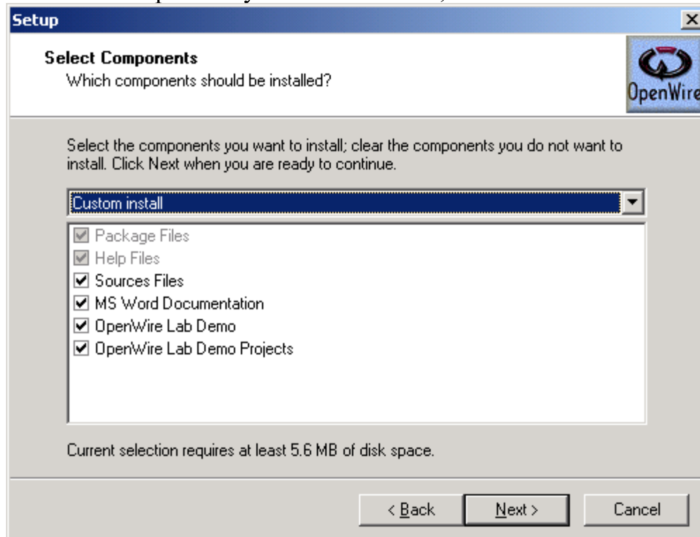


The content of this window will differ depend on the products you have installed on your system.

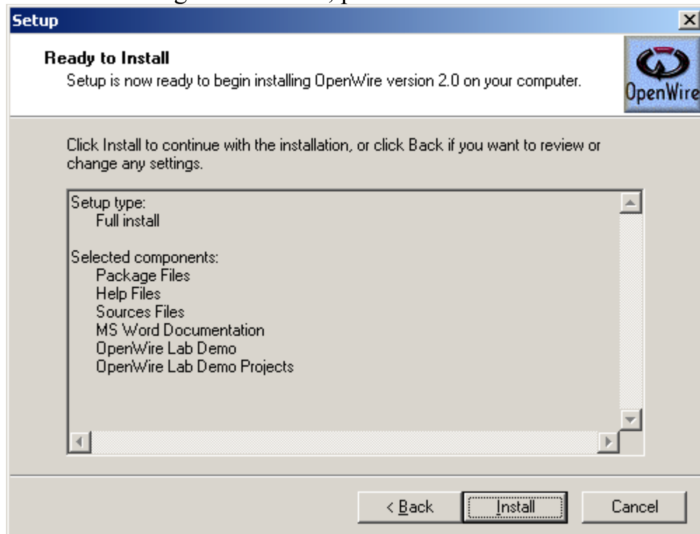
If you don't have any Delphi 5.0 or higher or C++ Builder 5.0 or higher installed, or the installation is unable to find the product, you will be able to install in a custom directory.

First make sure you have closed your C++ Builder and Delphi environments.

Select the product you wish to install, and click the Next button.

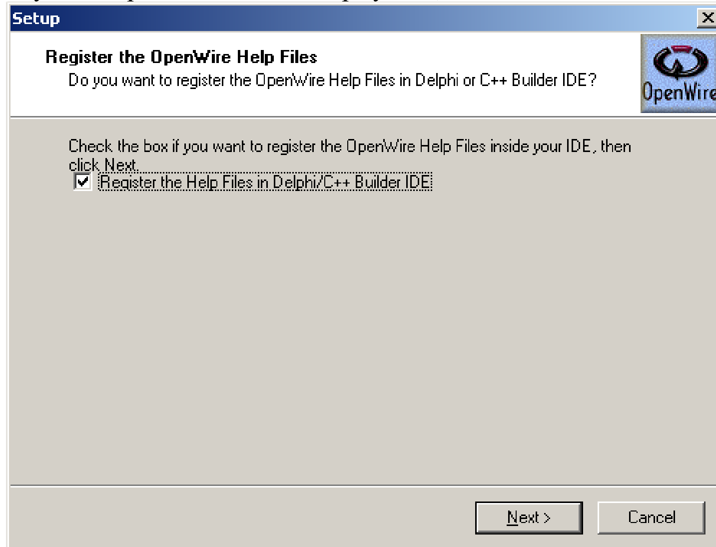


Here you can select the items you want to install.
After making the selection, press Next:



Here is your last chance to confirm the selection. If you are satisfied with the selection, press Install. The program will install OpenWire automatically.

After the installation the following screen will appear. If you would like to add the OpenWire help file to your Delphi/C++ Builder help system, check the box and Click Next.



This will be the end of the installation process. When the final window appear, Click the Finish Button.

WARNING: If the installation is unable to find your Delphi or C++ Builder, then you will see an additional window, asking for the directory where the files will be installed.

OpenWire Overview

In OpenWire, the connections among the components are established through small objects named Pins. OpenWire Version 1 defines 2 types of pins – SourcePins and SinkPins. Each component may have one or more pins of both types. One SourcePin can be connected to multiple SinkPins allowing very flexible and powerful designs. OpenWire 2 adds StatePins. StatePins can be connected in groups of pins via hidden internal objects named StateDispatchers. Each group of StatePins is capable of exchanging information about the change of state of a pin in the group, to be reflected in the rest of the pins. This allows maintaining of a state among the group of StatePins. In addition a single SourcePin can be connected to a StateDispatcher, serving this way as a State source for the StatePins connected to the dispatcher. Any number of SinkPins can be connected to the StateDispatcher as well, this way allowing the pins to be notified about a change in the State of the Dispatcher.

The main purpose of OpenWire is to allow different components to be able to connect without even knowing about each other. The real connection is established among the pins. The pins are communicating to each other using Delphi interfaces. This way if the interface has the same GUID and entries, both components are able to exchange data although two different companies may have designed them.

OpenWire doesn't contain any components; neither does it specify any component hierarchies. Instead specifies the basic pins hierarchy as well as some of the basic Interfaces. The purpose is to build a growing database of registered Interfaces over time allowing more and more components from different vendors to be able to work together. A nice feature is the fact that any pin can support multiple interfaces. This may sound complex but as you will see later implementing a pin support in a component is just few very simple lines of code and is even easier than supporting a pointer to another component as example. In addition OpenWire defines dynamic arrays of pins allowing easily creating components with multiple inputs and outputs organized into dynamic arrays i.e. you can change the number of inputs or outputs as result of other settings in the component. A good example is an "add" component that has a user defined number of inputs and output which is result of the sum of the inputs. OpenWire doesn't define the way you develop your components or a component hierarchy instead it focuses on pins and interfaces. Indeed you can take any existing component and add pins to it with a minimal effort, in many cases easier than adding a new property.

The mechanism of sending data from pin to pin is called Stream. There are 2 types of streams in OpenWire Version 1. Upstreams and Downstreams. You are sending data or notification downstream when the source pin calls interface functions in the sink pins, it is connected to. In this case the data gets delivered from source pin into sink pin. Upstream is the opposite. In this case the sink pin calls an interface

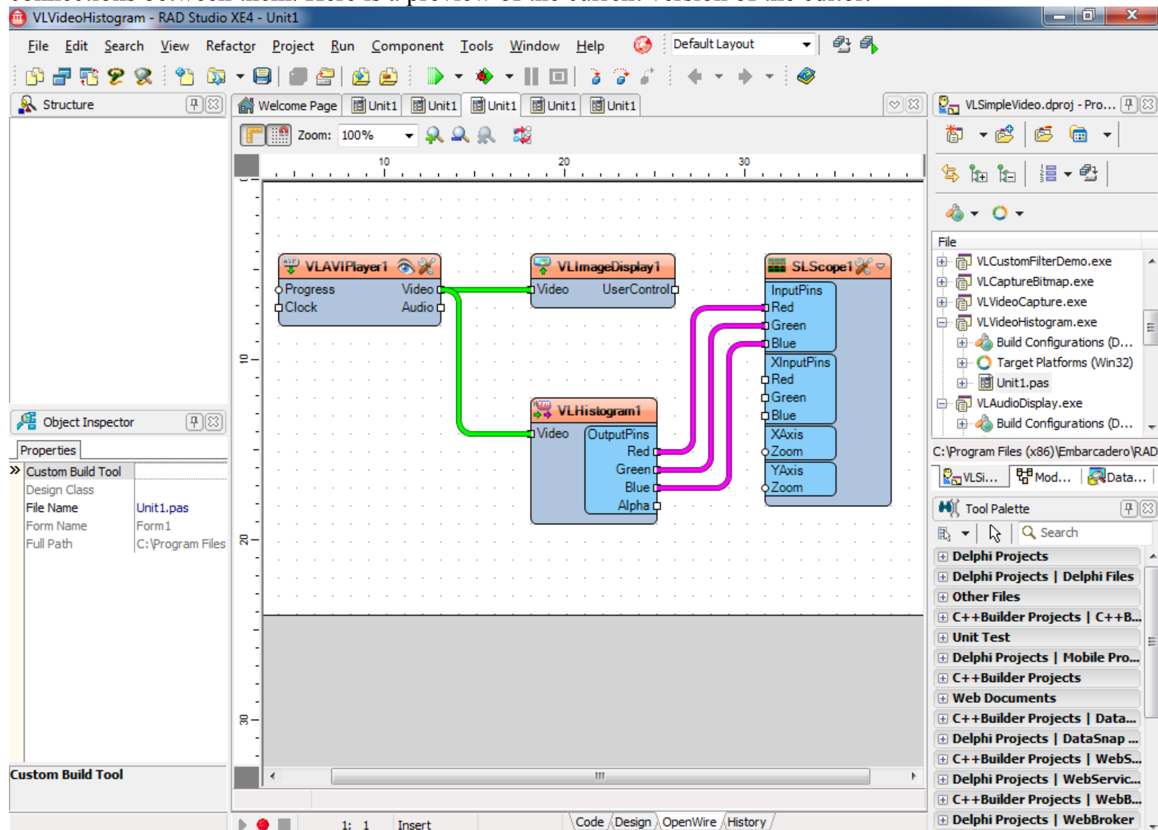
function inside the source pin, usually in order to ask for data or to send change of condition notification. OpenWire 2 adds another exchange mechanism – change of state broadcast. Change of state broadcast allows multiple components to maintain the same state among each other.

Platforms

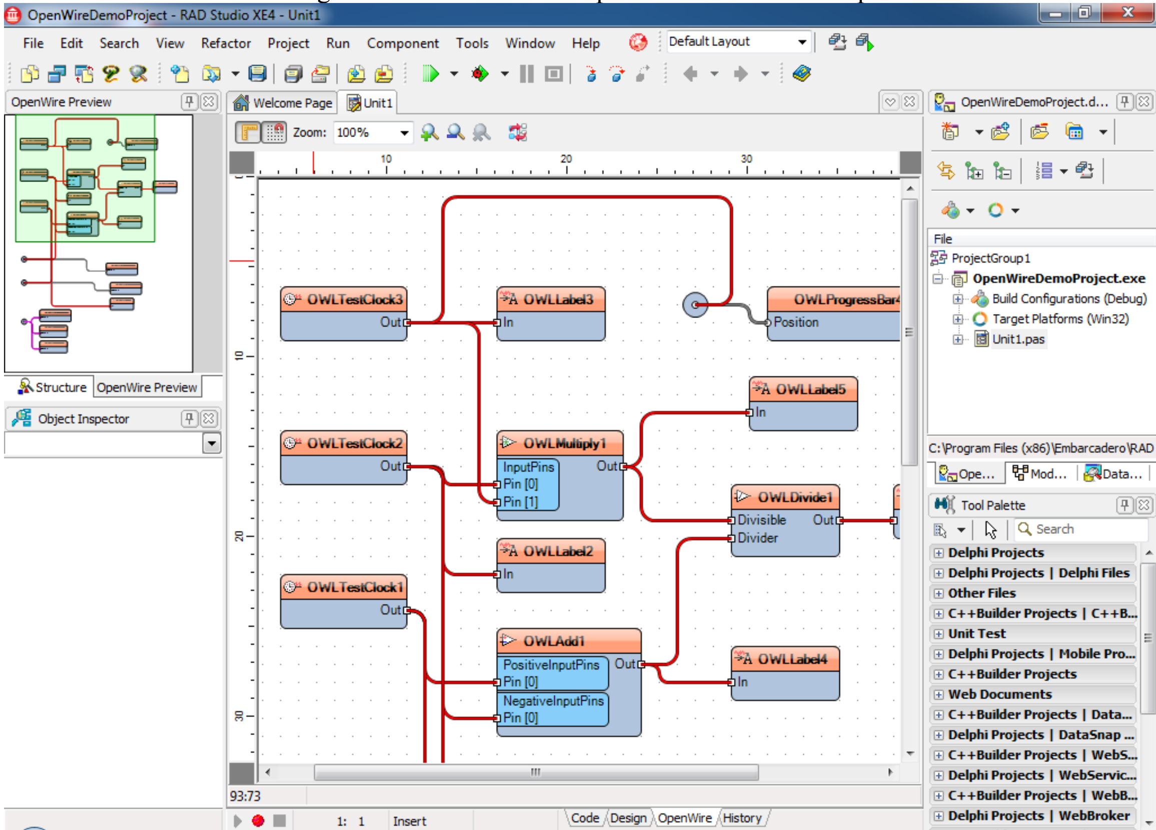
The current version of OpenWire supports Delphi 5.0, Delphi 6.0, Delphi 7.0, Delphi 2005, Delphi 2006, Delphi 2007, Delphi 2009, Delphi 2010, C++ Builder 5.0 and C++ Builder 6.0, C++ Builder 2006, C++ Builder 2007, C++ Builder 2009, C++ Builder 2010, and Lazarus under Windows and Linux.

The OpenWire Graphical Editor

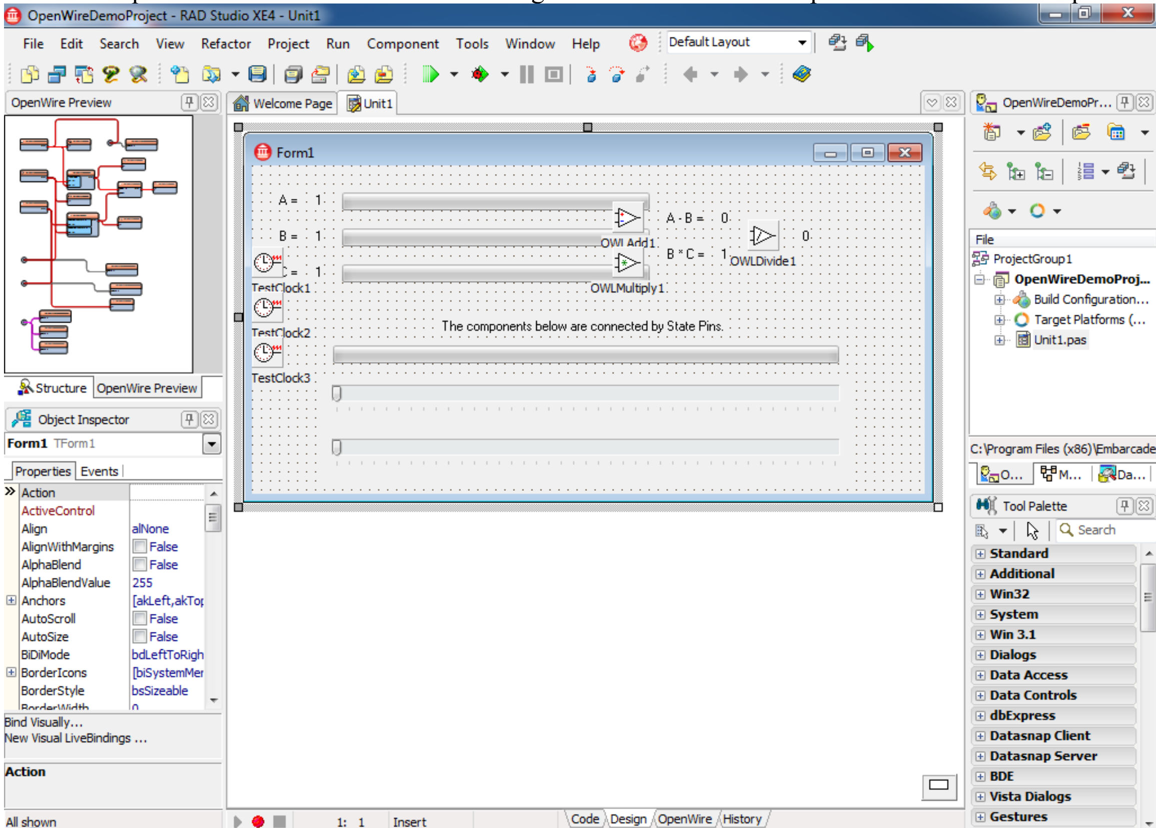
An OpenWire graphical editor is available. The editor allows users to connect pins by visually dragging connections between them. Here is a preview of the current version of the editor:



Here is the editor showing the connections in the OpenWire lab demo example:



This snap shot is taken with the editor showing the connections in the OpenWire lab demo example:

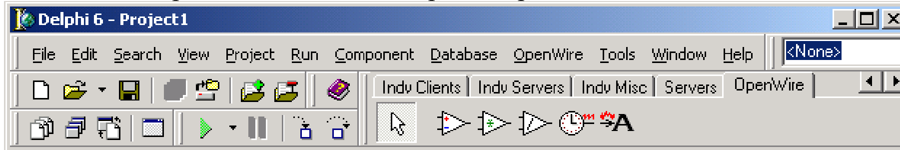


Naming Conventions

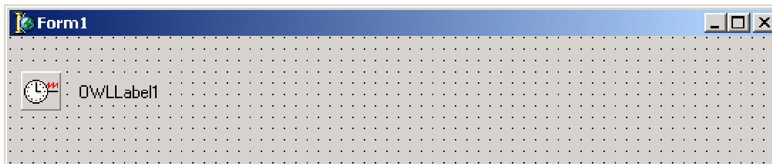
All the OpenWire object names start with TOW… Examples : TOWSourcePin, TOWSinkPin etc.
All the OpenWire exposed global function names start with OW… Example : OWRegisterStream
All the OpenWire interface type names start with IOW… Example : IOWStream

How to Use the Demo Package

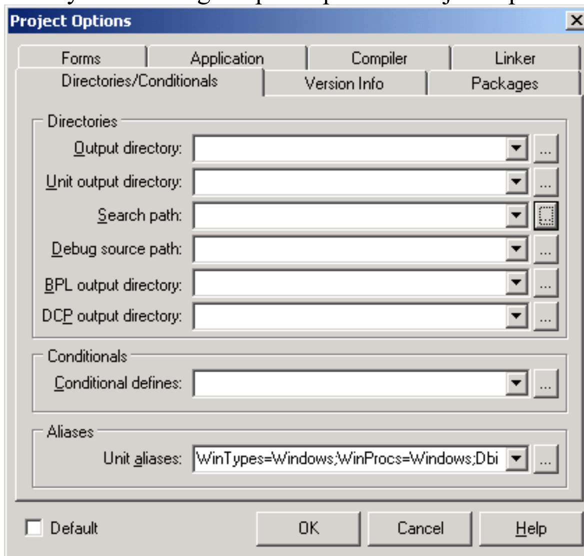
Select the OpenWire tab on the components palette.



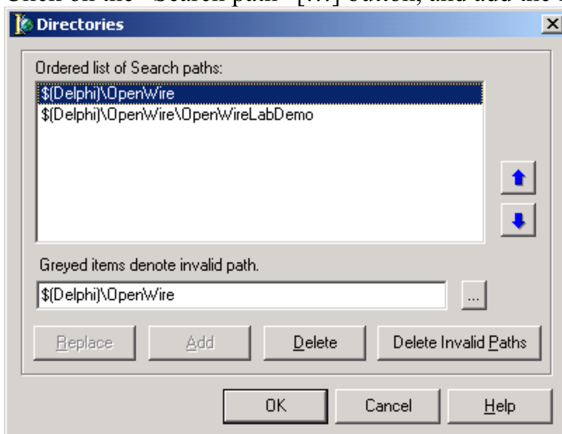
Click on TOWLTestClock component and drop it on the form, then click on TOWLLabel an also drop it on the form as shown below:



If you are using Delphi: Open the Project Options:

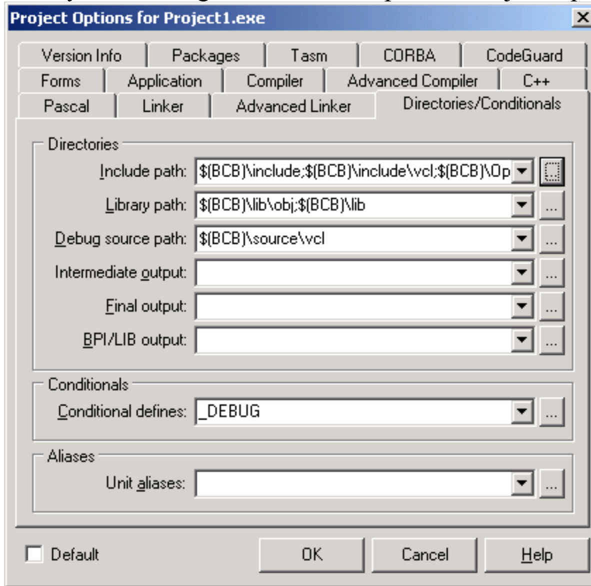


Click on the "Search path" […] button, and add the following paths:

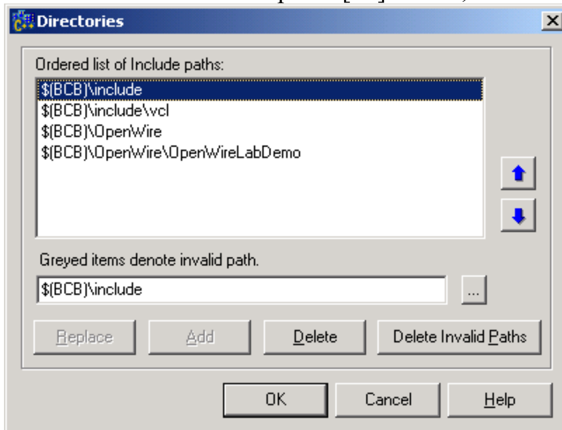


Click OK.

If you are using C++ Builder: Open the Project Options:

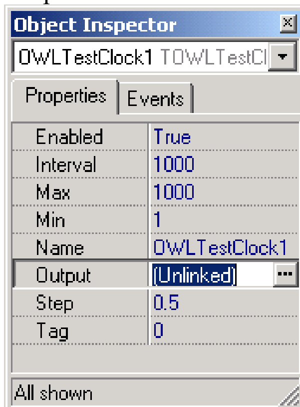


Click on the "Include path" […] button, and add the following paths:

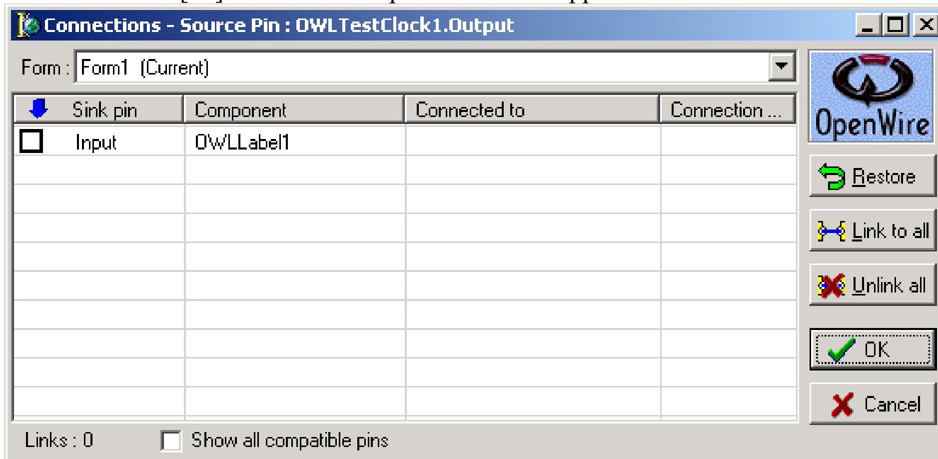


Click OK.

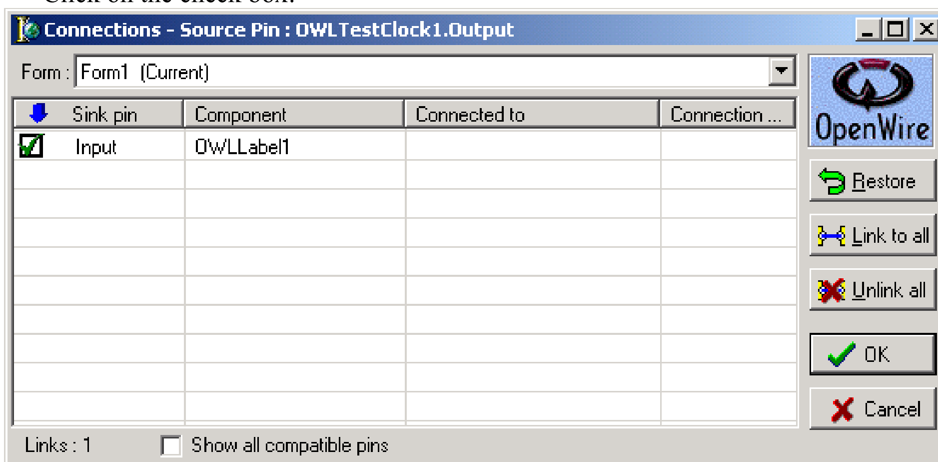
Click on the OWLTestClock1, switch to the Object Inspector. Select the Output property as shown on the picture:



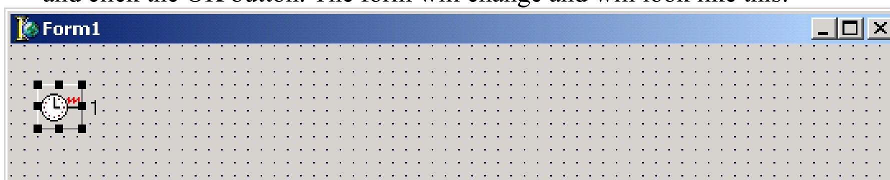
Click on the […] button and the pins editor will appear:



Click on the check box:

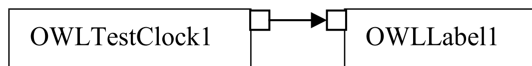


and click the OK button. The form will change and will look like this:

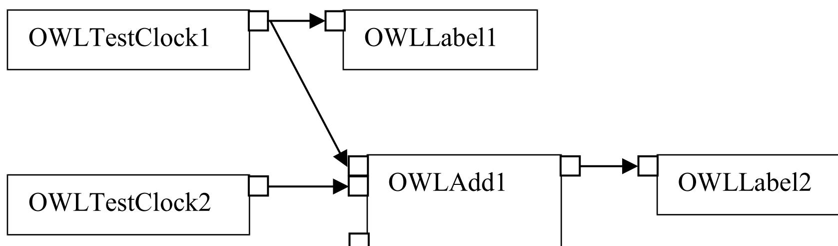


Save and run the application. The number in the label will start increasing every second.

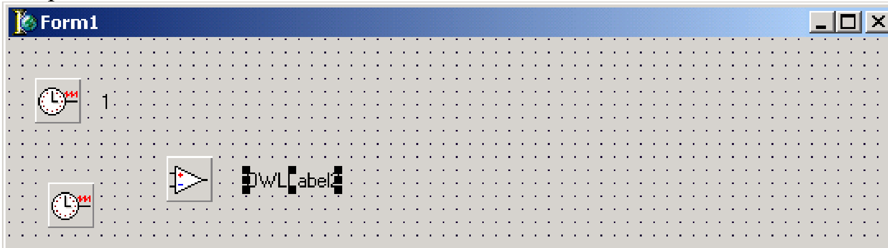
The OWLTestClock1 component will generate increasing numbers and will send them via the pin to the OWLLabel1 component. Here is how the diagram looks like:



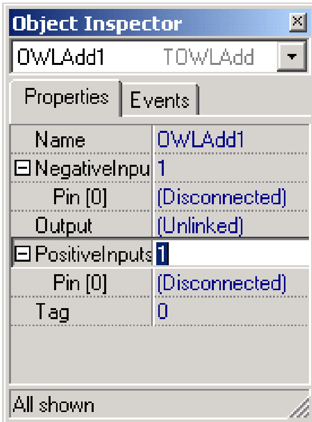
Now let's make the example a little bit more complex:



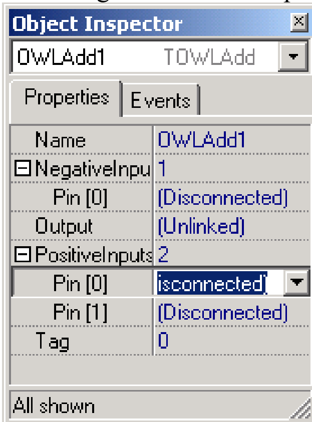
We need another TOWLTestClock component and another TOWLLabel as well as TOWLAdd components.



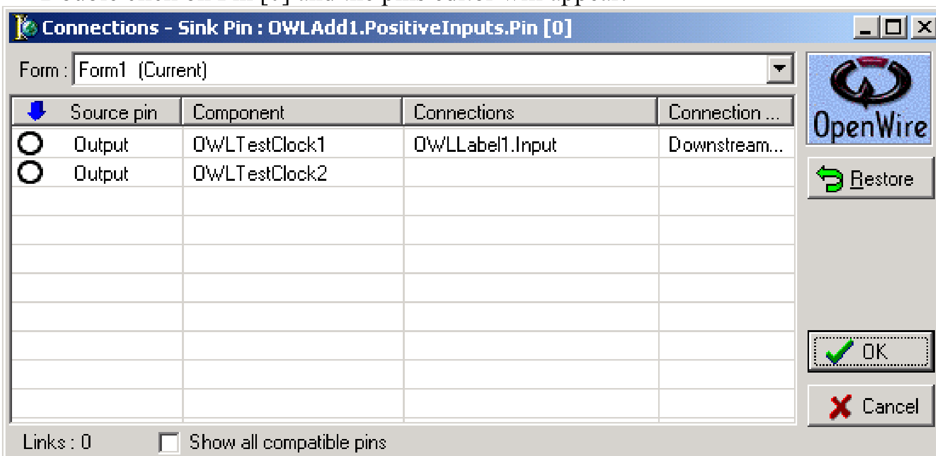
Select the OWLAdd1 and the switch to the object inspector:



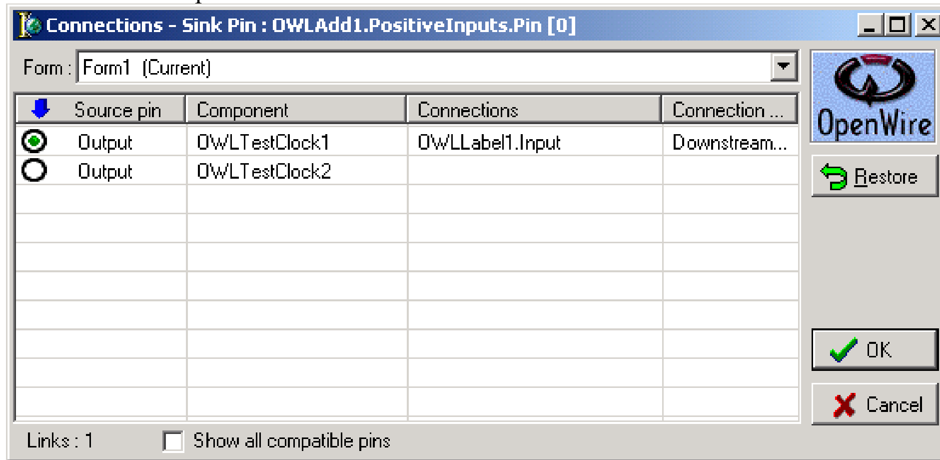
Change the amount of positive inputs to 2. A new pin will appear:



Double click on Pin [0] and the pins editor will appear:

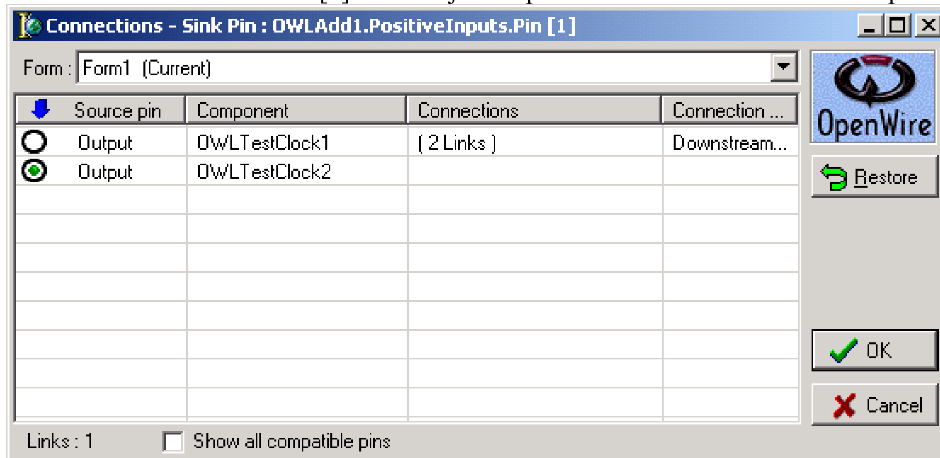


Select the Output of OWLTestClock1 :

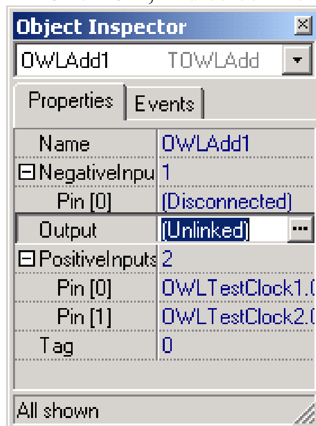


and click OK.

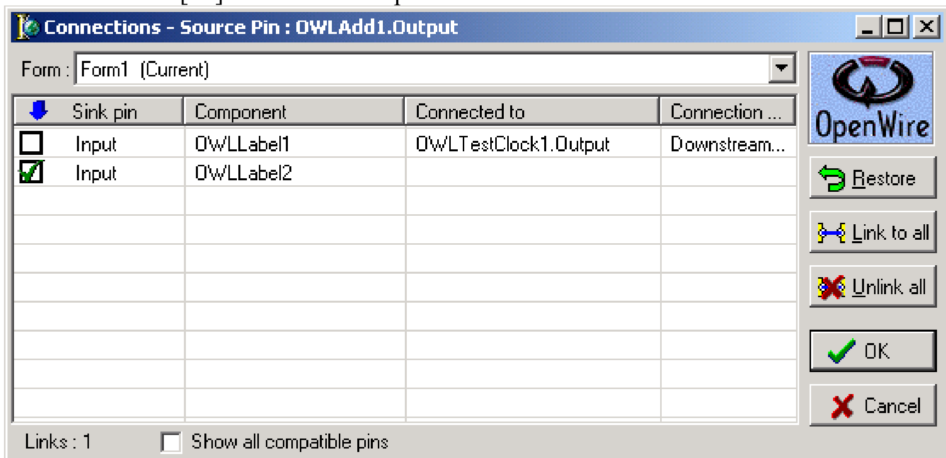
Then double click on Pin [1] in the object inspector and connect it with the Output of OWLTestClock2:



Click OK, and select the Output property in the Object inspector:



Click on the […] button select Input of OWLLabel2:



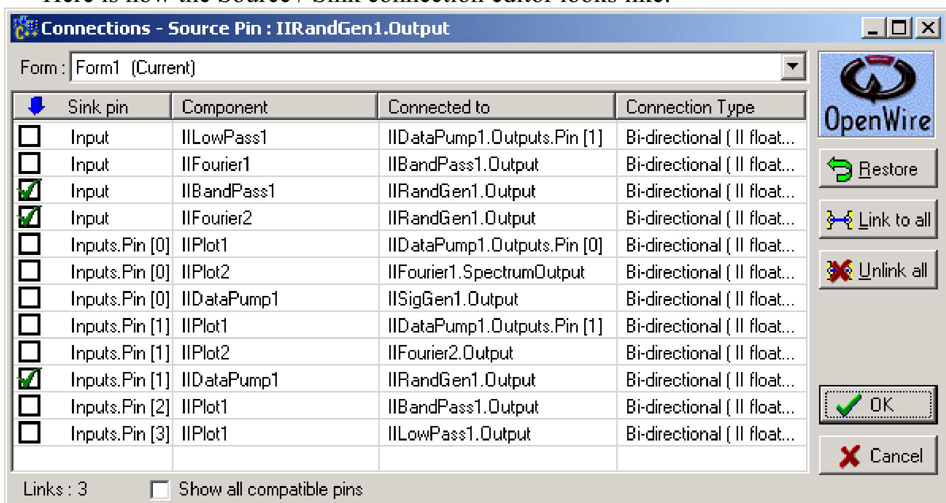
Click OK, compile and run. The application will show 2 numbers and one of them will increase 2 times faster than the other. The value will be result of adding the data sent by the 2 timers.

You can continue to experiment with more complex connections. Please keep in mind that the OpenWire demo lab package is just for demonstration purposes, and as such demonstrates just the very basic OpenWire functionality. You can see the source code of the examples and you can create much more powerful and robust components, based on OpenWire. One example of what can be achieved by OpenWire is the Armada components set developed by Innovative-Integration http://www.innovative-dsp.com/ Please feel free to check their web site. Please check also the OpenWire official site at http://www.openwire.org . Another great example of advanced OpenWire components is the VideoLab set of components, available from http://www.mitov.com .The OpenWire is under rapid development and expansion, and soon new features and components will be available for download.

Connecting Pins at Design Time (Using Property Editors)

OpenWire provides two property editors to support the design time pins connection. The editors are registered, so they will handle any component property of the type either sink or source pin. There are property editors for the pin lists as well. If you open a source pin property editor dialog, you can establish connections to multiple sink pins. When editing a sink pin, you can select only one source pin to be connected to it. A graphic editor expert similar to the appearance of LabView and HP VEE is under development, and will be available later this year.

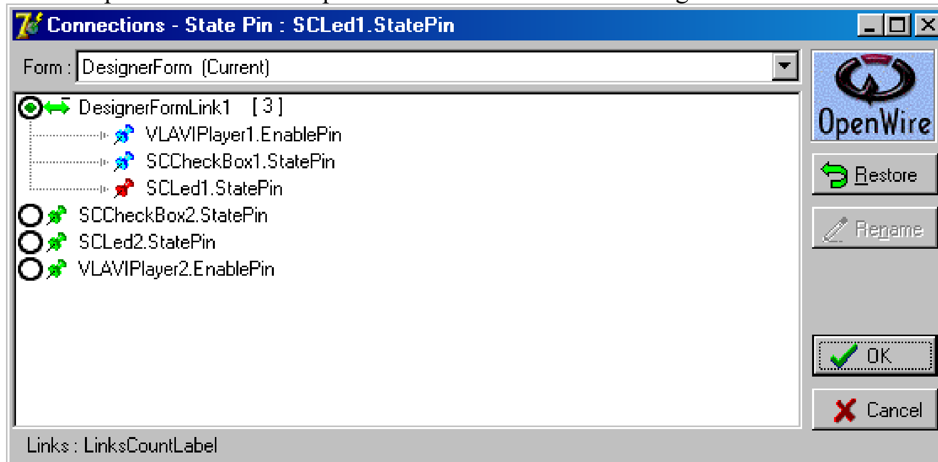
Here is how the Source / Sink connection editor looks like:



In this case we are editing a source pin. We have selected to connect the pin to 3 sink pins. In the first column we can see all the pins compatible with our source pin. In the second column are the components to which those pins belong. In the third column we can see if the sink pin is connected to another source pin.

In case there is a connection, we can see the negotiated Upstream and Downstream data type in the last column. We can link the source pin to all the sinks, unlink it or restore the original connection by using the buttons. In OpenWire 2 the editor is extended to support connecting Source/Sink pins to State pins and State Dispatchers.

State pins introduced in OpenWire 2 can be connected using the State Pin connection editor:



Here 3 pins are connected together via dispatcher named DesignerFormLink1. Another 3 pins are not connected. You can connect the pin either to a Dispatcher(group of pins), either to a non connected pin, creating this way a new dispatcher. Once created, you can rename a dispatcher to give it a more reasonable name. You can also use the Restore button, to restore the original connection of the pin, as it was at the moment the editor was opened.

Connecting Pins At Run Time (From Inside Your Code)

At runtime you can connect two pins directly using the Connect function:

```
Delphi Example :
  if MyComponent1.Pin1.Connect( OtherComponent.Pin2 ) then
    // Successful connection
```

```
C++ Builder Example :
  if( MyComponent1->Pin1->Connect( OtherComponent->Pin2 ))
    // Successful connection
```

You can connect two pins via Dispatcher using the ConnectByState function:

```
Delphi Example :
  if MyComponent1.Pin1.ConnectByState( OtherComponent.Pin2 ) then
    // Successful connection
```

```
C++ Builder Example :
  if( MyComponent1->Pin1->ConnectByState( OtherComponent->Pin2 ))
    // Successful connection
```

You can disconnect a pin using the Disconnect procedure.

```
Delphi Example :
  MyComponent1.Pin1.Disconnect;
```

```
C++ Builder Example :
  MyComponent1->Pin1->Disconnect();
```

Connect will return True in case the connection can be established and False in case it can’t. In order the connection to succeed the following conditions should be met:

1. One of the pins should be TOWSinkPin or TOWStatePin descendant and the other should be TOWSourcePin or TOWStatePin descendant,
2. One of the pins should implement an interface, which the other knows how to deal with.
3. The two pins should not be in functional dependency of each other – as example if the source pin has been calculated as result of the sink pin. In this case the source pin can't connect to the sink pin for an obvious reason.
4. In case of a long chain of pins connected in type dependency, the entire chain might be restricted to use a homogeneous interface. In this relatively rare case all of the pins will have to have at least one common interface. Otherwise the connection will fail.

If all of the above requirements are met, the connection will succeed and the Connect function will return True.

Understanding Basic OpenWire Pins

OpenWire Version 1 defines 2 types of pins TOWSourcePin and TOWSinkPin. They both are inherited from TOWPin. TOWPin is basic abstract class and can't be used in any components.

TOWObject

This is the root class for all the OpenWire objects. It provides the basic multithreading support.

You can lock an object for reading:

```
Delphi Example :
  var ReadLockSection : IOWLockSection

  ReadLockSection := APin.ReadLock();
    // The APin will be locked untill the end of the curent
    // function body.
```

```
C++ Builder Example :
  _di_IOWLockSection ReadLockSection = Pin->ReadLock();
    // The APin will be locked untill the end of the curent
    // function body.
```

You can lock an object for writing:

```
Delphi Example :
  var ReadLockSection : IOWLockSection

  ReadLockSection := APin.WriteLock();
    // The APin will be locked untill the end of the curent
    // function body.
```

```
C++ Builder Example :
  _di_IOWLockSection ReadLockSection = Pin->WriteLock();
    // The APin will be locked untill the end of the curent
    // function body.
```

You can make the object share a lock with another object:

```
Delphi Example :
  APin1.AddShareLock(APin2 );
```

```
C++ Builder Example :
  APin1->AddShareLock( APin2 );
```

You can separate the two objects locks:

```
Delphi Example :
  APin1.RemoveShareLock(APin2 );
```

```
C++ Builder Example :
  APin1->RemoveShareLock( APin2 );
```

TOWBasicPin

This is an abstract base class for the TOWPin class and internal proxy pins created to support design time pins across forms.

Two pins can be connected directly using the Connect member function.

```
Delphi Example :
  if MyComponent1.Pin1.Connect( OtherComponent.Pin2 ) then
    // Successful connection
```

```
C++ Builder Example :
  if( MyComponent1->Pin1->Connect( OtherComponent->Pin2 ))
    // Successful connection
```

Sometimes we want to make sure the pin we connect will receive data after another pin. This can be achieved by using the ConnectAfter function. This function connects the 2 pins, and makes sure the data recipient pin will be notified after the third pin is notified:

```
Delphi Example :
  if MyComponent1.Pin1.ConnectAfter( OtherComponent1.Pin2,
OtherComponent2.Pin3 ) then
    // Successful connection
```

```
C++ Builder Example :
  if( MyComponent1->Pin1->ConnectAfter( OtherComponent1->Pin2,
OtherComponent2->Pin3 ))
    // Successful connection
```

Two pins can be connected via Dispatcher using the ConnectByState function:

```
Delphi Example :
  if MyComponent1.Pin1.ConnectByState( OtherComponent.Pin2 ) then
    // Successful connection
```

```
C++ Builder Example :
  if( MyComponent1->Pin1->ConnectByState( OtherComponent->Pin2 ))
    // Successful connection
```

You can connect a pin to a State Dispatcher using ConnectToState function:

```
Delphi Example :
  if MyComponent1.Pin1.ConnectToState( AState ) then
    // Successful connection
```

```
C++ Builder Example :
  if( MyComponent1->Pin1->ConnectToState( AState ))
    // Successful connection
```

Sometimes we want to make sure the pin we connect will receive data after another pin. This can be achieved by using the ConnectToStateAfter function. This function connects the pin to a state, and makes sure the pin will be notified after the other pin is notified:

```
Delphi Example :
  if MyComponent1.Pin1.ConnectToStateAfter( AState,
OtherComponent.Pin2 ) then
    // Successful connection
```

```
C++ Builder Example :
  if( MyComponent1->Pin1->ConnectToStateAfter( AState, OtherComponent->Pin2 ))
    // Successful connection
```

At design time the connections can be established using a property editors. A graphic editor expert allowing establishing the connections visually is under development.

For details on connecting pins please see “Connecting Pins At Design Time” and “Connecting Pins At Run Time”.

You can disconnect a pin using the Disconnect procedure:

```
Delphi Example :
  MyComponent1.Pin1.Disconnect;
```

```
C++ Builder Example :
  MyComponent1->Pin1->Disconnect();
```

You can check if two pins can connect to each other using the CanConnectTo member function:

```
Delphi Example :
  if MyComponent1.Pin1.CanConnectTo( OtherComponent.Pin2 ) then
    // The pins can connect to each other
```

```
C++ Builder Example :
  if( MyComponent1->Pin1->CanConnectTo( OtherComponent->Pin2 ))
    // The pins can connect to each other
```

You can check if a pin can connect a state dispatcher by using CanConnectToState member function:

```
Delphi Example :
  if MyComponent1.Pin1.CanConnectToState( AState ) then
    // The pin can connect AState
```

```
C++ Builder Example :
  if( MyComponent1->Pin1->CanConnectToState( AState ))
    // The pin can connect AState
```

You can also check if two pins are compatible (can talk to each other through at least one type of interface). To do so you can use the IsCompatible member function:

```
Delphi Example :
  if MyComponent1.Pin1.IsCompatible( OtherComponent.Pin2 ) then
    // The pins are compatible
```

```
C++ Builder Example :
  if( MyComponent1->Pin1->IsCompatible( OtherComponent->Pin2 ))
    // The pins are compatible
```

It’s not very likely for you to use this function however.

To check whether or not two pins are connected together use the IsConnectedTo function:

```
Delphi Example :
  if MyComponent1.Pin1.IsConnectedTo( Pin2 ) then
    // The pins are connected
```

```
C++ Builder Example :
  if( MyComponent1->Pin1->IsConnectedTo( Pin2 ))
    // The pins are connected
```

To check if a pin is connected you can use the IsConnected function:

```
Delphi Example :
  if MyComponent1.Pin1.IsConnected then
    // The pin is connected
```

```
C++ Builder Example :
  if( MyComponent1->Pin1->IsConnected())
    // The pin is connected
```

The Notify function:

```
 function Notify( Operation : IOWNotifyOperation ) :
TOWNotifyResult; virtual; abstract;
```

will be covered in details later. It is used to send notification and deliver data Downstream or Upstream. The Operation object parameter is used to specify a interface specific, user defined operation (as example TOWSuppliedOperation, TOWStartOperation etc.) and if needed the object will contain the data to be delivered to the recipient.

TOWPin

This is the base class for all the OpenWire pins.

The overloaded function AddType shown below is very important:

```
  procedure AddType( ID : TGUID );  overload;
  procedure AddType( ID : TGUID; SubmitFunction : TOWSubmit );
      overload;
```

They define the interfaces to which the pin can send data and notifications. Whenever you define new pin capable of connecting to other pins implementing certain interface you should call this functions in order to describe how you will call the interface and which interface it is. They can take one or two parameters. The first one is the GUID of the interface you are in about to connect to. In Delphi you can use the interface name as well as shown in the sample below. In C++ Builder you can use __uuidof in order to obtain the interface GUID. The second parameter is a member function which will be called whenever the pin wants to notify or send data to the other pin. We will cover these two functions later. The second variant of the function is the one which was used most of the time in the previous versions of OpenWire. In the new versions there is a mechanism for subscribing a default submit functions when registering a new data type. OpenWire will use this function as a dispatcher if you don't specify your own dispatcher. In most of the cases it is what you want. Here is a sample of how to register a pin to be able to talk to another one implementing the IOWSomeInterface. This code will be usually part of the pins constructor:

```

Delphi Example :
  AddType(IOWSomeInterface, Notification );
  AddType(IOWSomeInterface );
```

```
C++ Builder Example :
  AddType(__uuidof(IOWSomeInterface), &Notification );
  AddType(__uuidof(IOWSomeInterface));
```

The RemoveType function:

```
procedure RemoveType( ID : TGUID );
```

Removes data type from the data types table for the pin. As result the pin will no longer be able to connect to pins implementing this type of data interface.

```
Delphi Example :
  RemoveType( IOWSomeInterface );
```

```
C++ Builder Example :
  RemoveType( __uuidof(IOWSomeInterface));
```

The ClearTypes function:

```
procedure ClearTypes();
```

Clears all the registered data type interfaces:

```
Delphi Example :
  ClearTypes;
```

```
C++ Builder Example :
  ClearTypes();
```

TOWSinkPin

The following properties and functions are specific to the TOWSinkPin :

Properties:

- SourcePin – The source pin to which the pin is connected at the moment. NIL if not connected.
- DownStreamLinkName – Returns the name of the DownStream connection in case the pin is connected. (Downstreams will be covered later)
- UpStreamLinkName – Returns the name of the UpStream connection in case the pin is connected. (Upstreams will be covered later)
- DownStreamID – Returns the GUID of the DownStream connection in case the pin is connected. (Downstreams will be covered later)
- UpStreamID – Returns the GUID of the UpStream connection in case the pin is connected. (Upstreams will be covered later)
- IgnoreUpstream – if the sink pin implements upstream it will always check whether or not the source pin you are connected to implements the stream, and if not will reject the connection. By setting this flag to true, the pin will connect without checking for the upstream support. This feature is not often used. (The upstreams and downstreams will be covered later.).

TOWMultiSinkPin

The following properties and functions are specific to the TOWMultiSinkPin :

Properties:

- SourceCount – the count of the source pins connected to the multi sink.
- Sources[] – List of Source Pins connected to the Sink Pin.
- DownStreamLinkName – Returns the name of the DownStream connection in case the pin is connected. (Downstreams will be covered later)
- UpStreamLinkName – Returns the name of the UpStream connection in case the pin is connected. (Upstreams will be covered later)
- DownStreamID – Returns the GUID of the DownStream connection in case the pin is connected. (Downstreams will be covered later)
- UpStreamID – Returns the GUID of the UpStream connection in case the pin is connected. (Upstreams will be covered later)
- IgnoreUpstream – if the sink pin implements upstream it will always check whether or not the source pin you are connected to implements the stream, and if not will reject the connection. By setting this flag to true, the pin will connect without checking for the upstream support. This feature is not often used. (The upstreams and downstreams will be covered later.).

TOWSourcePin

The following properties and functions are specific to the TOWSourcePin :

Properties :

- SinkCount – the count of the sink pins connected to the source.
- Sinks[] – List of Sink Pins connected to the Source Pin.

- FunctionSources – List of function sources. (This will be covered later.)
- DataTypeSources – List of type sources. (This will be covered later.)

```
Example : Accessing all the sink pins connected to a source pin.
Delphi Example :
  Var I       : Integer;
  Var SinkPin : TOWSinkPin;
  for I := 0 to SomeComponent.SomeSourcePin.SinkCount - 1 do
    begin
    SinkPin := SomeComponent.SomeSourcePin.Sinks[ I ];
    end;
```

```
C++ Builder Example :
  for( int i = 0; i < SomeComponent->SomeSourcePin->SinkCount; i ++ )
    {
    TOWSinkPin *SinkPin = SomeComponent->SomeSourcePin->Sinks[ i ];
    }
```

TOWStatePin

The following properties and functions are specific to the TOWStatePin:

Properties:

- PinCount – the count of pins connected via State Dispatcher.
- Pins[] – List of the pins connected via State Dispatcher.

```
Example : Accessing all the pins connected to a state pin.
Delphi Example :
  Var I    : Integer;
  Var APin : TOWPin;
  for I := 0 to SomeComponent.SomeStatePin.PinCount - 1 do
    begin
    APin := SomeComponent.SomeStatePin.Pins[ I ];
    end;
```

```
C++ Builder Example :
  for( int i = 0; i < SomeComponent->SomeStatePin->PinCount; i ++ )
    {
    TOWPin *APin = SomeComponent->SomeStatePin->Pins[ i ];
    }
```

TOWStateDispatcher

State dispatchers are created when 2 or more state pins are connected together. Source and Sink pins also can be connected via Dispatcher in some cases.

You can check if a pin is connected to a dispatcher by calling the ContainsPin function:

```
Delphi Example :
  if AState.ContainsPin( APin ) then
    // The pin is connected to AState
```

```
C++ Builder Example :
  if( AState.ContainsPin( APin ))
    // The pin is connected to AState
```

You can disconnect all the pins connected to the dispatcher and this way destroy it using the DisconnectAll procedure:

```
Delphi Example :
  AState.DisconnectAll;
```

```
C++ Builder Example :
  AState.DisconnectAll();
```

The following properties are specific to the TOWStateDispatcher:

- Name – the name of the State Dispatcher.
- PinCount – the count of the pins connected via the State Dispatcher.
- Pins[] – List of the pins connected via the State Dispatcher.

```
Example : Accessing all the pins connected to a State Dispatcher.
Delphi Example :
  Var I    : Integer;
  Var APin : TOWPin;
  for I := 0 to SomeState.PinCount - 1 do
    begin
    APin := SomeState.Pins[ I ];
    end;
```

```
C++ Builder Example :
  for( int i = 0; i < SomeState->PinCount; i ++ )
    {
    TOWPin *APin = SomeState->Pins[ i ];
    }
```

Downstreams and Upstreams

OpenWire Version 1 defines 2 types of streams – Upstreams and Downstreams. Downstream is the most often used way of transferring data among the pins. Downstream transfer occurs when a Source pin calls a member function of a sink pin interface inside the sink pins it is connected to. In order to force the Downstream to occur you simply have to call the Notify member function of a Source pin. And there should be at least one sink pin connected to the source pin. (The usage of the Notify function will be covered in details later.) Upstream event occurs, when the Notify function of a sink pin is called, and there is source pin connected to the sink pin. In some cases one pin can handle both upstream and downstream for one and the same type of data (OpenWire interface).

Change of State Broadcasting

OpenWire 2 introduces the StatePins. StatePins are using a different mechanism to exchange data (State information), called Change of State Broadcasting. When two or more StatePins are connected together, they form a group of pins synchronizing their state. The connection is maintained via hidden objects named StateDispatchers. When the state of one of the StatePins changes (the pin's Notify method has been called), the pin will send notification to each pin listed in the StateDispatcher, allowing them to reflect the change of state.

OpenWire Stream Interfaces

Each OpenWire pin can implement one or many interfaces. Some of the interfaces are defined by the OpenWire standard. They are called "Well-known interfaces" or "standard interfaces". Others can be defined by the user. Any user can submit an interface to the OpenWire development group and request the interface to be registered as a "Well-known interface". Some samples of "Well-known interfaces" include floating point data exchange interface, integer data exchange interface and some others. The current version of OpenWire defines only 6 types of "Well-known interfaces". However this code base is in about to expand rapidly.

What is an OpenWire interface? The answer is any interface inherited from IOWStream can be OpenWire interface if it is designed to be used by the pins. Here is the OpenWire definition of IOWStream:

```
  IOWStream = interface
    ['{2BFF1BE1-1698-4CFA-A427-9E0801C5B357}']
  end;
```

It's just empty interface. The following is the definition of the Floating point (Single)data exchange interface, as it is defined by OpenWire. This interface is used by any pin capable of receiving Floating point (Single) data :

```
  IOWBasicStream = interface(IOWStream)
    ['{561B072C-4191-49C6-9F22-21791EF977D9}']
    function DispatchData( DataTypeID : PDataTypeID; Operation :
IOWNotifyOperation; State : TOWNotifyState ) : TOWNotifyResult; stdcall;
  end;

  IOWDataStream = interface(IOWBasicStream)
    ['{CFDF94D7-5134-49D9-AC65-902BBC1CD140}']
  end;

  IOWFloatStream = interface(IOWDataStream)
    ['{67F6997B-7EB4-4E2F-8320-4A512B5F2BC7}']
  end;
```

The only necessary function is the one that will receive the data. You can define your own interfaces by selecting a new GUID and adding functions defined by you.

After creating the interface, you can decide to register a name and default dispatcher for the interface. We will discuss the dispatchers in detail later. The best place for the registration is the initialization section of the package file.

WARNING ! The default dispatcher must be available both in Design and Run time. DO NOT REGISTER THE DISPATCHER IN THE "Register" FUNCTION ! If you do so, the dispatcher will be available at design time only ! Do it in the initialization section.

Here is a sample of how you can do so in Delphi :

```
initialization
  OWRegisterDefaultHandler(IOWFloatStream,
                                    OWDefaultFloatNotificationHandler );
```

In C++ Builder you can use the #pragma startup or you can declare a fake global object and do the registration in the constructor :

```
#pragma startup MyOWRegistration;
void MyOWRegistration (void)
{
  OWRegisterDefaultHandler(IOWFloatStream,
                                    OWDefaultFloatNotificationHandler );
}
```

or

```
class TMyOWRegistrationClass
{
public:
  TMyOWRegistrationClass()
    {
    OWRegisterDefaultHandler(IOWFloatStream,
                                    OWDefaultFloatNotificationHandler );
    }
};

TMyOWRegistrationClass MyOWRegistration;
```

Here is a sample of OWDefaultFloatNotificationHandler implementation:

```
function OWDefaultFloatNotificationHandler( Sender : TOWPin; Handler :
IOWStream; DataTypeID : PDataTypeID; Operation : IOWNotifyOperation;
State : TOWNotifyState ) : TOWNotifyResult;
var
  Interf : IOWFloatStream;

begin
    Result := [];
    if( Handler.QueryInterface( IOWFloatStream, Interf ) = 0 ) then
      Result := Interf.DispatchData( DataTypeID, Operation, State );

end;
```

The dispatcher (Notifier) just calls the DispatchData method of the interface.

You can use the interface for both Upstream and Downstream. In most cases you will use it for Downstream only. In this case first you have to declare a pin inherited from TOWSinkPin or one of its descendants and then to implement the interface in the pin. You can use this interface for State Broadcasting as well. The OpenWire provides a set of basic pin types to handle some of the “Well-known interfaces”. Here is how OpenWire has defined the TOWFloatSinkPin pin:

```
type TOWDispatchEvent = procedure( DataTypeID : PDataTypeID;
       Operation : IOWNotifyOperation; CustomData : TObject ) of object;

type TOWFloatChangeEvent = procedure( Sender : TOWPin; AValue : Single )
of object;

TOWFloatSinkPin = class( TOWSinkPin, IOWFloatStream )
protected
FCustomData     : TObject;
FDispatchEvent : TOWDispatchEvent;
FOnDataChange   : TOWFloatChangeEvent;
FValue          : Single;

public
function DispatchData( DataTypeID : PDataTypeID; Operation :
    IOWNotifyOperation; State : TOWNotifyState ) :
        TOWNotifyResult; stdcall;

public
  constructor CreateEx( AOwner: TComponent; ADispatchEvent :
        TOWDispatchEvent; ACustomData : TObject = NIL );

  constructor Create( AOwner: TComponent; AOnDataChange :
        TOWFloatChangeEvent; ACustomData : TObject = NIL );

public
  property Value : Single read FValue;

end;
```

Everything that has to be done is implementing the DispatchData function for the IOWFloatStream interface. Here is the function as it is done in OpenWire:

```
function TOWFloatSinkPin.DispatchData( DataTypeID : PDataTypeID;
Operation : IOWNotifyOperation; State : TOWNotifyState ) :
TOWNotifyResult;  stdcall;
begin
  if( Operation.Instance() is TOWSuppliedSingleOperation ) then
    begin
    FValue := TOWSuppliedSingleOperation( Operation.Instance() ).Value;
```

```
      if( Assigned( FOnDataChange )) then
        FOnDataChange( Self, FValue );

    end;

  if( Assigned( FDispatchEvent )) then
    FDispatchEvent( DataTypeID, Operation, FCustomData );

  Result := [];
end;
```

This code is a bit more complex in order to make the pin easier for using. Here is a simplified version of the same code showing the bare minimum code needed:

```
function TOWFloatSinkPin.DispatchData( DataTypeID : PDataTypeID;
Operation : IOWNotifyOperation; State : TOWNotifyState ) :
TOWNotifyResult;  stdcall;
begin
  if( Assigned( FDispatchEvent )) then
    FDispatchEvent( DataTypeID, Operation, FCustomData );

  Result := [];
end;
```

The only code inside is used just to call the `FDispatchEvent` in case the event is assigned. We will see a sample of component using this pin type later.

Second you have to define a source pin capable of sending data or events through the interface. OpenWire defines such pin as follows:

```

TOWFloatSourcePin = class( TOWSourcePin )

  public
    constructor Create( AOwner: TComponent );

end;

constructor TOWFloatSourcePin.Create( AOwner: TComponent );
begin
  AddType( IOWFloatStream );
end;
```

For simplicity in this example the clocking support is omitted. This is enough. In this case the pin will use the default notifier (dispatcher) for the IOWFloatStream interface. You can also declare your own dispatcher:

```
type TOWFloatPinNotificationEvent = function (Handler : IOWFloatStream;
      Operation : IOWNotifyOperation; State : TOWNotifyState
      ) : TOWNotifyResult of object;


TOWExFloatSourcePin = class(TOWSourcePin )

  protected
    PinNotificationEvent         : TOWFloatPinNotificationEvent;
    FValue                       : Single;

  public
    constructor Create( AOwner: TComponent;
APinNotificationEvent : TOWFloatPinNotificationEvent );
```

```
  public
    function Notification( Handler : IOWStream;
      Operation : IOWNotifyOperation; State : TOWNotifyState ) :
  TOWNotifyResult; virtual;

end;

constructor TOWExFloatSourcePin.Create( AOwner: TComponent );
begin
  AddType( IOWFloatStream, Notification );
end;
```

The only new function here is the Notification function. As we will see later this function will be registered to be called for each sink pin connected to the source and registered to be handled through the IOWFloatStream interface. Here is how OpenWire implements the function:

```
function TOWExFloatSourcePin.Notification( Handler : IOWStream;
Operation : IOWNotifyOperation; State : TOWNotifyState ) :
TOWNotifyResult;
var
  Interf : IOWFloatStream;

begin
  Result := [];

  if( Handler.QueryInterface(IOWFloatStream,Interf) = 0 ) then
    begin
    if( Assigned( PinNotificationEvent )) then
       begin
       Result := PinNotificationEvent( Interf, DataTypeID, Operation,
State );
       Exit;
       end;

    if( nsNewLink in State ) then
      begin
      Interf.DispatchData( DataTypeID,
TOWSuppliedSingleOperation.Create( FValue ), State );
      Exit;
      end;

    Interf.DispatchData( DataTypeID, Operation, State );
    end;
 end;
```

In the function the Interf obtains interface of type IOWFloatStream from the Handler and calls the PinNotificationEvent the event usually points to a handler inside the component using the pin. There most often you will just call the DispatchData method with the value you want to pass to the sink pin. You will see sample of how to do that later. If the event is not assigned, then the DispatchData method will be called directly. If the Notification function has been called because a new connection has been established, the current FValue is wrapped in a TOWSuppliedSingleOperation object and the DispatchData is called.

Connecting and Handshaking

Handshaking is the process of 2 pins connecting to each other. 2 pins can connect to each other in case the following conditions are met:

1. One of the pins should be TOWSinkPin or TOWStatePin descendant and the other should be TOWSourcePin or TOWStatePin descendant.

2. One of the pins should implement an interface, which the other knows how to deal with. (The interfaces will be covered later.)
3. The two pins should not be in functional dependency of each other – as example if the source pin has been calculated as result of the sink pin. In this case the source pin can't connect to the sink pin for an obvious reason. (Functional dependency will be covered in detail later)
4. In case of a long chain of pins connected in type dependency, the entire chain might be restricted to use a homogeneous interface. In this relatively rare case all of the pins will have to have at least one common interface. Otherwise the connection will fail. (The type dependency will be described in details later.)

In order the two pins to connect to each other one of them should implement an interface the other knows how to deal with. Here is a sample of pin implementing an interface.

```
//
// Defining an interface.
IOWSampleStreamInterface = interface(IOWStream)
    ['{DC6A8530-E1C9-4F6F-AB8B-1242C391CF79}']
    procedure MyFunction ( Data : Single ); stdcall;
  end;

//
// Defining a sink pin implementing IOWSampleStreamInterface.
TOWSampleSinkPin = class( TOWSinkPin, IOWSampleStreamInterface )

  public
    procedure MyFunction ( Data : Single ); stdcall;

  end;
```

Next you have to create a pin capable to call functions inside the `IOWSampleStreamInterface`:

```

//
// Defining a source pin capable of calling functions
// inside IOWSampleStreamInterface.
TOWSampleSourcePin = class(TOWSourcePin )
  public
    constructor Create( AOwner: TComponent );

  public
    function Notification( Handler : IOWStream; Operation :
IOWNotifyOperation; State : TOWNotifyState ) :
TOWNotifyResult; virtual;

  end;

constructor TOWSampleSourcePin.Create( AOwner: Tcomponent );
begin
  inherited;

  // Register the Notification function to be called
  // for IOWSampleStreamInterface.
  AddType( IOWSampleStreamInterface, Notification );
end;
```

We will cover the implementation of the Notification function later. For now this is enough for both pins to be able to connect to each other.

One and the same pin can implement multiple interfaces. As example:

```
//
// Defining the interfaces.
```

```
IOWSampleStreamInterface1 = interface(IOWStream)
    ['{3634EB24-2ADD-49E7-BBF9-870A81BA3192}']
    procedure MyFunction1 ( Data : Single ); stdcall;
  end;

IOWSampleStreamInterface2 = interface(IOWStream)
    ['{3D4C0921-B05D-4D6B-A90C-662F8AE61C97}']
    procedure MyFunction2 ( Data : Single ); stdcall;
  end;

//
// Defining a sink pin implementing IOWSampleStreamInterface1,
// and IOWSampleStreamInterface 2.
TOWSampleSinkPin = class( TOWSinkPin,
IOWSampleStreamInterface1, IOWSampleStreamInterface2 )

  public
    procedure MyNotificationFunction1 ( Data : Single ); stdcall;
    procedure MyNotificationFunction2 ( Data : Single ); stdcall;

end;
```

You can also register a pin to be able to call functions inside multiple interfaces:

```
constructor TOWSampleSourcePin.Create( AOwner: Tcomponent );
begin
  inherited;

  AddType( IOWSampleStreamInterface1, MyNotificationFunction1 );
  AddType( IOWSampleStreamInterface2, MyNotificationFunction2 );
end;
```

In most of the cases the notification function will be different for each of the interfaces, but it is not a strict rule.

We are saying that two pins are connected when they agree on the interface that they will use to talk to each other. They can connect in each direction (upstream and downstream) using one and only one interface per direction. Once connected you can check the connection name in each direction through the DownStreamLinkName and the UpStreamLinkName properties of the TOWSinkPin. Each direction may use a different interface.

When connecting the pins, during the handshaking, the pins will negotiate the interface they are in about to use. The negotiation mechanism is very simple. The last pin type in the list of interfaces added through the AddType function, which matches one of the interfaces implemented, by the other pin, will be the pin type used for data exchange. The list of interfaces is always processed from the last to the first. In case you are testing with the two pins shown above – the result would be IOWSampleStreamInterface2. If the sink pin didn't implement the IOWSampleStreamInterface2, then the result would be IOWSampleStreamInterface1. If the last AddType was missing in the constructor for the source component, the connection would be established through IOWSampleStreamInterface1 as well. There is only one exception of that rule and it is in case there are type dependencies inside one of the components. This case is relatively rare. For most cases you can ignore it completely.

A Sink Pin can be registered to be function and/or type source for a certain source pin. In case the pin is a function source, it means that the value of the source pin is calculated inside the component from the value received from the sink pin (the source pin value is function of the sink pin value). In this case the source pin will not be able to send data to the sink pin, and the connection will be restricted. If the source pin is connected to a sink pin inside another component and this sink pin is a function source for a source pin in the other component, this other components source pin will not be able to connect to the sink pin of the first component for the same reason. This means that the function source restriction is propagated among the entire chain of sources and sinks. The type dependency is used in rare occasions. If a certain source pin is capable to connect to sink pins through different interfaces and has a type dependency on one or more sink pins inside the same component, it will restrict the interfaces it is capable of supporting in order to make sure the connection will be established using the same type interface through the entire type

chain. It may result in reconnecting already connected sink and source pins in the chain in order to ensure the homogenous connections. This mechanism can be ignored in most cases.

Standard (Well-known) Interfaces and Standard Pin Types

OpenWire defines a set of standard – called "Well-known" interfaces. The current version of OpenWire defines only 6 "Well-known interfaces" – IOWIntegerStream, IOWFloatStream, IOWRealStream, IOWBoolStream, IOWCharStream, IOWStringStream. OpenWire provides ready-to-use pins for those interfaces. The "Well-known interfaces" is a growing code base of interfaces available for usage by the developers. Whenever a developer needs to use a pin with certain capabilities – a good idea is to check whether or not a pin with such capabilities already exists, and if so to use the existing one. If not there might be a "Well-known interface" which could be used. Only if you can't find neither ready pin, nor "Well-known interface", to satisfy your needs, you should develop your own. How to develop and register your own interfaces will be covered later. Once developed and tested, if you think the interface could be used by others to exchange data, you can submit it to the OpenWire development team and if accepted it will appear in the "Well-known interfaces" code base. Information on how to submit your interfaces will be available on the OpenWire web site: www.openwire.org .

Clocking

OpenWire 2.1 introduces a standard way of providing clock for any OpenWire pin. The clocking is design for Downstream and is implemented via standard interface IOWClockStream.

Here is the clocking interface declaration:

```
  IOWClockStream = interface(IOWBasicStream)
    ['{48CDAF9F-00C7-4B45-999D-4EE25353A952}']
  end;
```

Here is the declaration, and the implementation of the newly introduced TOWClockSourcePin :

```
type
  TOWClockSourcePin = class( TOWSourcePin )

  protected
    function ClockNotification( Handler : IOWStream; DataTypeID :
PDataTypeID; Operation : IOWNotifyOperation; State : TOWNotifyState ) :
TOWNotifyResult; virtual;

  public
    procedure Clock();

  public
    constructor Create(AOwner: TComponent);

end;

constructor TOWClockSourcePin.Create(AOwner: TComponent);
begin
  inherited;
  AddType( IOWClockStream, ClockNotification );
end;

procedure TOWClockSourcePin.Clock();
begin
  Notify( TOWClockOperation.Create() );
end;

function TOWClockSourcePin.ClockNotification( Handler : IOWStream;
DataTypeID : PDataTypeID; Operation : IOWNotifyOperation; State :
TOWNotifyState ) : TOWNotifyResult;
var
```

```
  Interf : IOWClockStream;

begin
  Result := [];

  if( Handler.QueryInterface( IOWClockStream, Interf ) = 0 ) then
    Interf.DispatchData( DataTypeID, Operation, State );

end;
```

Because in OpenWire version 2.1 and up all the standard source pins are inherited from TOWClockSourcePin they are naturally capable of providing clock events.

Creating Components Using the Standard Interfaces and Pins

Using the Standard Source and Sink Pins In Your Components

Creating components using the standard already provided by OpenWire pins is very easy. It is matter of just few lines of code and pin support can be added to any existing component with very little effort. Here are few examples.

In our first example we will add sink pin support to a TLabel component, making it capable to display the received data. We will make it support only floating point data for now. (I.E. only IOWFloatStream will be implemented.)

Here is how we will declare our component inside the interface section in Delphi:

```
TDemoLabel = class(TLabel)
  protected
    FInputPin: TOWFloatSinkPin;

  protected
    procedure PinValueChange( Sender : TOWPin; AValue : Single );

  public
    constructor Create(AOwner: TComponent); override;
    destructor  Destroy; override;

  published
    property InputPin : TOWFloatSinkPin   read FInputPin write
FInputPin;
  end;
```

So the first thing we should do is to declare member variable FInputPin as TOWFloatSinkPin. Then we will declare a property to use this variable. Finally we will add the PinValueChange procedure. That is all you have to do in the declaration! As easy as adding a new property, right? But may be the catch is in the implementation then? Probably there is a whole bunch of code to be added there! Let’s see the implementation of the constructor, the destructor and the PinValueChange procedure:

```

constructor TDemoLabel.Create(AOwner: TComponent);
begin
  inherited;
  FInputPin := TOWFloatSinkPin.Create( Self, PinValueChange );
end;

destructor TDemoLabel.Destroy;
begin
  FInputPin.Free;
  inherited;
```

```
end;

procedure TDemoLabel.PinValueChange( Sender : TOWPin; AValue : Single );
begin
  Caption :=  FloatToStr( AValue );
end;
```

Our first component using OpenWire is ready! Let's take a look at each of the functions:

The constructor just creates a new pin, passing pointer to the component and the object address (__closure) of the PinValueChange procedure.

The destructor just deletes the pin.

The PinValueChange procedure is the most important one. This function actually receives the data from the pin and shows it in the TLabel caption.

Congratulations! You just learned how to create components having sink pins, using the already existed pins in OpenWire. You will learn how to implement your own pins later. Now let's see how difficult it is to create a component which provides data – i.e. has source pin.

To show how to use source pins inside component, we will create a timer component using TOWFloatSourcePin, and being able to connect and stream data to our TDemoLabel. First we will make the component being able to connect with the TLabel, then we will implement some real functionality inside – I.E. will make the component generate some increasing numbers from 1 – 100 with step 0.5. Then we will test the components together and see the data displayed inside our TDemoLabel.

Here is the code necessary to create and support the source pin. Indeed it looks almost the same as for the sink pin.

```
// TDemoTimer - basic version.
TDemoTimer = class(TTimer)
  protected
    FOutputPin  : TOWFloatSourcePin;

  public
    constructor Create(AOwner: TComponent); override;
    destructor  Destroy; override;

  published
    property OutputPin : TOWFloatSourcePin   read FOutputPin write
FOutputPin;
  end;
```

So as in our first sample, first we add member variable FOutputPin of type TOWFloatSourcePin. Then add a property to access the variable. Finally add the FloatPinNotification method.

Now let's focus on the implementation.

```

constructor TDemoTimer.Create(AOwner: TComponent);
begin
  inherited;
  FOutputPin := TOWFloatSourcePin.Create( Self );
end;

destructor TDemoTimer.Destroy;
begin
  FOutputPin.Free;
  inherited;
end;
```

That is all you need to do, in order to have a Source pin inside your component. Easy right? Now you can compile and place your two components on any form and connect them together. You can even drop multiple TDemoLabel components and connect all of them to your TDemoTimer. There is only one thing left to do. We have everything we need to connect the components, but we don't send any data. You have seen how easy was everything up to now, so it will be no surprise to learn that the only thing you need is one line of code:

```
FOutputPin.Value = FCounter;
```

In this case FCounter is a variable of type Single (floating point). The right place to put the Notify function obviously is inside OnTimer event or the Timer member procedure of the TTimer. Before we continue with the pins, we will have to say few words about the Delphi / C++ Builder TTimer component itself. There are few little details in the way the TTimer works. We don't want to use the OnTimer, because somebody may decide to assign another procedure to it. The obvious choice is the Timer member procedure. There is a hidden problem however. Borland optimized the timer to call the Timer procedure only if an OnTimer event is assigned. Whether bug or a feature this will make our lives more difficult. We will see a work around the problem just after we review the code to support the pins. So for now let's focus on the Timer procedure:

```
procedure TDemoTimer.Timer;
begin
  FOutputPin.Value := FCounter;

  FCounter := FCounter + 0.5;

  if( FCounter > 100 ) then
    FCounter := 0;

  inherited Timer;
end;
```

If it were not for the TTimer "feature" mentioned above we would have been almost done. The only thing left to be done would be just adding FCounter as a member variable to our class and assigning it to 1 in the constructor. Although it is out of the OpenWire topic we will show the necessary code to complete the TTimer:

```
TDemoTimer = class(TTimer)
  protected
    FOutputPin : TOWFloatSourcePin;
    FCounter   : Single;

  protected
    procedure Timer; override;
    procedure Loaded; override;

    procedure STestClockComponentTimer( Sender : TObject );

  public
    constructor Create(AOwner: TComponent); override;
    destructor  Destroy; override;

  published
    property OutputPin : TOWFloatSourcePin   read FOutputPin write
FOutputPin;
  end;

implementation

constructor TDemoTimer.Create(AOwner: TComponent);
begin
  inherited;
```

```
  FOutput := TOWFloatSourcePin.Create( Self );
  FCounter := 1;
  FOutputPin.Value := 1;  // Let's make the pin value the same as
FCounter.
end;

destructor TDemoTimer.Destroy;
begin
  FOutputPin.Free;
  inherited;
end;

procedure TDemoTimer.Loaded;
begin
  if( ( not Assigned( OnTimer ) ) and
( not ( csDesigning in ComponentState ) )) then
    OnTimer := STestClockComponentTimer;

end;

procedure TDemoTimer.Timer;
begin
  FOutputPin.Value := FCounter;

  FCounter := FCounter + 0.5;

  if( FCounter > 100 ) then
    FCounter := 0;

  inherited Timer;
end;

procedure TDemoTimer.STestClockComponentTimer( Sender : TObject );
begin
end;
```

The Loaded and the STestClockComponentTimer empty procedures have nothing to do with the OpenWire or our pins. They have been added just to work around the Borlands TTimer "feature". However now our component is completed and can be tested.

Congratulations! You have created your first two components using OpenWire. Now they can be placed on a form or TDtataModule and connected to each other and they will start exchanging data.

Using the Standard State Pins In Your Components

OpenWire 2 introduces a new type of pins – the State Pins. They are designed to allow multiple components to share a common state – Enabled/Disabled, Up/Down etc. Those pins are using a mechanism known as "Change of State Broadcasting". You can use the standard interfaces with state pins. OpenWire also provides some ready to use pins implementing the standard interfaces(data types).

In our example we will implement a TTrackBar descendant component using the standard TOWIntegerStatePin.

Here is how we will declare our component inside the interface section in Delphi:

```
  TDemoTrackBar = class( TTrackBar )
  protected
    FPositionPin : TOWIntegerStatePin;

  public
    constructor Create(AOwner: TComponent); override;
    destructor  Destroy; override;
```

```
  protected
    procedure Changed; override;

  protected
    procedure OnIntegerChangeEvent( Sender : TOWPin; AValue : Integer );

  published
    property PositionPin : TOWIntegerStatePin read FPositionPin write
FPositionPin;

  end;
```

The first thing we should do is to declare member variable FPositionPin as TOWIntegerStatePin. Then we will declare a property to use this variable. Then we will add the OnIntegerChangeEvent procedure. Finally we will override the Changed procedure in order to notify the pin about the changes in the component state. That is all you have to do in the declaration. Here is the implementation of the constructor, the destructor, the OnIntegerChangeEvent procedure, and the Changed procedure:

```
constructor TDemoTrackBar.Create(AOwner: TComponent);
begin
  inherited;
  FPositionPin := TOWIntegerStatePin.Create( Self,
OnIntegerChangeEvent );

end;

destructor  TDemoTrackBar.Destroy;
begin
  FPositionPin.Free();
  inherited;
end;

procedure TDemoTrackBar.OnIntegerChangeEvent( Sender : TOWPin; AValue :
Integer );
begin
  Position := AValue;
end;

procedure TDemoTrackBar.Changed;
begin
  inherited;
  FPositionPin.Value := Position;
end;
```

This is all, the component needs in order to support StatePins.

The constructor just creates a new State Pin, passing pointer to the component and the object address (__closure) of the OnIntegerChangeEvent procedure.

The destructor just deletes the pin.

The OnIntegerChangeEvent procedure is the most important one. This function actually receives the data from the pin and assigns the Value to the TrackBar's Position.

Finally inside the Changed procedure when a change of the position occurs, the FpositionPin gets notified, by assigning the Position to the FPositionPin.Value.

How the Notify Really Works

Looking at the above components it's kind of difficult to understand how the Notify will send the data to the TDemoLabel component as example. Here is how it works. Remember those AddType procedures we were discussing before? That is where the magic is. IOWFloatStream inherits from IOWDataStream. Here is the code for the default dispatcher for the IOWDataStream as it is implemented in OpenWire:

```
function OWDefaultDataNotificationHandler( Sender : TOWPin; Handler :
IOWStream; DataTypeID : PDataTypeID; Operation : IOWNotifyOperation;
State : TOWNotifyState ) : TOWNotifyResult;
var
  Interf : IOWDataStream;

begin
  Result := [];
  if( Handler.QueryInterface( IOWDataStream, Interf ) = 0 ) then
    Result := Interf.DispatchData( DataTypeID, Operation, State );

end;
```

This dispatcher will be called when a pin connected to another pin implementing IOWDataStream interface attempts to send data, and it has not registered its own dispatcher. In case the Source Pin sending the data has its own dispatcher, the default one will not be used. Here is a sample of the timer component using its own dispatcher:

```
TDemoTimer = class(TTimer)
  protected
    FOutput  : TOWExFloatSourcePin;
    FCounter : Single;

  protected
    procedure Timer; override;
    procedure Loaded; override;

    procedure STestClockComponentTimer( Sender : TObject );

  protected
    function FloatPinNotification( Handler : IOWFloatStream; Operation :
IOWNotifyOperation; State : TOWNotifyState ) :
TOWNotifyResult;

  public
    constructor Create(AOwner: TComponent); override;
    destructor  Destroy; override;

  published
    property Output : TOWExFloatSourcePin read FOutput write FOutput;
  end;

constructor TDemoTimer.Create(AOwner: TComponent);
begin
  inherited;
  FOutput := TOWExFloatSourcePin.Create( Self, FloatPinNotification );
  FCounter := 1;
end;

function TDemoTimer.FloatPinNotification( Handler : IOWFloatStream;
Operation : IOWNotifyOperation; State : TOWNotifyState ) :
TOWNotifyResult;
begin
  Result := [];
if( Operation.Instance() is TOWSuppliedOperation ) then
```

```
    Result := Handler.DispatchData( DataTypeID, Operation, State );

end;
```

Let's look at the code for the TOWExFloatSourcePin constructor and the Notification function:

```
constructor TOWExFloatSourcePin.Create( AOwner: TComponent;
APinNotificationEvent : TOWFloatPinNotificationEvent );
begin
  inherited Create( AOwner );
  PinNotificationEvent := APinNotificationEvent;

  AddType( IOWFloatStream, Notification );
end;

function TOWExFloatSourcePin.Notification( Handler : IOWStream;
Operation : IOWNotifyOperation; State : TOWNotifyState ) :
TOWNotifyResult;
var
  Interf : IOWFloatStream;

begin
  if( Assigned( PinNotificationEvent )) then
    if( Handler.QueryInterface(IOWFloatStream,Interf) = 0 ) then
       begin
       Result := PinNotificationEvent( Interf, Operation, State );
       Exit;
       end;

  Result := [];
end;
```

The AddType defines that when you call Notify for the Source pin, for any Sink pin implementing the IOWFloatStream a function named Notification should be called. The Notification function itself, calls PinNotificationEvent, and as you remember, we passed our component PinNotification as a parameter to the pin constructor, which assigns it to the PinNotificationEvent. So in other words for each sink pin implementing the IOWFloatStream interface the FloatPinNotification will be called. So if we have two TDemoLabel components connected to our TDemoTimer, and we call the Notify function, for each of them OpenWire will call FloatPinNotification, passing the IOWFloatStream interface to their sink pin as argument. This type of event is called Downstream event. In this case the Notify command has been called for the Source pin. If the IOWFloatStream was implemented for the source pin and the sink pin had registered by AddType function handlers to send data to the IOWFloatStream interface, then the pins would be capable of Upstream events, and you would be able to use Notify inside the Sink pin. The only difference is that in this case the event gets send to only one pin – the source pin, so far only one source pin can be connected to a sink pin. The IOWFloatStream is not designed for upstream events, and although they are possible to implement, they would have no value. There are however some interfaces in which upstream events are not only meaningful, but also necessary for the proper behavior of the interface.

There are some other ways of implementing source pin notification than the implementation shown above.

Let see again the FloatPinNotification handler:

```
function TDemoTimer.FloatPinNotification( Handler : IOWFloatStream;
Operation : IOWNotifyOperation; State : TOWNotifyState ) :
TOWNotifyResult;
begin
  Result := [];
if( Operation.Instance() is TOWSuppliedOperation ) then
    Result := Handler.DispatchData( DataTypeID, Operation, State );

end;
```

The Operation parameter is used to pass notification objects Down or Upstream. There are many notification objects, which could be send, and they depend on the particular interface. The most important one for floating point data is TOWSuppliedOperation. This object contains the data to be sent to the sink pins. The “if” statement checks if a new data has been sent I.E. if Operation is a TOWSuppliedOperation. The State parameter can be used to determine if the notification is called because a new connection is established to the Sink Pin. In this case State will contain nsNewLink and by checking for it you can add some code to handle this condition. Another valuable information received through the State is the nsLastIteration state. It gets send when the FloatPinNotification is called for the last pin in the list of pins connected to the source. It allows a performance optimizations to be done in this case. As example after this moment the Data will not be needed and can be changed or released. This feature is very valuable for ultra fast data processing application, but the topic goes beyond the scope of this document. The state plays some role for the dynamic order balancing as well, resulting in further speed improvements. The dynamic order balancing will be covered later.

Because we know that from the pin we always will send the FCounter we can simplify our design and also get the benefit of the nsNewLink state notification. Our new FloatPinNotification will look like this:

```
function TDemoTimer.FloatPinNotification( Handler : IOWFloatStream;
Operation : IOWNotifyOperation; State : TOWNotifyState ) :
TOWNotifyResult;
begin
  Result := Handler.DispatchData( DataTypeID,
TOWRequestedSingleOperation.Create( Fcounter ), State );

end;
```

and then our Notify function will not use the passed parameters and we can replace it with:

```
FOutput.Notify( IOWNotifyOperation.Create() );
```

Then our timer function will be:

```
procedure TDemoTimer.Timer;
begin
  FOutput.Notify( IOWNotifyOperation.Create() );

  FCounter := FCounter + 0.5;

  if( FCounter > 100 ) then
    FCounter := 1;

  inherited Timer;
end;
```

This approach is much simpler and was used often in OpenWire 1. OpenWire 2 introduced improved messaging system, and easier pin synchronization. This method is rarely needed in OpenWire 2.

Whenever you are satisfied with the default dispatcher for the data type, you should use it. The code of your components will be much smaller, and easier to understand.

When State pins and State Dispatchers are involved, the scenario is similar. One of the State pins will send a Notify Message Object. The Message Object will be delivered by the default or custom Notify Dispatchers to all the pins connected to the State Dispatcher. The code is almost identical to the code above.

Creating And Using Pins Implementing The Standard Interfaces

Now after we learned how to use the standard pins provided by OpenWire, it is time to try to create our own pins, capable of supporting multiple interfaces. Then we will modify the components from the last sample to use the just created pins.

We will start our work with a sink pin, and as a first step will make it accepting only one type of data (one interface) – the IOWFloatStream interface. Then we will add another interface. Here is how you declare a new pin implementing certain interface:

```
// Event type declaration
type TOWDemoFloatChangeEvent = procedure( Sender : TOWPin; AValue :
Single ) of object;

// Pin declaration
TOWDemoSinkPin = class( TOWSinkPin, IOWFloatStream )

  Protected
    FOnDataChange   : TOWDemoFloatChangeEvent;
    FValue          : Single;

  public
    function DispatchData( DataTypeID : PDataTypeID; Operation :
IOWNotifyOperation; State : TOWNotifyState ) : TOWNotifyResult; stdcall;

  public
    constructor Create( AOwner: TComponent; AOnDataChange :
TOWDemoFloatChangeEvent );

  end;
```

Now we will write the code for the constructor and the DispatchData method.

```
constructor TOWDemoSinkPin.Create( AOwner: TComponent; AOnDataChange :
TOWFloatChangeEvent );
begin
  inherited Create( AOwner );
  FOnDataChange := AOnDataChange;
end;

function TOWDemoSinkPin.DispatchData( DataTypeID : PDataTypeID;
Operation : IOWNotifyOperation; State : TOWNotifyState ) :
TOWNotifyResult;  stdcall;
begin
  if( Operation.Instance() is TOWSuppliedSingleOperation ) then
    begin
    FValue := TOWSuppliedSingleOperation( Operation.Instance() ).Value;
    if( Assigned( FOnDataChange )) then
      FOnDataChange( Self, FValue );

    end;

  Result := [];
end;
```

Our pin is ready to go. We can replace the TOWFloatSinkPin in out TLabel with TOWDemoSinkPin and the TLabel will still work the way it was before. But before we do that let see how difficult it is to add support for integer data to the same pin.

```

// Event type declarations
type TOWDemoFloatChangeEvent = procedure( Sender : TOWPin; AValue :
Single ) of object;

// Pin declaration
TOWDemoSinkPin = class( TOWSinkPin, IOWFloatStream, IOWIntegerStream )

  protected
```

```
    FOnFloatDataChange    : TOWDemoFloatChangeEvent
    FValue                : Single;

  public
    function DispatchData( DataTypeID : PDataTypeID; Operation :
IOWNotifyOperation; State : TOWNotifyState ) : TOWNotifyResult; stdcall;

  public
    constructor Create( AOwner: TComponent;
AOnFloatDataChange : TOWDemoFloatChangeEvent );

  end;
```

Here is how you would implement the constructor and the SendIntegerData method:

```
constructor TOWDemoSinkPin.Create( AOwner: TComponent;
AOnFloatDataChange : TOWDemoFloatChangeEvent );
begin
  inherited Create( AOwner );
  FOnFloatDataChange := AOnFloatDataChange;
end;

function TOWDemoSinkPin.DispatchData( DataTypeID : PDataTypeID;
Operation : IOWNotifyOperation; State : TOWNotifyState ) :
TOWNotifyResult;  stdcall;
begin
  if( Operation.Instance() is TOWSuppliedSingleOperation ) then
    begin
    FValue := TOWSuppliedSingleOperation( Operation.Instance() ).Value;
    if( Assigned(FOnFloatDataChange)) then
      FOnFloatDataChange( Self, FValue );

    end

  else if ( Operation.Instance() is TOWSuppliedIntegerOperation ) then
    begin
    FValue := TOWSuppliedIntegerOperation( Operation.Instance() ).Value;
    if( Assigned(FOnFloatDataChange)) then
      FOnFloatDataChange( Self, FValue );

    end;


  Result := [];
end;
```

Please note that the suggested mechanism differs than the one suggested in OpenWire 1. This one is much cleaner and easier to use. Let see now our TLabel using the new pin.

```
TDemoLabel = class(TLabel)
  protected
    FInput   : TOWDemoSinkPin;

  protected
    procedure OnDataChangedEvent(Sender : TOWPin; AValue : Single );

  public
    constructor Create(AOwner: TComponent); override;
    destructor  Destroy; override;

  published
```

```
    property Input : TOWDemoSinkPin read FInput write FInput;
  end;

constructor TDemoLabel.Create(AOwner: TComponent);
begin
  inherited;
  FInput := TOWDemoSinkPin.Create( Self, OnDataChangedEvent );

end;

destructor TDemoLabel.Destroy;
begin
  FInput.Free;
  inherited;
end;

procedure TDemoLabel.OnDataChangedEvent(Sender : TOWPin; AValue : Single
);
begin
  Caption := FloatToStr( Data );
end;
```

Now our component can accept both single (floating point) and integer data, through different interfaces. Please notice that the code is much shorter than in OpenWire 1. Just few lines of code and we have added new power to our component. Now you can compile and test the just created component together with the already existed TDemoTimer component. They should work together without any problems.

Now it's time to create a source pin capable of sending 2 different types of data. Then we will use our timer component to test the pin.

As first step we will make the pin supporting only floating point data. Then we will add Integer data support as well. Here is how we will declare the pin:

```
TOWDemoSourcePin = class(TOWSourcePin)
  protected
    FValue               : Single;

  protected
    function Notification( Handler : IOWStream; DataTypeID :
PDataTypeID; Operation : IOWNotifyOperation; State : TOWNotifyState ) :
TOWNotifyResult; virtual;

  protected
    procedure SetValue( AValue : Single );

  public
    constructor Create( AOwner: TComponent );

  public
    property Value : Single read FValue write SetValue;

end;
```

Please notice that the code is different than the one suggested in the OpenWire 1. Here is the implementation for the constructor and the FloatNotification function:

```
constructor TOWDemoSourcePin.Create( AOwner: TComponent);
begin
  inherited Create( AOwner );
  AddType( IOWFloatStream, Notification );
end;
```

```
procedure TOWDemoSourcePin.SetValue( AValue : Single );
begin
  if( FValue <> AValue ) then
    begin
    FValue := AValue;
    Notify( TOWSuppliedSingleOperation.Create( FValue ));
    end;

end;

function TOWDemoSourcePin.Notification( Handler : IOWStream;
DataTypeID : PDataTypeID; Operation : IOWNotifyOperation; State :
TOWNotifyState ) : TOWNotifyResult;
var
  Interf : IOWFloatStream;

begin
  Result := [];

  if( Handler.QueryInterface(IOWFloatStream,Interf) = 0 ) then
    begin
    if( nsNewLink in State ) then
      begin
      Interf.DispatchData( DataTypeID,
TOWSuppliedSingleOperation.Create( Value ), State );
      Exit;
      end;

    Interf.DispatchData( DataTypeID, Operation, State );
    end;

end;
```

With the AddType call in the constructor, we are registering the Notification to be called for the floating-point data interface the pin will call the Notification function.

The SetValue procedure is added to simplify the usage of the pin. It will check if the current value of the pin differs than the new one assigned to the Value property, and if so will send a notification with the new value.

The Notification first attempts to obtain an IOWFloatStream interface to the Handler it receives from OpenWire. If it succeeds it checks to see if the notification is because a new connection has been established. If so it sends the current pin FValue to the connected Sink Pin via the IOWFloatStream interface. Otherwise it sends whatever Operation has been received by the Notify function as a Operation parameter to the connected Sink Pin via the IOWFloatStream interface. You can add this pin to your TDemoTimer component, and test it, but before doing that, we will add Integer interface to the pin.

First we will add a new data type to the pin:

```
TOWDemoSourcePin = class(TOWSourcePin)
  protected
    FValue               : Single;

  protected
    function NotificationFloat( Handler : IOWStream; DataTypeID :
PDataTypeID; Operation : IOWNotifyOperation; State : TOWNotifyState ) :
TOWNotifyResult; virtual;

    function NotificationInteger( Handler : IOWStream; DataTypeID :
PDataTypeID; Operation : IOWNotifyOperation; State : TOWNotifyState ) :
TOWNotifyResult; virtual;
```

```
  protected
    procedure SetValue( AValue : Single );

  public
    constructor Create( AOwner: TComponent );

  public
    property Value : Single read FValue write SetValue;

end;
```

And now the implementation:

```
constructor TOWDemoSourcePin.Create( AOwner: TComponent );
begin
  inherited Create( AOwner );
  AddType( IOWIntegerStream, NotificationInteger );
  AddType( IOWFloatStream, NotificationFloat );
end;

function TOWDemoSourcePin.NotificationFloat( Handler : IOWStream;
DataTypeID : PDataTypeID; Operation : IOWNotifyOperation; State :
TOWNotifyState ) : TOWNotifyResult;
var
  Interf : IOWFloatStream;

begin
  Result := [];
  if( Handler.QueryInterface( IOWFloatStream,Interf) = 0 ) then
    begin
    if( nsNewLink in State ) then
      begin
      Interf.DispatchData( DataTypeID,
TOWSuppliedSingleOperation.Create( Value ), State );
      Exit;
      end;

    Interf.DispatchData( DataTypeID, Operation, State );
    end;

end;

function TOWFloatIntSourcePin.NotificationInteger( Handler : IOWStream;
DataTypeID : PDataTypeID; Operation : IOWNotifyOperation; State :
TOWNotifyState ) : TOWNotifyResult;
var
  Interf : IOWIntegerStream;

begin
  Result := [];
  if( Handler.QueryInterface( IOWIntegerStream,Interf) = 0 ) then
    begin
    if( nsNewLink in State ) then
      begin
      Interf.DispatchData( DataTypeID,
TOWSuppliedIntegerOperation.Create( Round( Value )), State );
      Exit;
      end;

    Result := Interf.DispatchData( DataTypeID, Operation, State )
```

```
    end;

end;
```

Notice the order of the AddType function calls. We first call AddType for Integer data, then for the floating point data. This means that the pin first will try to connect using the floating-point interface, and if it can't then it will try the integer format. The pin always will try to connect to the type registered last.
WARNING ! In the previous versions of OpenWire the registration was in reverse. The connection was established based on the first registered type, rather than on the last one. This is proved to be a wrong concept and was changed. Please update your code if necessary.

You can use one and the same Notification function for both formats, but that leads to a lot of bugs and is considered a very bad practice. It will not save any code, instead it will not allow the pin to participate in type dependencies. Even worse, in case you decide to create type dependency, the pin will use a wrong interface. Always use separated function handlers for each data format (interface).

Now it's time to test our pin with the TDemoTimer component we created before. Here is what you should change in the component:

```
TDemoTimer = class(TTimer)
  protected
    FOutput   : TOWExDemoSourcePin;
    FCounter : Single;

  protected
    procedure Timer; override;
    procedure Loaded; override;

    procedure STestClockComponentTimer( Sender : TObject );

  public
    constructor Create(AOwner: TComponent); override;
    destructor  Destroy; override;

  published
    property Output : TOWExDemoSourcePin read FOutput write FOutput;

  end;

constructor TDemoTimer.Create(AOwner: TComponent);
begin
  inherited;
  FOutput := TOWExDemoSourcePin.Create( Self );
  FCounter := 1;
  FOutput.Value := 1;
end;
```

The component is ready and can stream 2 different types of data. You can compile the component and test it with the TDemoLabel. If everything is fine, they will use the floating-point interface. You can easily test the integer interface, by either removing the AddType line for the floating-point interface, or by changing the AddType order.

Congratulations! You have just created your first pins of your own and implemented multiple data type support inside them. Now the only thing left to learn for you about the pins is how to define your own types of data (interfaces).

Defining Your Own Interfaces (Types Of Data)

To define new data type for OpenWire, you need to define a new interface descend from IOWStream. You should consider defining a new data type only in case there is no one available in the list of "Well-

known interfaces”. Defining a new interface means that no other components will be able to connect to your pins, unless you provide the interface to other developers one way or another. OpenWire provides a set of “Well-known interfaces”, but there are cases when you have to handle your own special type of data format. Then your only option is developing interface of your own. The first step is to select a name for the new interface. OpenWire recommends that all the interface names should start with IOW…. Second, you should create a new GUID for your interface. The GUID is the way OpenWire to distinguish the different interface, so it should be unique. Delphi and C++ Builder both are capable of generating unique GUIDs for you. Just press Ctrl+Shift+G in the editor, and they will generate a new GUID and will place it in the code. So here is a sample of a new OpenWire interface defining a new data type:

```
type IOWDemoStream = interface(IOWStream)
    ['{57C4D24C-66B6-419E-9319-1047C6B834CC}']
  end;
```

The just defined interface is ready to be used, but there is no real functionality in it. You can use it to test a pin, but you have no way to supply any data through it. For the purpose of the sample we will assume that your data is a special record type. As example:

```
type TMyData = record
  Data  : Pointer;
  Count : Integer;
end;
```

We need a mechanism to pass this data type through the just created interface. In order to do so, we need to add only one function or procedure to our interface:

```
type IOWDemoStream = interface(IOWStream)
    ['{57C4D24C-66B6-419E-9319-1047C6B834CC}']
    procedure SendMyData( Data : TMyData ); stdcall;
  end;
```

In most cases this function should be more than enough for our interface to operate under OpenWire. You can add any other functions you think you need to control the stream, synchronize the components, notify for conditions or whatever you think is necessary to support your data. However the more functions you add, the more support code will be necessary for each component to deal with your data. If there is not real necessity to add more functions restrict yourself to just one. OpenWire 2 introduces the Operation objects used to send command Upstream or Downstream. You can use the Operation object inside your interface.

Here is a sample:

```
type IOWDemoStream = interface(IOWStream)
    ['{57C4D24C-66B6-419E-9319-1047C6B834CC}']
    function SendMyDataByOperation(Operation : IOWNotifyOperation ) :
TOWNotifyResult; stdcall;
  end;
```

In this case the Operation will be used to pass a control message along with the data. It’s up to you to define new Notify Operation objects and how they will be used. However when releasing the interface to other developers, you should give them description of those objects and probably some samples on how to use them. Otherwise they will be unable to use your interface. The same is true for the TMyData record.

Here is a sample of a Notify Operation class declaration:

```
  TOWSuppliedMyDataOperation = class( TOWSuppliedOperation )
  public
    Value : TMyData;

  public
    constructor Create( AValue : TMyData );

  end;
```

The constructor will just initialize the Value member variable.

After defining the interface, you can (And should!) register it inside the unit Register section by using the OWRegisterStream procedure. Here is a sample how to do that:

```
procedure Register;
begin
  OWRegisterStream( IOWDemoStream, 'My demo stream' );
end;
```

By doing that, you are assigning a string name for this type of data. All the property editors and experts now will know how to display information about your data type, and it will be much easier to work with them. This registration is not a requirement, but is very important.

It's highly recommended that you register a default data dispatcher for your new data type. It will make it much easier for the users of your interface to develop pins supporting it.

```
function OWDefaultDemoStreamNotificationHandler( Sender : TOWPin;
Handler : IOWStream; DataTypeID : PDataTypeID; Operation :
IOWNotifyOperation; State : TOWNotifyState ) : TOWNotifyResult;
begin
  if( Operation.Instance() is TOWSuppliedMyDataOperation ) then
    ( Handler as IOWDemoStream ).SendMyData( TOWSuppliedMyDataOperation(
Operation.Instance()).Value );

  Result := [];
end;

initialization
  OWRegisterDefaultHandler( IOWDemoStream,
OWDefaultDemoStreamNotificationHandler );
```

In case you are using the second version of the interface as we discussed it earlier, the dispatcher will look like:

```
function OWDefaultDemoStreamNotificationHandler( Sender : TOWPin;
Handler : IOWStream; DataTypeID : PDataTypeID; Operation :
IOWNotifyOperation; State : TOWNotifyState ) : TOWNotifyResult;
begin
  Result := Handler.SendMyDataByOperation( Operation );
end;

initialization
  OWRegisterDefaultHandler( IOWDemoStream,
OWDefaultDemoStreamNotificationHandler );
```

This default dispatcher assumes that you will send the data by calling the Notify method of the pin passing a TOWSuppliedMyDataOperation object:

```
APin.Notify( TOWSuppliedMyDataOperation.Create( MyData ));
```

Any component which doesn't do so should register it's own notify dispatcher.

That is all you have to do in order to have a new OpenWire data type (interface).

Creating A Pin Implementing Your Interface

Now it’s time to create a pin using the newly created interface. For that purpose we will use the following interface:

```

type TMyData = record
  Data : Single;
end;
```

```
type IOWDemoStream = interface(IOWStream)
    ['{57C4D24C-66B6-419E-9319-1047C6B834CC}']
    procedure SendMyData( Data : TMyData ); stdcall;
  end;
```

We will declare our own Supplied object:

```
TOWSuppliedMyDataOperation = class( TOWSuppliedOperation )
  public
    Value : TMyData;

  public
    constructor Create( AValue : TMyData );

end;

constructor TOWSuppliedMyDataOperation.Create( AValue : TMyData );
begin
  inherited Create();
  Value := AValue;
end
```

The TMyData is somewhat useless, but is good enough to demonstrate our sample. Here is a sink pin implementing the new interface:

```
// Event type declaration
type TOWDemoSendEvent = procedure ( Data : TMyData ) of object;

// Pin declaration
TOWDemoSinkPin = class( TOWSinkPin, IOWDemoStream )
  protected
    SendMyDataEvent : TOWDemoSendEvent;

  public
    procedure SendMyData( Data : TMyData ); stdcall;

  public
    constructor Create( AOwner: TComponent;
ASendMyDataEvent : TOWDemoSendEvent );

end;
```

Now we will create the constructor, and the SendMyData method:

```
constructor TOWDemoSinkPin.Create( AOwner: TComponent;
ASendMyDataEvent : TOWDemoSendEvent );
begin
  inherited Create( AOwner );
  SendMyDataEvent := ASendMyDataEvent;
end;

procedure TOWDemoSinkPin.SendMyData( Data : TMyData );
begin
  if ( Assigned(SendMyDataEvent )) then
      SendMyDataEvent( Data );
end;
```

The pin is ready. Now here is a version of our TDemoLabel using the newly created pin:

```
TDemoLabel = class(TLabel)
  protected
    FInput    : TOWDemoSinkPin;
```

```
  protected
    procedure SendMyLabelData( MyData : TMyData );

  public
    constructor Create(AOwner: TComponent); override;
    destructor  Destroy; override;

  published
    property Input : TOWDemoSinkPin read FInput write FInput;
  end;

constructor TDemoLabel.Create(AOwner: TComponent);
begin
  inherited;
  FInput := TOWDemoSinkPin.Create( Self, SendMyLabelData );
end;

destructor TDemoLabel.Destroy;
begin
  FInput.Free;
  inherited;
end;

procedure TDemoLabel.SendMyLabelData( MyData : TMyData );
begin
  Caption := FloatToStr( MyData.Data );
end;
```

The only noticeable difference is the SendMyLabelData function. The function just assigns the proper value to the caption.

Creating A Pin Capable Of Sending Data Through your Interface

Now we can start working on our source pin, and make it capable of sending data through our interface. Here is our pin:

```
TOWDemoSourcePin = class(TOWSourcePin )
  protected
    FValue : Single;

  protected
    procedure SetSingle( AValue : Single );

  public
    function Notification( Handler : IOWStream; DataTypeID :
PDataTypeID; Operation : IOWNotifyOperation; State : TOWNotifyState ) :
TOWNotifyResult; virtual;

  public
    constructor Create( AOwner: TComponent );

  public
    property Value : Single read FValue write SetSingle;

  end;
```

Here is the implementation for the constructor, the Notification, and the SetSingle function:

```
constructor TOWDemoSourcePin.Create( AOwner: TComponent );
begin
```

```
  inherited Create( AOwner );
  AddType(IOWDemoStream, Notification);
end;

function TOWDemoSourcePin.Notification( Handler : IOWStream;
DataTypeID : PDataTypeID; Operation : IOWNotifyOperation; State :
TOWNotifyState ) : TOWNotifyResult;
var
  Interf : IOWDemoStream;
  MyData : TMyData;

begin
  Result := [];

  if( Handler.QueryInterface(IOWDemoStream,Interf) = 0 ) then
    begin
    if( nsNewLink in State ) then
      begin
      MyData.Data := FValue;
      Interf.SendMyData( MyData );
      Exit;
      end;

    if( Operation.Instance() is TOWSuppliedMyDataOperation ) then

Interf.SendMyData( TOWSuppliedMyDataOperation( Operation.Instance() ).Va
lue );

    end;

end;

procedure TOWDemoSourcePin.SetSingle( AValue : Single );
var
  MyData : TMyData;

begin
  if( FValue <> AValue ) then
    begin
    FValue := AValue;
    MyData.Data := AValue;
    Notify( TOWSuppliedMyDataOperation.Create( MyData ));
    end;

end;
```

There is nothing new in this implementation, so we will move to our final step, which will be – creating a component to test our new source pin. Here is the changed version of TDemoTimer:

```
TDemoTimer = class(TTimer)
  protected
    FOutput  : TOWDemoSourcePin;
    FCounter : Single;

  protected
    procedure Timer; override;
    procedure Loaded; override;

    procedure STestClockComponentTimer( Sender : TObject );
```

```
  public
    constructor Create(AOwner: TComponent); override;
    destructor  Destroy; override;

  published
    property Output : TOWDemoSourcePin read FOutput write FOutput;
  end;

constructor TDemoTimer.Create(AOwner: TComponent);
begin
  inherited;
  FOutput := TOWDemoSourcePin.Create( Self );
  FCounter := 1;
  FOutput.Value := 1;
end;
```

```
procedure TDemoTimer.Timer;
var
  MyData : TMyData;

begin
  MyData.Data := FCounter;
  FOutput.Value := FCounter;

  FCounter := FCounter + 0.5;

  if( FCounter > 100 ) then
    FCounter := 1;

  inherited Timer;
end;
```

There is nothing special here and the implementation doesn't differ much than what we have already seen. By assigning a new value to FOutput.Value we invoke the SetValue procedure. SetValue will call Notify with a new TOWSuppliedMyDataOperation object and will pass the new data. If there are any Sink Pins connected, the Notification will be invoked for each of them, and will pass the data to the SendMyData procedure.

This was the last and by far the most complex development of pins we are in about to cover here. You can also make the pin to accept some other types of data, but this topic was covered already.

Registering Your Interface As a Standard One

You can register your interface as a "Well-known interface". Information how to register your interface, will be posted on the OpenWire's web site at www.openwire.org.

Function Dependencies

Very often the result of a source pin is calculated based on one or more sink pins. As example if you have a filter, the output is some function of the input. Obviously it's not possible to connect the output to the input. OpenWire provides a standard way of describing such a relationship. The mechanism is called registering a function source. The source pins have a function called FunctionSources.Add for this purpose. Here is a sample how you can use this function:

```
constructor TSTestComponent.Create(AOwner: TComponent);
begin
  inherited;
  FInput  := TTestSinkPin.Create( Self, UpdateData, PinNotification );
  FOutput := TTestSourcePin.Create( Self, UpdateData, PinNotification );
  FOutput.FunctionSources.Add( FInput );
```

```
end;
```

In this case the FOutput depends on FInput and OpenWire will not allow them to connected to each other, neither will it allow the end of a chain of connected function sources to be connected with its beginning.

Type Dependencies

If you have a component with a sink and a source pin, and both pins can work with multiple types of data, you may want to force the output (the source pin) to use the same type of data, as the input (the sink pin). In this case you are trying to establish type source for the source pin (the output). You can use the DataTypeSources.Add function for this purpose.
Here is a sample, how you can do that:

```
constructor TSTestComponent.Create(AOwner: TComponent);
begin
  inherited;
  FInput  := TTestSinkPin.Create( Self, UpdateData, PinNotification );
  FOutput := TTestSourcePin.Create( Self, UpdateData, PinNotification );
  FOutput.DataTypeSource.Add( FInput );

end;
```

In this case OpenWire will connect the sink pin and the source pin to other components only if both connections can use the same interface (data type). This functionality ensures that the entire chain of data will be homogeneous. I.E. all the pins in the chain will exchange the same type of data.

Dynamic Streaming Order Balancing

OpenWire is designed and optimized for very high performance. In fact it has been designed for fast data acquisition systems and can transfer data within the range of tens of millions of samples per second (> 20Ms), event on 500 MHz Pentium systems. One of the problems you will encounter when dealing with such speed is the need to perform additional copy operations when multiple sink pins are connected to a single source. In this case on a first glance you have to copy the data for each sink pin. This is not necessary however, and you must avoid it in order to achieve high performance. The key is a Well-known technology named lazy evaluation. You are using this technology every day, when you are using strings, so far the Delphi String and the C++ Builder AnsiString are designed using lazy evaluation. We will not go very deep into this technology, so far it is not real part of the OpenWire, but we will discuss the basic principles involved. The idea is that if none of the recipients is in about to modify the data we can keep it at the same location and just pass a pointer to the data. We need to have a counter indicating how many pointers we have to the data. Whenever somebody is not interested any more of the data, we just release the pointer and decrease the counter. When the counter becomes 0 no one needs the data, and the data can be destroyed. If somebody wants to modify his data, we should check if the counter is one – i.e. there are no other owners then we can directly apply those changes over the data. When the counter is larger than 1 we need to make a separated copy of the data and direct the pointer there. The new copy will have its own counter set to 1 and we will decrease the counter of pointers to the original data. Using this mechanism means that we can deliver data to multiple recipients without a single copy operation and achieve otherwise unimaginable speeds. There is one very important detail however. After delivering the data to the last pin in the list of sink pins, our source pin can release the data. In this case, we can release the ownership of the data, even before we deliver it. This way if the sink pin modifies the data it will be the only owner and a copy operation will not take place. If any of the other pins attempt modification a copy operation will be performed. It is obvious that in order to optimize the system we should try to deliver the data last to the pin, which most often and most recently had modified the data. This way the system optimizes itself over the time achieving the best possible performance. OpenWire has build in mechanism to perform this important and complex operation. This mechanism is called Dynamic streaming order balancing and is very simple to use. Remember the format of the notification functions?

```
function Notification( Handler : IOWStream; Operation :
IOWNotifyOperation; State : TOWNotifyState ) :
TOWNotifyResult; virtual;
```

Pay close attention to the returned result and the State. We already discussed the TOWNotifyState. One thing we mentioned was that there is a state called nsLastIteration. This state indicates that we are dealing with the last sink pin in the list of sink pins and we can release the ownership of the data. The “return” parameter is a set of conditions. Currently there is only one possible element in the set and this is the nrDataChanged condition. We have never used this condition in our samples, because we were working with small size of data, and there were not any large copy operations we needed to optimize for speed. However when you are using lazy evaluation and you are modifying the data inside the pin you must return nrDataChanged as a condition. This will allow OpenWire to perform the Dynamic streaming order balancing, and you will achieve much better performance. Here is a sample of a source pin implementing such optimization:

```
function TDemoTimer.FloatPinNotification( Handler : IOWFloatStream;
Operation : IOWNotifyOperation; State : TOWNotifyState ) :
TOWNotifyResult;
begin
  if( Operation.Instance() is TIISuppliedBufferOperation ) then

if( Handler.SendSomeData( TIISuppliedBufferOperation(Operation.Instance(
)).DataBuffer)
      begin
      // The data has been modified by the SendSomeData function
      Result := [ nrDataChanged ];
      Exit;
      end;

  Result := [];
 end;
```

In this sample we assume that the SendSomeData function returns True if the data has been modified. Another even much better approach is to design your interface, so the data handling function to return TOWNotifyResult. Here is an example of such interface taken from a very high performance OpenWire based VCL library designed by Innovative-Integration www.innovative-dsp.com :

```
IOWIIFloatStream = interface(IOWStream)
    ['{58041272-EF20-4696-87E6-9EE46ABFBD89}']
    function Operation( Operation : IOWNotifyOperation; State :
TOWNotifyState ) : TOWNotifyResult; stdcall;
  end;
```

In this case the Operation function returns TOWNotifyResult and can indicate whether or not the data has been modified inside. Also this way any other features added to the TOWNotifyResult in the future will be immediately available in your function. Here is how your notify function will look like in this case:

```
function TDemoTimer.IIFloatBlockPinNotification( Handler :
IOWFloatStream; Operation : IOWNotifyOperation; State : TOWNotifyState )
: TOWNotifyResult;
begin
  if( Operation.Instance() is TIISuppliedBufferOperation ) then
    begin
    Result := Handler.Operation( Operation, State );
    Exit;
    end;

  Result := [];
 end;
```

Here is how your data handling method will look like:

```
function TAComponent.IIDataOperation( Operation : IOWNotifyOperation;
State : TOWNotifyState ) :
TOWNotifyResult;
```

```
begin
  if( NeedToModifyTheData )
    begin
    // Modify the data here!
    Result := [ nrDataChanged ];
    end

  else
    begin
    // Use the data without modification here!
    Result := [];
    end;

end;
```

You may never need to use or support the Dynamic streaming order balancing. Most of the applications use small data to transfer and the speed is not as critical. In those cases you can always return Result := []; . However if the very high speed is critical for you and you are using some form of lazy evaluation, the mechanism is here working and ready for you to use it.

Dynamic Pin Lists (Arrays)

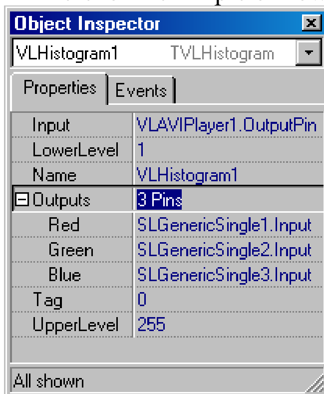
In many cases you know how many inputs and outputs you need for your component. A filter as example has usually one input and one output, so it is easy. Sometimes however you may need to allow the user to specify the number of pins. As example if you have a TAdd component whose output is the sum of some number inputs, you don't know how many inputs the user will need, and you can't provide that number while you are designing the component. It is possible to create collection properties and to handle that through them, but it's a lot of work, and is not very convenient for the end user.

OpenWire provides two very convenient and easy-to-use classes named TOWPinList and TOWPinListOwner.

TOWPinList is designed for those cases when the amount of pins supported by the list is based on another property. As example if you have a graph, the number of pins should match the number of channels on the graph.

TOWPinListOwner is designed for the case when the list should control the amount of pins. It allows you to specify the pins and automatically allocates and releases them.

Here is an example of how the Pin List properties will appear in the Object Inspector:



You can add a pin to the TOWPinList in your code by using the Add method or the AddNamed method:

```
Delphi Example :
  PinList.Add( AnySinkOrSourcePin );
  PinList.AddNamed( AnySinkOrSourcePin, 'My pin' );

C++ Builder Example :
  PinList->Add( AnySinkOrSourcePin );
  PinList->Add( AnySinkOrSourcePin, "My pin" );
```

Add named just describes how the pin will appear in the object inspector.

You can delete pin using the Remove or Delete methods:

```
Delphi Example :
  PinList.Remove( AnySinkOrSourcePin );
// or
  PinList.Delete( 3 );  // Removes the third pin

C++ Builder Example :
  PinList->Remove( AnySinkOrSourcePin );
// or
  PinList->Delete( 3 ); // Removes the third pin
```

When using TOWPinListOwner you can resize the list just by assigning the Count:

```
Delphi Example :
  PinListOwner.Count := 5;

C++ Builder Example :
  PinListOwner->Count = 5;
```

You can obtain the pins count from the Count property.
You can access the pins using the Pins property:

```
Delphi Example :
  var I   : Integer;
  var Pin : TOWPin;

  for I := 0 to PinList.Count - 1 do
    begin
    Pin := PinList.Pins[ I ];
    end;

C++ Builder Example :
  for( int i = 0; i < PinList->Count; i ++ )
    {
    TOWPin *Pin = PinList->Pins[ i ];
    }
```

Creating Components Using Pin Lists

Let's create a simple component with multiple inputs using TOWPinListOwner. We will assume that the component will multiply all the inputs and will send the result through an output(source pin). Here is the declaration of the component:

```
TOWLMultiply = class(TComponent)
  private
    FInputDataArray : array of Single;

  protected
    FOutput     : TOWFloatSourcePin;
    FInputs     : TOWPinListOwner;

  protected
    procedure SendData( DataTypeID : PDataTypeID; Operation :
IOWNotifyOperation; CustomData : TObject );
    function  PinNotification( Handler : IOWFloatStream;
Operation : IOWNotifyOperation; State : TOWNotifyState ) :
TOWNotifyResult;

    function  CreateInputPin( APinListOwner : TOWPinList ) : TOWPin;
    procedure DestroyInputPin( APinListOwner : TOWPinList );
```

```
  public
    constructor Create(AOwner: TComponent); override;
    destructor  Destroy; override;

  published
    property Output : TOWFloatSourcePin  read FOutput write FOutput;
    property Inputs : TOWPinListOwner read FInputs write FInputs;
  end;
```

The only new thing for us is the declaration of the Inputs. However it's very much alike the declaration of a single pin. The only new methods for us are the CreateInputPin and DestroyInputPin. We will focus on them after taking a look at the implementation:

```
constructor TOWLMultiply.Create(AOwner: TComponent);
begin
  inherited;
  FOutput := TOWFloatSourcePin.Create( Self, PinNotification );
  FInputs := TOWPinListOwner.CreateEx( Self, 2, 100, CreateInputPin,
DestroyInputPin );
end;

destructor  TOWLMultiply.Destroy;
begin
  FInputs.Free;
  FOutput.Free;
  inherited;
end;

function TOWLMultiply.CreateInputPin( APinListOwner : TOWPinList ) :
TOWPin;
var
  Pin : TOWFloatSinkPin;

begin
  SetLength( FInputDataArray, APinListOwner.Count + 1 );
  FInputDataArray[ APinListOwner.Count ] := 1.0;

  Pin := TOWFloatSinkPin.Create( Self, SendData,
TObject( APinListOwner.Count ));
  FOutput.FunctionSources.Add( Pin );
  Result := Pin;
end;

procedure TOWLMultiply.DestroyInputPin( APinListOwner : TOWPinList );
begin
  SetLength( FInputDataArray, APinListOwner.Count );
end;

function TOWLMultiply.PinNotification( Handler : IOWFloatStream;
Operation : IOWNotifyOperation; State : TOWNotifyState ) :
TOWNotifyResult;
var
  I     : Integer;
  Value : Single;

begin
  Value := 1;

  for I := 0 to FInputs.Count - 1 do
    Value := Value * FInputDataArray[ I ];
```

```
  Handler.DispatchData( DataTypeID,
TOWSuppliedSingleOperation.Create( Value ), State );
  Result := [];
end;

procedure TOWLMultiply.SendData( DataTypeID : PDataTypeID; Operation :
IOWNotifyOperation; CustomData : TObject );
begin
  if( Operation.Instance() is TOWSuppliedSingleOperation ) then
    begin
    FInputDataArray[ Integer( CustomData ) ] :=
TOWSuppliedSingleOperation( Operation.Instance() ).Value;
    FOutput.Notify( TOWNotifyOperation.Create() );
    end;

end;
```

In the constructor we just create the source pin for the output and a TOWPinListOwner for the inputs. The constructor of the TOWPinListOwner has 4 parameters. The first and the second are the minimum and the maximum size of the list – in this case we should have no less than 2 and no more than 100 inputs. The other two parameters are an event handler, which gets called when new pin is needed and an optional handler called when the pin will be destroyed. We must provide the pin creation event. Otherwise the pin list will not be able to know what type of pin we need to create.

Here is a sample of the simplest possible such event handler:

```
function TOWLMultiply.CreateInputPin( APinListOwner : TOWPinList ) :
TOWPin;
begin
  Result := TOWFloatSinkPin.Create( Self, SendData, NIL ));
end;
```

In this case we just create a new pin and return it. The deletion handler is optional. In our case we are using it to destroy some internal data associated with the pin, when the pin is destroyed. We will not go into any details how the component works so far that they are not related to the pin lists.

When you use the TOWPinList you have to maintain the creation and the destruction of the pins your self. Here is an example of a constructor creating a few pins and organizing them in pin list:

```
constructor TJustTestComponent.Create(AOwner: TComponent);
begin
  inherited;
  FCombo := TOWPinList.Create( Self, False );
  FCombo.AddNamed( TOWTestSinkPin.Create( Self, UpdateData,
PinNotification ), 'Test Input1');
  FCombo.AddNamed( TOWTestSinkPin.Create( Self, UpdateData,
PinNotification ), 'Test Input2');
  FCombo.AddNamed( TOWTestSinkPin.Create( Self, UpdateData,
PinNotification ), 'Test Input3');
  FCombo.AddNamed( TOWTestSourcePin.Create( Self, UpdateData,
PinNotification ), 'Test Output1');
  FCombo.AddNamed( TOWTestSourcePin.Create( Self, UpdateData,
PinNotification ), 'Test Output2');
  FCombo.AddNamed( TOWTestSourcePin.Create( Self, UpdateData,
PinNotification ), 'Test Output3');
end;
```

The second parameter in the TOWPinList constructor indicates whether or not the pins should be destroyed when they are deleted from the pin. If the parameter is True, the Pin List will delete them automatically. Otherwise we will have to manually delete them.

Writing Threading Safe Components with OpenWire

OpenWire 2.4 and higher is threading safe out of the box, however in some cases you may have to write some additional code in order to write fully thread safe components. This is a huge topic and goes beyond the scope of this manual but we will cover some basic ideas here.

Each pin has a thread safe locking mechanism. When you send data from a pin to another, all the ins in the connection will be locked until the data is delivered. In addition when you have a function dependency set between source and sink pin, they by default will share a lock, and both will be locked at the same time. This allows the component writer to be sure that the In and Out pin of the component are locked while the data is processed and race conditions will not happen. Often however the component will include properties that affect the calculations, but are accessible from the rest of the application. If you have a multithreading component that is a signal generator, and you have a property that sets the amplitude of the signal, you must make sure the property is threading safe. There are few ways to do so. You can lock directly the output pin of the component while setting the property. A more flexible approach is to create a separated locking object inside the component and make it share lock with the output pin. Here is an example of this technique:

```
type TMySignalGen = class(TComponent)
  protected
    FLock      : TOWObject;
    FOutputPin : TOWRealSourcePin;
    FAmplitude : Integer;

  protected
    procedure SetAmplitude( Value : Real );
    function  GetAmplitude() : Real;
...
...
  public
    constructor Create(AOwner: TComponent); override;
    destructor  Destroy(); override;

  published
    property Amplitude  : Real read GetAmplitude write SetAmplitude;
    property OutputPin  : TOWRealSourcePin read FOutputPin write
FOutputPin;

 end;

...
...

constructor TMySignalGen.Create(AOwner: TComponent);
begin
  inherited;
  FLock := TOWObject.Create();
  FOutputPin := TOWRealSourcePin.Create( Self, DispatchOperation );
  FLock.AddShareLock( FOutputPin );
end;

destructor  TMySignalGen.Destroy();
begin
  FOutputPin.Free();
  FLock.Free();
  inherited;
end;

procedure TMySignalGen.SetAmplitude( Value : Real );
var
  AWriteLock : IOWLockSection;
```

```
begin
  AWriteLock := FLock.WriteLock();
  FAmplitude := Value;
end;

function  TMySignalGen.GetAmplitude() : Real;
var
  AReadLock : IOWLockSection;

begin
  AReadLock := FLock.ReadLock();
  Result := FAmplitude;
end;
```

Conclusion

OpenWire is a powerful tool for creating flexible and expandable VCL components. It is easy to use and does require very little code to support the functionality. The OpenWire components are very easy to use and allow writing of complex applications with zero lines of code.