OptiSystem

Tutorials - C++ Component

Optical Communication System Design Software

Version 14

OptiSystem

Tutorials - C++ Component

Optical Communication System Design Software

Disclaimer

Optiwave makes no representation or warranty with respect to the adequacy of this documentation or the programs which it describes for any particular purpose or with respect to its adequacy to produce any particular result. In no event shall Optiwave, its employees, its contractors or the authors of this documentation, be liable for special, direct, indirect, or consequential damages, losses, costs, charges, claims, demands, or claim for lost profits, fees, or expenses of any nature or kind.

Technical support

If you purchased Optiwave software from a distributor that is not listed here, please send technical questions to your distributor.

Optiwave — Canada/US

Tel	(613) 224-4700	**E-mail**	support@optiwave.com
Fax	(613) 224-4706	**URL**	www.optiwave.com

Optiwave Japan — Japan

Tel	+81.43.375.2644	**E-mail**	support@optiwave.jp
Fax	+81.43.375.2644	**URL**	www.optiwave.jp

Optiwave Europe — Europe

Tel	+33 (0) 494 08 27 97	**E-mail**	support@optiwave.com
Fax	+33 (0) 494 33 65 76	**URL**	www.optiwave.eu

Table of contents

Introduction

IMPORTANT!

*To create and compile the C++ code for the **Cpp Component** you must have either Microsoft Visual Studio 2013 Community or Professional installed on your computer. (**Note:** The Cpp Component is not currently compatible with Microsoft Visual Studio 2015!).*

OptiSystem's **Cpp Component,** similar in operation to the **MATLAB Component**, allows users to create their own custom components, written directly in C++, for use in OptiSystem projects. The user can also create their own internal graphs and results that can be displayed within OptiSystem.

The component can be configured in two modes:

Release mode - The component project is compiled as a dynamic link library (.dll) in release mode. OptiSystem will load the “dll” during runtime. The signal data is passed directly from OptiSystem to the “dll” and then, after processing, from the “dll” back to OptiSystem. This is the mode to use when you wish to run OptiSystem and your C++ component within it.

Debug mode - The component project is compiled as a stand-alone application (.exe) in debug mode. Running the component project from within the developer environment will allow the user to enter the code and set breakpoints. OptiSystem creates copies of the signal data in various text files which will be loaded by the project. The project knows where to obtain these files and their formatting by an XML data file it loads at runtime.

The full documentation and a pre-configured project is available at the Optiwave C++ Component Home Page

The project is large with many classes available to the user. Due to the number of configuration settings involved, it is recommended to create a copy of this default project and add code to it for each component the user wishes to define. The user may add as many classes as they wish to this project but it is not recommended that the user removes or alters the classes already provided other than at the defined entry point: *Calculate_API()* in class **DS_SystemManager.cpp**.

The project has been set up so that very little changes must be made to switch between the Release and Debug modes. The user will not need to alter any of their

code, just change some configuration settings and add a line of code to tell the project where to find the XML file mentioned above. The two different modes are explained in detail in Tutorial 1: Binary Signal Manipulation.

The most effective method to learn the usage of OptiSystem's C++ component is to follow the tutorials below. They will teach the user how to access and manipulate the basic data structures, configure the project for release and debug mode, and use many of the convenience functions built into the project.

It is recommended that the user create a copy of the default project available at the Optiwave C++ Component Home Page for each tutorial, then follow along for the manipulations of the C++ code and the creation of the OptiSystem project.

This document contains the following sections.

Tutorials

C++ Component Tutorials

This section contains the following introductory tutorials for the Cpp (C++) Component.

Tutorial 1: Release and Debug modes, Basic manipulation of a Binary Signal

This tutorial describes how to configure a project in either Release mode or Debug mode. It also demonstrates how to manipulate a binary signal.

Release mode of operation

Part 1: Create the ".dll" for the component

Follow the steps below to ensure that the project is configured properly to create a "dll" in release mode appropriate for OptiSystem.

Step **Action**

Verification of project configurations

1 **Open** the **CppCoSimulation.sln** located in the base directory of the provided project in Visual Studio 2013.

2 **Verify** that the whole project is set to Release mode (x64) (see Fig 1*)*.

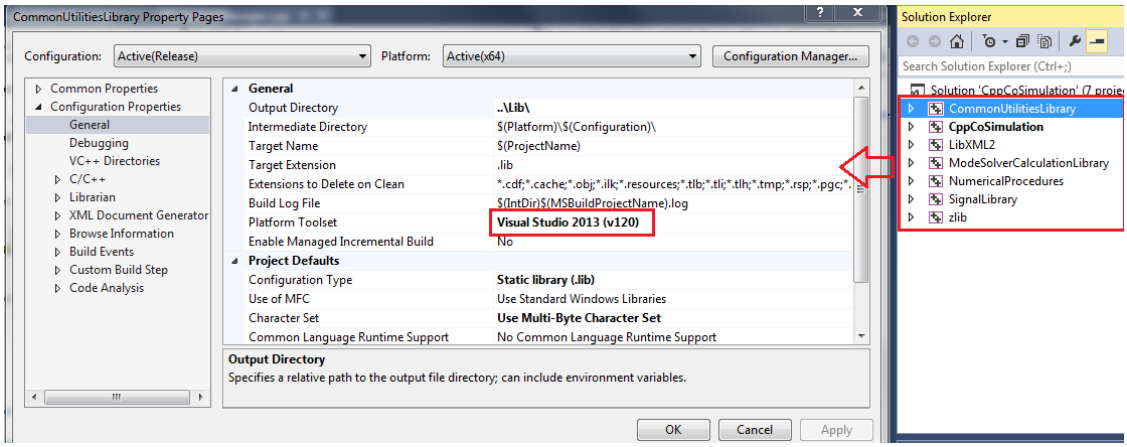
3 **Right-click** on each directory listed in the Solution Explorer and make sure that the following key configuration values are set (the rest may stay as default)

a. Go to **Properties/Configuration Properties/General**

b. Verify that **Platform Toolset** is set to Visual Studio 2013 (v120) and the Configuration/Platform is Active(Release)/Active(x64) (see Fig 2)
If it is not, your project won't compile and/or link to OptiSystem correctly

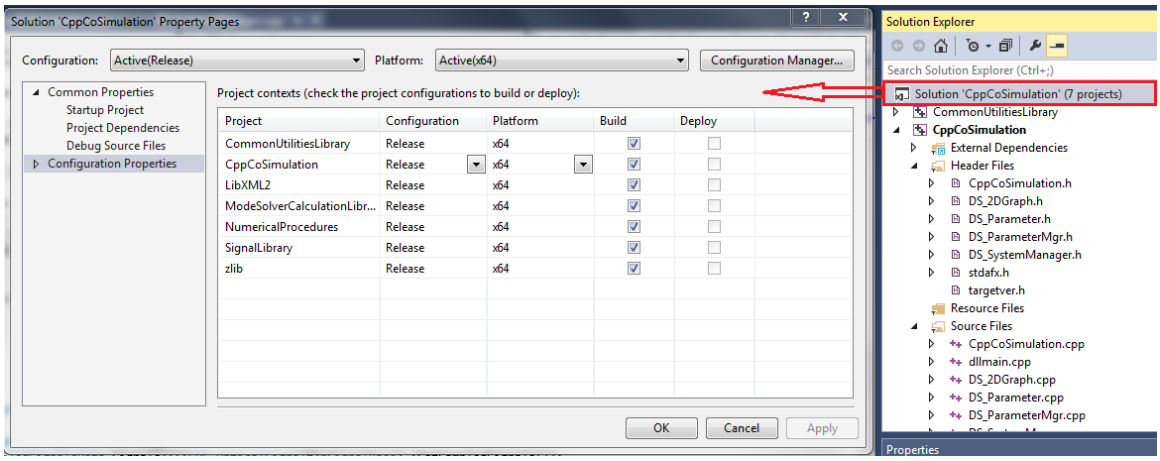
Figure 1

Figure 2



c. **Right-click** on *Solution 'CppCoSimulation'*, select ***Properties/Configuration Properties***. Verify that the active configuration is release. Verify all projects are configured for release x64 and the build boxes are checked (Fig 3)

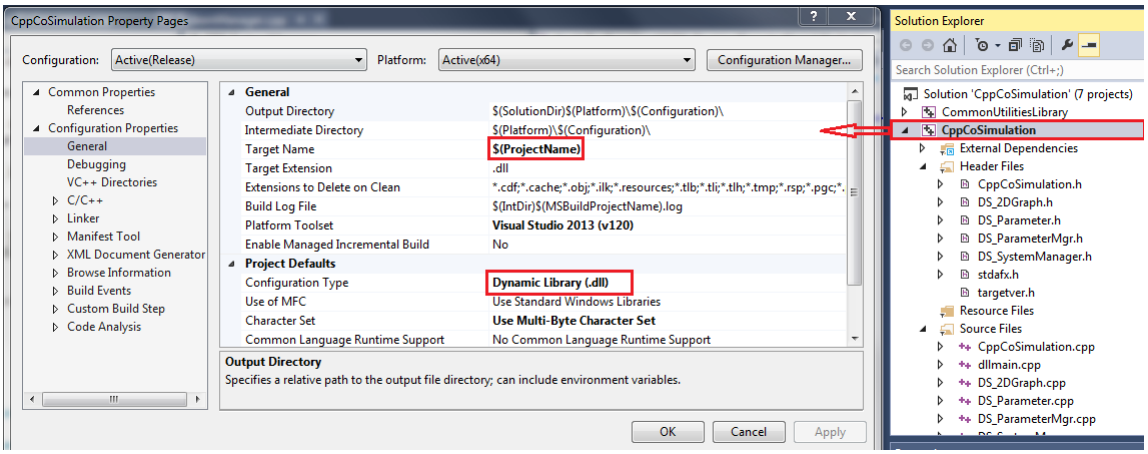
Figure 3



d. **Right-click** on **CppCosimulation**, select ***Properties/Configuration Properties/General***. **Verify** that Configuration Type is Dynamic Library (Fig 4).

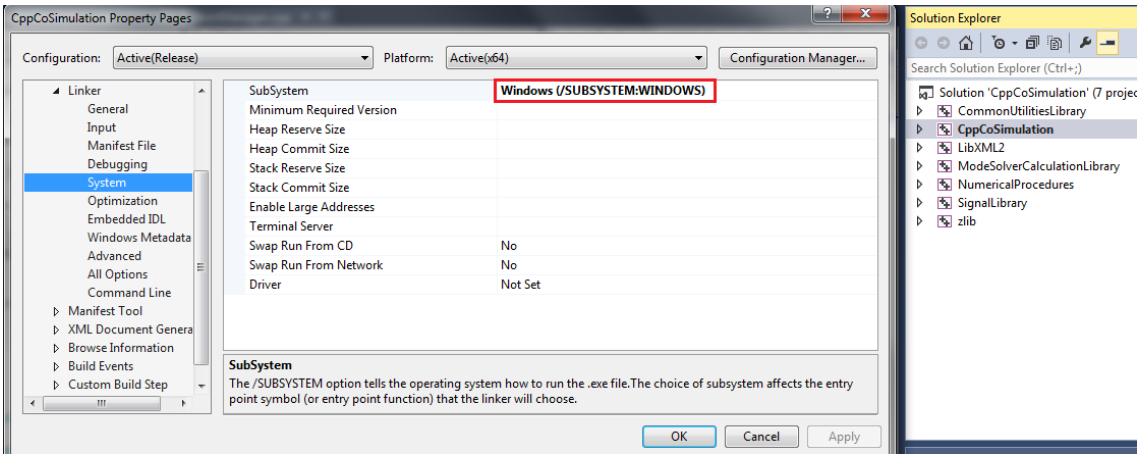
If you wish, you may also change the Target Name from the default (which will name the resulting dynamic library as "CppCoSimulation.dll").

Figure 4



e. Under ***Configuration Properties/Linker/System*** **Verify** that SubSystem is set to Windows (Fig 5).

Figure 5



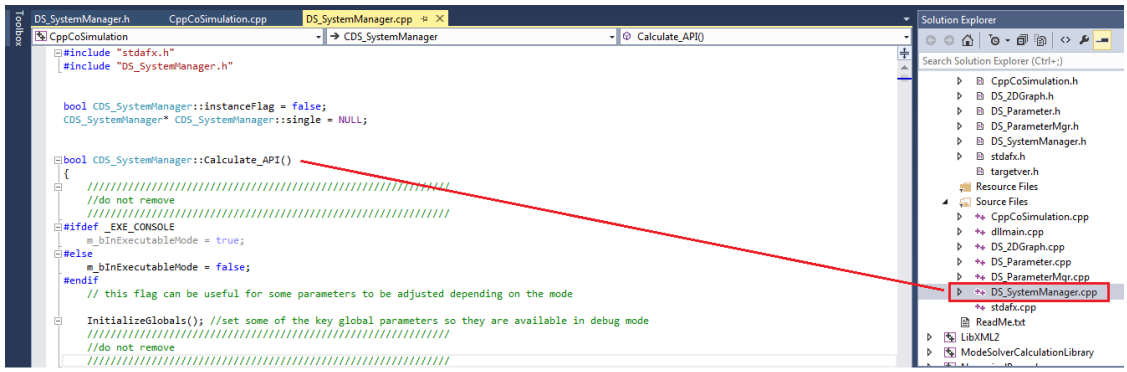
Implementation of code

4 Under *CppCoSimulation/Source Files*, **double-click** on DS_SystemManager.cpp

5 Within this file, navigate to the function bool CDS_SystemManager:Calculate_API(). (Fig 6)

This represents the entry point for your code into the project. For all of your projects, you will start your code here. You may create your own classes and functions which will be called from here.

Figure 6



Loading port data

6 **Copy** the following lines into the function (make sure to paste directly after "InitializeGlobals();" or the "DO NOT REMOVE" comment block)

```
////////////////////////////////////////////// BINARY SIGNAL OUTPUT//////////////////////////////////////////////////
CDS_SignalBase* pOOSignal1 = GetSignalFromOutputPort(1);
//create an output port

if (pOOSignal1 != NULL)
{
if (pOOSignal1->GetSignalName() == "BinarySignal")
{
```

*The first line loads the signal from the first (and in this case only) port. If you had a second port to load, you would add: CDS_SignalBase * pIOSignal2 = GetSignalFromInputPort(2). The IF conditions have been added as a safety precaution. This will ensure that the calculation will only go ahead if the input port is defined with the proper signal type*

Getting binary data

7 Copy the following lines into the function

```
CDS_BinarySignal binarySignal1 = GetBinarySignal(pIOSignal1);
////////////////////////////////////////////// BINARY SIGNAL LOADING//////////////////////////////////////////////////
```

This binary signal class holds both the sequence of bits (vector<long> binarySignal1.m_Bits) and the bit rate (binarySignal1.GetBitRate()).

Manipulating data

8 Now that the data has been placed on a vector, we can manipulate this to our requirements. In this case we are simply performing a binary inverse operation. **Copy** the following lines into the function

```
////////////////////////////////////////////// DATA MANIPULATION//////////////////////////////////////////////////
for (int i = 0; i < binarySignal1.m_Bits.size(); i++)
{
binarySignal1.m_Bits[i] = (binarySignal1.m_Bits[i] + 1) % 2;
}
//if value = 0 -> 1, if value = 1 -> 0
////////////////////////////////////////////// DATA MANIPULATION//////////////////////////////////////////////////
```

Create an output port

9 **Copy** the following lines into the function

```
////////////////////////////////////////////// BINARY SIGNAL OUTPUT//////////////////////////////////////////////////
CDS_SignalBase* pOOSignal1 = GetSignalFromOutputPort(1);
//create an output port

if (pOOSignal1 != NULL)
{
if (pOOSignal1->GetSignalName() == "BinarySignal")
{
```

This will create an output port for our data for OptiSystem to access. Checks have been added to make sure that the calculation will only go ahead if the output port on the C++ component is configured correctly

Set the output port signal to the binary signal

10 **Copy** the following lines into the function

```
PutBinarySignal(binarySignal1, pOOSignal1);
//put the binary signal onto the output port
```

Code for error checking

11 **Copy** the following lines into the function
Ensures that we close all the brackets for our safety precautions

```
}
else
{
return 0; //returns warning to OptiSystem -- output is not set correctly to binary
} //if (pOOSignal1->GetSignalName() == "BinarySignal")
} //if (pOOSignal1 != NULL)
else
{
return 0; //returns warning to OptiSystem -- output put not created
}
} //if (pIOSignal1 != NULL)
} //if (pIOSignal1->GetSignalName() == "BinarySignal")
SaveOutputDataToFiles();//for debugging purposes
////////////////////////////////////
return 1; //for successful run
```


Note: The full code and an OptiSystem project for this function (*Tutorial1SupplementaryFiles.zip)* is available at the Optiwave C++ Component Home Page. Replace the **DS_SystemManager.cpp** file in your project with the one given in the zip file. If you want to run in debug, you must also change the location of the XML given in **CppCoSimulation.cpp**. In addition change the location of the files in the OptiSystem projects to your particular files and directories

Note we have added some code right at the beginning of the routine (as follows)

```
////////////////////////////////////////////////////////////
//do not remove
////////////////////////////////////////////////////////////
        #ifdef _EXE_CONSOLE
                m_bInExecutableMode = true;
        #else
                m_bInExecutableMode = false;
        #endif
        // this flag can be useful for some parameters to be adjusted depending on the mode

        InitializeGlobals(); //set some of the key global parameters so they are avaliable in debug mode
////////////////////////////////////////////////////////////
//do not remove
////////////////////////////////////////////////////////////
```

This code is not necessary to run this example, but it is a good idea to keep in all future projects.

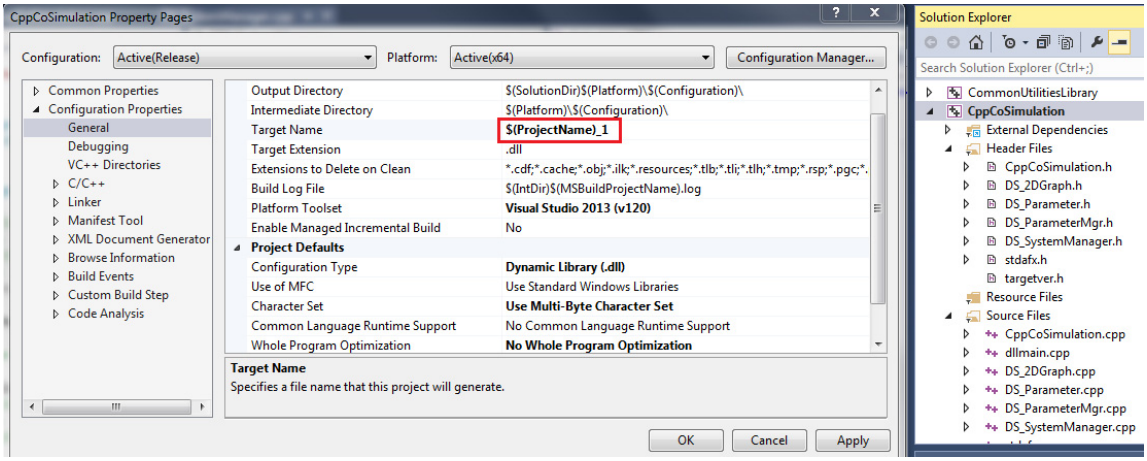
The variable *m_bInExecutableMode* will be used in the next example. The command *InitializeGlobals();* is to make sure all global parameters are initialized correctly when we are running in debug mode.

Build the dll

12 The configurations and code are now finished, all that remains is to build the dynamic library. Within Visual Studio's menu, **select** *Build/Build Solution*

If for some reason the dll is locked, you won't be able to build successfully. If this is the case, you will either have to exit OptiSystem in order to release the dll or rename your output dll. The simplest method is to rename the Target name of the dll, for example by renaming from ***$(ProjectName)*** *to* ***$(ProjectName)_1*** *(you can keep changing this name for each build) (See Fig 7)*

Figure 7



Note: Make sure to load the appropriate dll in Step 3 (Part 2).

Part 2: Creating an OptiSystem project to use the component

In part 1 we created a dll for the component, we must now link this to an OptiSystem project.

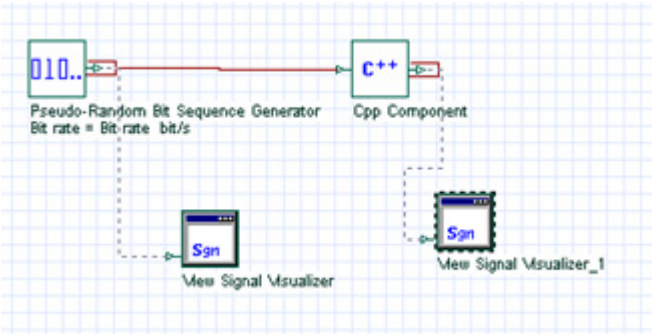
Step **Action**

Verification of project configurations

1 **Open** a session of OptiSystem and **create** the following system layout (Fig 8)

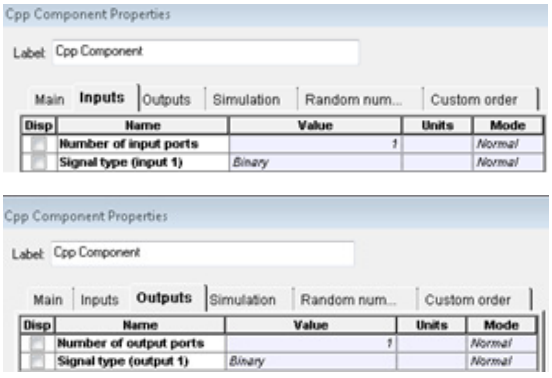
The Cpp Component can be found under Default/External Software Tools/CoSimulationLibrary. Your PRBS may not connect to the ***Cpp Component*** *yet as you must first set the appropriate port structure*

Figure 8 Cpp component setup in OptiSystem



2 **Set** the input and output ports to binary (Fig 9)

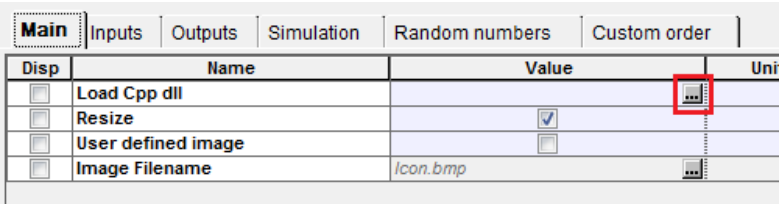
Figure 9 Cpp Component port settings



3 **Load** the dll (Fig 10)

Note: If you kept your folder name as CppCoSimulation and the target name as the default, your dll file will be at ***[your location]\CppCoSimulation\x64\Release\CppCoSimulation.dll]***

Figure 10 Load Cpp dll

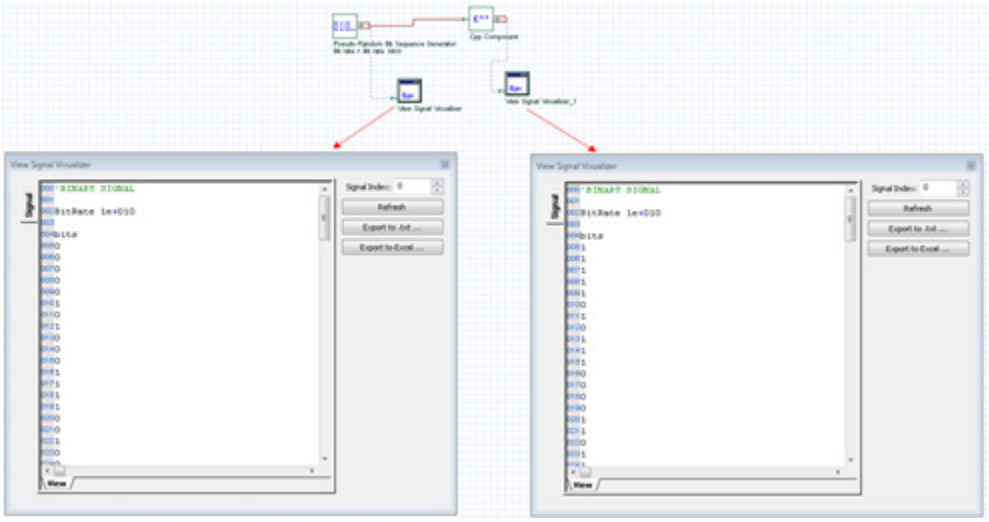


4 **Run** the project

The configuration of the project is now complete. If you now run your simulation, you should get results similar to the following where the output data is the inverse of the input (see Fig 11)

Note: If you receive a calculation warning "*Calculating Cpp Component...Warning while calculating component Cpp Component*" check to make sure that both the input and output ports are configured correctly to the binary format.

Figure 11 Cpp Component simulation results



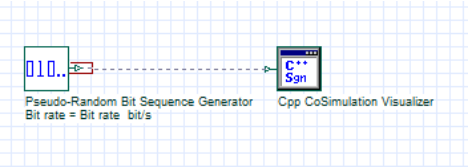
Debug mode of operation

One cannot directly enter the debugger of their component project from OptiSystem because OptiSystem is built in release mode (which is why the previous dll was also built in release mode). OptiSystem provides a solution to this problem by duplicating onto text files all the signals that are to enter into the **Cpp Component.** These text files can then be loaded into the component project when running in debug mode. In this case, no modifications to the code in **Calculate_API** are required. Below we explain how to create the appropriate data files for the component and how to load them through the debugger.

Part 1: Creating an OptiSystem project to output data

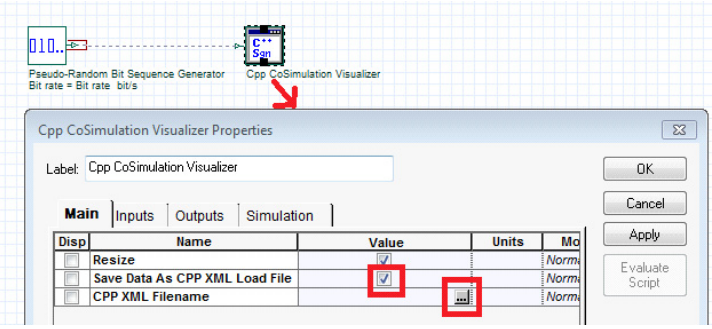
Step	Action
1	**Open** a session of OptiSystem and **create** the following system layout (Fig 12) *The **Cpp Cosimulation Visualizer** can be found under **Default/Visualizer Library/**. This special visualizer serves to duplicate the data that would be sent to the Cpp Component. We must now configure it appropriately.*

Figure 12



Step	Action
2	**Right-click** on Cpp CoSimulation Visualizer and select Properties (Fig 13)

Figure 13



3 Make sure ***Save Data As CPP XML Load File*** is checked (Fig 13)

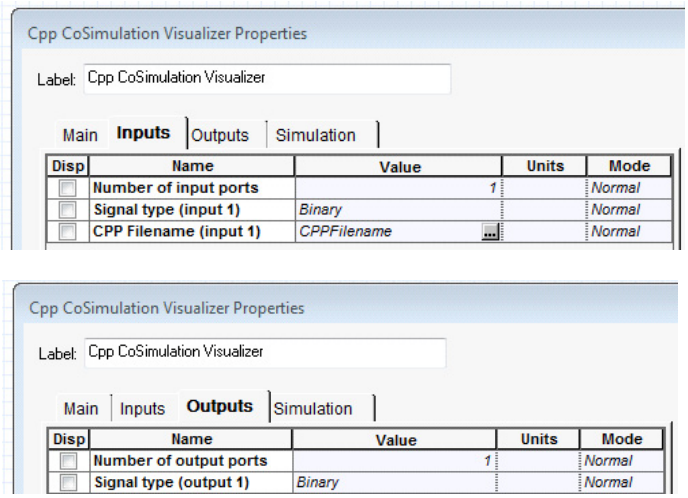
4 Choose a location and Filename for the XML file by clicking on the grey box. In this example we will call the file *testXML.txt* and save it at *[your Cpp project location]\CppCoSimulation\testXML.txt*

The XML file is a text file in XML format that will inform the component about all the ports and the locations of the data files.

5 Set the input and output ports to be exactly the same as the **Cpp Component** (binary) - Fig 14

At the present time, the output port is not necessary, but we have included it for future functionality

Figure 14



6 On ***CPP Filename*** click on the grey box and select a name and location for the binary signal data.

*For this example you could choose **[your Cpp project location]\CppCoSimulation\CPPFilename.txt]***

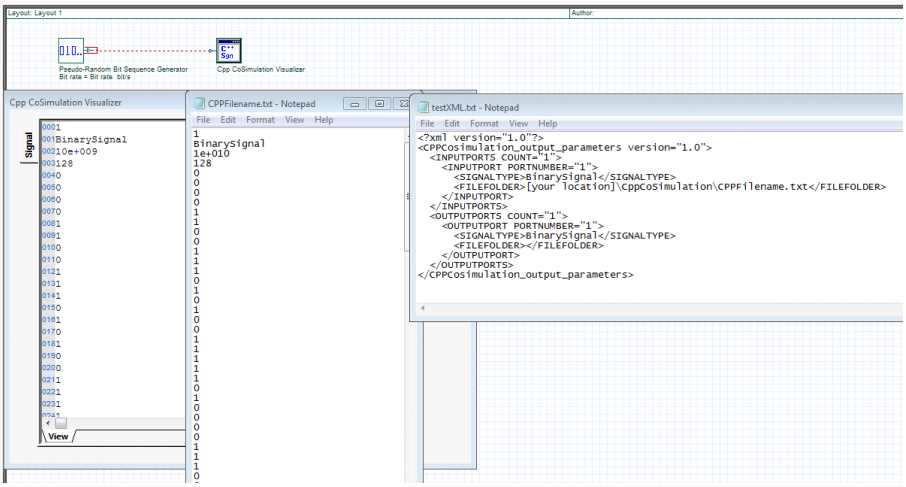
Note: You won't have to enter this filename in your code explicitly as this information will be recorded in the XML file of step 4.

7 **Run** the program

If you double click on the visualizer, you will see the data for the port displayed in a format similar to the view signal visualizer. There is an arrow to scroll through the data on each port. The data you see on this visualizer is duplicated in the data file specified above and the XML file holds the information about the ports. These data files are shown in Fig 15.

Note: Do not manipulate these data files, they are auto generated by OptiSystem

Figure 15



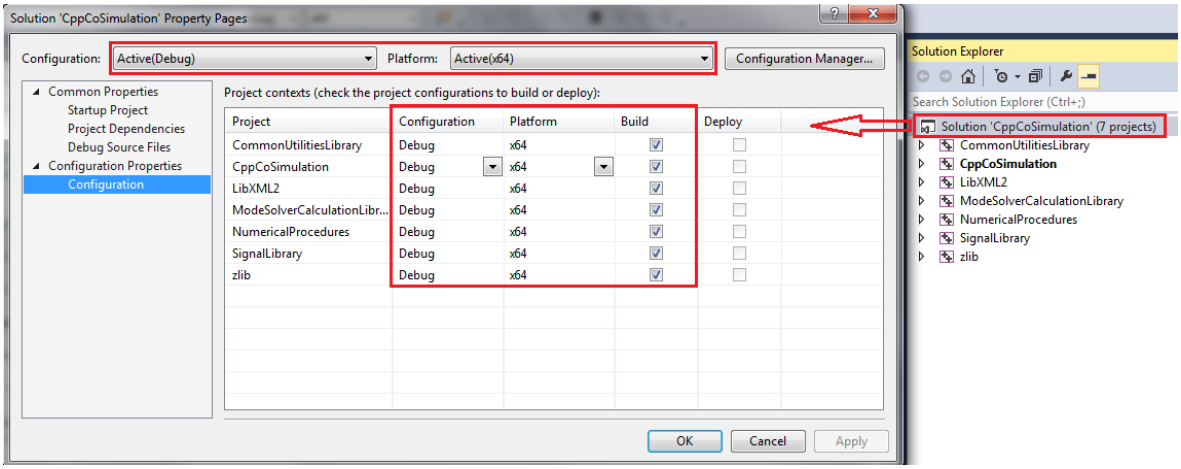
Part 2: Creating a standalone ".exe" for debugging

Step	Action
1	**Open** CppCoSimulation.sln located in the base directory of the provided project in Visual Studio 2013
2	**Set/Verify** that the project is set to Debug x64 mode (Fig 16)

Figure 16 Properties/Configuration Properties/General

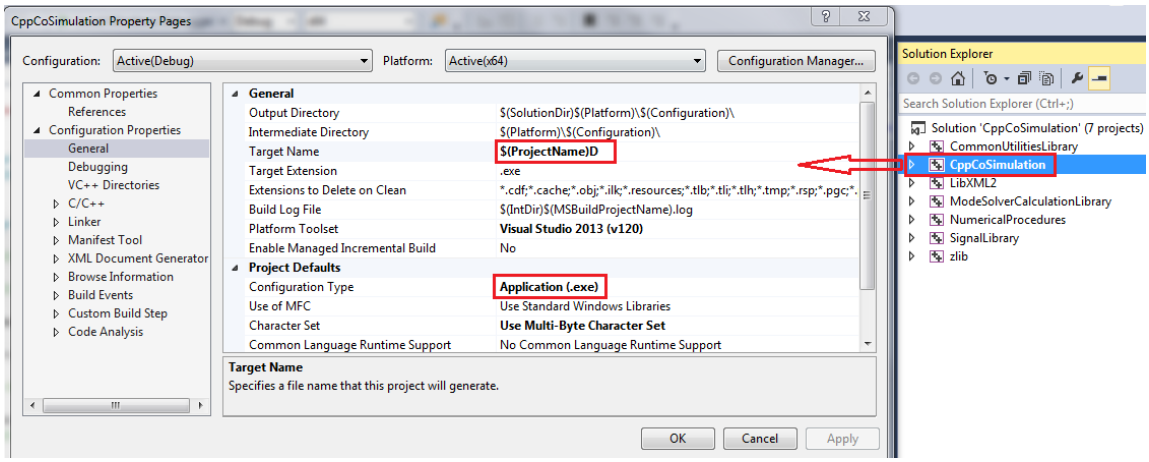
3 **Right-click** on *Solution 'CppCoSimulation'*, **Select** *Properties/Configuration Properties*. **Set/Verify** all projects are configured for Debug x64 and the build boxes are checked (Fig 17)

Figure 17



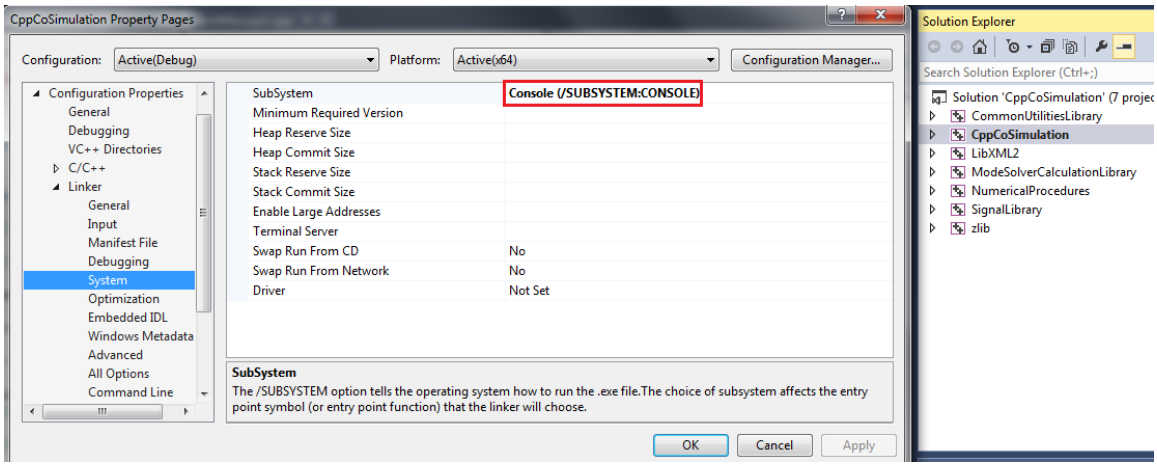
4 **Right-click** on **CppCoSimulation**. Under Configuration Properties/General, **Set/Verify** that Configuration Type is Application (see Fig 18). You may also change the Target Name from the default (which will name the resulting executable as "CppCoSimulationD.exe").

Figure 18



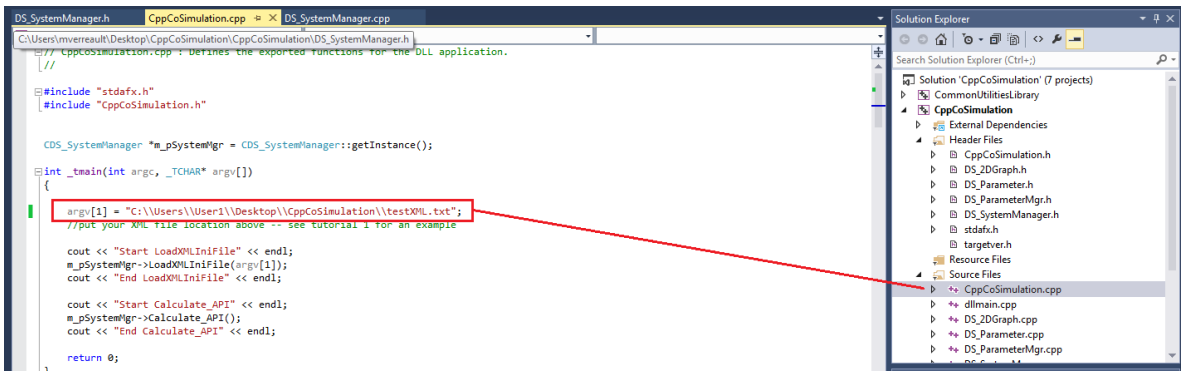
5 Under *Configuration Properties/Linker/System*, **Set/Verify** that SubSystem is set to Console (See Fig 19).

Figure 19



6 Under CppCoSimulation/Source Files, **left double click** on **CppCoSimulation.cpp**

Figure 20 CppCoSimulation.cpp



7 Within this file (see Fig 20), **navigate** to the function “int _tmain(int argc, _TCHAR* argv[])”. This is the starting point for your stand alone configuration. **Add** the following line to your code:

```
argv[1] = “[your Cpp project location]\\CppCoSimulation\\testXML.txt”;
```

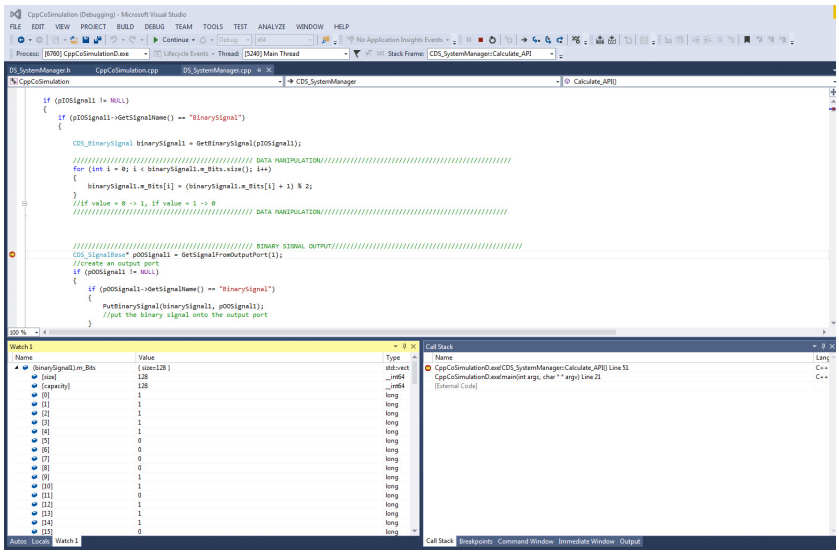
Note: You will have to modify this to your file location. The above would be the case if a user named “testUser” had placed the CppCoSimulation project on their Desktop and had saved the testXML.txt file at this location. Note the use of “\\” instead of “\”, which is necessary because the C++ string class treats “\” as a special character. The “_tmain” function is ignored when we build as a dll, so there is no need to remove it if you switch back to release and dll mode

8 **Rebuild** your solution

9 **Set** the break points and execute

Fig 21 shows a breakpoint set in Calculate_API just after the manipulation of the data. One can see that we can now access all the data and class information in the debugger

Figure 21 Break point set in Calculate_API



End of Tutorial 1

Tutorial 2: Basic Manipulation of Mary Signals

In this tutorial, the user is introduced to the Mary signal class. In addition, it will be demonstrated how to read in global parameters, define and read in local parameters, and create results and graphs

Release mode of operation

Part 1: Create the ".dll" for the component

Step	Action
1	**Verify** the release configuration of the project as with Tutorial 1
2	**Open DS_SystemManager.cpp** and go to function *Calculate_API()*
	Loading port data
3	**Copy** the following lines into the function

```
////////////////////////////////////////////////MARY SIGNAL LOADING////////////////////////////////////////////////
CDS_SignalBase* pIOSignal1 = GetSignalFromInputPort(1);
//Load the data from the first input port. At this point we are just loading it into our parent data class:CDS_Signal-
Base
if (pIOSignal1 != NULL)
{
if (pIOSignal1->GetSignalName() == "MarySignal")
{
```

The first line loads the signal from the first (and in this case only) port. This is exactly the same as in Tutorial 1 The IF conditions have been added as a safety precaution. This will ensure that the calculation will only go ahead if the input port is defined with the proper signal type.

Getting binary data

4	**Copy** the following lines into the function

```
CDS_MarySignal marySignal1 = GetMarySignal(pIOSignal1);
//get Mary signal from input port
```

Similar to the binary, the Mary signal class holds both the sequence of symbols (vector<double> marySignal1.m_Symbols) and the symbol rate (marySignal1.GetSymbolRate()).

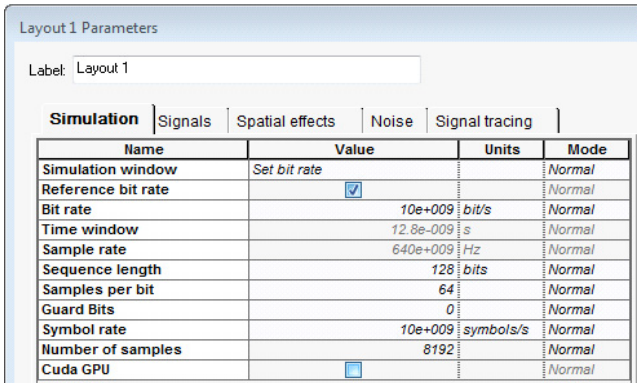
Get the global parameters

5	In this project, as a simple demonstration, we will load the global layout parameter of the sequence length and then just redisplay it on our local results. Any of OptiSystem's layout parameters can be accessed by name using the appropriate function. For example, to access the layout parameters listed in Fig 1 we could write

```
double dBitRate = GetGlobalParameterDouble("Bit rate");
```


```
double dTimeWindow = GetGlobalParameterDouble("Time window");
double dSampleRate = GetGlobalParameterDouble("Sample rate");
long nSequenceLength = GetGlobalParameterLong("Sequence length");
long nSamplesPerBit = GetGlobalParameterLong("Samples per bit");
long nGuardBits= GetGlobalParameterLong("Guard Bits");
double dSymbolRate = GetGlobalParameterDouble("Symbol rate");
long nNumberOfSamples= GetGlobalParameterLong("Number of samples");
bool bCudaGPU= GetGlobalParameterBool("Cuda GPU");
```

Figure 1



Note: For double precision parameters we use "GetGlobalParameterDouble", for long parameters we use "GetGlobalParameterLong" and for booleans we use "GetGlobalParameterBool"

6 **Copy** the following lines into the function

```
////////////////////////////////////////////////////////////////////////////////////////////
//Get global parameter
long nSequenceLength = GetGlobalParameterLong("Sequence length");
//get the sequence length from the layout. The data is in long format.
```

Get the user defined component parameters

When we set up the OptiSystem project (to be shown later), we will define a double precision component parameter called "Multiplication Factor". Here we will write the code to access this. It is very similar to obtaining the global parameters above except the function name is now "GetParameterDouble" for doubles, "GetParameterLong" for longs and "GetParameterBool" for booleans.

7 **Copy** the following lines into the function

```
////////////////////////////////////////////////////////////////////////////////////////////
//Get user defined parameter
double dMultiplicationFactor = GetParameterDouble("Multiplication Factor");
```

Manipulating data

Now that the data has been placed on a vector, and we have read in our parameters we can manipulate to our requirements. In this case we are simply multiplying the Mary data by the defined multiplication factor.

8 **Copy** the following lines into the function

```
////////////////////////////////////////////////////////////////////////////////////////////////////
//Data manipulation
////////////////////////////////////////////////////////////////////////////////////////////////////
//multiply MARY elements by given factor
for (int i = 0; i < marySignal1.m_Symbols.size();i++)
{
marySignal1.m_Symbols[i] *= dMultiplicationFactor;
}
```

Adding a result

Adding a result, which will be displayed in OptiSystem, is very simple. Using the command below, the value of the variable *nSequenceLength* will be set to the name “Global sequence length”

9 **Copy** the following lines into the function

```
////////////////////////////////////////////////////////////////////////////////////////////////////
//Add a result
////////////////////////////////////////////////////////////////////////////////////////////////////
AddResult("Global sequence length", nSequenceLength);
```

Create a 2D graph in OptiSystem

Creating a graph is a three step process. The user must: allocate and define the graph, create data arrays for the axis data and set this data to the graph. In this case we are simply plotting the manipulated Mary data.

10 **Copy** the following lines into the function

```
////////////////////////////////////////////////////////////////////////////////////////////////////
//Add a graph
////////////////////////////////////////////////////////////////////////////////////////////////////
Allocate2DGraph("Mary sequence");
CDS_2DGraph* pGraph1 = Get2DGraph("Mary sequence");
//allocates the graph, the name Mary sequence is simply a unique identifier for this graph. Choose any name you
wish.

pGraph1->SetGraphTitle("Mary values versus sequence element");
pGraph1->SetTitleX("Mary Element");
pGraph1->SetTitleY("Mary Value");
//sets up title and axis information

DOUBLEVECTOR arrX1;  //x coordinate
DOUBLEVECTOR arrY1;  //y coordinate
//**Note** data to be graphed must be vectors of doubles

arrX1.resize(marySignal1.m_Symbols.size());
arrY1.resize(marySignal1.m_Symbols.size());
```


```
for (int i = 0; i < marySignal1.m_Symbols.size(); i++)
{
arrX1[i] = (double)(i + 1);
arrY1[i] = marySignal1.m_Symbols [i];
}
//places the x and y axis data onto the appropriate vectors

pGraph1->SetXData(arrX1);
pGraph1->SetYData(arrY1);
//adds the axis data to the graph.
```

Create an output port

11 **Copy** the following lines into the function

```
////////////////////////////////////////////// MARY SIGNAL OUTPUT//////////////////////////////////////////////////
CDS_SignalBase* pOOSignal1 = GetSignalFromOutputPort(1);
//create an output port
if (pOOSignal1 != NULL) //safety
{
if (pOOSignal1->GetSignalName() == "MarySignal") //safety
{

{
```

This will create an output port for our data for OptiSystem to access.

Set the output signal data sequence

12 **Copy** the following lines into the function

```
PutMarySignal(marySignal1, pOOSignal1);
//puts the mary signal onto the output port
////////////////////////////////////////////// MARY SIGNAL OUTPUT//////////////////////////////////////////////////
}
else
{
return 0;
//returns warning to OptiSystem -- output is not set correctly to binary
} //if (pOOSignal1->GetSignalName() == "MarySignal")
} //if (pOOSignal1 != NULL)
else
{
return 0; //returns warning to OptiSystem -- output put not created
}
}
}
```

The above is to close the brackets for our safety precautions

Note: The full code and an OptiSystem project for this function (*Tutorial2SupplementaryFiles.zip)* is available at the Optiwave C++ Component Home Page. Replace the **DS_SystemManager.cpp** file in your project with the one given in the zip file. If you want to run in debug, you must also change the location of the XML given in **CppCoSimulation.cpp**. In addition change the

location of the files in the OptiSystem projects to your particular files and directories

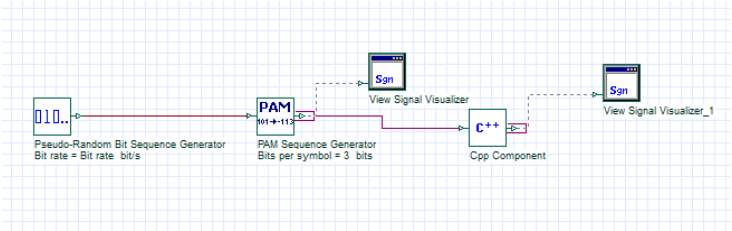
Part 2: Creating an OptiSystem project to use the component

Step **Action**

Verification of project configurations

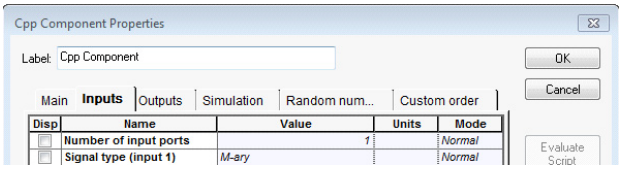
1 **Open** a session of OptiSystem and **create** the following system layout (Fig 2)
The PAM Sequence Generator is used to create the Mary signal.

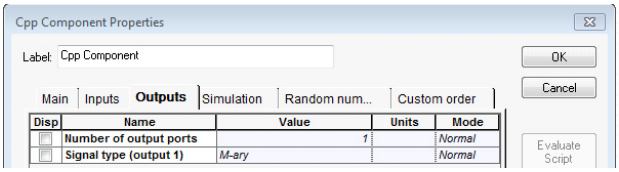
Figure 2 Cpp component setup in OptiSystem



2 **Set** the input and output ports to M-ary (see Fig 3)

Figure 3 Cpp Component port settings

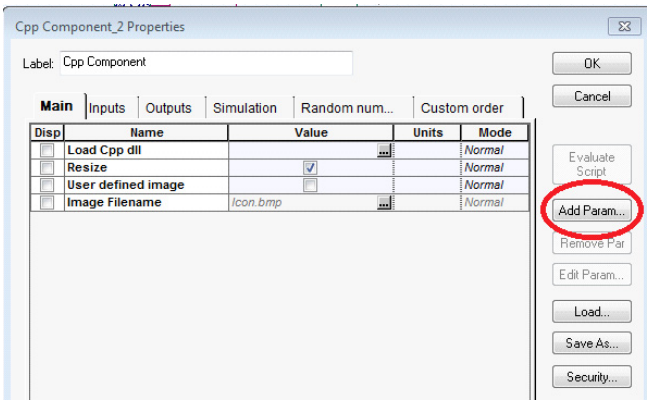




Add the component parameter

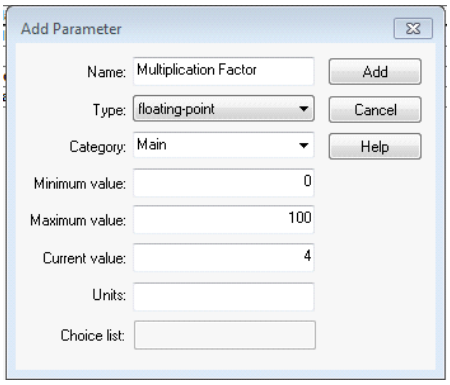
3 We must now add the parameter “Multiplication Factor” that we will access in the code. Under the main tab, **Click** on Add Parameter (Fig 4)

Figure 4



4 A dialog window will pop up which we add the values as below

Figure 5



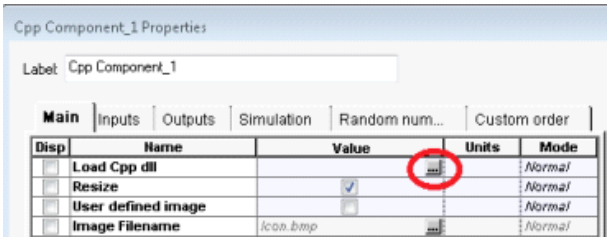
The type floating-point will be read in as a double in our code. This parameter can now be accessed on the main tab of our component.

Note: It is important to make sure the variable type defined in the component (as shown in Fig 5: “floating-point”) is opened by the correct function (GetParameterDouble (“Multiplication Factor”)); otherwise the value will return zero in your code.

5 **Load** the dll

Note: If you kept your folder name as CppCoSimulation and the target name as the default, your dll file will be at ***[your location]\CppCoSimulation\x64\Release\CppCoSimulation.dll***

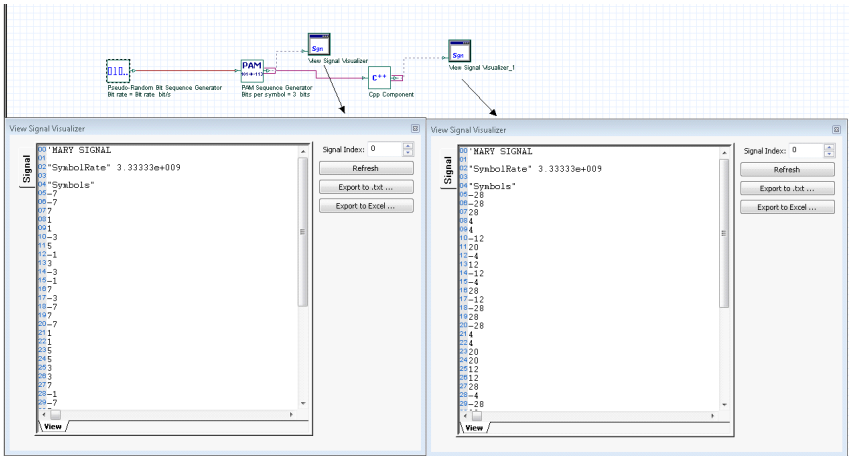
Figure 6



6 **Run** the project

The configuration of the project is now complete. If you now run your simulation, you should obtain results where the output data has values four times that of the input.

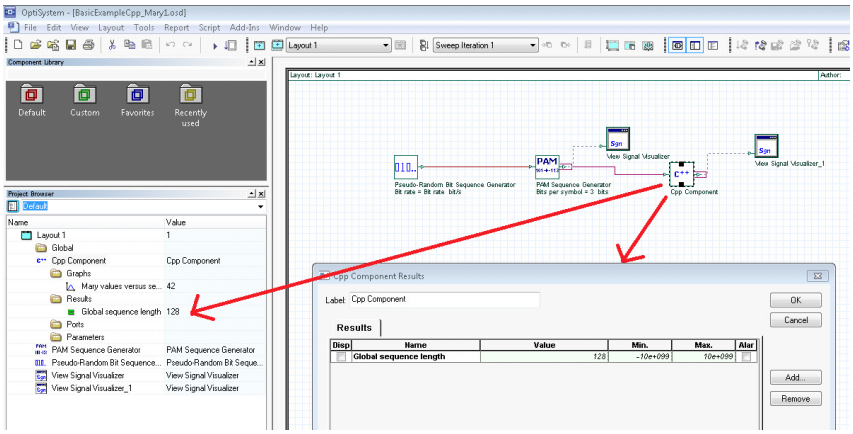
Figure 7



Component Results

The added result “Global sequence length” can be accessed just like the built-in components: right-clicking on the component and selecting “Component results” as well as listed in the project browser.

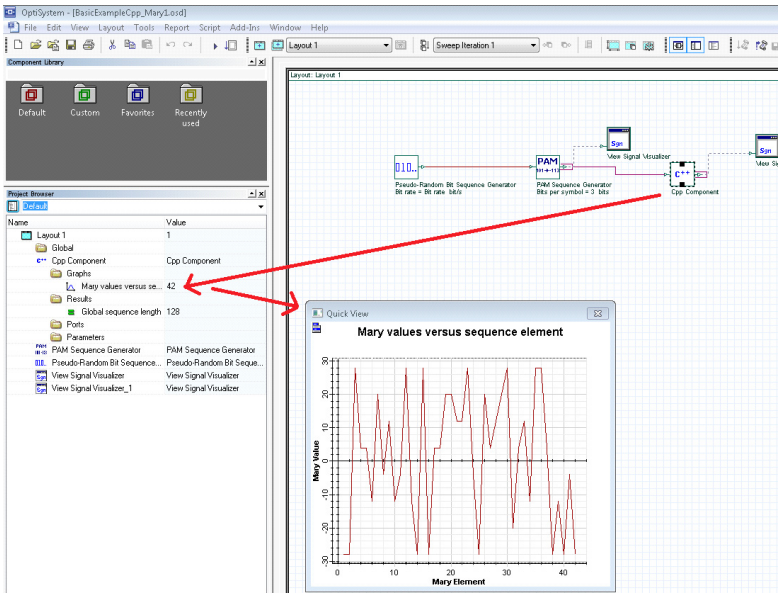
Figure 8



Component Graphs

The graph we created for the Mary sequence can also be accessed by the standard method in the component browser.

Figure 9



Debug mode of operation

The project configuration for debug mode is the same as in Tutorial 1. However, care must be taken in this case. In debug mode, we cannot retrieve any parameters defined for the Cpp Component in OptiSystem because we are running in standalone mode. In this case, we cannot obtain the parameter “Multiplication Factor” using:

```
double dMultiplicationFactor = GetParameterDouble(“Multiplication Factor”);
```

We will have to instead define it explicitly as

```
double dMultiplicationFactor  = 4;
```

Instead of having to comment out and retype code every time you switch from Release to Debug mode, we use the binary flag determined from the predefined project configuration set at the beginning of Calculate_API:

```
#ifdef _EXE_CONSOLE
m_bInExecutableMode = true;
#else
m_bInExecutableMode = false;
#endif
// this flag can be useful for some parameters to be adjusted depending on the mode
```

We can then replace the code:

```
double dMultiplicationFactor = GetParameterDouble("Multiplication Factor");
```

With:

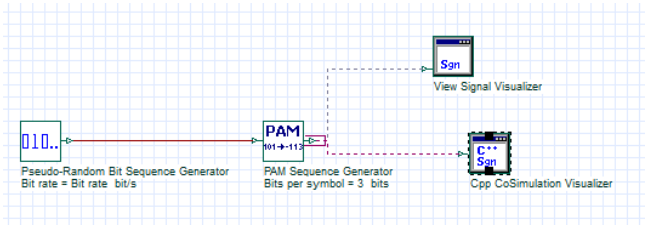
```
if (!m_bInExecutableMode)
{
dMultiplicationFactor = GetParameterDouble("Multiplication Factor");
}
else
{
dMultiplicationFactor = 4.0;
}
```

With this condition, when compiling in release/dll mode we obtain the parameter from the component. When compiling in debug/exe mode we set the parameter explicitly. It is recommended that the user utilizes this method for all definitions where they need to redefine them between release and debug modes. For the other steps, they are very similar to tutorial 1

Step	Action
1	**Switch** to debug mode

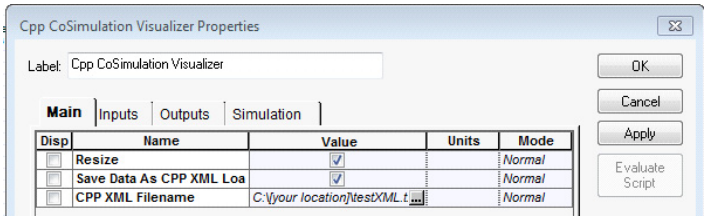
2 **Create** a system with a Cpp Cosimulation Visualizer

Figure 10



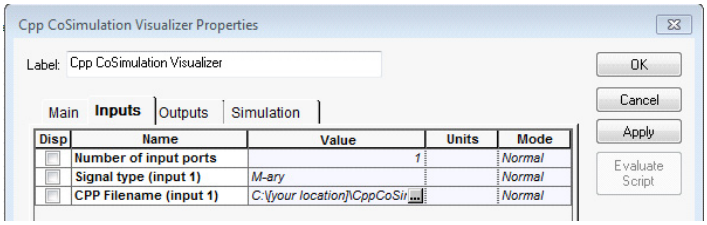
3 **Choose** the appropriate location for an XML file (for example "[your Cpp project location]\CppCoSimulation\testXML.txt"

Figure 11



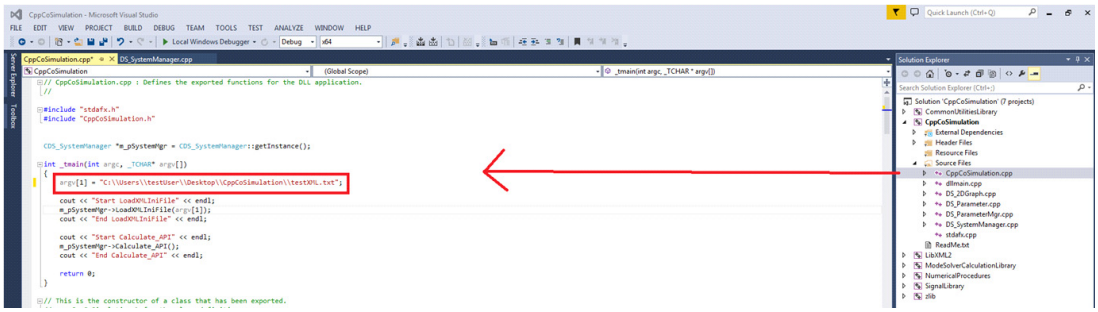
4 Make sure in this case that the input port is M-ary

Figure 12



5 Put the appropriate XML location into “int_tmain(...)”

Figure 13



Plotting intermediate graphs with gnuplot

One can create intermediate graphs simply in the C++ component with predefined convenience functions if gnuplot is installed (please go to http://sourceforge.net/projects/gnuplot/files/gnuplot/ for the latest build). The predefined function prototypes can be seen in DS_SystemManager.h

```
////////////////////////////////////////////////////////////////////////////////
//Convenience plotting functions using gnuplot
//In order for these to work, you must have gnuplot for windows installed on your computer
////////////////////////////////////////////////////////////////////////////////
void gnuplot2D(DOUBLEVECTOR x, DOUBLEVECTOR y, string title = "", string legend = "", string xlabel = "", string ylabel = "");
void gnuplot2D(DOUBLEVECTOR x1, DOUBLEVECTOR y1, DOUBLEVECTOR x2, DOUBLEVECTOR y2, string title = "", string legend1 = "", string legend2 =
"", string xlabel = "", string ylabel = "");
void gnuplot3DTopViewRealVector(DOUBLEVECTOR x, DOUBLEVECTOR y, DOUBLEVECTOR z, string title = "", string xlabel = "", string ylabel = "");
void gnuplot3DSurfaceRealVector(DOUBLEVECTOR x, DOUBLEVECTOR y, DOUBLEVECTOR z, string title = "", string xlabel = "", string ylabel = "");
void gnuplot3DTopViewTransverseMode(CNDS_OpticalTransverseMode mode, string type = "abs", string title = "");
void gnuplot3DSurfaceTransverseMode(CNDS_OpticalTransverseMode mode, string type = "abs", string title = "");
```

The 3D plotting functions will be discussed in a later tutorial. Here we have created two overloaded 2D plotting functions. The first will plot one set of (x,y) data which must be in double vector format. The second will plot two sets of (x,y) data against each other. You may add titles and labels if desired.

For example, we plot the output M-ary data using the gnuplot 2D function:

In this case, we already created the appropriate data structures when we created the OptiSystem graphs. We used arrX1 for the x coordinate data and arrY1 for the y coordinate data:

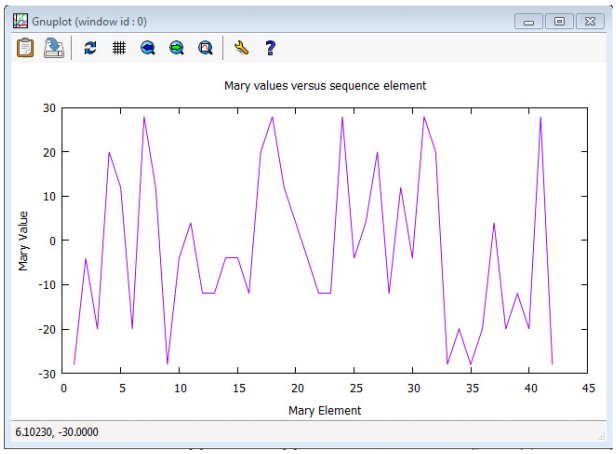
```
pGraph1->SetXData(arrX1);
pGraph1->SetYData(arrY1);
//adds the axis data to the graph.
```

Therefore after these we can simply add the lines

```
if (m_bInExecutableMode)
gnuplot2D(arrX1, arrY1, “Mary values versus sequence element”, “”, “Mary Element”, “Mary Value”);
//in executable mode plot the graph with gnuplot
```


which will duplicate the graph we created for the component in gnuplot when the program is run

Figure 14



The “if (m_bInExecutableMode)” statement has been added so that the graph doesn't show up when running in dll mode.

End of Tutorial 2

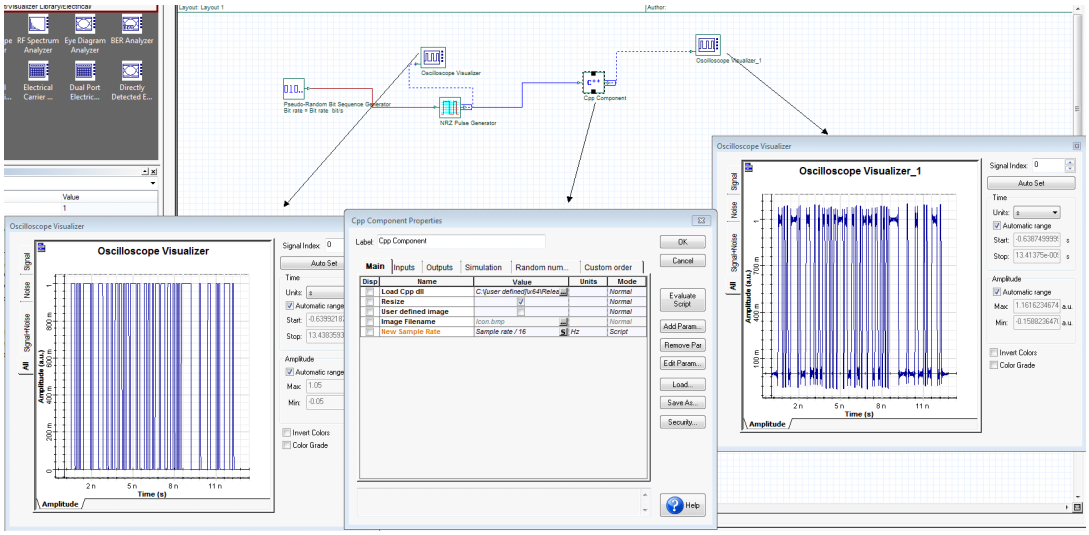
Tutorial 3: Basic Manipulation of Electrical Signals

Note: For this tutorial it is expected the user will copy the code provided in the supplementary files. Here we only describe the key elements of this code.

In this tutorial we introduce the electrical signal class and demonstrate how one can use our built-in functions of the class

The OptiSystem project and results are shown below. We re-sample the electrical signal to the defined parameter “New Sample Rate” which we have set as “Sample rate / 16", which in this case will be 40GHz

Figure 1



Note: The full code and an OptiSystem project for this function (*Tutorial3SupplementaryFiles.zip)* is available at the Optiwave C++ Component Home Page. Replace the **DS_SystemManager.cpp** file in your project with the one given in the zip file. If you want to run in debug, you must also change the location of the XML given in **CppCoSimulation.cpp**. In addition change the location of the files in the OptiSystem projects to your particular files and directories.

Description of the code

Load the electrical signal

```
////////////////////////////////////////////// ELECTRICAL SIGNAL LOADING//////////////////////////////////////////////
CDS_SignalBase* pIOSignal1 = GetSignalFromInputPort(1);
//Load the data from the first input port. At this point we are just loading it into our parent data class:CDS_SignalBase
if (pIOSignal1 != NULL) //safety
{
        if (pIOSignal1->GetSignalName() == "ElectricalSignal") //safety
        {

                ////////////////////////////////////////////////////////////////////////////////////////////////////////////
                ////////////////////////////////////// Get Electrical Signal from Input Port 1 ////////////////////////////////////////
                CNDS_ElectricalSampledSignal electricalSampledSignal = GetElectricalSampledSignal(pIOSignal1);
                ////////////////////////////////////// Get Electrical Signal from Input Port 1 ////////////////////////////////////////
                ////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

The base class for the electrical sampled signals is **CNDS_ElectricalSampledSignal** which we obtain from the port using the *GetElectricalSampledSignal(pIOSignal1)*; function. Other electrical signals the user can access are electrical sampled noise and electrical individual samples (used in feedback applications). These will be discussed in future tutorials. Refer to **SignalLibrary/NDS_ElectricalSampledSignal.h** for all the data and functions available for this class. For example, the data stored in this class is:

```
// enum types
enum enumSignalDomain
{
        domainTime,
        domainFrequency
};

// signal domain = Time or Frequency
long            m_nDomain;
// SAMPLE RATE
double          m_dSampleRate;
// ELECTRICAL FIELD (TIME DOMAIN)
// vectors with signal amplitude components in the REAL part:
COMPLEXVECTOR   m_arrAmplitude;
```

The complex amplitude data is stored in *m_arrAmplitude*. Note that the information can be defined in the time or frequency domain. Initially, all information (from OptiSystem in release mode or data files in debug mode) is read in the frequency domain. To switch domains, use the commands:

```
electricalSampledSignal.SetDomain(CNDS_ElectricalSampledSignal::domainTime);
```

or

```
electricalSampledSignal.SetDomain(CNDS_ElectricalSampledSignal::domainFrequency);
```

depending on your requirement. The *SetDomain* function will first check which domain the signal is currently in. If it is already in the domain required, no further

action will be performed. If it is different the appropriate FFT/IFFT will be applied on *m_arrAmplitude*.

Set data vector for signal before resampling (testing purposes in gnuplot)

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//create vector of data before resampling for gnuplot2D
electricalSampledSignal.SetDomain(CNDS_ElectricalSampledSignal::domainTime);
DOUBLEVECTOR vecXBeforeResample, vecYBeforeResample;

for (int j = 0; j < electricalSampledSignal.GetSize(); j++)
{
		vecXBeforeResample.push_back(j * GetGlobalParameterDouble("Time window") / electricalSampledSignal.GetSize());
		vecYBeforeResample.push_back(real(electricalSampledSignal.m_arrAmplitude[j]));
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

We will be plotting a graph in gnuplot to compare the real part of the signal (in this case there is no phase rotation and so no imaginary part) before and after resampling. Here we set up the vector for the before data.

We first make sure the signal is in the time-domain so that when we access *m_arrAmplitude*. Then

```
vecXBeforeResample.push_back(j * GetGlobalParameterDouble("Time window") / electricalSampledSignal.GetSize());
```

sets the x-coordinate (time points vector). This gives the appropriate time divisions over the total time window. Next the command

```
vecYBeforeResample.push_back(real(electricalSampledSignal.m_arrAmplitude[j]));
```

sets the vector of amplitude data, which is in the time domain

Set new sample rate

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Read parameter data entered into Cpp component interface in OptiSystem if in dll mode or set value if in exe mode
////////////////////////////////////////////////////////////////////////////////////////////////////////////////
double dSampleRate;
if (m_bInExecutableMode)
{
		dSampleRate = 40.0e9;
}
else
{
		dSampleRate = GetParameterDouble("New Sample Rate");
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

Recall that we have defined in the OptiSystem component the parameter “New Sample Rate” and if we are in debug mode we must define it explicitly as 40GHz (or whichever value is desired).

Resampling

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// resample signal using built-in routines
electricalSampledSignal.DownsampleToFrequencyLimits(-dSampleRate / 2.0, dSampleRate / 2.0);
//Cut the signal to this frequency range you can take advantage of the built-in resampling routines in CNDS_ElectricalSampledSignal
//but this is not necessary if you have your own routines that you wish to use.
long nSize = PowerOfTwo(electricalSampledSignal.GetSize());
electricalSampledSignal.Resample(nSize);
//Resample to the newly defined sample rate over the time window
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

We wish to re-sample the data to the new sample rate. This is performed by our built in class functions

DownsampleToFrequencyLimits this removes any of the higher frequency components.

A restriction in OptiSystem is that all signals must be 2n samples long. The next two commands

```
long nSize = PowerOfTwo(electricalSampledSignal.GetSize());
electricalSampledSignal.Resample(nSize);
```

make sure this the case. Either cutting the frequency range slightly or zero padding it.

Set data vector for signal after resampling (testing purposes in gnuplot) and plotting

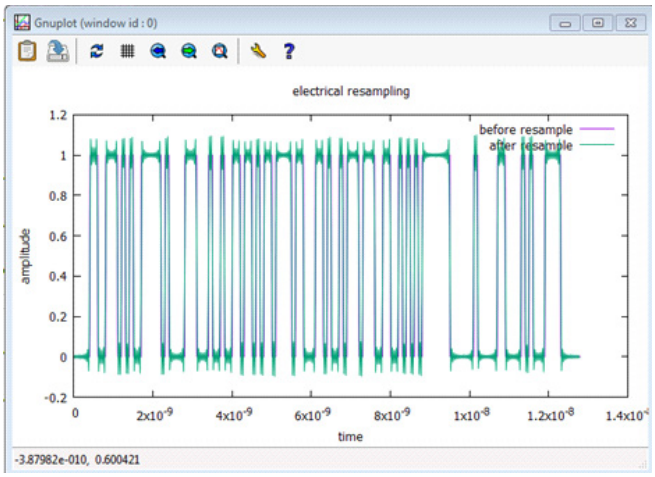
```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//create vector of data after resampling for gnuplot2D
electricalSampledSignal.SetDomain(CNDS_ElectricalSampledSignal::domainTime);
DOUBLEVECTOR vecXAfterResample, vecYAfterResample;
for (int j = 0; j < electricalSampledSignal.GetSize(); j++)
{
                vecXAfterResample.push_back(j * GetGlobalParameterDouble("Time window") / electricalSampledSignal.GetSize());
                vecYAfterResample.push_back(real(electricalSampledSignal.m_arrAmplitude[j]));
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

The above is similar to the creation of the data vectors for before resampling. The data will still be over the same time window, but now the time-divisions will be different.

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// plot data
if (m_blnExecutableMode)
        gnuplot2D(vecXBeforeResample, vecYBeforeResample, vecXAfterResample, vecYAfterResample, "electrical resampling",
                "before resample", "after resample", "time", "amplitude");
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

The data is plotted from the call above. The figure below shows the result of this call in gnuplot (Notice that this is the same result as we see in the OptiSystem visualizers)

Figure 2



Place electrical sampled signal on the output port

```
CDS_SignalBase* pOOSignal1 = GetSignalFromOutputPort(1);
//create an output port
if (pOOSignal1 != NULL) //safety
{
        if (pOOSignal1->GetSignalName() == "ElectricalSignal") //safety
        {
        ////////////////////////////////////////////////////////////////////////////////////////////////////////////////
        ///// create electrical signal Output Port 1 /////
                PutElectricalSampledSignal(electricalSampledSignal, pOOSignal1);
        }
        else
        {
                return 0;
                //returns warning to OptiSystem -- output is not set correctly to electrical
        } //if (pOOSignal1->GetSignalName() == "ElectricalSignal")
} //if (pOOSignal1 != NULL)
//////////////////////////////////////////// ELECTRICAL SIGNAL OUTPUT////////////////////////////////////////////////
else
{
        return 0; //returns warning to OptiSystem -- output not created
}
```

The output port pOOSignal1 is created the same way as in the first two tutorials.

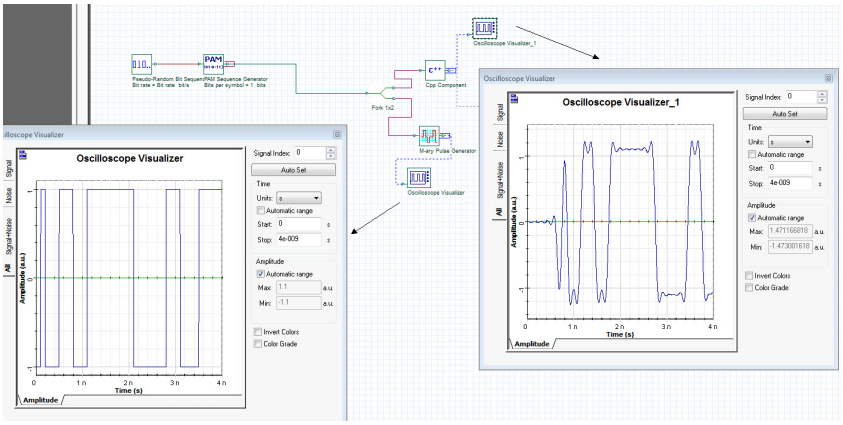
End of Tutorial 3

Tutorial 4: Electrical pulse shaper with M-ary input

Note: For this tutorial it is expected the user will copy the code provided in the supplementary files. Here we only describe the key elements of this code.

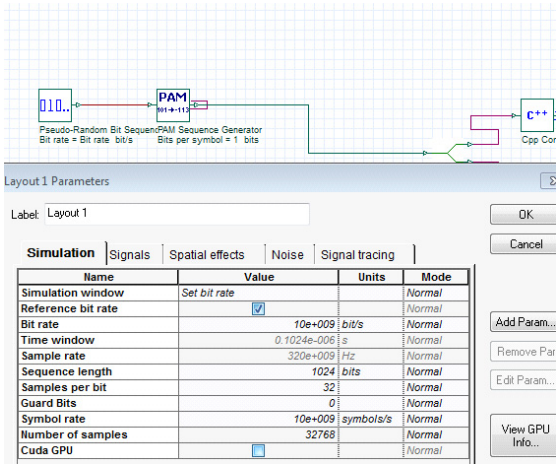
OptiSystem already has a number of pulse shapers. In this example we show how to create a custom pulse shaper where the input is an M-ary signal. A pulse shape is determined from a file of filter taps data (the FIR of the particular filter) which is applied to the M-ary input to create a shaped electrical signal output. For this example, we will use the impulse response of a root-raised cosine pulse, but the user can supply any response file they wish. The component will create a pulse shape as shown in Oscilloscope Visualizer_1 below.

Figure 1



For simplicity we set up a two level M-ary system with possible values of {-1,1} only. In this case then, the bit rate will be the same as the symbol rate.

Figure 2



Note: in this example the layout parameter “Symbol rate” will not be used, only “Bit rate” and “Samples per bit”. The symbol rate will be determined with the component from the input M-ary signal.

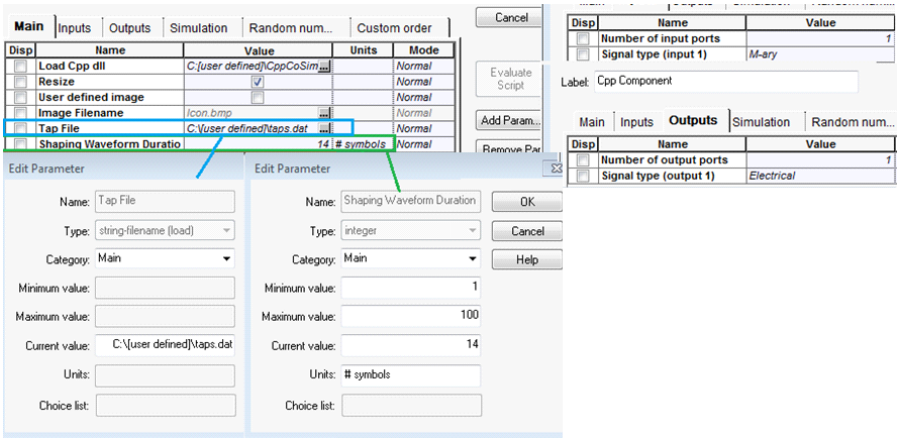
Description of the code

Component setup

Two parameters are added to the component:

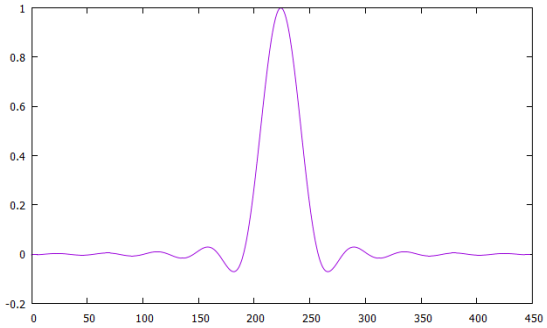
- “Tap File” which is of string-filename (load) type
- “Shaping Waveform Duration” which is the time window of the shaping pulse, but instead of being given in seconds, it is given in terms of # symbols.

Figure 3



The input port is of type “M-ary” and the output is of type “Electrical”. The tap file used in this example represents the impulse root-raised cosine pulse as below. Note that the number of taps = Shaping Waveform Duration * Symbols per bit + 1= 14 * 32 + 1 = 449. This requires knowledge of the symbols per bit. We can calculate this from the bit rate (layout parameter) and the symbol rate (carried in the input Mary signal).

Figure 4



The file is in the form:

-0.00250627

-0.00250402

-0.00245201

-0.00235037

...

where each line represents the value at the next tap.

The time window in seconds is calculated by:

```
double dTimeWindow = (double)GetParameterLong("Shaping Waveform Duration") / dSymbolRate;
//the Time Window is defined in number of symbols
```

Note: The full code, tap file and an OptiSystem project for this function (*Tutorial4SupplementaryFiles.zip)* is available at the Optiwave C++ Component Home Page. Replace the **DS_SystemManager.cpp** and **DS_SystemManager.h** files in your project with the ones given in the zipfile. If you want to run in debug, you must also change the location of the XML given in **CppCoSimulation.cpp**. In addition change the location of the files in the OptiSystem projects to

We have created two functions in the class: *ConvertToImpulses*, which converts the input M-ary signal to a sequence of electrical “impulses” (the first sample in each symbol is the M-ary value). *ApplyTapFunction*, applies the impulse response to our impulse sequence to shape the pulses.

It is important to note in *ConvertToImpulses* that when we created the electrical signal, we not only had to define the amplitudes of the signal sequence, we also had to define the bandwidth and sample rate:

```
electricalSampledSignal.m_dSampleRate = GetGlobalParameterDouble("Sample rate");
electricalSampledSignal.m_Bandwidth.SetBandwidth(electricalSampledSignal.m_dSampleRate);
```

Additionally we made sure this signal was in the time domain first before we defined the amplitudes

```
CNDS_ElectricalSampledSignal electricalSampledSignal;
electricalSampledSignal.SetDomain(CNDS_ElectricalSampledSignal::domainTime);
ConvertToImpulses(electricalSampledSignal, vnMARYData1);
```

In *ApplyTapFunction* we use the relation that the FFT of the convolution is equal to the product of the FFTs of the convolving function. The OptiSystem C++ project provides built-in FFT and IFFT routines based on FFTW. The FFT function is invoked as:

```
fftw_plan plan = m_Plan.CreatePlan(nNumberPointsInFunction, FFTW_FORWARD, FFTW_ESTIMATE);
RunFFTW(plan, function, function_fft);
m_Plan.DestroyPlan(plan);
```

which performs a FFT on function to the resulting function_fft. Both of these must be defined as CComplex pointers like:

```
CComplex* function = new CComplex[nNumberPointsInFunction];
CComplex* function_fft = new CComplex[nNumberPointsInFunction];
```

The IFFT function is invoked as:

```
fftw_plan plan = m_Plan.CreatePlan(nNumberPointsInFunction, FFTW_BACKWARD, FFTW_ESTIMATE);
RunFFTW(plan, function_fft, function);
m_Plan.DestroyPlan(plan);
```

finally, for normalization purposes, after the IFFT, we must divide the resulting function by *nNumberPointsInFunction*.

End of Tutorial 4

Tutorial 5: Binary controlled optical switch.

Note: For this tutorial it is expected the user will copy the code provided in the supplementary files. Here we only describe the key elements of this code.

In this tutorial we introduce the optical sampled signal class by creating a binary-controlled optical switch. The optical signal is sent through an RC filter to simulate a time delay in the switching. The component setup and the results of the project are shown below

Figure 1

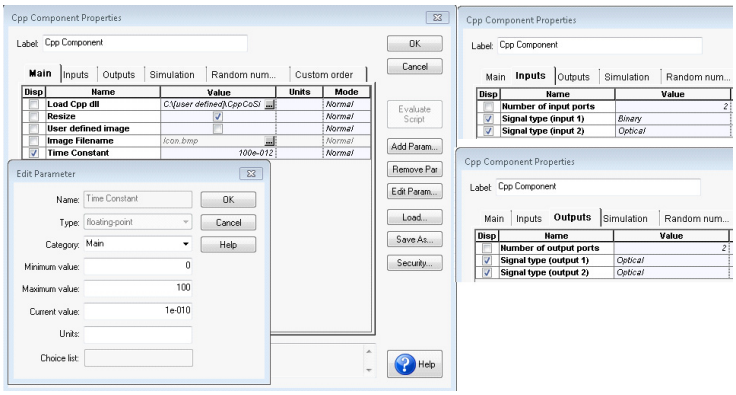
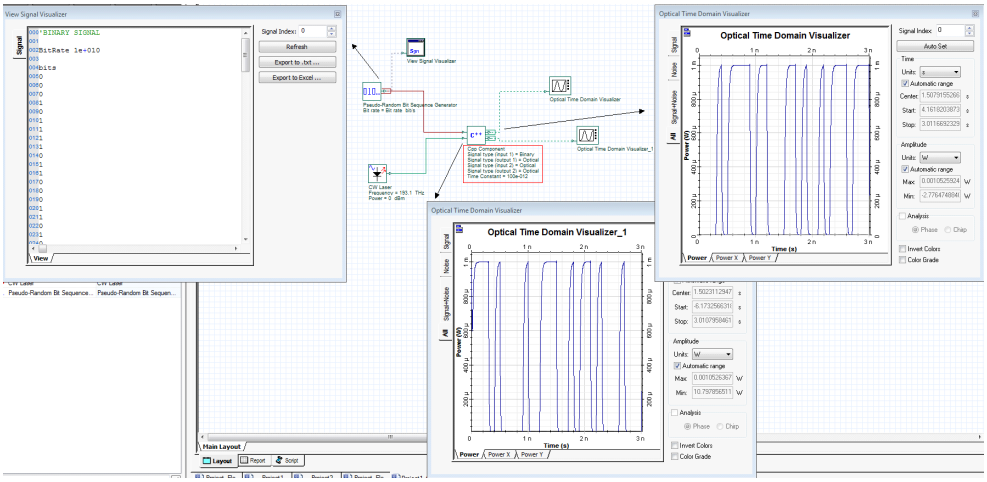


Figure 2



Where the top visualizer is the RC filtered value of the optical signal determined from the binary input and the bottom visualizer RC filters the optical signal from the inverse of the binary input

Optical signals

The optical signal classes are the most complex of all the signal classes. Those to be accessed by the user are:

Sampled signals

CNDS_OpticalSampledSignal, header located in SignalLibrary/NDS_OpticalSampledSignal.h

Individual samples (used for feedback applications)

CNDS_OpticalIndividualSample, header located in SignalLibrary/NDS_OpticalIndividualSample.h

Parameterized signals

CNDS_OpticalParameterizedSignal, header located in SignalLibrary/NDS_OpticalParameterizedSignal.h

Noise bins

CNDS_OpticalNoiseBin, header located in SignalLibrary/NDS_OpticalNoiseBin.h

All other optical classes listed in this directory do not need to be accessed by the user

Domains

```
// signal domain = Time or Frequency
long   m_nDomain;
enum enumSignalDomain
{
    domainTime,
    domainFrequency
};
```

Similarly to the electrical signal, we can convert between the time and frequency domain. The built in function SetDomain will apply the appropriate FFT/IFFT during the conversion

Amplitudes and polarization

```
// vectors with signal amplitude components :
// optical signal in m_arrAmplitudeX - polarizationConstant
// optical signal polarization in X and Y - polarizationArbitrary
 COMPLEXVECTOR    m_arrAmplitudeX;
 COMPLEXVECTOR    m_arrAmplitudeY;
CNDS_Polarization      m_Polarization;

enum enumPolarizationType
{
     polarizationNone,
     polarizationConstant,
     polarizationArbitrary
};
```

The amplitude data is stored in the complex vectors

```
COMPLEXVECTOR m_arrAmplitudeX;
COMPLEXVECTOR m_arrAmplitudeY;
```

The phase information is accounted for by the complex format of the amplitude. How this is stored depends on the polarization format being used. The default format has m_arrAmplitudeX storing the total amplitude, m_arrAmplitudeY is zero and polarization information stored in the Stokes vector class:

```
CNDS_Polarization          m_Polarization;

// Stokes vector (not normalized)
 double m_dS0;
double m_dS1;
double m_dS2;
double m_dS3;
```

We can instead (and we will do this in the example) store the X and Y polarization amplitudes separately on m_arrAmplitudeX and m_arrAmplitudeY respectively. The conversion between the storage modes is obtained by the function:

```
SetPolarizationState(CNDS_OpticalSampledSignal::polarizationConstant);
```

To store as Stokes vectors

```
SetPolarizationState(CNDS_OpticalSampledSignal::polarizationArbitrary);
```

To store as X and Y amplitudes

Bandwidth

The bandwidth information is stored in the parent class (**CNDS_SignalBase**)

```
CNDS_SignalBandwidth m_Bandwidth;
```

which stores

```
double m_dLowerFrequency;
double m_dUpperFrequency;
```

The average between these two frequencies will be the optical signal's carrier frequency and the difference is the bandwidth

Spatial information

In this tutorial, we only consider single mode signals, which do not contain any spatial mode data. Accessing the transverse mode data is not needed in this tutorial and will be discussed in Tutorial 6."

```
CNDS_OpticalTransverseMode  m_ModeX;
CNDS_OpticalTransverseMode  m_ModeY;
```

Note: The full code and an OptiSystem project for this function *(Tutorial5SupplementaryFiles.zip)* is available at the Optiwave C++ Component Home Page. Replace the **DS_SystemManager.cpp** and **DS_SystemManager.h** files in your project with the ones given in the zipfile. If you want to run in debug, you must also change the location of the XML given in **CppCoSimulation.cpp**. In addition change the location of the files in the OptiSystem projects to your particular files and directories.

Key points

1 - While CNDS_OpticalSampledSignal is the class of the optical sampled signal, it is possible to have multiple signals on the same port. This happens because the input signal can be the sum of multiple modulated optical signals on different carrier frequencies and there can also be multiple transverse modes and polarizations. In OptiSystem each different carrier frequency and transverse mode (and in some cases polarization) will be described by a unique CNDS_OpticalSampledSignal. Therefore when optical data is read in from the port, the function call used is

```
VECTOR_OpticalSampledSignals vectorSignal1 = GetOpticalSampledSignalsVector(pIOSignal2);
```

where we defined in the header:

```
typedef std::vector< CNDS_OpticalSampledSignal>     VECTOR_OpticalSampledSignals;
```

For example if we had on the input port a single polarization optical signal that was modulated on two carrier frequencies and each frequency had two transverse modes, vectorSignal1 would be of size 4 and would have elements:

```
vectorSignal1[0]--> CNDS_OpticalSampledSignal for wavelength 1, transverse mode 1
vectorSignal1[1]--> CNDS_OpticalSampledSignal for wavelength 1, transverse mode 2
vectorSignal1[2]--> CNDS_OpticalSampledSignal for wavelength 2, transverse mode 1
vectorSignal1[3]--> CNDS_OpticalSampledSignal for wavelength 2, transverse mode 2
```

In many cases there will only be one mode and therefore vectorSignal1[0] would be the only element.

2 - To manipulate the amplitude data of the optical signal directly, we require that the signals are in time domain format and the X and Y amplitudes carry each polarization separately. Thus we call the functions

```
vectorSignal1[i].SetDomain(CNDS_OpticalSampledSignal::domainTime);
vectorSignal2[i].SetDomain(CNDS_OpticalSampledSignal::domainTime);
//First the domain of each input signal will be set to time domain using the function SetDomain()
vectorSignal1[i].SetPolarizationState(CNDS_OpticalSampledSignal::polarizationArbitrary);
vectorSignal2[i].SetPolarizationState(CNDS_OpticalSampledSignal::polarizationArbitrary);
//The polarization for optical signals can be represented in different ways, here we want separate arrays of X and Y
```

3 - The amplitude information for the optical signal is protected so we must use the accessors functions:

```
CComplex* pEx1 = NULL;
CComplex* pEy1 = NULL;
CComplex* pEx2 = NULL;
CComplex* pEy2 = NULL;

//For each sampled signal and polarization modify the amplitude depending on the binary value to emulate a switch.
for (int i = 0; i < vectorSignal1.size(); i++)
{
        //Getting complex field data from current sampled signal
        pEx1 = vectorSignal1[i].GetDataX();
        pEy1 = vectorSignal1[i].GetDataY();

        pEx2 = vectorSignal2[i].GetDataX();
        pEy2 = vectorSignal2[i].GetDataY();
```

Note that the accessors are passing pointers so any manipulations we do on them will modify the underlying *m_arrAmplitudeX* and *m_arrAmplitudeY* in the class.

4- The optical data is placed on the output port by the function

```
PutOpticalSampledSignalsVector(vectorSignal1, pOOSignal1);
PutOpticalSampledSignalsVector(vectorSignal2, pOOSignal2);
```

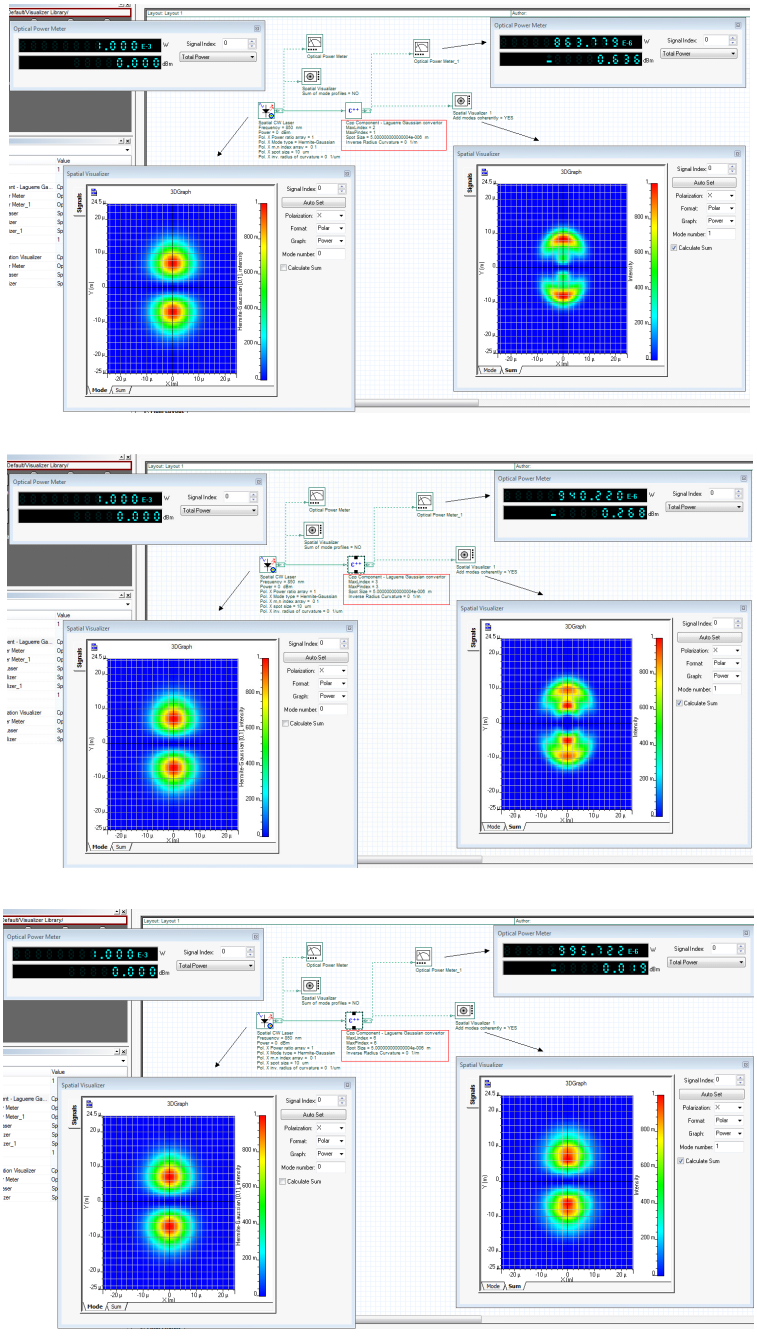
End of Tutorial 5

Tutorial 6: Optical transverse mode converter

Note: For this tutorial it is expected the user will copy the code provided in the supplementary files. Here we only describe the key elements of this code.

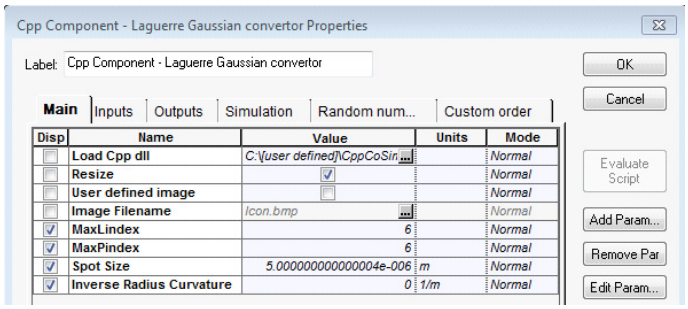
In this example we demonstrate how one can access and manipulate the transverse mode information in the optical sampled signal. We build a transverse mode converter where the input transverse field will be converted to a defined set of Laguerre-Gaussian modes using a predefined overlap integral calculation class. Below we show the case when the input mode is the [0 1] Hermite Gaussian mode. As the figures show, for increasing number of Laguerre-Gaussian modes in the basis, the input transverse field can be more accurately captured.

Figure 1 From top to bottom: LG (MaxL/MaxP: 2/1); LG (MaxL/MaxP: 3/3); LG (MaxL/MaxP: 6/6)



In this case we have added four parameters to the component. The Laguerre-Gaussian mode basis will be all the modes in the range [0...MaxLindex,0...MaxPindex]

Figure 2



Note: The full code and an OptiSystem project for this function (*Tutorial6SupplementaryFiles.zip)* is available at the Optiwave C++ Component Home Page. Replace the **DS_SystemManager.cpp** file in your project with the ones given in the zipfile. If you want to run in debug, you must also change the location of the XML given in **CppCoSimulation.cpp**. In addition change the location of the files in the OptiSystem projects to your particular files and directories.

Transverse modes

The transverse mode information is stored in the **CNDS_OpticalTransverseMode** class. The data in this class is a matrix of complex elements *CDS_ComplexMatrix m_mAmplitude* where the rows (x) and columns (y) represent the coordinates in the transverse plane. The spatial representation of the basis of Laguerre-Gaussian modes are put into the vector *std::vector< CNDS_OpticalTransverseMode> vecTranverseModes*.

Conversion of transverse modes

We convert the original set of transverse modes to the new basis using the convenience class CNP_ChangeTransverseModes:

```
CNP_ChangeTransverseModes changeTransverseModes;
changeTransverseModes.Calculate(vectorSignal1, vecTranverseModes, vecTranverseModes);
```

This will rewrite the transverse modes of our input optical signal (*vectorSignal1*, which were originally Hermite-Gaussian) into the new Laguerre-Gaussian basis given in *vecTranverseModes*. Note that we have passed *vecTranverseModes* twice because they represent the X and Y polarization transverse modes. In this case the two polarizations have identical transverse modes. If you step into the **CNP_ChangeTransverseModes** class, you will see that the conversion is accomplished in a standard way using the overlap integrals between the initial transverse mode and the new basis.

The new transverse modes are then attached to the optical signal

```
sampledSignalTemp.m_ModeX = vecNewTransverseModesX[j];
sampledSignalTemp.m_ModeY = vecNewTransverseModesY[j];
```

where *sampledSignalTemp* will eventually be copied over to an element of *vectorSignal1*.

Debug plotting using gnuplot

We have added some intermediate plotting functionality for 3D plotting

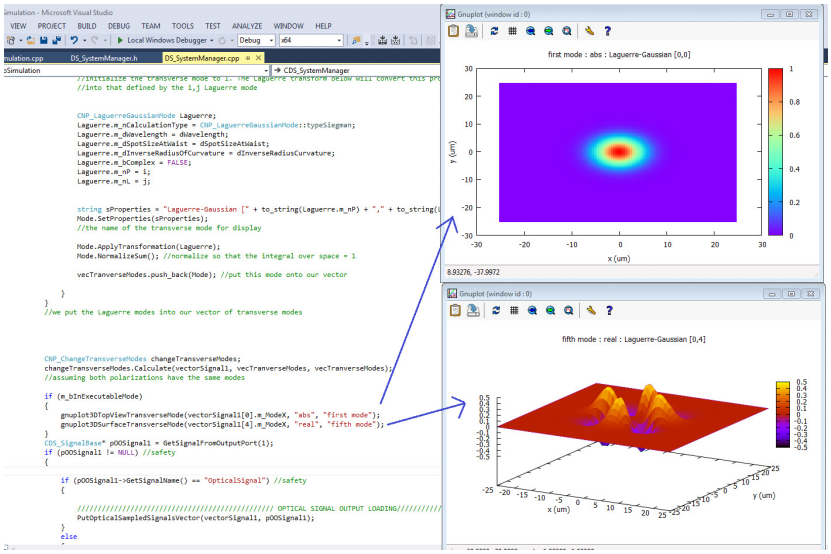
```
void gnuplot3DTopViewRealVector(DOUBLEVECTOR x, DOUBLEVECTOR y, DOUBLEVECTOR z, string title = "", string xlabel = "", string ylabel = "");
void gnuplot3DSurfaceRealVector(DOUBLEVECTOR x, DOUBLEVECTOR y, DOUBLEVECTOR z, string title = "", string xlabel = "", string ylabel = "");

void gnuplot3DTopViewTransverseMode(CNDS_OpticalTransverseMode mode, string type = "abs", string title = "");
void gnuplot3DSurfaceTransverseMode(CNDS_OpticalTransverseMode mode, string type = "abs", string title = "");
```

In the first two functions you directly define the coordinates (x, y) and values (z) to be plotted. For convenience, the second two functions have be written so you can plot the transverse mode directly. The variable string type has types “abs”, “real”, “imag” and “phase” for the part of the complex data you wish to plot. In the above code we plot as an example

```
gnuplot3DTopViewTransverseMode(vectorSignal1[0].m_ModeX, "abs", "first mode");
//plots top view of absolute value of the first transverse mode
gnuplot3DSurfaceTransverseMode(vectorSignal1[4].m_ModeX, "real", "fifth mode");
//surface plot of the real component of the fifth transverse mode
```

Figure 3



End of Tutorial 5

Tutorial 7: Working with optical parameterized signals and noise bins

Note: For this tutorial it is expected the user will copy the code provided in the supplementary files. Here we only describe the key elements of this code.

The optical parameterized signal is used for cases when it is not necessary to consider sampled signals in a system (for example an average power analysis). This class is much faster to calculate with than the full sampled signals. In addition, for some optical devices, the bandwidth of the noise can be very large. Often too large for the noise to be computationally practical using a sampled signal description. In this case it is convenient to describe the noise in a parameterized way similar to the optical parameterized signal

In both these cases, the power is stored in the **CNDS_Polarization** class:

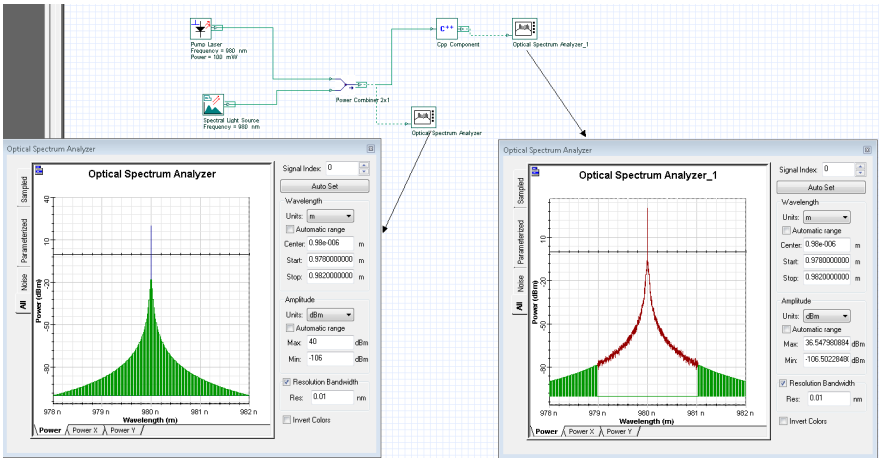
```
// stores the power and state of polarization
CNDS_Polarization      m_Polarization;
CNDS_SignalBandwidth   m_Bandwidth;
```

where **CNDS_Polarization** *m_Polarization* stores the power and state of polarization and **CNDS_SignalBandwidth** *m_Bandwidth* stores the bandwidth in the signal's parent class

The following example shows a simple manipulation of the parameterized signal and noise bins. The “Pump Laser” generates the parameterized signal, the “Spectral Light Source” generates the noise bins in a Lorentzian profile. In this example, the signals are read in, their power is multiplied by a factor of 10, the parameterized signal is converted to a sampled signal, the noise-bins within the bandwidth of the newly created sampled signal are added to it and finally the sampled signal and remaining noise bins are put on the output port.

Note: The full code and an OptiSystem project for this function (*Tutorial7SupplementaryFiles.zip*) is available at the Optiwave C++ Component Home Page. Replace the **DS_SystemManager.cpp** file in your project with the ones given in the zipfile.

Figure 1



Getting the signals

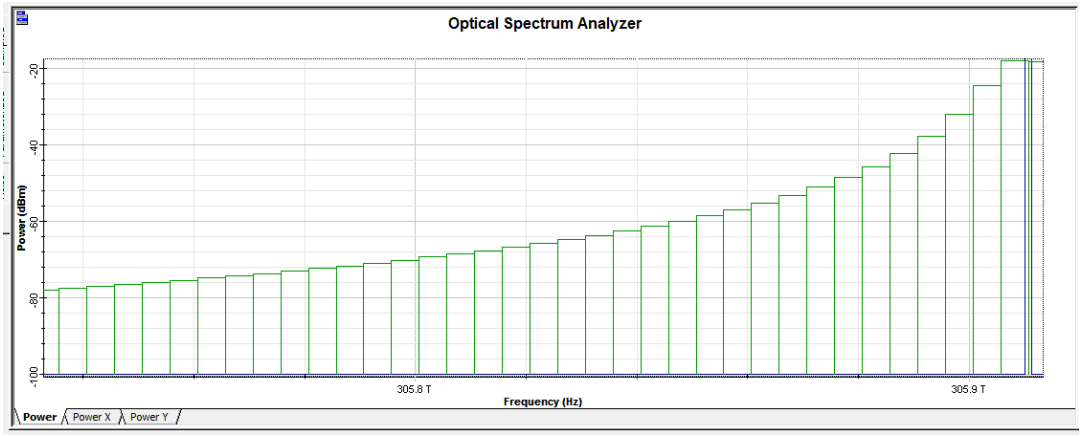
To obtain the parameterized signals and noise bins, use the commands:

```
VECTOR_OpticalParameterizedSignals parameterizedSignal1 = GetOpticalParameterizedSignalsVector(pIOSignal1);
//get the parameterized signals from the input port
VECTOR_OpticalNoiseBins noiseBins1 = GetOpticalNoiseBinsVector(pIOSignal1);
//get the noise bins from the input port
```

Note that they are both getting the signal from the same input port (pIOSignal1) because all the various signal types are carried on the ports simultaneously
As with the sampled signals, the data is read in as vectors. Each vector element in the parameterized signal corresponds to a different wavelength mode. However, they cannot carry any transverse mode information.

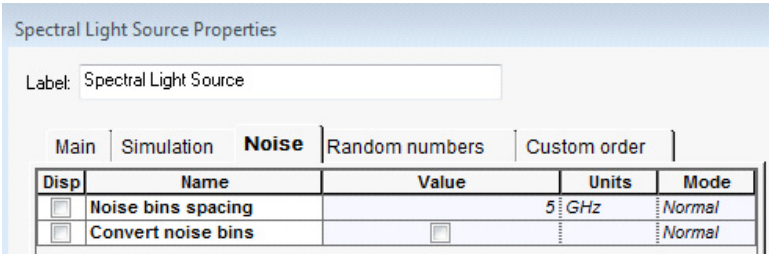
For the noise bins, the vector has another significance. Each vector element of the noise bin corresponds to a region of the frequency space, as shown below. In each region, it is assumed that the noise has a constant power.

Figure 2



The size of this noise bin depends on the component that generated it. In this case the “Spectral Light Source”. The parameter “Noise bins spacing” which sets this size is shown below (note also the convert noise bins has been unchecked).

Figure 3



Multiplication of parameterized signal power

Obtain the current power and multiply it by 10 (**GetPower** returns Watts).

```
parameterizedSignal1[i].m_Polarization.SetPower(parameterizedSignal1[i].m_Polarization.GetPower() * 10);
// multiply parameterized signal power by factor of 10
```

Convert the parameterized signal to a sampled signal

The project already has built-in functions to perform this conversion operation. The user need only instantiate an optical sampled signal passing the parameterized signal and the sample rate (which is obtained from the layout parameters).

```
CNDS_OpticalSampledSignal signal = CNDS_OpticalSampledSignal(parameterizedSignal1[i], dSampleRate);
//above is a built in initializer which will automatically convert the optical parameterized signal into a sampled signal
```


Multiply noise power and add to the sampled signal

```
for (int j = 0; j < noiseBins1.size(); j++)
{
        noiseBins1[j].m_Polarization.SetPower(noiseBins1[j].m_Polarization.GetPower() * 10);
        //multiply noise by factor of 10
        if (noiseBins1[j].m_Bandwidth.GetLowerFrequency() >= signal.m_Bandwidth.GetLowerFrequency() &&
        noiseBins1[j].m_Bandwidth.GetUpperFrequency() <= signal.m_Bandwidth.GetUpperFrequency())
        {
                signal.Add(noiseBins1[j]);
        }
        //If this noise bin is within the bandwith of our new converted sampled signal, add it to the signal
        //(will convert this noise bin to sampled automatically)
        //Any noise bin successfully added to the sampled signal will be zeroed afterwards.
        //All noise bins outside of the bandwidth will be untouched.
}
```

The method for multiplying the noise power is similar to the parameterized signal. The "Add" function in **CNDS_OpticalSampledSignal** is overloaded such that if a noise bin is passed, it will be automatically converted to a sampled signal. The "if" statement is used so that if this noise bin is within the bandwidth of the sampled signal, it will be added, and then this noise-bin will be zeroed. If it is not within the sampled signal, it is ignored

Output of signals

Both the newly created sampled signal and the remaining noise bins (those not added to the sampled signal) are put on the output port.

```
PutOpticalSampledSignalsVector(sampledSignals1, pOOSignal1);
//put the new sampled signal on the output port
PutOpticalNoiseBinsVector(noiseBins1, pOOSignal1);
//put the remaining noise bins on the oputput port (the ones not within the bandwidth of the sampled signal
```

End of Tutorial 7

Appendix 1: Overview of signal types

This section describes how to access and set the properties of the OptiSystem signal classes.

For all signal types, the input and output ports must first be created:

```
Input: CDS_SignalBase* pIOSignal1 = GetSignalFromInputPort(1);
Output: CDS_SignalBase* pOOSignal1 = GetSignalFromOutputPort(1);
```

Binary signals

The binary signal holds two pieces of information: the binary sequence (stored as a vector of longs) and the bit rate.

From the **CDS_BinarySignal** header file these are in the variables:

```
double        m_dBitRate;
LONGVECTOR    m_Bits;
```

which have been left public for convenience.

Accessing the signal

To access the binary signal from the input port use:

```
CDS_BinarySignal binarySignal1 = GetBinarySignal(pIOSignal1);
```

and one can now access each element in the signal directly:

```
binarySignal1.m_Bits[i]
```

To get the bit-rate for this signal use:

```
double dBitRate = binarySignal1.GetBitRate();
```

or directly:

```
double dBitRate = binarySignal1. m_dBitRate;
```

Setting the signal

To place the binary signal onto the output port use:

```
PutBinarySignal(binarySignal1,pOOSignal1);
```


M-ary signals

This signal type is similar to the binary except a vector of doubles is used to store the sequence data instead of a vector of longs.

From the **CDS_MarySignal** header file these are in the variables:

```
double        m_dSymbolRate;
DOUBLEVECTOR     m_Symbols;
```

which have been left public for convenience.

Accessing the signal

To access the Mary signal from the input port use:

```
CDS_MarySignal marySignal1 = GetMarySignal(pIOSignal1);
```

The M-ary signal holds two pieces of information: the M-ary sequence (stored as a vector of doubles) and the symbol rate.

To access the vector of M-ary values use:

```
marySignal1.m_Symbols[i]
```

To get the symbol-rate for this signal use:

```
double dSymbolRate = marySignal1.GetSymbolRate();
```

or directly,

```
double dSymbolRate = marySignal1. m_ dSymbolRate;
```

Setting the signal

To place the M-ary signal onto the output port use:

```
PutMarySignal(marySignal1, pOOSignal1);
```

Electrical signals

Electrical sampled signals

The basic class for the electrical sampled signals is **CNDS_ElectricalSampledSignal**. Refer to **SignalLibrary/NDS_ElectricalSampledSignal.h** for all the data and functions available for this class. For example, the data stored in this class is:

```
// enum types
enum enumSignalDomain
{
        domainTime,
        domainFrequency
};

// signal domain = Time or Frequency
long      m_nDomain;
// SAMPLE RATE
double   m_dSampleRate;
// ELECTRICAL FIELD (TIME DOMAIN)
// vectors with signal amplitude components in the REAL part:
COMPLEXVECTOR   m_arrAmplitude;
```

The complex amplitude data is stored in *m_arrAmplitude*. Note that the information can be defined in the time or frequency domain. Initially, all information (from OptiSystem in release mode or data files in debug mode) is read in the frequency domain. To switch domains, use the commands:

```
electricalSampledSignal.SetDomain(CNDS_ElectricalSampledSignal::domainTime);
```

or,

```
electricalSampledSignal.SetDomain(CNDS_ElectricalSampledSignal::domainFrequency);
```

depending on your requirement. The *SetDomain* function will first check which domain the signal is currently in. If it is already in the domain required, no further action will be performed. If it is different, the appropriate FFT/IFFT will be applied on *m_arrAmplitude*.

Accessing the signal

```
CNDS_ElectricalSampledSignal electricalSampledSignal = GetElectricalSampledSignal(pIOSignal1);
```

Setting the signal

```
PutElectricalSampledSignal(electricalSampledSignal, pOOSignal1);
```


Electrical sampled noise

The base class for the electrical sampled signals is **CNDS_ElectricalSampledNoise**. Refer to **SignalLibrary/NDS_ElectricalSampledNoise.h** for all the data and functions available for this class. The data for the noise is stored in another electrical sampled signal (refer to the header)

```
// vectors with noise amplitude components
CNDS_ElectricalSampledSignal    m_Amplitude;
// vectors with normalized PSD
CNDS_ElectricalSampledSignal    m_NormalizedPSD;
```

Accessing the signal

```
CNDS_ElectricalSampledNoise electricalSampledNoise = GetElectricalSampledNoise(pIOSignal1);
```

Setting the signal

```
PutElectricalSampledNoise(electricalSampledNoise, pOOSignal1);
```

Electrical individual samples

The base class for the electrical individual sample is **CNDS_ElectricalIndividualSample**. Refer to **SignalLibrary/NDS_ElectricalIndividualSample.h**

In this case there is only one piece of amplitude data, at the particular sample (from the header):

```
CComplex        m_ccAmplitude;
```

These signals are particularly useful for feedback applications where one needs to analyze the signal one sample at a time.

Accessing the signal

```
CNDS_ElectricalIndividualSample electricalIndividualSample = GetElectricalIndividualSample(pIOSignal1);
```

Setting the signal

```
PutElectricalIndividualSample(electricalIndividualSample, pOOSignal1);
```

Optical signals

The optical ports can have multiple signals on them. This happens because the input signal can be the sum of multiple modulated optical signals on different carrier frequencies and there can also be multiple transverse modes and polarizations. In OptiSystem each different carrier frequency and transverse mode (and in some cases polarization) will be described by a unique signal. Therefore when optical data is read in from the port, a vector of the signals is created. For the different types of optical signals the following vectors have been defined:

```
typedef std::vector< CNDS_OpticalSampledSignal>      VECTOR_OpticalSampledSignals;
typedef std::vector< CNDS_OpticalIndividualSample>  VECTOR_OpticalIndividualSamples;
typedef std::vector< CNDS_OpticalParameterizedSignal> VECTOR_OpticalParameterizedSignals;
typedef std::vector< CNDS_OpticalNoiseBins> VECTOR_OpticalNoiseBins
```

Optical sampled signals

The basic class for the optical sampled signals is **CNDS_OpticalSampledSignal**. Refer to **SignalLibrary/NDS_OpticalSampledSignal.h** for all the data and functions available for this class. For example, the data stored in this class includes:

Domains

```
// signal domain = Time or Frequency
long   m_nDomain;
enum enumSignalDomain
{
   domainTime,
   domainFrequency
};
```

Similarly to the electrical signal, we can convert between the time and frequency domain. The built in function *SetDomain* will apply the appropriate FFT/IFFT during the conversion.

Amplitudes and polarization

```
// vectors with signal amplitude components :
// optical signal in m_arrAmplitudeX - polarizationConstant
// optical signal polarization in X and Y - polarizationArbitrary
 COMPLEXVECTOR    m_arrAmplitudeX;
 COMPLEXVECTOR    m_arrAmplitudeY;
 CNDS_Polarization       m_Polarization;

 enum enumPolarizationType
 {
      polarizationNone,
      polarizationConstant,
      polarizationArbitrary
 };
```

The amplitude data is stored in the complex vectors

```
COMPLEXVECTOR m_arrAmplitudeX;
COMPLEXVECTOR m_arrAmplitudeY;
```


The phase information is accounted for by the complex format of the amplitude. How this is stored depends on the polarization format being used. The default format has *m_arrAmplitudeX* storing the total amplitude, *m_arrAmplitudeY* being zero and polarization information stored in the Stokes vector class

```
CNDS_Polarization m_Polarization;
// Stokes vector (not normalized)
double m_dS0;
double m_dS1;
double m_dS2;
double m_dS3;
```

We can instead store the X and Y polarization amplitudes separately on *m_arrAmplitudeX* and *m_arrAmplitudeY* respectively. The conversion between the storage modes is obtained by the function (to store as Stokes vectors)

```
SetPolarizationState(CNDS_OpticalSampledSignal::polarizationConstant);
```

and (to store as X and Y amplitudes)

```
SetPolarizationState(CNDS_OpticalSampledSignal::polarizationArbitrary);
```

Bandwidth

The bandwidth information is stored in the parent class (**CNDS_SignalBase**):

```
CNDS_SignalBandwidth m_Bandwidth;
```

which stores

```
double m_dLowerFrequency;
double m_dUpperFrequency;
```

The average between these two frequencies will be the optical signal's carrier frequency and the difference is the bandwidth.

Spatial information

The transverse mode information is stored in:

```
CNDS_OpticalTransverseMode  m_ModeX;
CNDS_OpticalTransverseMode  m_ModeY;
```

The data in this class is a matrix of complex elements **CDS_ComplexMatrix** *m_mAmplitude* where the rows and columns represent the coordinates in the transverse plane

Accessing the signal

```
VECTOR_OpticalSampledSignals vectorSignal = GetOpticalSampledSignalsVector(pIOSignal1);
```

Setting the signal

```
PutOpticalSampledSignalsVector(vectorSignal, pOOSignal1);
```

Optical individual samples

The base class for the optical individual samples is **CNDS_OpticalIndividualSample** (for further details please see **SignalLibrary/NDS_OpticalIndividualSample.h**)

In this case there is only two pieces of amplitude data, at the particular sample (from the header):

```
CComplex    m_ccAmplitudeX;
CComplex    m_ccAmplitudeY;
```

Which terms are non-zero depend on whether the polarization information is carried in the Stokes parameters of in the amplitudes, similar to the optical sampled signal.

These signals are particularly useful for feedback applications where one needs to analyze the signal one sample at a time. Also similar to the optical sampled signal, the carrier bandwidth information is stored. However, these signals do not carry transverse modes and so are only suitable for single mode operations.

Accessing the signal

```
VECTOR_OpticalIndividualSamples vectorIndividual = GetOpticalIndividualSamplesVector(pIOSignal1);
```

Setting the signal

```
PutOpticalIndividualSamplesVector(vectorIndividual, pOOSignal1);
```


Optical parameterized signals

The base class for the optical parameterized signals is **CNDS_OpticalParameterizedSignal** (for further details please see **SignalLibrary/NDS_OpticalParameterizedSignal.h**)

The purpose of this class is for cases when it is not necessary to consider sampled signals in a system (for example when performing average power analysis). This class is much faster to calculate with than the full sampled signal class.

The information stored in this class is

```
// stores the power and state of polarization
CNDS_Polarization    m_Polarization;

// stores statistical properties of the signals
CNDS_StatisticalSignalParameters   m_Parameters;

// common channels stores the power ratio and properties for each channel
// for extra functionality in specialized components. Not generally necessary.
CNDS_CommonChannelsData    m_CommonChannels;
```

The user does not need to use *m_CommonChannels*. The power of the signal is stored in *m_Polarization*:

```
double m_dLinewidth;
double m_dExtinctionRatio;
```

In addition, in the parent class, the carrier bandwidth is stored

```
CNDS_SignalBandwidth m_Bandwidth
```

There is no transverse mode information stored in this class so it is only appropriate for describing a single mode.

Accessing the signal

```
VECTOR_OpticalParameterizedSignals vectorParameterized = GetOpticalParameterizedSignalsVector(pIOSignal1)
```

Setting the signal

```
PutOpticalParameterizedSignalsVector(vectorParameterized, pOOSignal1);
```

Optical noise bins

The base class for the optical noise bins is **CNDS_OpticalNoiseBins**(for further details please see **SignalLibrary/NDS_OpticalNoiseBins.h**)

For some optical devices, the bandwidth of the noise can be very large. Often too large for the noise to be computationally practical using a sampled signal description. In this case it is convenient to describe the noise in a parameterized way similar to the optical parameterized signal.

The data stored in this class is the power

```
// stores the power and state of polarization
CNDS_Polarization       m_Polarization;
```

and the bandwidth through this signal's parent class.

In many OptiSystem components, the "Convert noise bins" option will convert a subset of the noise (within a defined, smaller bandwidth) to sampled signals using a Gaussian random function on the power

Accessing the signal

```
VECTOR_OpticalNoiseBins vectorNoiseBin = GetOpticalNoiseBinsVector(pIOSignal1);
```

Setting the signal

```
PutOpticalNoiseBinsVector(vectorNoiseBin, pOOSignal1);
```


Appendix 2: Debugging tips (release mode)

In this section we present some built-in convenience functions for the user to debug their programs when running their projects in release mode.

Gnuplot

One can create intermediate graphs in the C++ component with predefined convenience functions if gnuplot is installed (please go to: http://sourceforge.net/projects/gnuplot/files/gnuplot/ for the latest build). The predefined function prototypes can be seen in DS_SystemManager.h.

```
////////////////////////////////////////////////////////////////////////////////
//Convenience plotting functions using gnuplot
//In order for these to work, you must have gnuplot for windows installed on your computer
////////////////////////////////////////////////////////////////////////////////
void gnuplot2D(DOUBLEVECTOR x, DOUBLEVECTOR y, string title = "", string legend = "", string xlabel = "", string ylabel = "");
void gnuplot2D(DOUBLEVECTOR x1, DOUBLEVECTOR y1, DOUBLEVECTOR x2, DOUBLEVECTOR y2, string title = "", string legend1 = "", string legend2 =
"", string xlabel = "", string ylabel = "");
void gnuplot3DTopViewRealVector(DOUBLEVECTOR x, DOUBLEVECTOR y, DOUBLEVECTOR z, string title = "", string xlabel = "", string ylabel = "");
void gnuplot3DSurfaceRealVector(DOUBLEVECTOR x, DOUBLEVECTOR y, DOUBLEVECTOR z, string title = "", string xlabel = "", string ylabel = "");
void gnuplot3DTopViewTransverseMode(CNDS_OpticalTransverseMode mode, string type = "abs", string title = "");
void gnuplot3DSurfaceTransverseMode(CNDS_OpticalTransverseMode mode, string type = "abs", string title = "");
```

Examples were provided in tutorials 2, 3, 4 and 6 while running in debug mode. These can also be run in release mode however the user must ensure the working directory is writable because the gnuplot functions create temporary files (for example)

```
void CDS_SystemManager::gnuplot2D(DOUBLEVECTOR x, DOUBLEVECTOR y, string title, string legend, string xlabel, string ylabel)
{


    int nRand = rand(); //to make new name for file so it doesn't conflict with any other temporary files

    //create temporary data file
    string fileName = "tempDat" + to_string(nRand) + ".dat";
    ofstream ViewOut(fileName);
    for (int i = 0; i < x.size(); i++)
    {
        ViewOut << x[i] << " " << y[i] << "\n";
    }
    ViewOut.close();

    //create gnuplot script file
    string scriptName = "plot" + to_string(nRand) + ".gp";
    ofstream scrip(scriptName);
    scrip << "set style data lines \n";
    scrip << "set title \"" + title + "\" \n";
    scrip << "set xlabel \"" + xlabel + "\" \n";
    scrip << "set ylabel \"" + ylabel + "\" \n";
    scrip << "plot '" + fileName + "' title '" + legend + "'   \n";
    scrip.close();
```

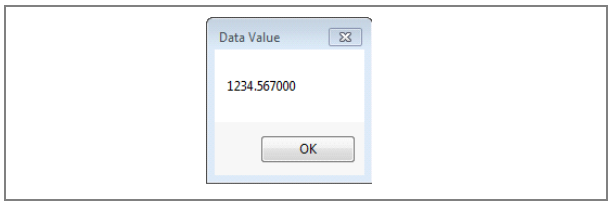
If the user cannot write to the working directory, they should explicitly set the path for the files for "fileName" and "scriptName".

Information boxes

If the user wants a popup window to display a piece of information during the run, they can use code like the example below:

```
double dDataValue =  1234.567;
MessageBox(0, std::to_string(dDataValue).c_str(), "Data Value", MB_OK);
```

the command std::to_string(dDataValue).c_str() converts the data to a string for display



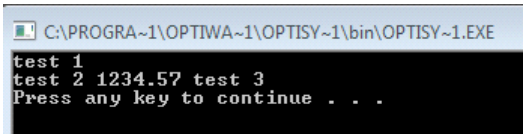
The component will pause at this point until the user dismisses the popup window.

Console output

The class "Console" has been added so that the user can display information in a console during execution of their component. For example the code below:

```
Console myConsole;
myConsole << "test 1\n";
double dDataValue = 1234.567;
myConsole << "test 2 " << dDataValue << " test 3 \n";
system("pause");
```

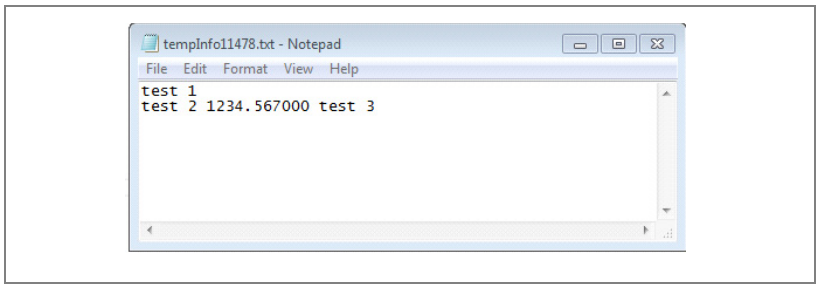
will result in:



The class "Console" has been added so that the user can display information in a console during execution of their component. For the code:

```
Console myConsole;
string buffer;
myConsole << "test 1\n";
buffer += "test 1\n";
double dDataValue = 1234.567;
myConsole << "test 2 " << dDataValue << " test 3 \n";
buffer += "test 2 " + to_string(dDataValue) + " test 3 \n";
system("pause");
Notepad(buffer);
```

will produce the console output as above, then after execution, will copy the contents of the buffer onto Windows Notepad as follows:



Note: We will also be introducing a new Console feature in OptiSystem 14.1 which will automatically output buffer data (from any Cpp component) to the OptiSystem console window.

Optiwave
7 Capella Court
Ottawa, Ontario, K2E 7X1, Canada

Tel.: 1.613.224.4700
Fax: 1.613.224.4706

E-mail: support@optiwave.com
URL: www.optiwave.com