Ulyssix ALTAIR Data Process Manual

Revision B 07/06/2018

Ulyssix Technologies, Inc
7470 New Technology Way, Suite B, Frederick, MD 21703-9461
Tel: 301-846-4800 ~ Fax: 301-846-0686 ~ www.ulyssix.com

Table of Contents

Table of Figures

Introduction

ALTAIR uses Data Processes for mathematical and logical manipulation of data that is decommutated from a telemetry stream. Telemetry data can come from wide variety of sources and some data sources require computation of the raw data to extract the measured value. For example, the raw data from a thermocouple is a non-linear voltage that represents a temperature. Small changes in voltage might be difficult to identify in a display, but a mathematical algorithm in a Data Process can convert the non-linear voltage to a linear temperature.

There are three types of Data Processes in ALTAIR: Formulas, C# Function, and DLL Plug-Ins. A Formula Data Process is built using the Formula Editor in ALTAIR's Process Edit window. A C# Function is built by either typing or pasting C# text code into the C# Function Editor in ALTAIR's Process Edit window. A DLL Plug-In is a compiled dynamic link library developed using Microsoft .NET development tools. These files must run in a managed .NET environment.

A Data Process is defined once and can be used on multiple decom parameters. For example, a Data Process named DivideBy16 that takes the current value of the selected decom parameter and returns that value divided by sixteen could be used to convert a 16-bit integer to a fixed pointer number with 12-bits to the left of the decimal point and 4-bits to the right of the decimal point. Once defined, the Data Process DivideByt16 can be selected for multiple decom words.

As previously mentioned, there are three types of Data Processes in ALTAIR: Formulas, C# Function, and DLL Plug-Ins. Both Formulas and C# Functions are Local Processes. Local Process are stored in the Altair XML setup file and must be copied and pasted as text between setups. The DLL Plug-In Data Processes are available for any ALTAIR XML setup on a computer with the Data Process DLL.

Choosing the Correct Data Process Type

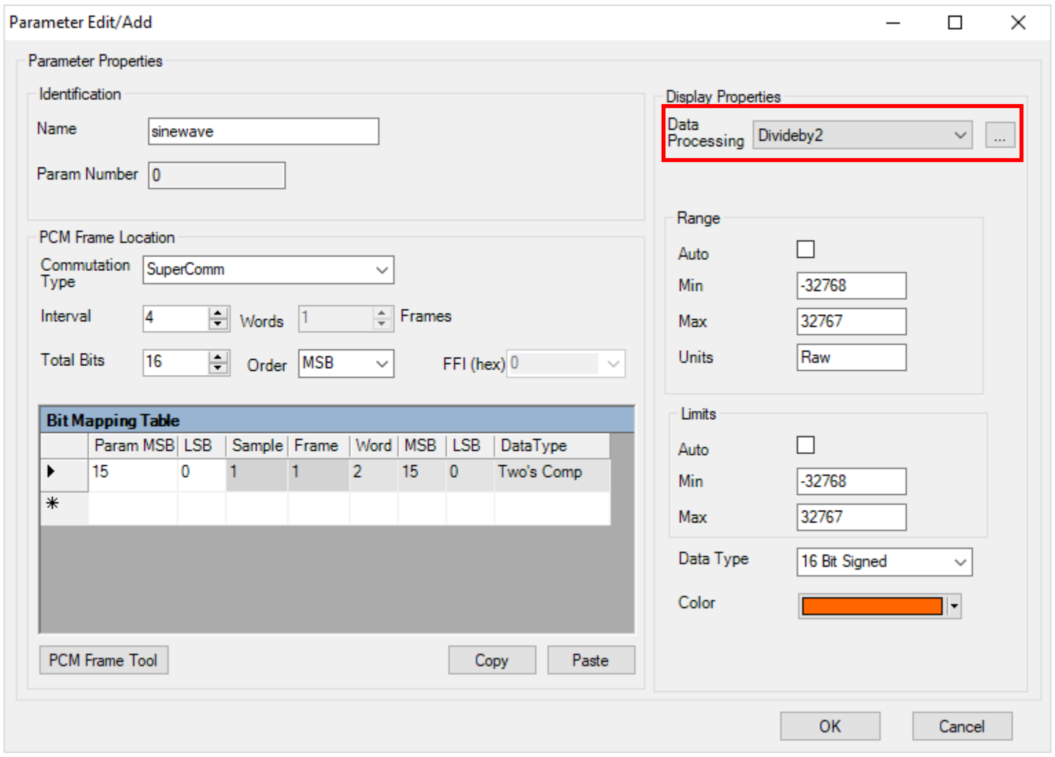
Formulas are best for simply mathematical operations that depend on the current value of one or more decom parameters. They are fast and easy to implement. Formulas do not have any bitwise operations. Formulas do not have any memory to store previous values. Formulas are stored in the ALTAIR XML setup file.

C# Functions have all mathematical function available in C#, including bitwise operations. C# Functions use class global variables to store data from previous calculated values. The added features of the C# Functions come with some added complexity to coding. C# Function are stored in the ALTAIR XML setup file.

The DLL Plug-Ins have all the flexibility of the C# Function, can access external C# Libraries, and can ask the user to specify values. For example, a DLL Plug-In Data Process for Quadratic Scaling could ask the user to define the constants A, B, and C in the quadratic scaling formula C*x^2 + B*x + c. The DLL Plug-In Data Processes where the user define constants adds flexibility. Because the DLL Plug-Ins are compiled, they are easy to transport between computers running ALTAIR. The DLL Plug-In is slightly harder to develop and troubleshoot because error checking requires running ALTAIR and the Microsoft .NET Development tools. A DLL Plug-in can contain multiple independent Data Processes. A DLL can contain an entire library of Data Processes.

Navigating to the Process Edit Window

A Data Process is a mathematical and logical operation that is applied to one or more decom parameters to compute a value. To access the Data Process, select a decom parameter and launch its Parameter Edit/Add window. This can be done by double clicking on a decom parameter in the Parameter View window or by launching the Decom window, selecting the Parameter tab, selecting the desired parameter, and then clicking the Edit button. In the upper right corner of the Parameter Edit/Add window there is a drop-down box to select an existing Data Process and browse button.



Bit Mapping Table

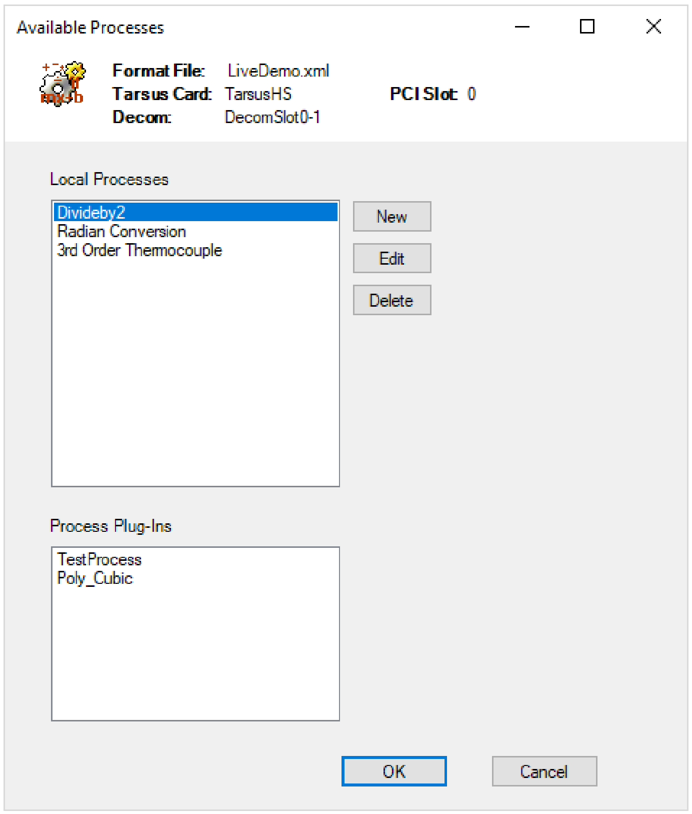
	Param MSB	LSB	Sample	Frame	Word	MSB	LSB	DataType
▶	15	0	1	1	2	15	0	Two's Comp
*								

ALTAIR Parameter Edit/Add Window

The drop-down box contains a list of existing Data Processes. This includes all Formula, C# Function, and DLL Plug-Ins Data Processes. The default entry is None; this turns off the Data Process for this decom parameter. The browse button launches the Available Processes window.

Available Processes Window

The Available Processes has two list boxes.


Available Processes

Format File: LiveDemo.xml
Tarsus Card: TarsusHS PCI Slot: 0
Decom: DecomSlot0-1

Local Processes

Divideby2
Radian Conversion
3rd Order Thermocouple

Process Plug-Ins

TestProcess
Poly_Cubic

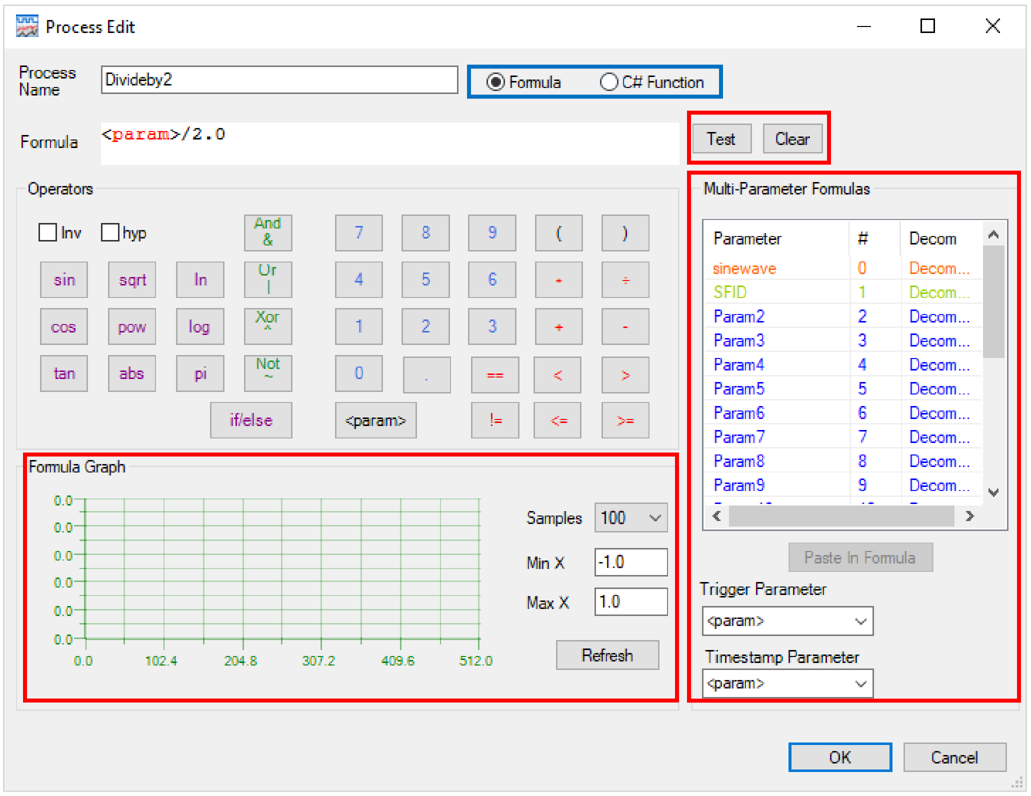
ALTAIR Available Processes Window

The top list box contains the Local Data Processes. By definition, all Formula and C# Functions are Local Data Processes. The Local Data Processes are stored in the ALTAIR XML setup file. In order to use these in another ALTAIR XML setup file, the text for the Formula for C# Function must be copied and pasted into the second ALTAIR XML setup file. The bottom list box contains the Data Processes from the DLL Plug-Ins. These Data Processes are available to any ALTAIR XML setup file launched on the computer that has the DLL Plug-In.

Clicking the New button creates a new data process. Selecting a Data Process from the Local Processes list box enables the Edit button and the Delete button. The Edit button launches the Process Edit window for the selected Data Process. The Delete button removes the selected Data Process from the Altair XML setup file.

Process Edit Window

The Process Edit window has two configurations controlled by the Formula and C# Function radio buttons located at the top center of the window (blue rectangle in the image below). Changing the selected radio button switches between the Formula Editor and the C# Function editor. The Formula Graph section, Test button, Clear button, and Multi-Parameter Formulas section are common between both the Formula Editor and the C# Function Editor (red rectangles in image below). The Process Edit window is resizable. Increasing the size of the Process Edit window will increase the size of the Formula / C# Function Text Exit window. A larger text edit size is conducive to developing code. Due to the large number of controls, the window will not shrink past its default size.



Process Name: Divideby2

[✓ Formula] C# Function

Formula: `<param>/2.0`

Samples: 100

Min X: -1.0

Max X: 1.0

Parameter	#	Decom
sinewave	0	Decom...
SFID	1	Decom...
Param2	2	Decom...
Param3	3	Decom...
Param4	4	Decom...
Param5	5	Decom...
Param6	6	Decom...
Param7	7	Decom...
Param8	8	Decom...
Param9	9	Decom...

Trigger Parameter: <param>

Timestamp Parameter: <param>

ALTAIR Process Edit Window

Formula Graph section setups the test conditions applied by clicking either the Refresh button or the Test button. The Samples drop-down box determines the number of iterations for evaluating the Data Process. The options for the Samples drop-down box are: 10, 100, and 1000. The Min X text box and Max X text box determine the starting and end values for the evaluation. The results of the evaluation are plotted in the graph.

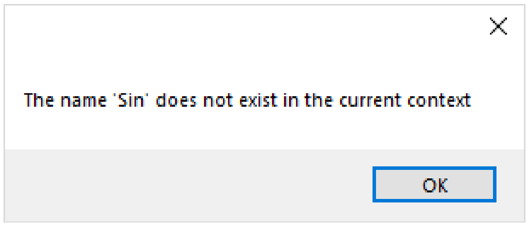
The Clear button deletes all text in Formula and C# Function text box.

The Test button evaluates the Data Process from the parameters defined in the Formula Graph section. The evaluation begins at Min X and continues with Samples number of equally spaced values to Max X. The results of the evaluation are plotted in graph. The Test button also error checks Data Process. If there are errors, a Process Error window appears with the list of errors.

The error messages displayed are actual messages generated by the compiler which can sometimes be misleading. Errors are generally due to missing operators, mismatched parentheses, or simple typing errors. When encountering an error, review the formula item-by-item to be certain there is an operation between every function, and all parentheses are used properly. Parentheses should always be used to assure proper precedence. If a Data Process with an error is saved and the decom parameter is used in a display (Meter, Strip Chart, etc), ALTAIR will display a warning message that there is an error with the Data Process and the display will not be updated. If this occurs, please delete all displays using the decom parameter with the erroneous Data Process.

Formula Editor in the Process Edit Window

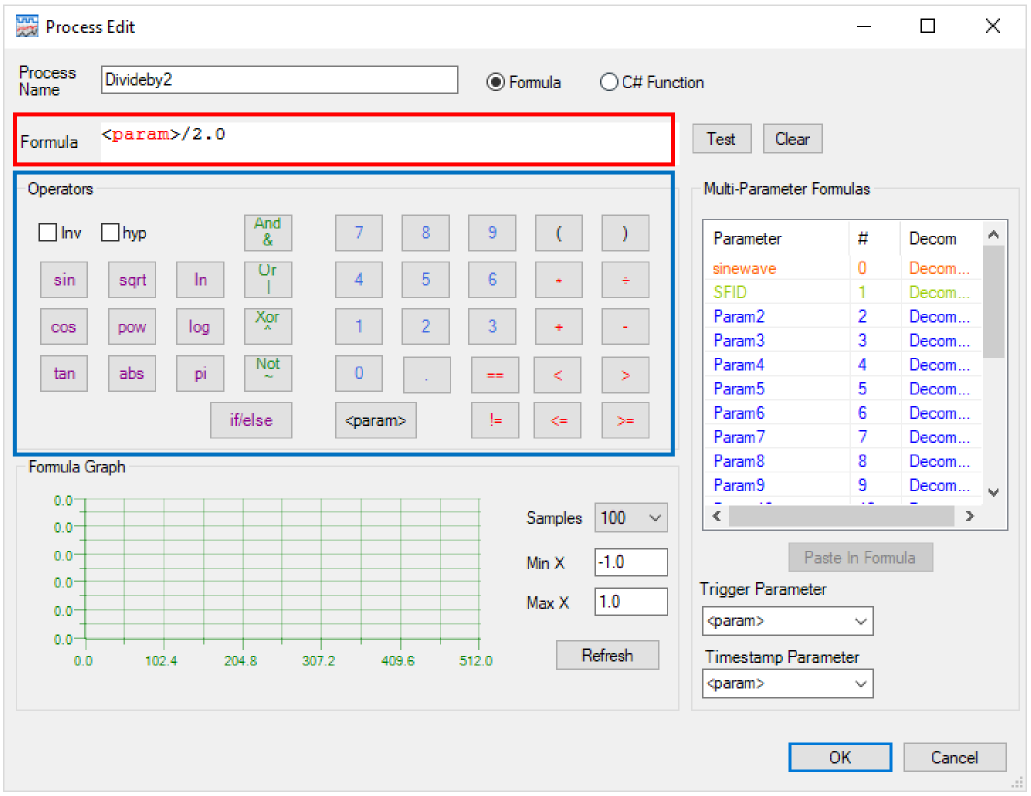
When the Formula radio button is selected, the Process Edit window displays the Formula Editor (see image below). The Formula Text Editor displays the current formula using syntax highlighting for key words (red rectangle in image below). The Formula Text Editor is case sensitive, including the syntax highlighting. For example, sin(param) will compile but Sin(param) will throw the following error.



```
The name 'Sin' does not exist in the current context
```

Formula Editor is Case Sensitive

The Formula Editor contains mathematical and logical operations in the Operators section (blue rectangle in image below). The mathematical operators include trigonometric, exponential, and logarithmic operators. The logical operators include if-then-else as well as logical And, Or, Xor, and Not operators. Please note that there are no bitwise operators. For bit manipulator, please use a C# Function. After completing a Formula, always use the Test button to evaluate and error check. It is always best to catch the errors before the Formula is saved to the ALTAIR XML setup file.



Process Name: Divideby2 [✓ Formula] C# Function

Formula: `<param>/2.0`

Multi-Parameter Formulas

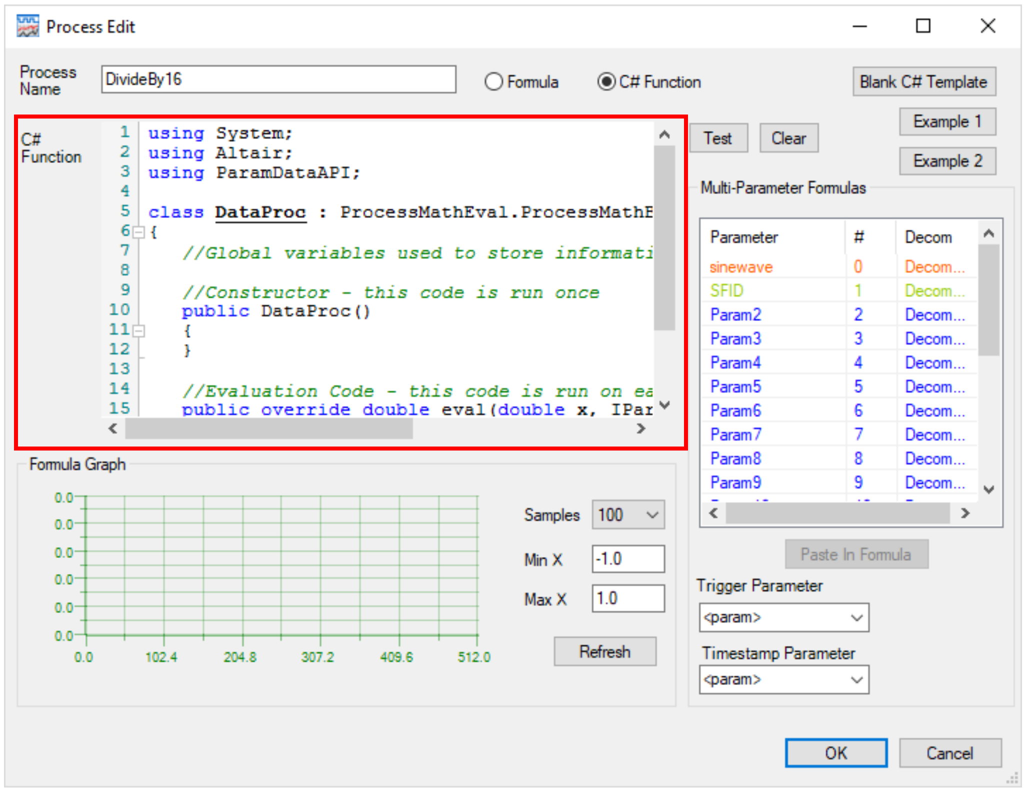
Parameter	#	Decom
sinewave	0	Decom...
SFID	1	Decom...
Param2	2	Decom...
Param3	3	Decom...
Param4	4	Decom...
Param5	5	Decom...
Param6	6	Decom...
Param7	7	Decom...
Param8	8	Decom...
Param9	9	Decom...

Samples: 100
Min X: -1.0
Max X: 1.0
Trigger Parameter: <param>
Timestamp Parameter: <param>

ALTAIR Formula Editor in the Process Edit Window

C# Function Editor in the Process Edit Window

When the C# Function radio button is selected, the Process Edit window displays the C# Function Editor (see image below). The C# Function Text Editor displays the current C# function using syntax highlighting for key words (red rectangle in image below). The C# Function Text Editor is case sensitive; it follows the same rules and C#. If editing an existing Data Process, the C# Function Text Editor displays the previous code. For a new Data Process, the C# Function Editor is blank. In the upper right corner there are three buttons: Blank C# Template, Example 1, and Example 2.



Process Name: DivideBy16

[○ Formula] [✓ C# Function]

C# Function

```
using System;
using Altair;
using ParamDataAPI;

class DataProc : ProcessMathEval.ProcessMathE
{
    //Global variables used to store informati

    //Constructor - this code is run once
    public DataProc()
    {
    }

    //Evaluation Code - this code is run on ea
    public override double eval(double x, IPar
```

Parameter	#	Decom
sinewave	0	Decom...
SFID	1	Decom...
Param2	2	Decom...
Param3	3	Decom...
Param4	4	Decom...
Param5	5	Decom...
Param6	6	Decom...
Param7	7	Decom...
Param8	8	Decom...
Param9	9	Decom...

Samples: 100
Min X: -1.0
Max X: 1.0

Trigger Parameter: <param>
Timestamp Parameter: <param>

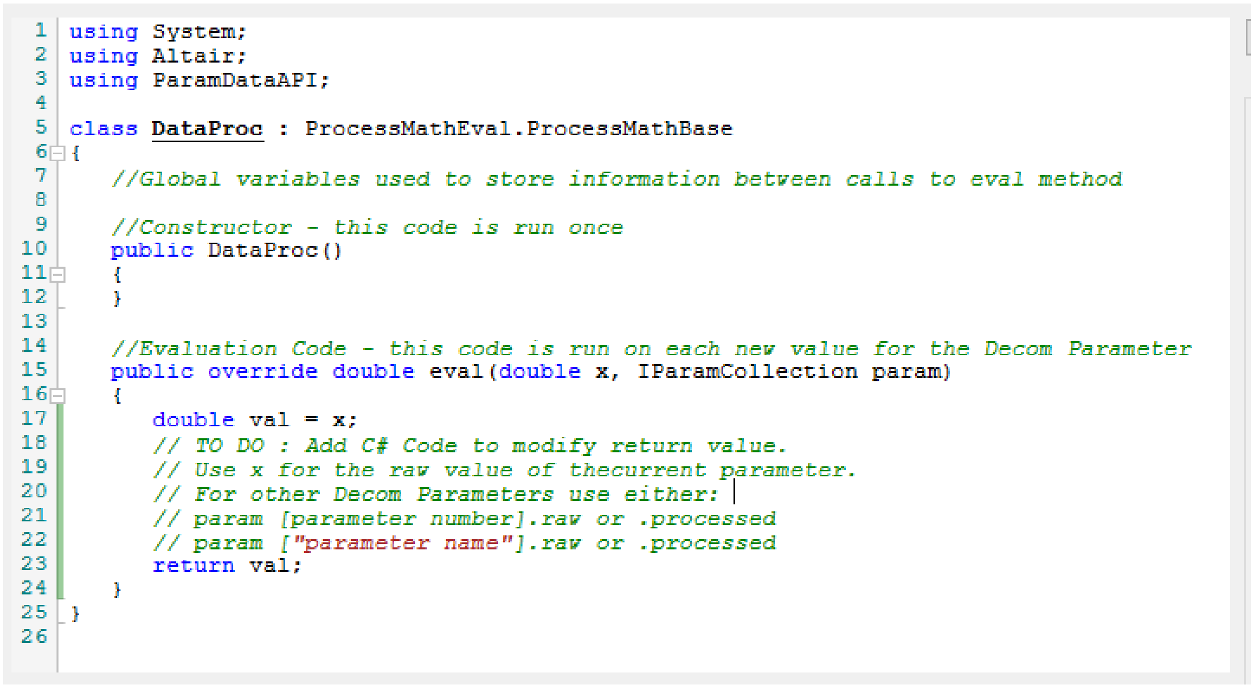
ALTAIR C# Function Editor in Process Edit Window

The ALTAIR Data Process window is useful for developing a simple C# Function Data Process. However, more complicated Data Processes would be benefit from the use of Microsoft .NET Development tools for C# like Visual Studio or .NET Core. Using these tools provides IntelliSense autocomplete and more informative debugging. IntelliSense is particularly useful for investigating the functions available and their syntax for C# libraries.

Blank C# Template

Clicking the Blank C# Template button populates the C# Function Text Editor with the required structure for a C# Function Data Process. The C# Function begins with three using statements: System, Altair, and ParamDataAPI. For more information on the ParamDataAPI, please see the ParamDataAPI section in this manual.

Those three using statements are required and should not be changed. Next, is the class definition for DataProc with the base class ProcessMathEval.ProcessMathBase. The DataProc class contains the code that is called for the Data Process. The name, DataProc cannot be changed. Next, is a comment specifying the location for any global variables. If a value needs to be pass from each call to the Data Process, it should be stored as a global variable. For example, if the Data Process is a Box Car Average, then the array to store the data for the Box Car would be a global variable (see the code for Example 1). After the global variables is the constructor. The constructor is called when the code is initialized. The constructor is where any variables or arrays should be initialized. After the constructor is the eval function. The name, data type, and parameters of the eval function cannot be changed. The eval function is called every time that new data is sent to the Data Process. This is where any computations should occur. The Blank C# Template is below.

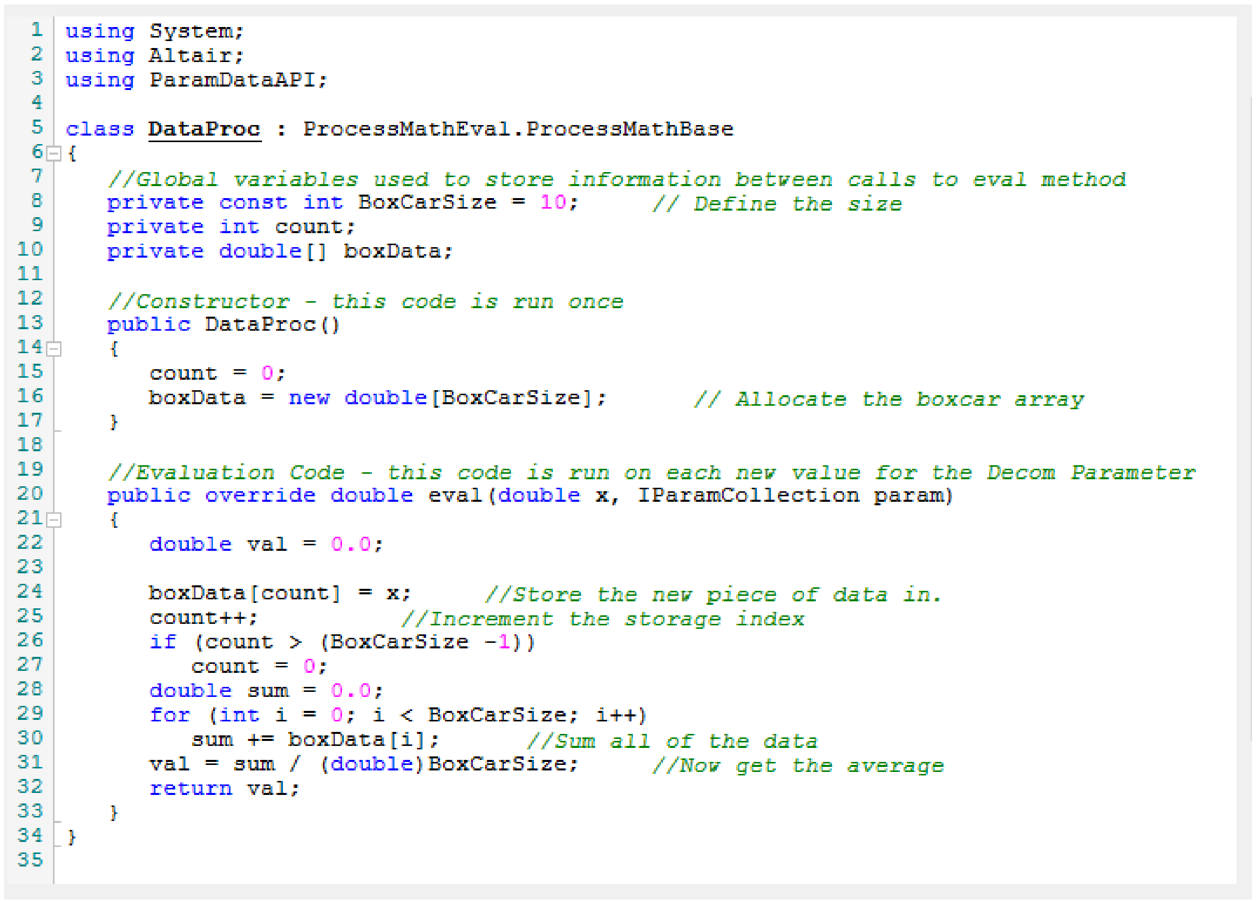


```
using System;
using Altair;
using ParamDataAPI;

class DataProc : ProcessMathEval.ProcessMathBase
{
   //Global variables used to store information between calls to eval method

   //Constructor - this code is run once
   public DataProc()
   {
   }

   //Evaluation Code - this code is run on each new value for the Decom Parameter
   public override double eval(double x, IParamCollection param)
   {
      double val = x;
      // TO DO : Add C# Code to modify return value.
      // Use x for the raw value of thecurrent parameter.
      // For other Decom Parameters use either:
      // param [parameter number].raw or .processed
      // param ["parameter name"].raw or .processed
      return val;
   }
}
```

C# Function – Blank C# Template

C# Function Example 1

Clicking the Example 1 button populates the C# Function Text Editor with the C# Function for a ten-value box car average. Example 1 demonstrates using the global variables to store the number of values in the box car average (BoxCarSize = 10), the number of entries added to the box car (count), and the array to hold the box car values (boxData). The constructor initializes the integer count to 0 and defines the double array boxData to have BoxCarSize samples. Every time the eval function is called, data is added to the array at position count. Then count is incremented. If count is larger than the size BoxCarSize -1, then count is reset to zero. Then the average of the samples is commutated and retuned from the function.

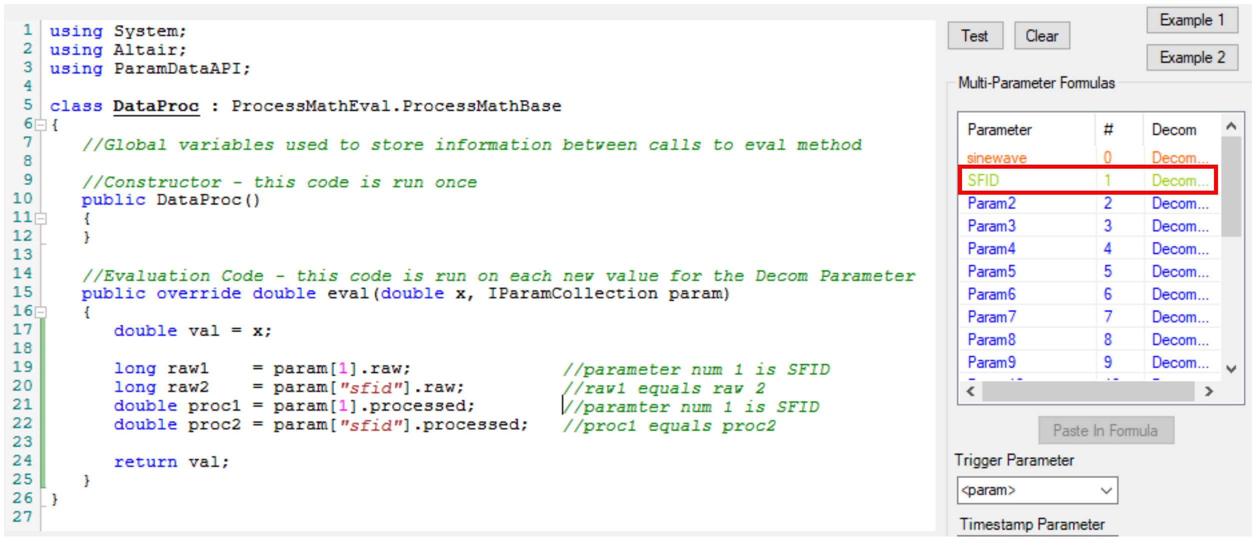


```
using System;
using Altair;
using ParamDataAPI;

class DataProc : ProcessMathEval.ProcessMathBase
{
   //Global variables used to store information between calls to eval method
   private const int BoxCarSize = 10;      // Define the size
   private int count;
   private double[] boxData;

   //Constructor - this code is run once
   public DataProc()
   {
      count = 0;
      boxData = new double[BoxCarSize];       // Allocate the boxcar array
   }

   //Evaluation Code - this code is run on each new value for the Decom Parameter
   public override double eval(double x, IParamCollection param)
   {
      double val = 0.0;

      boxData[count] = x;      //Store the new piece of data in.
      count++;           //Increment the storage index
      if (count > (BoxCarSize -1))
         count = 0;
      double sum = 0.0;
      for (int i = 0; i < BoxCarSize; i++)
         sum += boxData[i];       //Sum all of the data
      val = sum / (double)BoxCarSize;      //Now get the average
      return val;
   }
}
```

C# Function – Example 1 Box Car Average

Getting Data from Other Parameters

The raw value of the current parameter is available as the double x as defined in the function eval. The second parameter in the function eval is named param and is of type IParamCollection. The IParamCollection param is an object that contains every decom parameter as variable types IParamData. A decom parameter can be addressed either using its name or its decom parameter number. The data for decom parameter can be accessed through IParamData by the properties “raw” and “processed.” Raw data is accessed before any Data Processes are applied and is a data type long (64-bit signed integer). Processed data is accessed after any Data Processes are applied is a data type double (double precision floating point). This information is noted in the comments on lines 21 and 22 in the Blank C# Template.



```
using System;
using Altair;
using ParamDataAPI;

class DataProc : ProcessMathEval.ProcessMathBase
{
   //Global variables used to store information between calls to eval method

   //Constructor - this code is run once
   public DataProc()
   {
   }

   //Evaluation Code - this code is run on each new value for the Decom Parameter
   public override double eval(double x, IParamCollection param)
   {
      double val = x;

      long raw1     = param[1].raw;              //parameter num 1 is SFID
      long raw2     = param["sfid"].raw;         //raw1 equals raw 2
      double proc1 = param[1].processed;         //paramter num 1 is SFID
      double proc2 = param["sfid"].processed;    //proc1 equals proc2

      return val;
   }
}
```

Parameter	#	Decom
sinewave	0	Decom...
SFID	1	Decom...
Param2	2	Decom...
Param3	3	Decom...
Param4	4	Decom...
Param5	5	Decom...
Param6	6	Decom...
Param7	7	Decom...
Param8	8	Decom...
Param9	9	Decom...

Example of using IParamCollection to Access Other Parameters

In the example code above, there are examples of using the IParamCollection variable param to access the same decom parameter by four methods. Please note in the image above that in the Multii-Parameter Formulas list box that the parameter named SFID is assigned Parameter Number 1 (red rectangle in the image above).

1. Accessing the raw value using the parameter number to index the IParamCollection to a specific IParamData.
2. Accessing the raw value using the parameter name to index the IParamCollection to a specific IParamData. The parameter name case sensitive.
3. Accessing the processed value using the parameter number to index the IParamCollection to a specific IParamData. The parameter name case sensitive.
4. Accessing the processed value using the parameter name to index the IParamCollection to a specific IParamData. The parameter name case sensitive.

In the above example, if the decom parameter does not have a Data Process, then raw and processed would have the same/similar value. Please note that this equality is defined as “within reason” given the differences between the accuracy of a double precision floating point number and an integer.

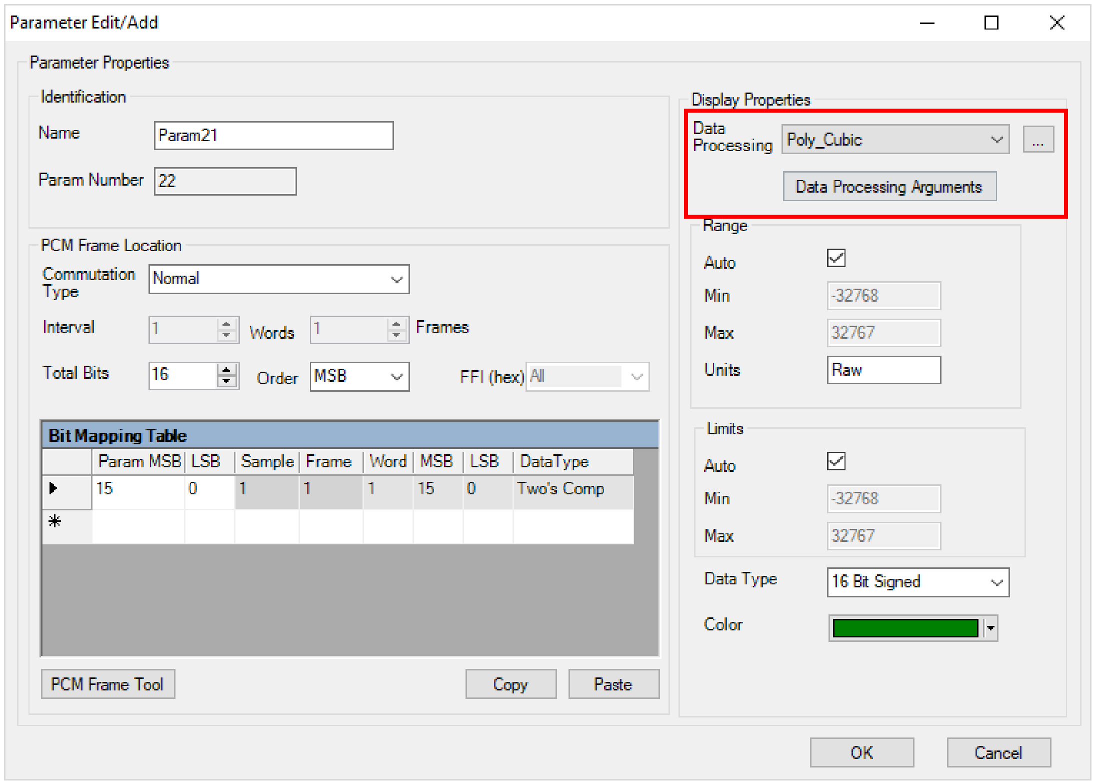
A detailed API for the IParamCollection and IParamData is available in the section ParamDataAPI.

DLL Plug-In

The DLL Plug-In is a dynamic link library using managed Microsoft .NET environment. The best tools for creating a DLL Plug-In is either Microsoft Visual Studio or Microsoft .NET Core. As of the writing of this document, Microsoft offers the free Community version of Visual Studio as well as the free Microsoft .NET Core. Both are integrated development environments that provide syntax highlighting, Intellisense auto-complete, compiling, and debugging. The DLL Plug-In can be written in any language that supports the managed .NET environment, but this document will focus solely on using C# language.

The recommended version of the .NET Framework is 4.0. .NET 4.0 is the last version compatible available for Windows XP and is included in most installation of Windows 7 and Windows 10.

One of the major advantages of the DLL Plug-In is that it allows for user defined constants. If the Data Process selected in the Data Process drop-down box requires user defined constants, the Data Processing Arguments button appears (see red rectangle in the image below). The use of arguments in the DLL Plug-In is optional. See Example Poly_Cubic below for more details.


Parameter Edit/Add

Parameter Properties

Identification

Name: Param21

Param Number: 22

PCM Frame Location

Commutation Type: Normal

Interval: 1 Words 1 Frames

Total Bits: 16 Order: MSB FFI (hex): All

Bit Mapping Table

	Param MSB	LSB	Sample	Frame	Word	MSB	LSB	DataType
▶	15	0	1	1	1	15	0	Two's Comp
*								

Display Properties

Data Processing: Poly_Cubic

Range

Auto: ✓

Min: -32768

Max: 32767

Units: Raw

Limits

Auto: ✓

Min: -32768

Max: 32767

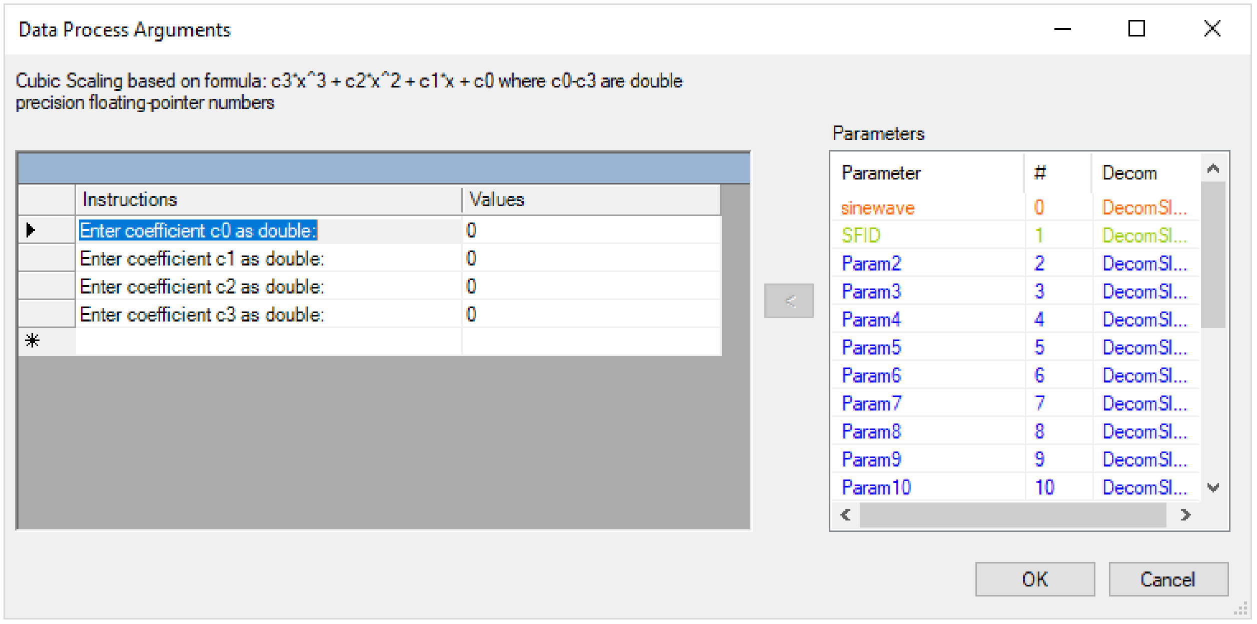
Data Type: 16 Bit Signed

Color: 🟩

Parameter Edit/Add Window with Data Processing Arguments Button

Data Processing Arguments Window

Clicking the Data Processing Arguments button launches the Data Process Arguments window. The Data Processing Arguments window has a brief description of the Data Process at the top as well as a table to define all of the required arguments. Each item in the data table has an instruction and a value. Please note that it is up to each DLL Plug-In developer to provide the proper instructions and to implement the required error checking for user entry. ALTAIR does minimal error checking to ensure that the correct data type is used. Each DLL Plug-In should ensure that operations like dividing by zero are prevented.



Cubic Scaling based on formula: c3*x^3 + c2*x^2 + c1*x + c0 where c0-c3 are double precision floating-pointer numbers

Instructions	Values
Enter coefficient c0 as double:	0
Enter coefficient c1 as double:	0
Enter coefficient c2 as double:	0
Enter coefficient c3 as double:	0

Parameters

Parameter	#	Decom
sinewave	0	DecomSl...
SFID	1	DecomSl...
Param2	2	DecomSl...
Param3	3	DecomSl...
Param4	4	DecomSl...
Param5	5	DecomSl...
Param6	6	DecomSl...
Param7	7	DecomSl...
Param8	8	DecomSl...
Param9	9	DecomSl...
Param10	10	DecomSl...

Data Process Arguments Window for DLL Plug-In

In the example above, please note that both the Data Process description and the instruction for each value state the data type for the value. For Cubic Polynomial Scaling all four constants are doubles.

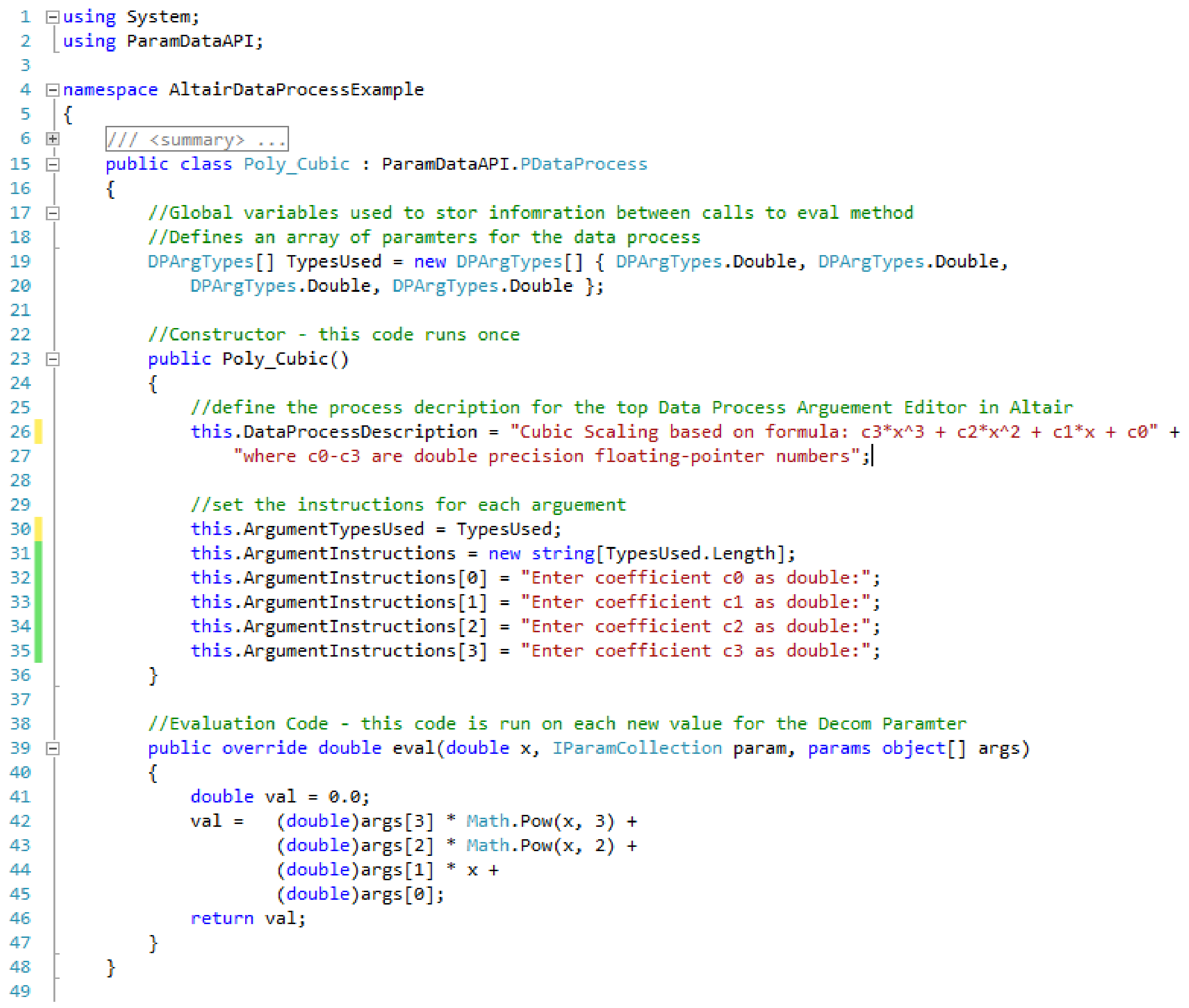
Example Code for Poly_Cubic

The C# code for the example Poly_Cubic is similar to the C# Function example code. These two different types of Data Processes are derived from the same base. The C# Function is simplified for ease of use and therefore has a reduced feature set.

The DLL Plug-In C# code begins with using statements for System and ParamDataAPI on lines 1 and 2. On line four there is a definition for the namespace. In .NET, Namespace is a collection of classes that are interoperable without the need for using statements. After the namespace is the class definition for the Data Process Poly_Cubic which includes the base class ParamDataAPI.PDataProcess. This is a different base class from C# Function, which used the base class ProcessMathEval.ProcessMathBase. The name of the class can be changed. The class name is displayed in the Parameter Add/Edit window, in this example Poly_Cubic.

The structure inside of the Poly_Cubic class matches the pattern of the C# Function. There is a section for Global Variables, a constructor for the class, and a function named eval. The eval function has two different possible prototypes. The first has two parameters: double x and IParamCollection param. The second prototype adds a third parameter, object[] args. Either prototype can be used, but if the object[] args is not included, the eval function will not have access to the user defined arguments.

```
public override double eval(double x, IParamCollection param)
public override double eval(double x, IParamCollection param, params object[] args)
```



```
using System;
using ParamDataAPI;

namespace AltairDataProcessExample
{
    /// <summary> ...
    public class Poly_Cubic : ParamDataAPI.PDataProcess
    {
        //Global variables used to stor infomration between calls to eval method
        //Defines an array of paramters for the data process
        DPArgTypes[] TypesUsed = new DPArgTypes[] { DPArgTypes.Double, DPArgTypes.Double,
            DPArgTypes.Double, DPArgTypes.Double };

        //Constructor - this code runs once
        public Poly_Cubic()
        {
            //define the process decription for the top Data Process Arguement Editor in Altair
            this.DataProcessDescription = "Cubic Scaling based on formula: c3*x^3 + c2*x^2 + c1*x + c0" +
                "where c0-c3 are double precision floating-pointer numbers";

            //set the instructions for each arguement
            this.ArgumentTypesUsed = TypesUsed;
            this.ArgumentInstructions = new string[TypesUsed.Length];
            this.ArgumentInstructions[0] = "Enter coefficient c0 as double:";
            this.ArgumentInstructions[1] = "Enter coefficient c1 as double:";
            this.ArgumentInstructions[2] = "Enter coefficient c2 as double:";
            this.ArgumentInstructions[3] = "Enter coefficient c3 as double:";
        }

        //Evaluation Code - this code is run on each new value for the Decom Paramter
        public override double eval(double x, IParamCollection param, params object[] args)
        {
            double val = 0.0;
            val =   (double)args[3] * Math.Pow(x, 3) +
                    (double)args[2] * Math.Pow(x, 2) +
                    (double)args[1] * x +
                    (double)args[0];
            return val;
        }
    }

```

DLL Plug-In Example Code for Cubic Polynomial Scaling

The only global variable is the array of DPArgTypes named TypesUsed. DPArgTypes is an enum defined in ParamDataAPI. The array TypesUsed defines the data types for the arguments for the DLL Plug-In Poly_Cubic.

The constructor includes the definition of the process description string: ArgumentTypesUsed, the data types for the arguments: ArgumentTypesUsed, and the definition of the array of strings for the instructions for the arguments: ArgumentInstrunctions. In the example above, please note that process description and the argument instructions the text displayed in the Data Process Argument Window image above. The variable ArgumentTypesUses is passed the value of the global variable TypesUsed.

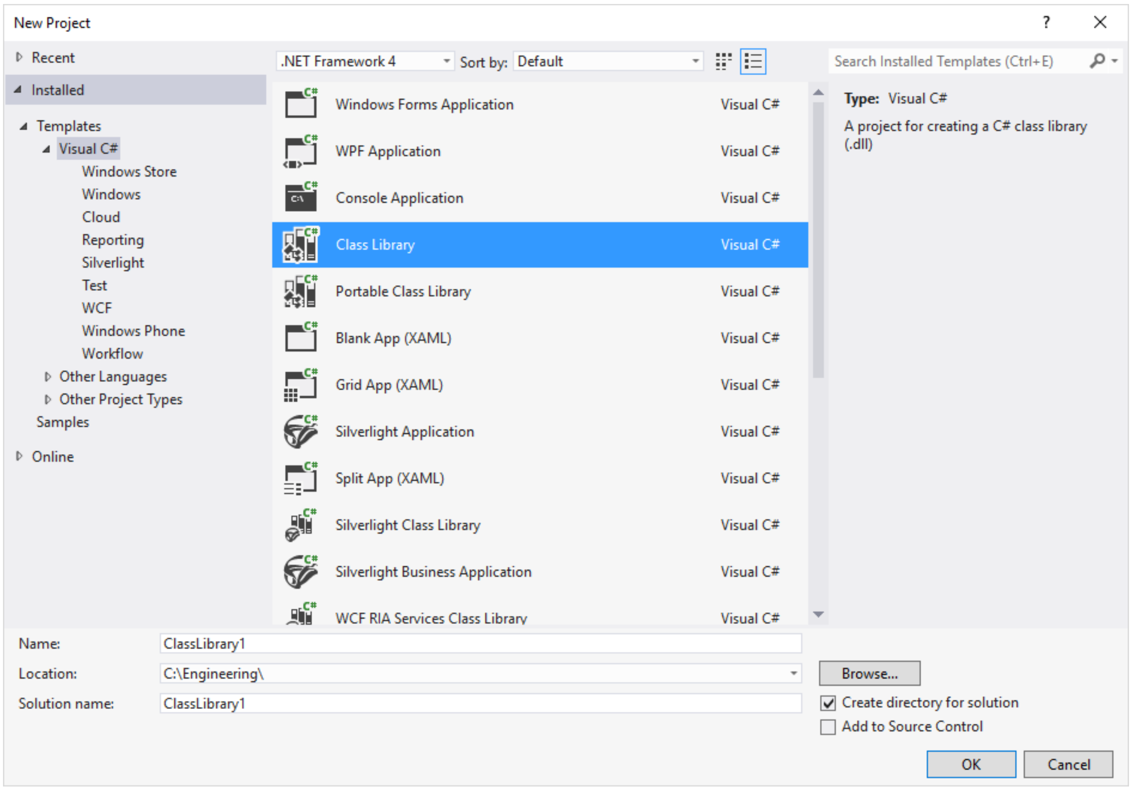
In the function eval, the parameter args is used to access the user defined arguments. Since args is an array of objects, the array much be addressed and then properly cast to the correct C# data type. In the Poly_Cubic example the arguments are data type doubles. In the calculation of cubic equation, each argument from the args array is cast to a double. After the calculation, the double val is returned from the function.

Setting Up a C# Project to Develop an Example Data Process DLL Plug-In

This section of the manual will focus on using configuring a Visual Studio C# project to build a DLL Plug-in. A DLL Plug-In can contain multiple classes where each class is a Data Plug-In.   An example Visual Studio C# project is available as part of the Data Process Software Development Kit (SDK) from Ulyssix. Please contact Ulyssix to receive a copy of the SDK   Contact Ulyssix for help using other development environments or Microsoft .NET managed languages.

The images for this tutorial are from Visual Studio 2012. Both newer and older versions of Visual Studio will work.

1. Open Visual Studio and select New Project to launch the New Project window.


New Project
?
Recent
Installed
Templates
Visual C#
Windows Store
Windows
Cloud
Reporting
Silverlight
Test
WCF
Windows Phone
Workflow
Other Languages
Other Project Types
Samples
Online
Search Installed Templates (Ctrl+E)
Windows Forms Application Visual C#
WPF Application Visual C#
Console Application Visual C#
Class Library Visual C#
Portable Class Library Visual C#
Blank App (XAML) Visual C#
Grid App (XAML) Visual C#
Silverlight Application Visual C#
Split App (XAML) Visual C#
Silverlight Class Library Visual C#
Silverlight Business Application Visual C#
WCF RIA Services Class Library Visual C#
Browse...
Add to Source Control
OK
Cancel

.NET Framework 4 · Sort by: Default

Type: Visual C#

A project for creating a C# class library (.dll)

Name: ClassLibrary1

Location: C:\Engineering\

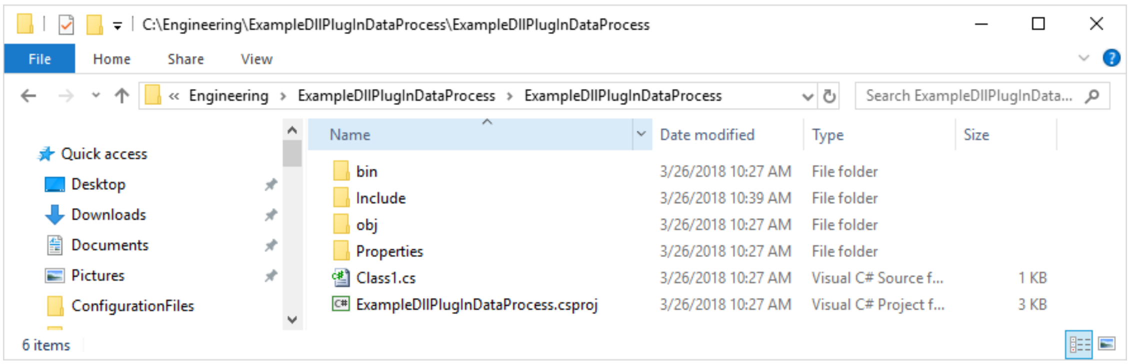
Solution name: ClassLibrary1

☑ Create directory for solution

Visual Studio New Project Window

2. In the drop-down box at the top, select .NET Framework 4. Select Class Library from the list in the center. At the bottom, enter the name of the DLL Plug-In and use the Browse button to select the location. Then click OK to continue
3. Visual Studio will create the project and launch the C# development environment.
4. The DLL Plug-In requires a reference to the ParamDataAPI.dll. This file is provided is part of the ALTAIR software and is also part of the Data Process SDK.
    a. It is highly recommended that ParamDataAPI.dll version 18.1 or higher is used. To check the version number, right click on the ParamDataAPI file, select Properties, and then select the Details tab.

b. To add the ParamDataAPI.dll to your Visual Studio project, open a Windows Explorer window and navigate to the base directory of your example Data Process Plug-In. The base directory contains a folder with the name of your project, open that folder and create a new folder named Include.


C:\Engineering\ExampleDllPlugInDataProcess\ExampleDllPlugInDataProcess
File
Home
Share
View
« Engineering › ExampleDllPlugInDataProcess › ExampleDllPlugInDataProcess
Search ExampleDllPlugInData...
Quick access
Desktop
Downloads
Documents
Pictures
ConfigurationFiles
6 items

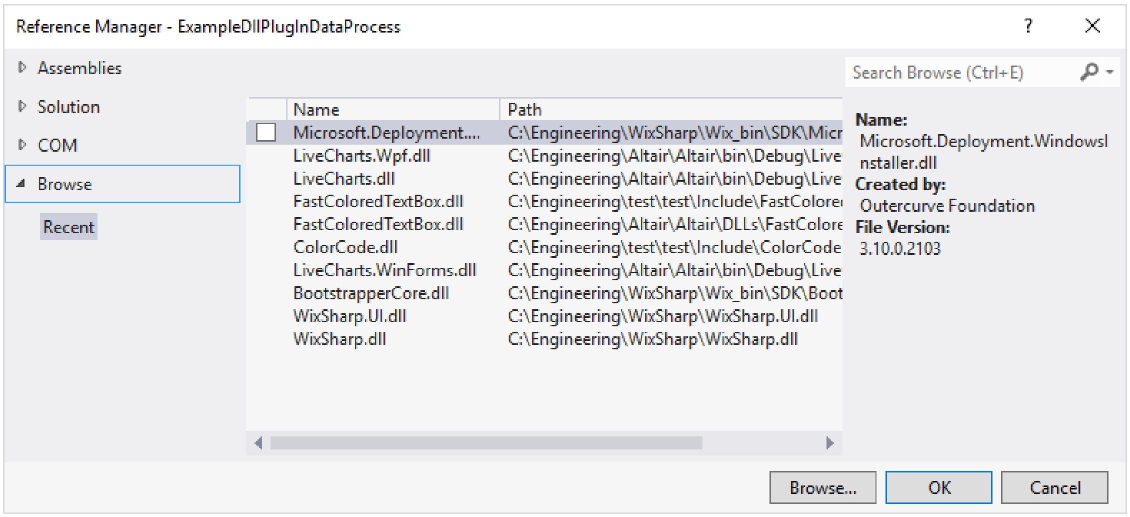
Name	Date modified	Type	Size
bin	3/26/2018 10:27 AM	File folder	
Include	3/26/2018 10:39 AM	File folder	
obj	3/26/2018 10:27 AM	File folder	
Properties	3/26/2018 10:27 AM	File folder	
Class1.cs	3/26/2018 10:27 AM	Visual C# Source f...	1 KB
ExampleDllPlugInDataProcess.csproj	3/26/2018 10:27 AM	Visual C# Project f...	3 KB

Windows Explorer Showing Location for the Include Folder

c. Paste the ParamDataAPI.dll into the Include folder.

d. Return to Visual Studio and open the Solution Explorer window. This window is often collapsed on one side of the C# development environment. It is also available using the View menu and selecting Solution Explorer.

e. Right click on Reference heading in the Solution Explorer and select Add Reference from the pop up window to launch the Reference Manger window.



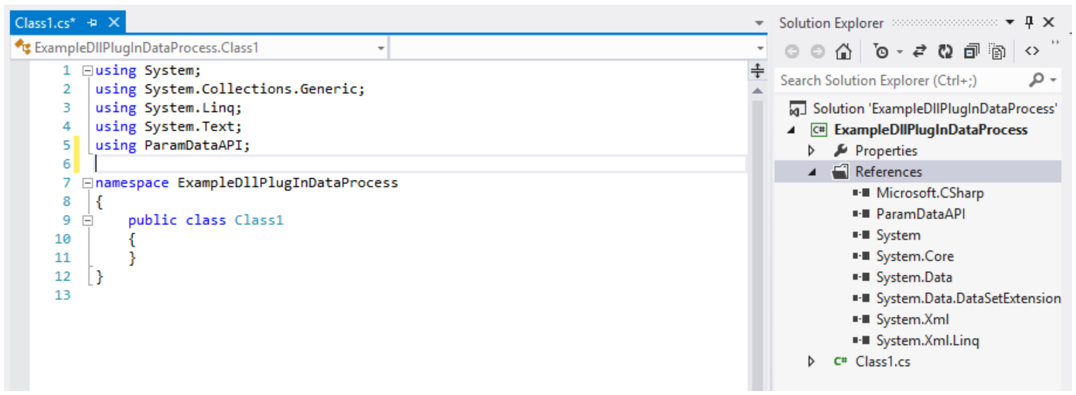
Name	Path
Microsoft.Deployment....	C:\Engineering\WixSharp\Wix_bin\SDK\Micr
LiveCharts.Wpf.dll	C:\Engineering\Altair\Altair\bin\Debug\Live
LiveCharts.dll	C:\Engineering\Altair\Altair\bin\Debug\Live
FastColoredTextBox.dll	C:\Engineering\test\test\Include\FastColore
FastColoredTextBox.dll	C:\Engineering\Altair\Altair\DLLs\FastColore
ColorCode.dll	C:\Engineering\test\test\Include\ColorCode
LiveCharts.WinForms.dll	C:\Engineering\Altair\Altair\bin\Debug\Live
BootstrapperCore.dll	C:\Engineering\WixSharp\Wix_bin\SDK\Boot
WixSharp.UI.dll	C:\Engineering\WixSharp\WixSharp.UI.dll
WixSharp.dll	C:\Engineering\WixSharp\WixSharp.dll

Name:
Microsoft.Deployment.WindowsInstaller.dll
Created by:
Outercurve Foundation
File Version:
3.10.0.2103

Visual Studio Reference Manager

f. Click the Browse button at the bottom and navigate to the Include folder. Select ParamDataAPI.dll and click Add. Click OK in the Reference Manager window to complete the task. ParamDataAPI will now appear in the list of References.

5. Add the command “using ParamDataAPI” to list of usings at the top of the file.

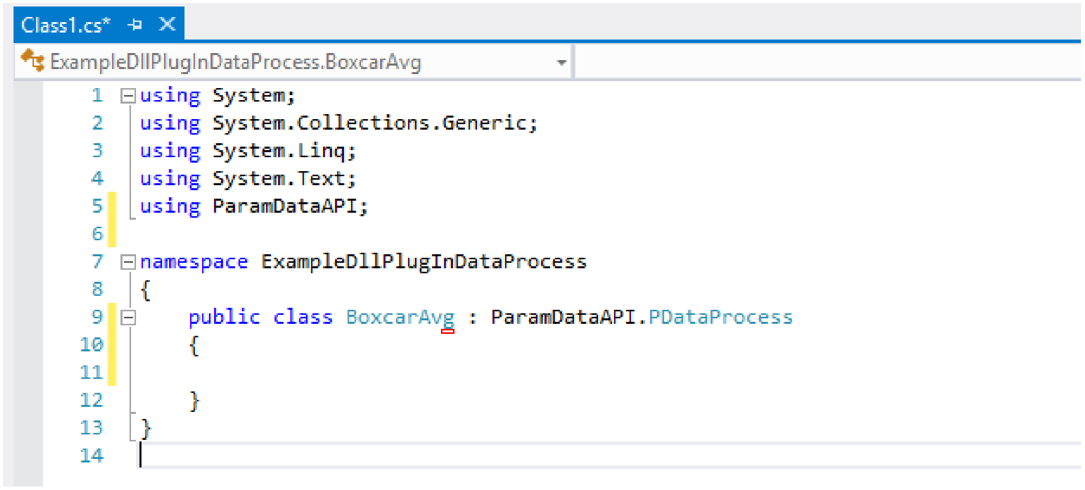

Class1.cs*
ExampleDllPlugInDataProcess.Class1
Solution Explorer
Search Solution Explorer (Ctrl+;)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using ParamDataAPI;

namespace ExampleDllPlugInDataProcess
{
    public class Class1
    {
    }
}
```

```
Solution 'ExampleDllPlugInDataProcess'
  ExampleDllPlugInDataProcess
    Properties
    References
      Microsoft.CSharp
      ParamDataAPI
      System
      System.Core
      System.Data
      System.Data.DataSetExtension
      System.Xml
      System.Xml.Linq
    Class1.cs
```

C# Development Environment with ParamDataAPI

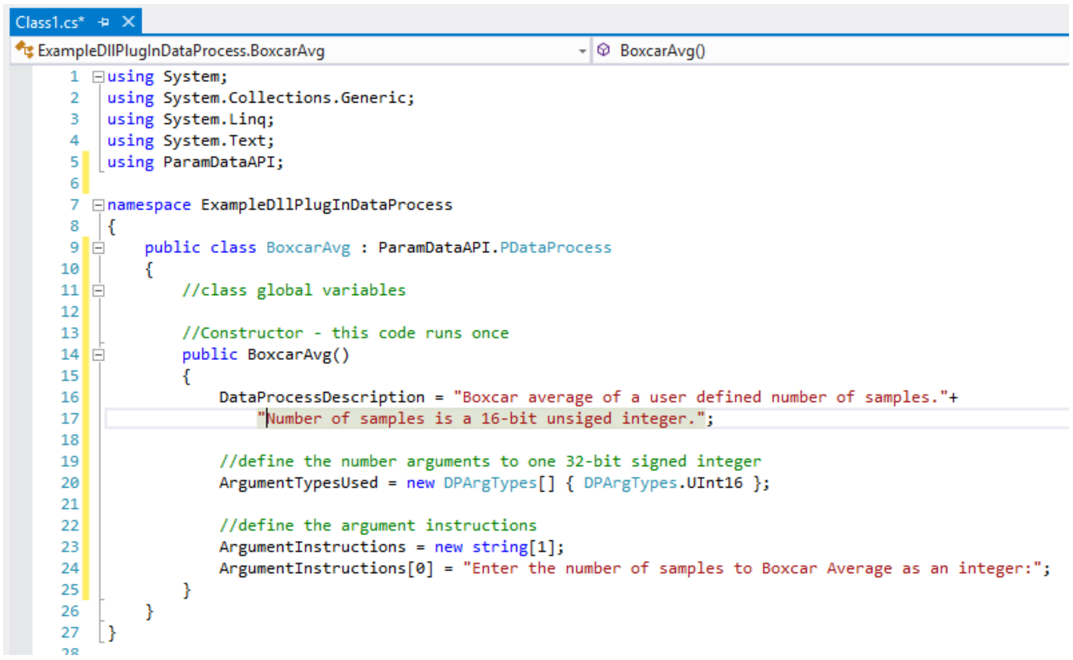
6. Each class included in the name space implements its own Data Process. The class can be renamed. Each class must implement the Param.PDataProcess interface. Do this by adding the following command to the end of the class definition: ": ParamterDataAPI.PDataProcess"



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using ParamDataAPI;

namespace ExampleDllPlugInDataProcess
{
    public class BoxcarAvg : ParamDataAPI.PDataProcess
    {

    }
}

```

Implement the PDataProcess Interface for the Class

7. Add a comment to denote the location for class global variables and add the constructor for class. The constructor must be public and cannot contain any parameters. In the constructor, add the DataProcessDescription, ArgumentTypesUsed, and the ArgumentInstructions. The ArgumentsTypesUsed in this example is one unsigned 16-bit. This argument defines the size of the boxcar.

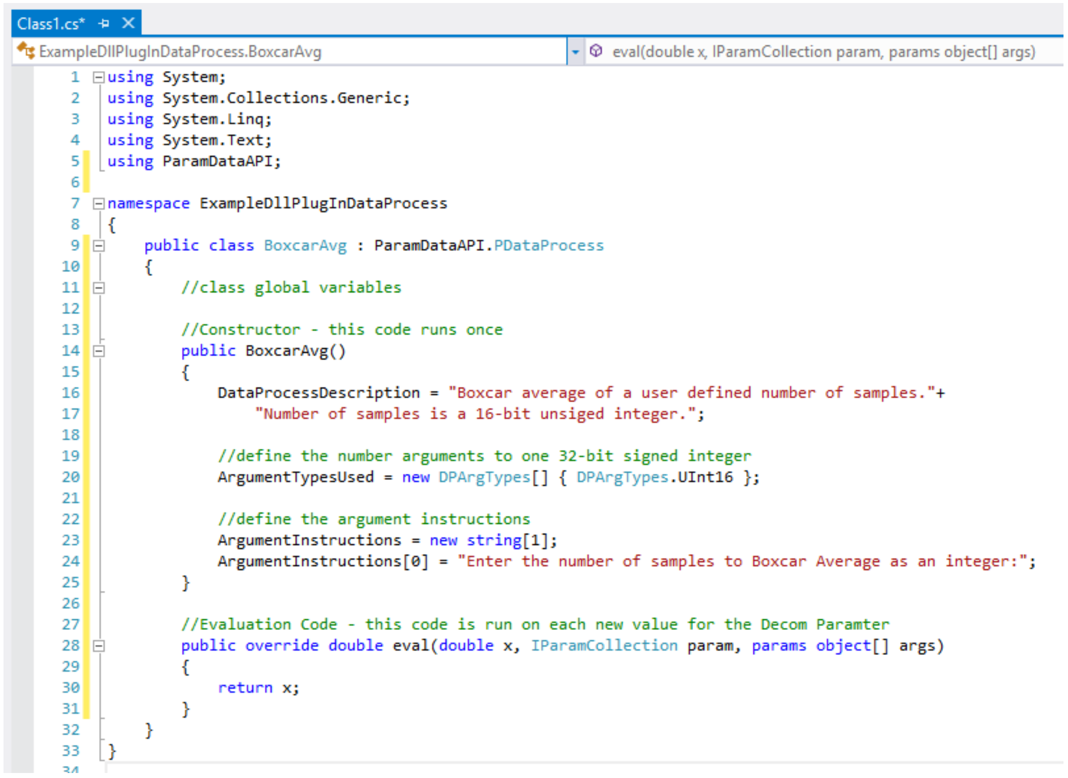


Class1.cs* ExampleDllPlugInDataProcess.BoxcarAvg BoxcarAvg()

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using ParamDataAPI;

namespace ExampleDllPlugInDataProcess
{
    public class BoxcarAvg : ParamDataAPI.PDataProcess
    {
        //class global variables

        //Constructor - this code runs once
        public BoxcarAvg()
        {
            DataProcessDescription = "Boxcar average of a user defined number of samples."+
                "Number of samples is a 16-bit unsiged integer.";

            //define the number arguments to one 32-bit signed integer
            ArgumentTypesUsed = new DPArgTypes[] { DPArgTypes.UInt16 };

            //define the argument instructions
            ArgumentInstructions = new string[1];
            ArgumentInstructions[0] = "Enter the number of samples to Boxcar Average as an integer:";
        }
    }
}
```

Add the Constructor with PDataProcess Variables

8. Add the implementation for the eval function. The eval function must be public. The eval function optionally can include the params object array. In the example below, the return value is the parameter x. The Data Process returns the unchanged raw value. This completes the requirements for the Data Process to compile and be used in Altair.

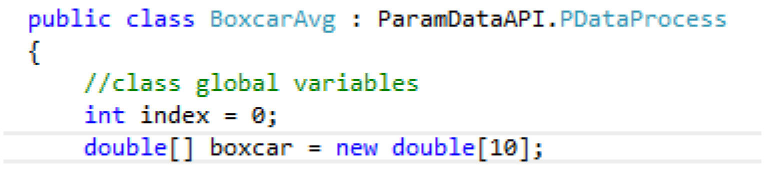


Class1.cs* ExampleDllPlugInDataProcess.BoxcarAvg eval(double x, IParamCollection param, params object[] args)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using ParamDataAPI;

namespace ExampleDllPlugInDataProcess
{
    public class BoxcarAvg : ParamDataAPI.PDataProcess
    {
        //class global variables

        //Constructor - this code runs once
        public BoxcarAvg()
        {
            DataProcessDescription = "Boxcar average of a user defined number of samples."+
                "Number of samples is a 16-bit unsiged integer.";

            //define the number arguments to one 32-bit signed integer
            ArgumentTypesUsed = new DPArgTypes[] { DPArgTypes.UInt16 };

            //define the argument instructions
            ArgumentInstructions = new string[1];
            ArgumentInstructions[0] = "Enter the number of samples to Boxcar Average as an integer:";
        }

        //Evaluation Code - this code is run on each new value for the Decom Paramter
        public override double eval(double x, IParamCollection param, params object[] args)
        {
            return x;
        }
    }
}
```

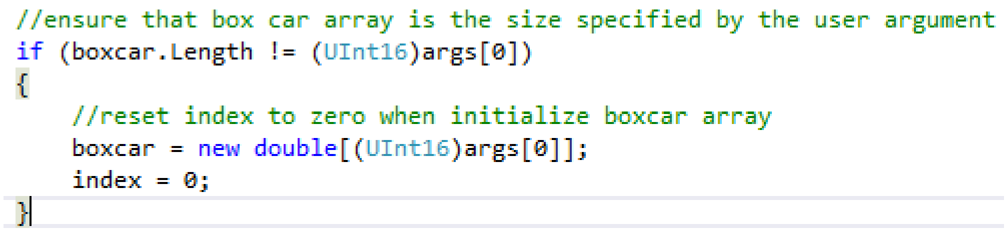
Add the eval Function

9. The next step is to add the needed C# code to implement the Boxcar Average.
    a. Two global variables are needed:
        i. An array of doubles is needed to hold the samples for the boxcar average.
        ii. An integer to hold the index in the array for the next sample to be added.



```
public class BoxcarAvg : ParamDataAPI.PDataProcess
{
    //class global variables
    int index = 0;
    double[] boxcar = new double[10];
```

    b. Code in the eval function to implement the correct size of the box car average.



```
//ensure that box car array is the size specified by the user argument
if (boxcar.Length != (UInt16)args[0])
{
    //reset index to zero when initialize boxcar array
    boxcar = new double[(UInt16)args[0]];
    index = 0;
}
```

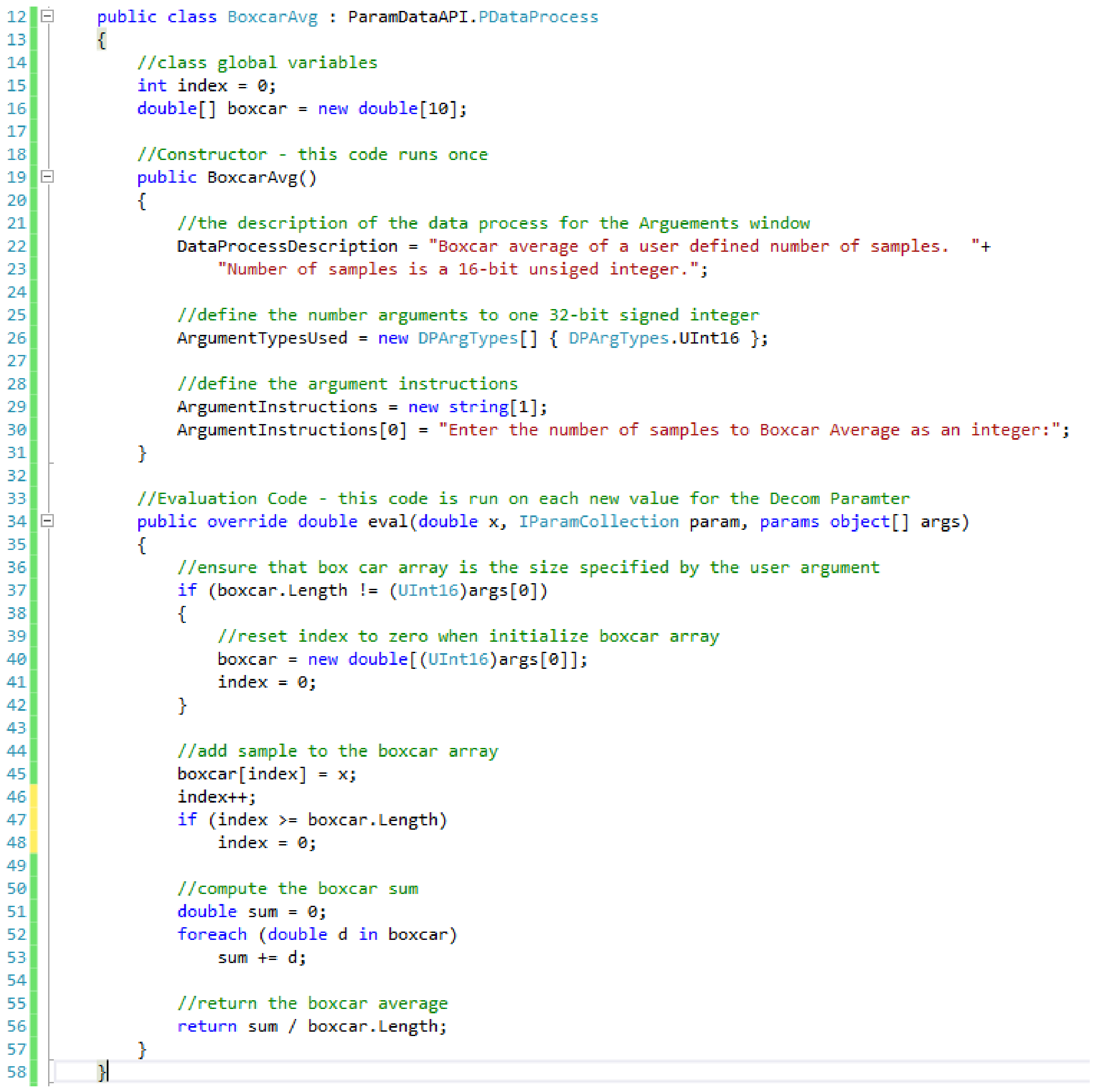
    c. Code to add a sample to the box car array.

```
//add sample to the boxcar array
boxcar[index] = x;
index++;
if (index >= boxcar.Length)
    index = 0;
```

    d. Code to compute the boxcar sum and return the boxcar average.

```
//compute the boxcar sum
double sum = 0;
foreach (double d in boxcar)
    sum += d;

//return the boxcar average
return sum / boxcar.Length;
```

10. Here is the complete code for the Boxcar Average Data Process:



```
    public class BoxcarAvg : ParamDataAPI.PDataProcess
    {
        //class global variables
        int index = 0;
        double[] boxcar = new double[10];

        //Constructor - this code runs once
        public BoxcarAvg()
        {
            //the description of the data process for the Arguements window
            DataProcessDescription = "Boxcar average of a user defined number of samples.  "+
                "Number of samples is a 16-bit unsiged integer.";

            //define the number arguments to one 32-bit signed integer
            ArgumentTypesUsed = new DPArgTypes[] { DPArgTypes.UInt16 };

            //define the argument instructions
            ArgumentInstructions = new string[1];
            ArgumentInstructions[0] = "Enter the number of samples to Boxcar Average as an integer:";
        }

        //Evaluation Code - this code is run on each new value for the Decom Paramter
        public override double eval(double x, IParamCollection param, params object[] args)
        {
            //ensure that box car array is the size specified by the user argument
            if (boxcar.Length != (UInt16)args[0])
            {
                //reset index to zero when initialize boxcar array
                boxcar = new double[(UInt16)args[0]];
                index = 0;
            }

            //add sample to the boxcar array
            boxcar[index] = x;
            index++;
            if (index >= boxcar.Length)
                index = 0;

            //compute the boxcar sum
            double sum = 0;
            foreach (double d in boxcar)
                sum += d;

            //return the boxcar average
            return sum / boxcar.Length;
        }
    }
```

Completed Boxcar Average DLL Plug-In Data Process

11. In the Solution Explorer, right click on the solution and select Build from the pop-up menu. If there are no errors, Visual Studio will build the DLL. Go to the destination folder (\bin\debug for Debug mode and \bin\release for Release mode) and copy the compiled DLL. Navigate to the ALTAIR folder (usually C:\Program Files(x86)\Ulyss\Altair) and locate the Process_Plugins folder. Paste the DLL into the Process_Plugins folder.

ParamDataAPI

The ParamDataAPI is a DLL that is part of the ALTAIR software and is required in every C# Function Data Process. It is included by the statement “using ParamDataAPI” on line three of the Blank C# Template. ParamDataAPI includes the enum DPArgTypes and the classes IParamCollection, IParamData, and PDataProcess. IParamCollection is an object that contains multiple instances of the IParamData. There is one instance of IParamData for each decom parameter in ALTAIR.

The IParamCollection variable “param” is included in the eval function in every C# Function Data Process (see line 15 in the C# Blank Template). In a C# Function Data Process, the IParamCollection variable “param” is used to access any decom parameter.

Enums

The enum DPArgTypes defines the data type used for an argument in DLL Plug-In Data Process. The arguments are stored as generic objects and require being cast back to the correct data type. The enum allows the DLL Plug-In to define the data type as well communicate the data type back to ALTAIR.

```
enum DPArgTypes
{
	Int8 = 0,
	Int16 = 1,
	Int32 = 2,
	Int64 = 3,
	UInt8 = 4,
	UInt16 = 5,
	UInt32 = 6,
	UInt64 = 7,
	Float = 8,
	Double = 9,
	String = 10,
	UInt32Bin = 11,
	ParamName = 12,
	ParamNumber = 13,
	ParamNameTrigger = 14,
	ParamNumberTrigger = 15,
	Enum = 16
}
```

The enum ProcessDataType defines the data type for the return type of the Data Process as well as the parameter x in the function eval. Currently, only the FloatingPoint ProcessData type is implemented.

```
public enum ProcessDataType
{
	Integer = 0,
	FloatingPoint = 1,
}
```

The enum ProcessType defines the process type of the Data Process.

```
public enum ProcessType
```

```
{
	Formula = 0,
	Fuction = 1,
	PlugIn = 2,
}
```

Class IParamData

IParaData is a class that holds the description and data of a decom parameter. The following properties and methods for the IParamData class are listed below:

double IParamData.GetSampleRate

Inputs:	None.
Returns:	Sample rate in samples per second as a double floating-point number.
Purpose:	Get the sample rate from IParamData.
Remarks:	None.
Example:	double sRate = IParam.GetSampleRate; //IParamData
	double sRate = IParamCol[1].GetSampleRate; //IParamCollection

string IParamData.ParamName

Inputs:	None.
Returns:	Decom parameter name from this IParamData.
Purpose:	Get the decom parameter name from IParamData.
Remarks:	None.
Example:	string name = IParam.ParamName; //IParamData
	string name = IParamCol[1].ParamName; //IParamCollection

uint IParamData.ParamNumber

Inputs:	None.
Returns:	Decom parameter as an unsigned integer from this IParamData.
Purpose:	Get the decom parameter number from IParamData.
Remarks:	None.
Example:	uint num = IParam.ParamNumber; //IParamData
	uint num = IParamCol["SFID"].ParamNumber; //IParamCollection

double IParamData.processed

Inputs:	None.
Returns:	Current value for the parameter as a double precision float. The processed value is after any Data Process is applied.
Purpose:	Accesses the current processed value of the IParamData.
Remarks:	None.
Example:	long raw = IParam.processed; //IParamData
	long raw = IParamCol[1].processed; //IParamCollection
	long raw = IParamCol["SFID"].processed; //IParamCollection

long IParamData.raw

Inputs:	None.
Returns:	Current value for the parameter as a signed 64-bit integer. The raw value is before any Data Process is applied.
Purpose:	Accesses the current raw value of the IParamData.
Remarks:	None.

Example:	`long raw = IParam.raw;                        //IParamData`
	`long raw = IParamCol[1].raw;                  //IParamCollection`
	`long raw = IParamCol[”sfid”].raw;             //IParamCollection`

ulong IParamData.TimeStamp_1uS

Inputs:	None.
Returns:	Current timestamp for the parameter in microseconds into the current year. Includes leap day if applicable.
Purpose:	Accesses the timestamp of the current IParamData value.
Remarks:	None.

Example:	`ulong uS = IParam.TimeStamp_1uS;              //IParamData`
	`ulong uS = IParamCol[1]. TimeStamp_1uS;       //IParamCollection`
	`ulong uS = IParamCol[”SFID”]. TimeStamp_1uS;  //IParamCollection`

ulong IParamData.TimeStamp_100nS

Inputs:	None.
Returns:	Current timestamp for the parameter in hundreds of nanoseconds into the current year. Includes leap day if applicable.
Purpose:	Accesses the timestamp of the current IParamData value.
Remarks:	None.

Example:	`ulong nS100 = IParam.TimeStamp_100nS;             //IParamData`
	`ulong nS100 = IParamCol[1]. TimeStamp_100nS;      //IParamCollection`
	`ulong nS100 = IParamCol[”SFID”]. TimeStamp_100nS;//IParamCollection`

Class IParamCollection

This is analogous to using the Multi-Parameter Formula list box in a Formula Data Process. The available properties and methods for the IParamCollection are listed below:

IParamData IParamCollection[int index]

Inputs:	Integer parameter number for desired decom parameter.
Returns:	IParamData for the decom parameter number.
Purpose:	Accesses a IParamData in IParamCollection by its parameter number.
Remarks:	Invalid parameter number throws exception.
Example:	`IParamData pData = IParamCol[1];`

IParamData IParamCollection [string name]

Inputs:	String parameter name. Case sensitive.
Returns:	IParamData for the decom parameter name.
Purpose:	Accesses a IParamData in IParamCollection by its parameter name.
Remarks:	Invalid parameter name throws exception.
Example:	IParamData pData = IParamCol [”SFID”];

Int IParamCollection.Count

Inputs:	None.
Returns:	Integer number of IParamData inside IParamCollection.
Purpose:	Get the number of IParamData in the IParamCollection.
Remarks:	None.
Example:	int paramCount = IParamCol .Count;

Int IParamCollection.GetIndexForParamName(string name)

Inputs:	String parameter name. Case sensitive.
Returns:	Parameter number as integer. Returns -1 if name is not found.
Purpose:	Get the parameter number by name. Allows searching for a parameter Name with error checking if the parameter number is less than zero.
Remarks:	None.
Example:	int paramNum= IParamCol .GetIndexForParamName(”SFID”);

Class PDataProcess

The PDataProcess class provides information about the Data Process and

string ProcessName

Inputs:	None.
Returns:	The name of the Data Process as a string.
Purpose:	To access the name of the Data Process
Remarks:	None.
Example:	string name = ProcessName;

string DataProcessDescription

Inputs:	None.
Returns:	The description of the Data Process as a string.
Purpose:	To access the description of the Data Process
Remarks:	None.
Example:	string desc = DataProcessDescription;

ProcessType PType

Inputs:	None.
Returns:	Enum for the process type of the Data Process.

Purpose: To determine if the Data Process is a Formula, Function, or Plug-In.
Remarks: None.
Example: ProcessType procType = PType;

ProcessDataType PDataType

Inputs: None.
Returns: Enum for the data type for the Data Process.
Purpose: To determine if the Data Process returns an Integer or Floating Point.
Remarks: Only the Floating Point ProcessDataType is implemented.
Example: ProcessDataType procDataType = PDataType;

string[] ArgumentInstructions

Inputs: None.
Returns: Array of strings that contain the instructions for all arugmentes.
Purpose: Accesses instructions for the use and requirements of arguments.
Remarks: None.
Example: string instruc = ArgumentInstructions[0];

DPArgTypes[] ArgumentTypesUsed

Inputs: None.
Returns: Array of enum DPArgTypes for the data type used for the argument.
Purpose: Accesses data type for the arguments of the Data Process.
Remarks: None.
Example: DPArgTypes dt = ArgumentTypeUsed[0];

string[][] ArgumentEnums

Inputs: None.
Returns: Array of Enums where a single Enum is an array of strings.
Purpose: To define all of the Enums, and their values, used in the Data Process.
Remarks: The first index is the Enum. The second index is the string value of the Enum.
Example: string[] MyEnum = ArgumentEnum[0][];
Example: string EnumTwo = MyEnum[1];

string[] ParamNamesUsed

Inputs: None.
Returns: Array of parameter names used in the Data Process.
Purpose: Accesses parameter names used in the Data Process.
Remarks: None.
Example: string name0 = ParamNamesUsed[0];

int[] ParamNumbersUsed

Inputs: None.
Returns: Array of parameter numbers used in the Data Process.

Purpose:	Accesses parameter numbers used in the Data Process.
Remarks:	None.
Example:	int num0 = ParamNumbersUsed[0];

int TimeStampParam

Inputs:	None.
Returns:	Number of parameter that is used to calculate time stamp.
Purpose:	Determine which parameter number is used to calculate the time stamp.
Remarks:	A value of -1 indicates the TimeStampParam is the same as the parameter for the Data Process.
Example:	int tsParam = TimeStampParam;

int TriggerParam

Inputs:	None.
Returns:	Number of parameter that is used to trigger the Data Process calculation.
Purpose:	Determine which parameter number is used as the trigger.
Remarks:	A value of -1 indicates the Trigger Param is the same as the parameter for the Data Process.
Example:	int trigParam = TriggerParam;

double eval (double x, IParamCollection param)

Inputs:	double x is the raw value of the decom parameter to which the Data Process is applied. IParamCollection param is an object containing all of the IParamData for the telemetry setup.
Returns:	The result of the Data Process.
Purpose:	Function prototype used in a DLL Plug-In Data Process.
Remarks:	Function prototype using double precision floating point numbers and the reference to IParamCollection.
Example:	See example DLL Plugin;

double eval (double x, IParamCollection param, params object[] args)

Inputs:	double x is the raw value of the decom parameter to which the Data Process is applied. IParamCollection param is an object containing all of the IParamData for the telemetry setup. params object[] args is an array of objects that contain all of the arguments for the Data Process. Each object must be cast to its proper type.
Returns:	The result of the Data Process.
Purpose:	Function prototype used in a DLL Plug-In Data Process.

Remarks:	Function prototype using double precision floating point numbers and the reference to IParamCollection and the array of objects for the arguments.
Example:	See example DLL Plugin;

long eval (long x, IParamCollection param)

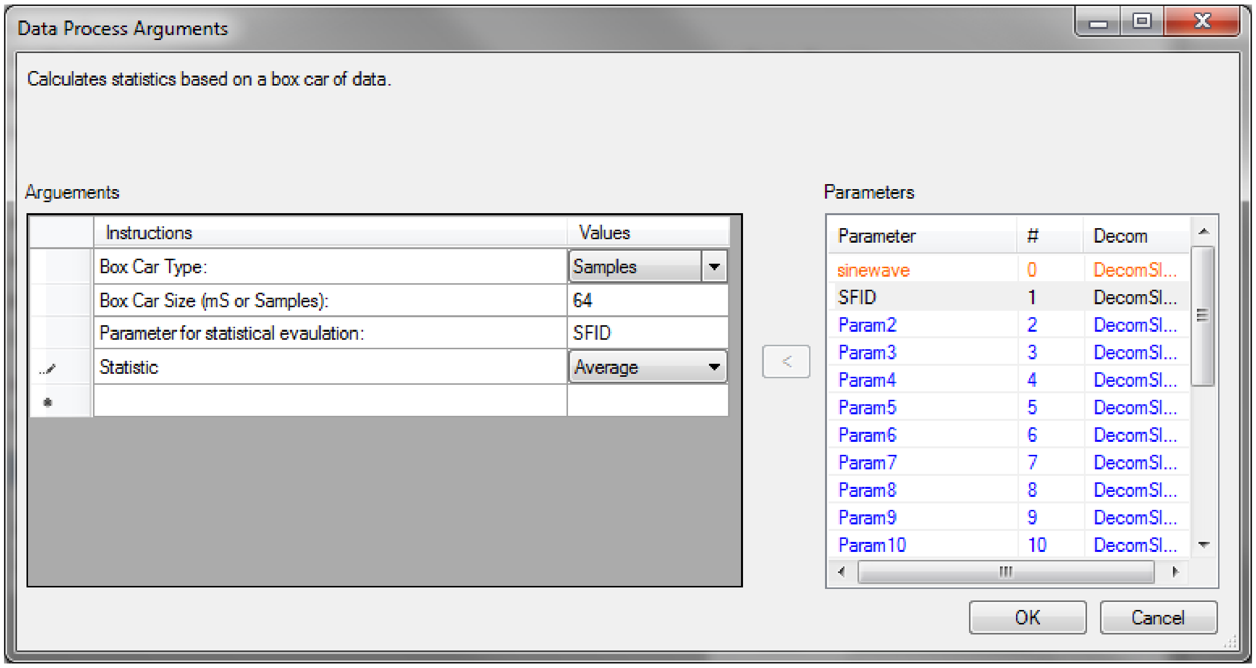
Inputs:	long x is the raw value of the decom parameter to which the Data Process is applied. IParamCollection param is an object containing all of the IParamData for the telemetry setup.
Returns:	The result of the Data Process.
Purpose:	Function prototype used in a DLL Plug-In Data Process.
Remarks:	Function prototype using signed 64-bit integers and the reference to IParamCollection. Not currently implemented.
Example:	See example DLL Plugin;

long eval (long x, IParamCollection param, params object[] args)

Inputs:	long x is the raw value of the decom parameter to which the Data Process is applied. IParamCollection param is an object containing all of the IParamData for the telemetry setup. params object[] args is an array of objects that contain all of the arguments for the Data Process. Each object must be cast to its proper type.
Returns:	The result of the Data Process.
Purpose:	Function prototype used in a DLL Plug-In Data Process.
Remarks:	Function prototype using signed 64-bit integers including the reference to IParamCollection and the array of objects for the arguments. Not currently implemented.
Example:	See example DLL Plugin;

Appendix A – Using DPArgsType.Enum

The DPArgsType.Enum allows the DLL Plug-In to implement a combo box based on an Enum in the Data Process Argument window. This feature is used to limit the options that a user can enter for an argument.



Calculates statistics based on a box car of data.

Arguements

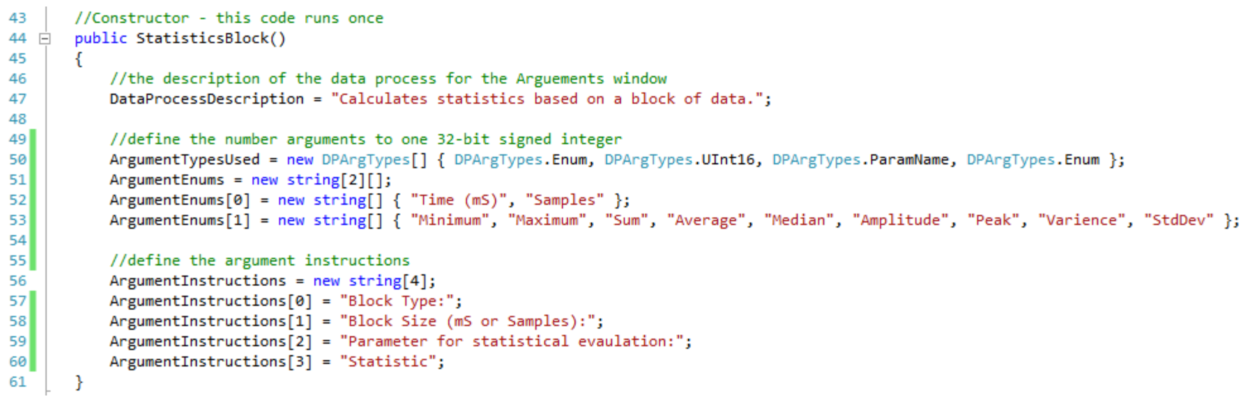
Instructions	Values
Box Car Type:	Samples
Box Car Size (mS or Samples):	64
Parameter for statistical evaulation:	SFID
Statistic	Average

Parameters

Parameter	#	Decom
sinewave	0	DecomSl...
SFID	1	DecomSl...
Param2	2	DecomSl...
Param3	3	DecomSl...
Param4	4	DecomSl...
Param5	5	DecomSl...
Param6	6	DecomSl...
Param7	7	DecomSl...
Param8	8	DecomSl...
Param9	9	DecomSl...
Param10	10	DecomSl...

Data Process Argument Window with Combo Boxes

In the example above, the first and fourth arguments are combo boxes. The combo boxes are implemented by setting the ArguementTypesUsed to DPArgTypes.Enum and then adding the desired strings to the PDataProcess ArgumentEnums. The ArgumentEnum is a jagged two dimensional array of strings. The first array index selects the desired Enum. The second array index selects the desired entry of the Enum.



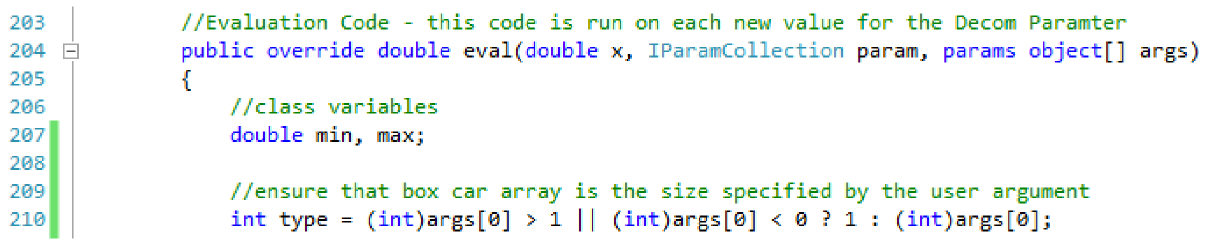
```
43      //Constructor - this code runs once
44      public StatisticsBlock()
45      {
46          //the description of the data process for the Arguements window
47          DataProcessDescription = "Calculates statistics based on a block of data.";
48
49          //define the number arguments to one 32-bit signed integer
50          ArgumentTypesUsed = new DPArgTypes[] { DPArgTypes.Enum, DPArgTypes.UInt16, DPArgTypes.ParamName, DPArgTypes.Enum };
51          ArgumentEnums = new string[2][];
52          ArgumentEnums[0] = new string[] { "Time (mS)", "Samples" };
53          ArgumentEnums[1] = new string[] { "Minimum", "Maximum", "Sum", "Average", "Median", "Amplitude", "Peak", "Varience", "StdDev" };
54
55          //define the argument instructions
56          ArgumentInstructions = new string[4];
57          ArgumentInstructions[0] = "Block Type:";
58          ArgumentInstructions[1] = "Block Size (mS or Samples):";
59          ArgumentInstructions[2] = "Parameter for statistical evaulation:";
60          ArgumentInstructions[3] = "Statistic";
61      }
```

Code Example of an Enum Combobox

Line 50 defines the DPArgTypes array. The first and fourth elements are DPArgsType.Enum. To define the Enum values, the PDataProcess ArgumentEnum is defined on lines 51-53. Line 51 defines

ArgumentEnum as two dimension jagged array that contains two string arrays. The first string array contains strings for the combo box for the Block Type (as defined on line 57). The second string array array contains strings for the combo box for Statistic (as defined on line 60).

The arguments parameter, object[] args, in to the function eval contains the user defined arguments from the Data Process Argument window. This argument for a DPArgType.Enum is the integer representing the index of the selected parameter. This integer should be inside of size of the ArguementEnum array, but error checking it is always a wise decision. Below is an implementation of error checking the argument.



```
203          //Evaluation Code - this code is run on each new value for the Decom Paramter
204          public override double eval(double x, IParamCollection param, params object[] args)
205          {
206              //class variables
207              double min, max;
208
209              //ensure that box car array is the size specified by the user argument
210              int type = (int)args[0] > 1 || (int)args[0] < 0 ? 1 : (int)args[0];
```

Example Code for Accessing a DPArgsType.Enum

On line 210, the code above casts args[0] to an integer and limits the allowed value to 0 or 1. The first argument is Block Type. Block Type should be either index of 0 or 1 because the Enum only has two options: Time (mS) or Samples.