IADS Programming User Guide

Jul 2020
Curtiss-Wright Document SSD-IADS-140

Table of Contents

1. Programming Resources

This guide details methods to programmatically extend IADS by using step by step examples many of which have working projects with source code that are available for download from the IADS Web site.

1.1 Overview

IADS provides programmatic extension to the core system using several techniques. Within the IADS Client, plug-ins can be written using Microsoft COM technology, including new displays, additions to the derived computational engine and custom data export extensions. The IADS Client also extends an automation interface for easy scripting. For access to external data and configuration files IADS provides a set of COM based libraries which can be linked into your programs. Finally, for data processing, IADS includes a standard Ethernet protocol for custom real time data interfaces and a command and control interface for real time operation.

2. Custom ActiveX Display Plugins

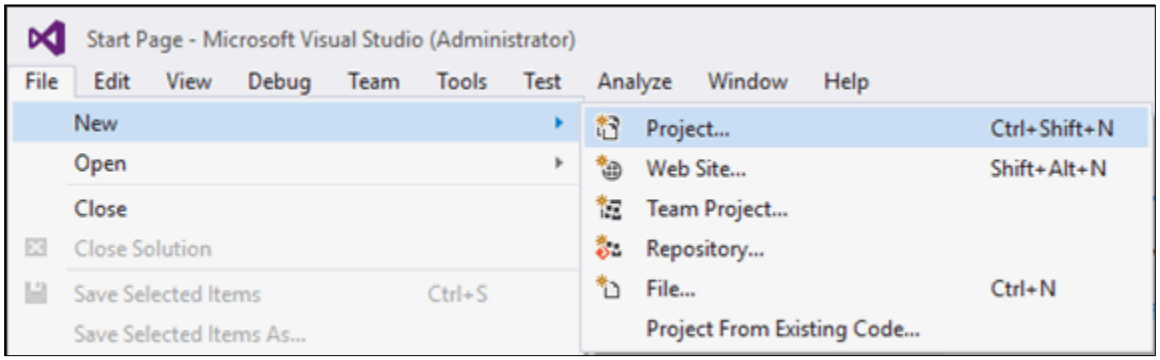
For more background on how to build an ActiveX display, download the sample ActiveX Display project from the Curtiss Wright IADS website and read the comments within the code: https://iads.symvionics.com/support/programming-examples/

Warning- Be careful about pasting code directly from this tutorial. For instance, Visual Studio encapsulates strings in different quotation marks (`""`) than the standard quotes in Word. (“”). You may need to type certain things out manually or edit existing code slightly.

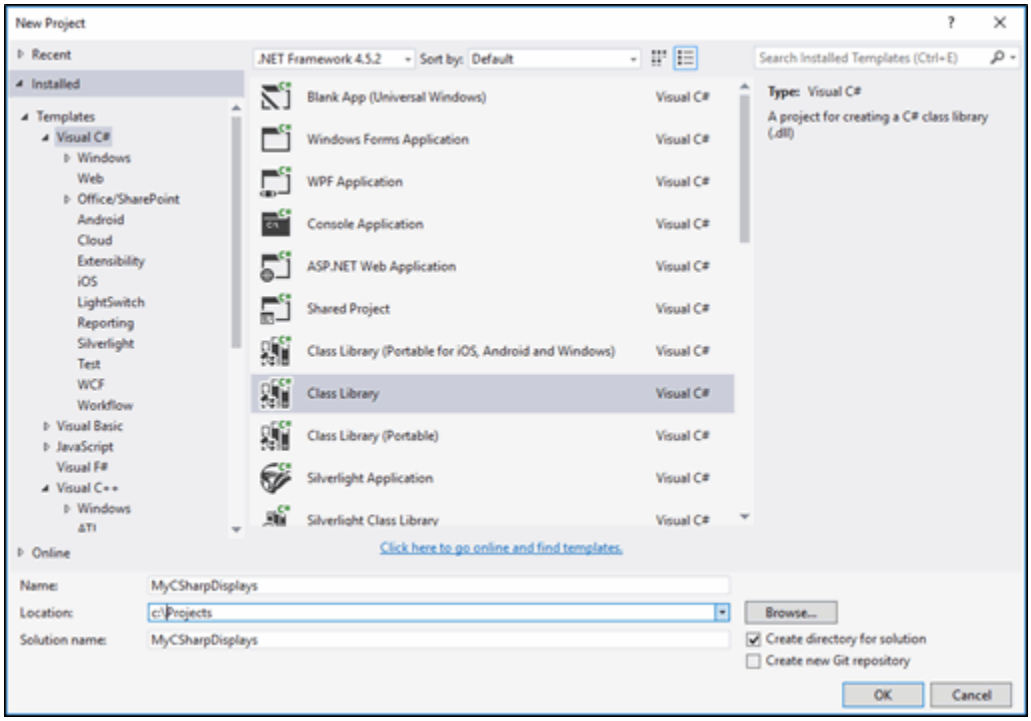
2.1 Creating an IADS custom ActiveX control using C# VS2015

This tutorial assumes you are using Microsoft Visual Studio 2015. It should apply to other versions with minimal modification. This instruction will guide you through the process of creating a custom display for IADS using the project wizard in Visual C#.

1) Open up VS2015 and Select **File** > **New** > **Project**.



2) In the New Project dialog that appears, choose the **Other Languages** > **Visual C#** tier and click the **Class Library** option. At this point, please read the next step before you finish completing the dialog. There are some important considerations when choosing the proper project name.

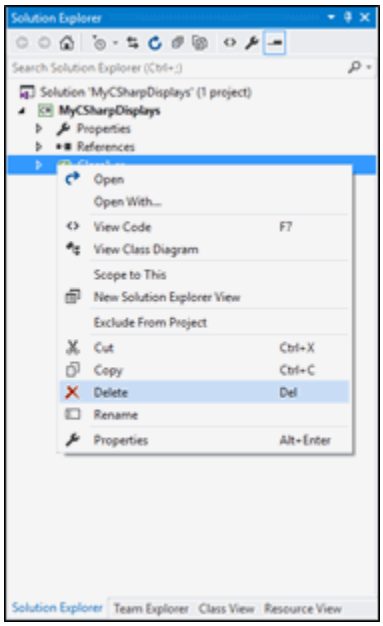


3) The project name you choose will become part of the display identifier name (also known as the ProgID, see note below). When it comes time to use your control in IADS, users will insert your new control into the "Display Builder" toolbox based solely upon its name (more on this later). Plan on creating many displays in one "project" (most common and easier to manage the code). Choose a general project name like "AircraftGauges" or "FluidSystemDisplays". One way to look at it is that the project name is akin to the "Genus" of your display, so shoot for generality. Consider prefixing the project name with your organization like "NASA" or "Lockheed", as it may easier for users to locate your control the "Display Builder" list (i.e. NasaFluidSystemDisplays or LockheedAircraftGauges).

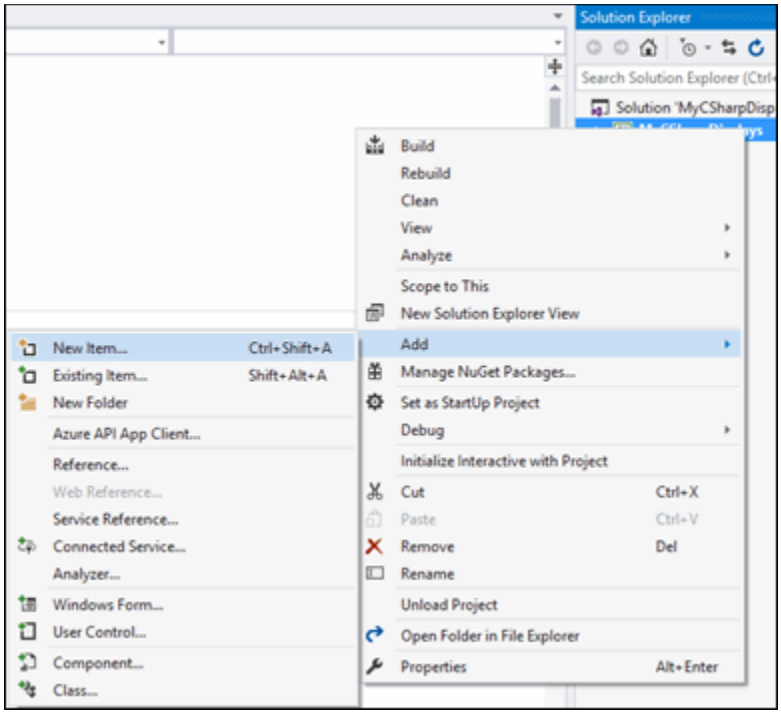
   Note: Microsoft refers to your display's name as its "ProgID" (also known as the Program ID). This is the string equivalent of your GUID (Global Unique Identifier) for the function. These Ids are placed in the Microsoft registry (directly from your project's ".rgs" file), allowing your object to be created without any knowledge of the location of your "Dll" on the file system. Of course, this assumes that it is registered using the "regsvr32" program (consult the Microsoft documentation).

   Now, in the fields at the bottom of the dialog, enter the project name, location, and the solution name.

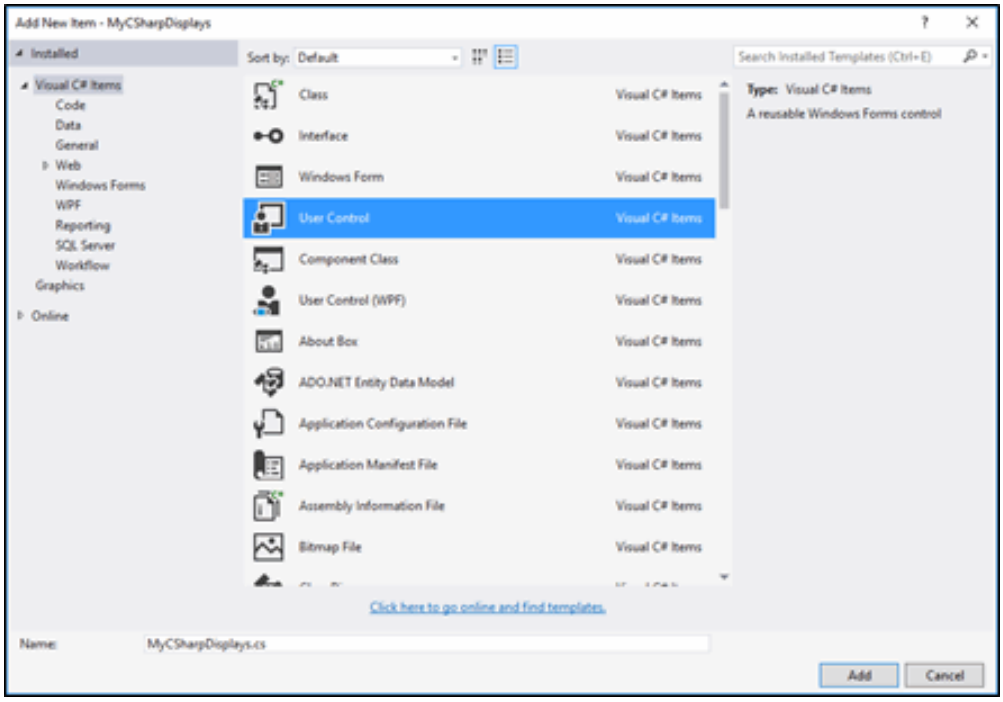
4) After pressing OK, the project will be ready for editing. Before we begin, delete the Class1.cs file. We will not need the file.



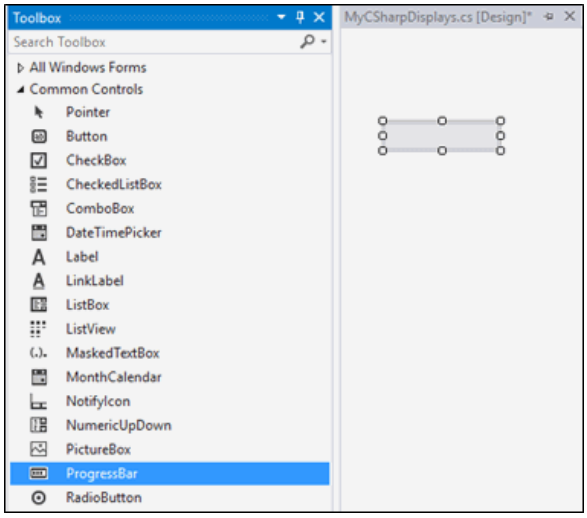
5) Right-click on your project in the Solution Explorer and choose **Add** > **New Item**.



6) In the "Add New Item" dialog select **User Control**. Enter the name of your display in the field and click the **Add** button.

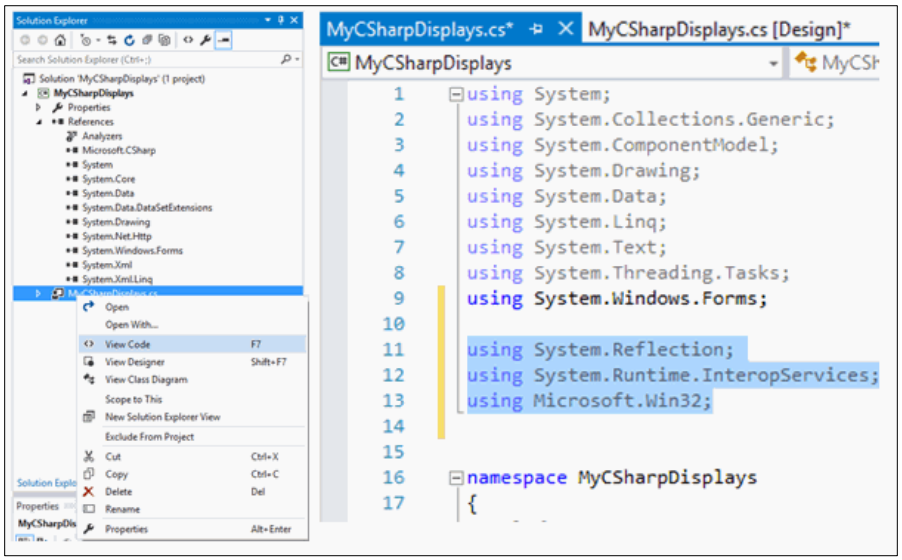


7) Visual Studio now displays the "Design" view of the display and shows a blank form. You can use the Visual Studio "Toolbox" to add a visual object. Drop a ProgressBar into the form; it is located under the "Common Controls" tier in the Toolbox.



8) Now that we are done adding components, we will need to do some work in the code section. Go to the Solution Explorer and right-click on your "displays.cs" file. Select **View Code**. In your "displays.cs" file, add the following using clauses:
using System.Reflection;


```
using System.Runtime.InteropServices;
using Microsoft.Win32;
```

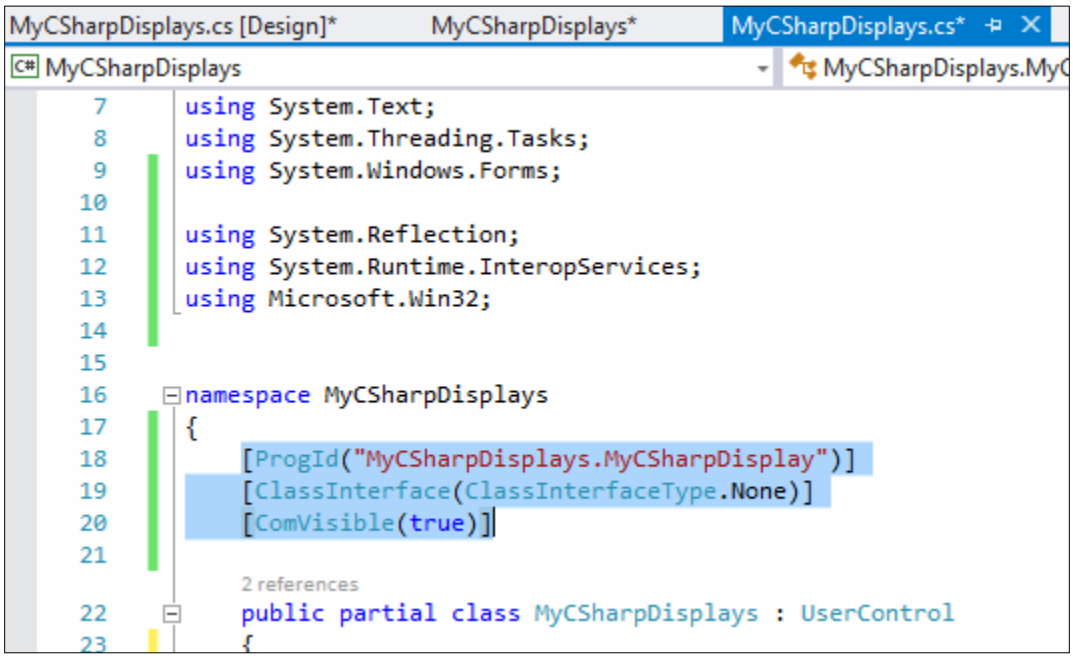


9) Now add the following attributes to your class:

```
[ProgId("MyCSharpDisplays.MyCSharpDisplay")]
[ClassInterface(ClassInterfaceType.None)]
[ComVisible(true)]
```

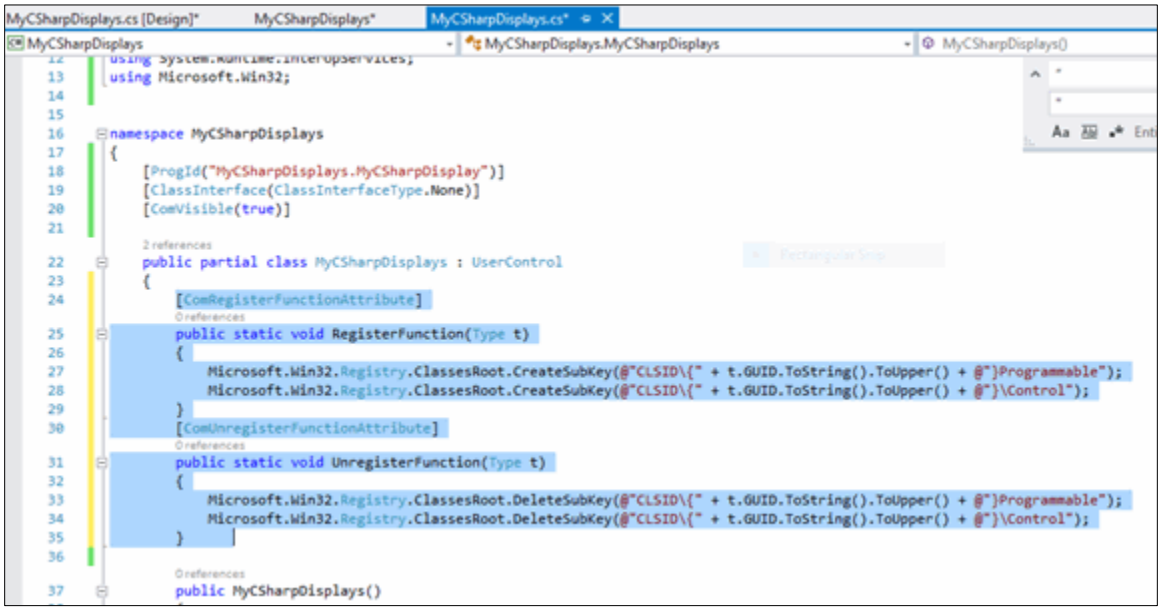
Note that the ProdId attribute is the same "ProgID" as mentioned in step 3. This will become your display's name inside of IADS, so choose appropriately. If you have followed the steps correctly, the name should be in the form of ProjectName.ClassName.

There are several ClassInterfaceType options. "None" provides the fewest default properties beyond the ones you explicitly specify.


MyCSharpDisplays.cs [Design]*
MyCSharpDisplays*
MyCSharpDisplays.cs*
MyCSharpDisplays
MyCSharpDisplays.MyC
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Reflection;
using System.Runtime.InteropServices;
using Microsoft.Win32;
namespace MyCSharpDisplays
{
[ProgId("MyCSharpDisplays.MyCSharpDisplay")]
[ClassInterface(ClassInterfaceType.None)]
[ComVisible(true)]
2 references
public partial class MyCSharpDisplays : UserControl
{

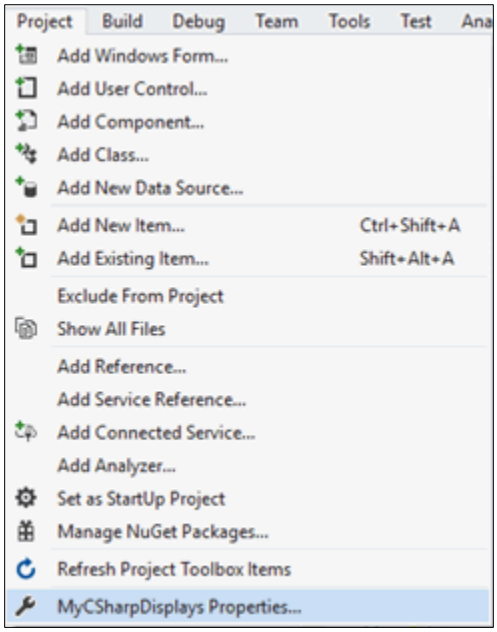
10) Add the following methods to perform the COM registration functions. COM is the interface method that IADS will use to send/receive data, save, and load your display. The registration mechanism is the manner in which IADS will identify your display after it is installed on a PC. If you skip this step, you will not be able to see your display in the IADS Display Builder dialog.

```
[ComRegisterFunctionAttribute]
        public static void RegisterFunction(Type t)
        {
            Microsoft.Win32.Registry.ClassesRoot.CreateSubKey(@"CLSID\{" +
t.GUID.ToString().ToUpper() + @"}Programmable");
            Microsoft.Win32.Registry.ClassesRoot.CreateSubKey(@"CLSID\{" +
t.GUID.ToString().ToUpper() + @"}\Control");
        }
  [ComUnregisterFunctionAttribute]
        public static void UnregisterFunction(Type t)
        {
            Microsoft.Win32.Registry.ClassesRoot.DeleteSubKey(@"CLSID\{" +
t.GUID.ToString().ToUpper() + @"}Programmable");
            Microsoft.Win32.Registry.ClassesRoot.DeleteSubKey(@"CLSID\{" +
t.GUID.ToString().ToUpper() + @"}\Control");
        }
```

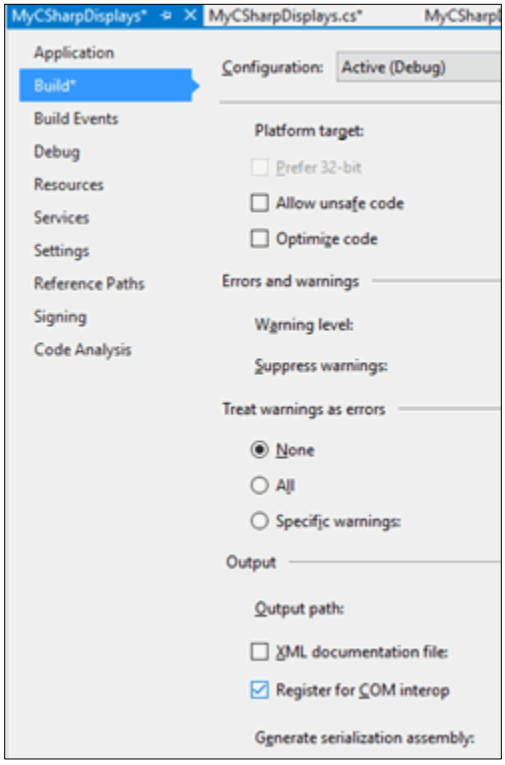


The RegisterFunction is called when the display is registered during the execution of the RegAsm utility. The UnregisterFunction is called when a display is unregistered. For more information on registering a display before use, consult the online Microsoft documentation.

11) Ensure the display is compiled with the necessary COM code so that it can communicate with IADS. In the "Project" drop down menu, select **Properties**.



12) Under the "Build" tab scroll down to the bottom and check the **Register for COM interop** option.



13) After this step is complete, save your work and continue onto the next section.

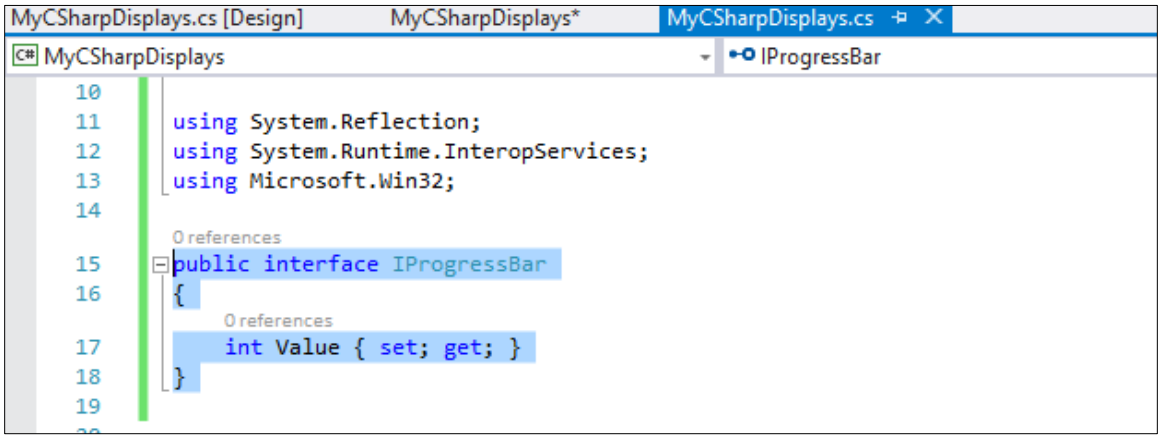
2.1.1 Adding properties to your new display using C# VS2015

At this step in the process you will add properties to your display. Think of properties as "data injection ports" or "interface plugs". They are attributes of your display, for example, text color, needle angle, scale factor; any feature you want the user to change or animate. To give a concrete example, the included demo project is code for an "Attitude Indicator" that simulates an aircraft dial. It has properties for "Roll", "Pitch" and "Heading" as well as "Sky Color" and "Ground Color".

Any property that you include in your display will be an access point on which the user can modify its contents/characteristics/behavior. Changing the "Pitch" property in my attitude indicator example would, as expected, cause the display to rotate its graphics to indicate the new pitch angle. As so, you need to understand the scope of your display's behavior and provide your users every property that you foresee them changing (within reason); and supply the code that responds to these property values and outputs the appropriate response. When this is complete, the user can drive any of these properties, with data from IADS, simply by dragging and dropping a parameter on the display; or they can set any of these properties to a constant value using the "right-click" properties sheet of the display. The best part is that all you have to do is worry about what properties to add and how to implement them and IADS will take care of ALL of the data related issues.

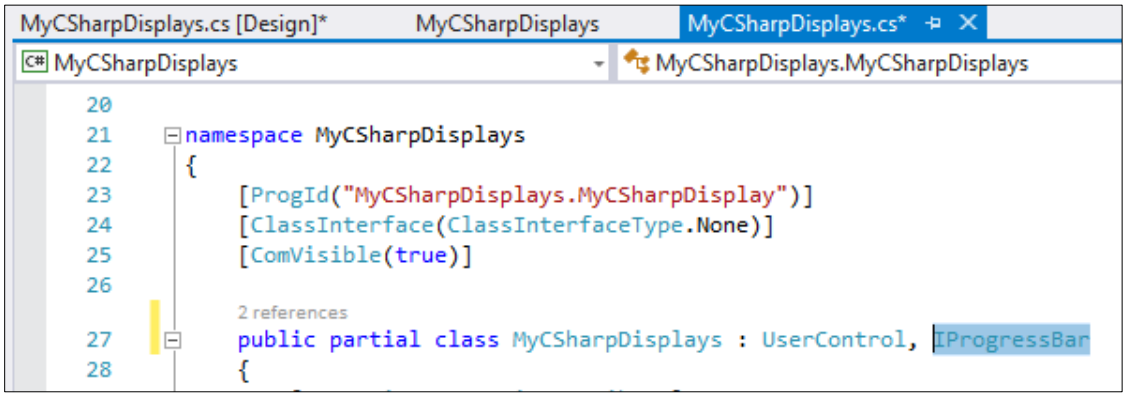
1) Add a public interface to the class. This is the interface which we will put all our properties that we want to expose to the user. In this example, we have added the Value property to provide access to the Value property of the .Net ProgressBar control.

```
public interface IProgressBar
{
   int Value {set; get;}
}
```



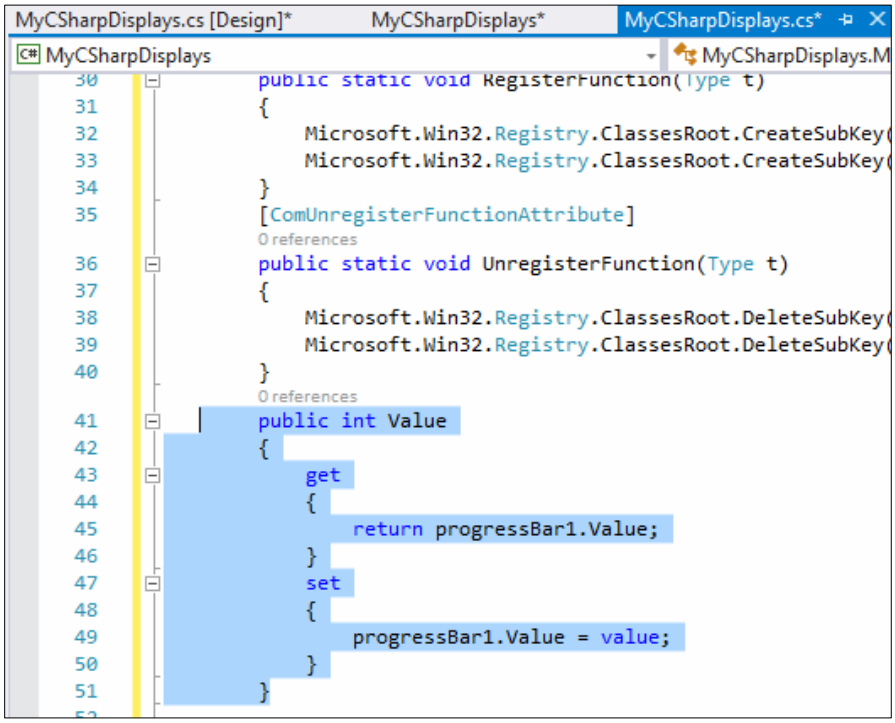
2) Add the interface to the User Control class:

```
public class MyCSharpDisplay : Usercontrol, IProgressBar
```



3) The next step is to implement the created code for each new property we add. In this example, we will focus on the set and get functions for the "Value" property.

4) For the set function, we will need to capture the incoming value and push it into the progress bar object. Once that is complete, our progress bar should redraw and display the new value. Please be aware that the value from the outside might be pushed on this display at a very high frequency. It might be prudent to add code to determine if the incoming property value is different than the existing value of a property and forgo the setting on the progress bar. This is vital optimization you might need to consider when building more complex displays. For this simple example, we will skip this step. The get function is easy to implement. Simply return the value of our progress bar that we have created for this property.

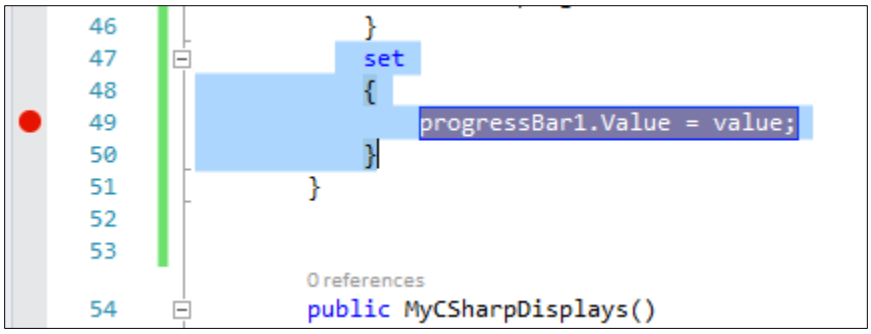


```
public int Value
{
   get
   {
         return progressBar1.Value;
   }
   set
   {
         progressBar1.Value = value;
   }
}
```

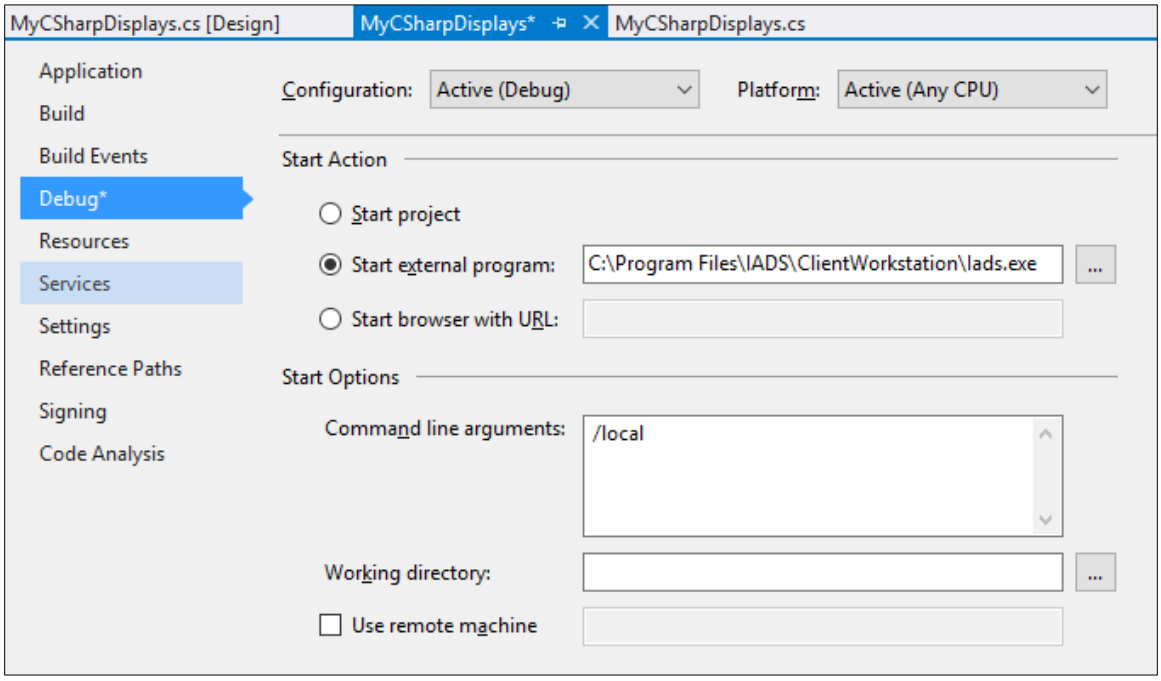
5) At this point you can modify the code in the display to perform your specific needs.
6) See section 2.3 to add your new display to IADS.

2.1.2 Debugging your new display in IADS using C# VS2015

1) In the development environment, place a break point in one of the "Set" method for testing.



In Visual Studio, select the **Project** > **Properties** drop down menu. In the "Debug" tab, set the "Start external program" field to the IADS application executable (Iads.exe). The exe is located in the "C:\Program Files\Iads" directory. Build your project and click the "Go" command IADS will start.



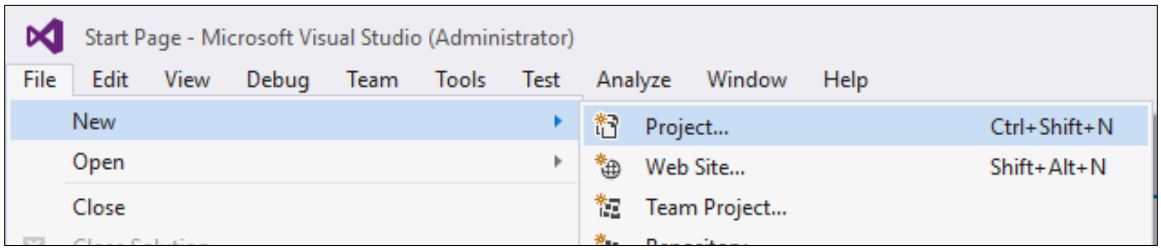
2) Drag-n-drop your display onto the new Analysis Window as explained in section 2.3. Once that is complete, save your configuration. Choose a parameter from the Parameter Tool and drop it onto the display. After the parameter is attached to a property, your break point should now hit in the debugger. You can now step through your drawing code if necessary.

2.2 Creating an IADS custom ActiveX control using C++ VS2015

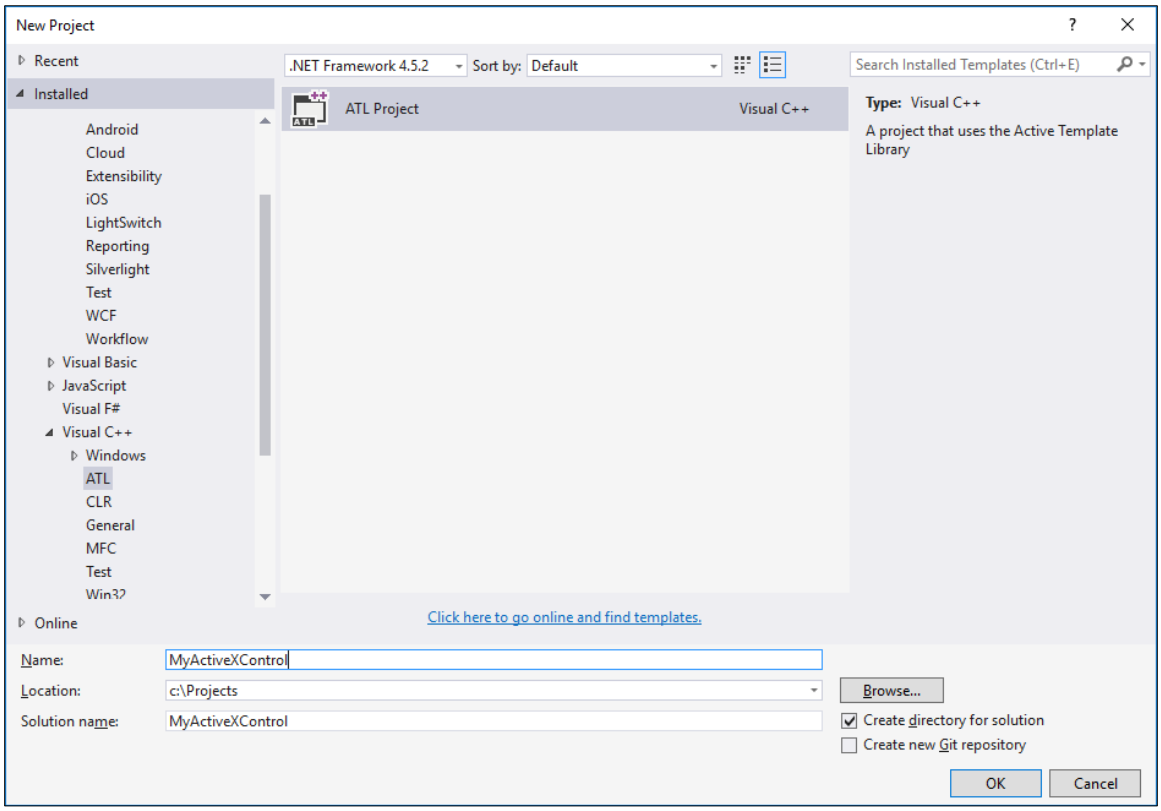
This tutorial assumes you are using Microsoft Visual Studio 2015. It should apply to other versions with minimal modification. This instruction will guide you through the process of creating a custom display for IADS using the ATL COM Wizard in C++ VS2015.

Warning- Make sure to create a project with the same bitness (x64,x86) as your IADS client. Your interfaces will not work if they try and run on a different architecture.

1) Open up VS2015 and Select **File** > **New** > **Project**.



2) In the New Project dialog that appears, choose the **Visual C++** > **ATL** tier and click the **ATL Project** option. At this point, please read the next step before you finish completing the dialog. There are some important considerations when choosing the proper project name.



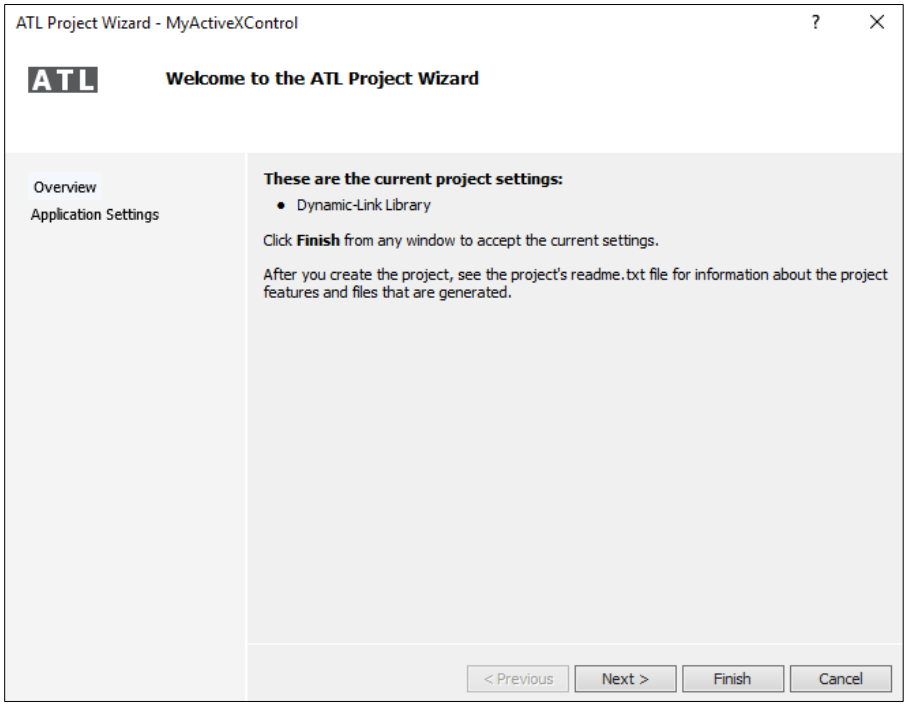
3) The project name you choose will become part of the display identifier name (aka ProgID, see note below). When it comes time to use your control in IADS, users will insert your new

control into the “Display Builder” toolbox based solely upon its name (more on this later). Plan on creating many displays in one “project” (most common and easier to manage the code). Choose a general project name like “AircraftGauges” or “FluidSystemDisplays”. One way to look at it is that the project name is akin to the “Genus” of your display, so shoot for generality. Consider prefixing the project name with your organization like “NASA” or “Lockheed”, as it may easier for users to locate your control the “Display Builder” list (i.e. NasaFluidSystemDisplays or LockheedAircraftGauges).

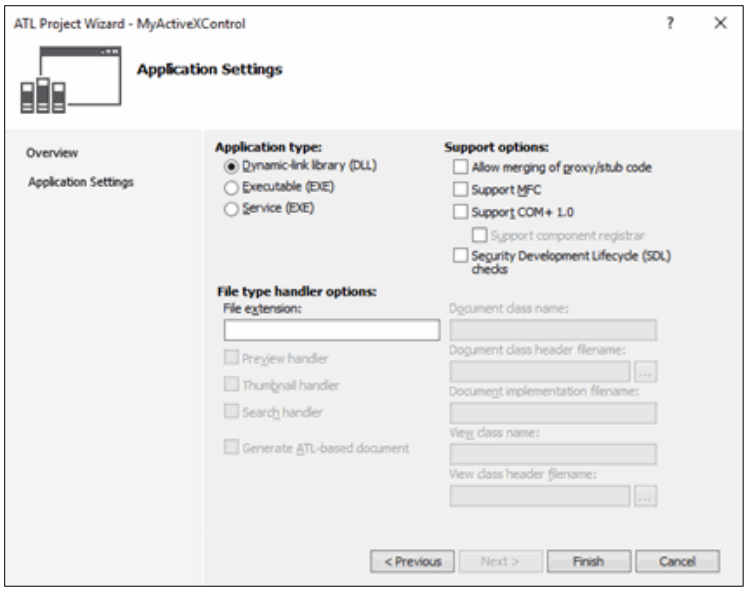
Note: Microsoft refers to your function’s name as its “ProgID” (aka Program ID). This is the string equivalent of your GUID (Global Unique Identifier) for the function. These Ids are placed in the Microsoft registry (directly from your project’s “.rgs” file), allowing your object to be created without any knowledge of the location of your “Dll” on the file system. Of course, this assumes that it is registered using the “regsvr32” program (consult the Microsoft documentation).

Now, in the fields at the bottom of the dialog, enter the project name, location, and the solution name.

4) After pressing **OK**, the “ATL Project Wizard” dialog will appear as below.

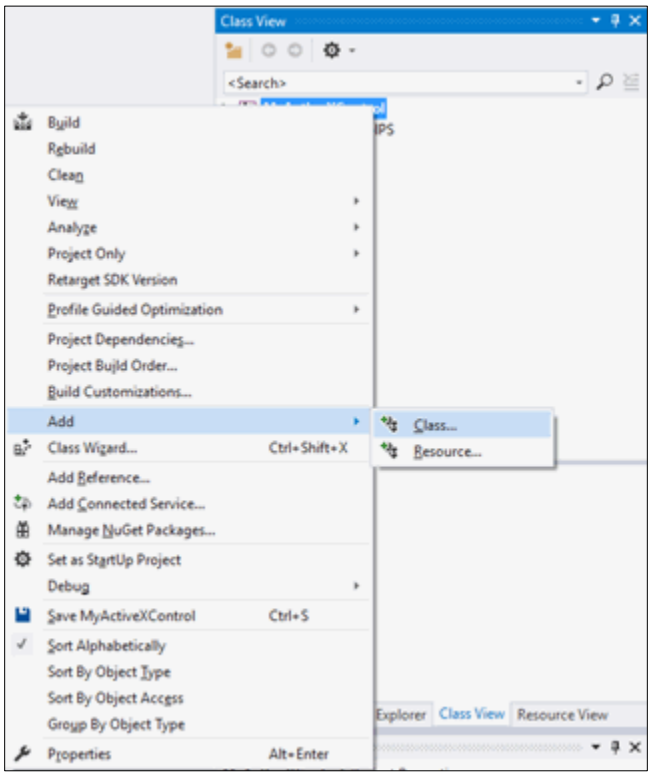


5) Click the **Next** button in the Wizard. On the new wizard page, ensure that the "Dynamic Link Library (DLL)" is checked. Every display that runs in IADS is of

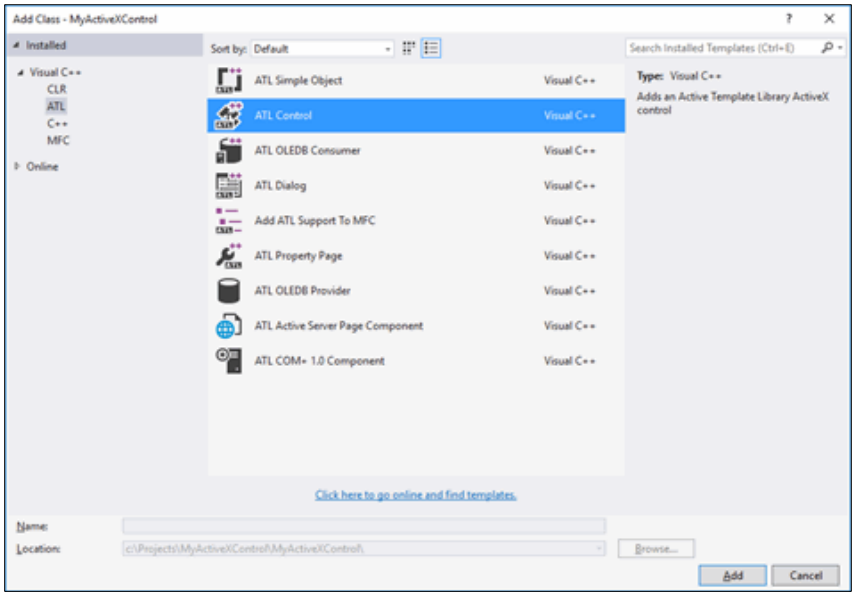


type DLL because it allows for maximum speed in displaying graphics. Press the **Finish** button and the Wizard will set up your project.

6) Next, go to the "ClassView" tab in Visual Studio's workspace and right-click on the project name. Choose **Add** > **Class**.

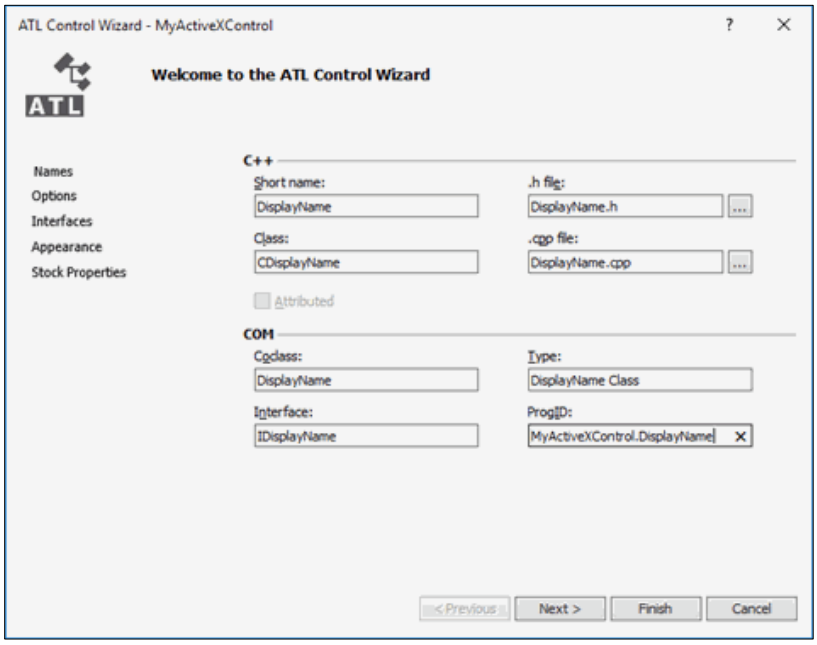


7) Upon adding a new class you will be presented with a dialog. Click the **ATL** tier and **ATL Control** as shown below. When that is complete, press the **Add** button.

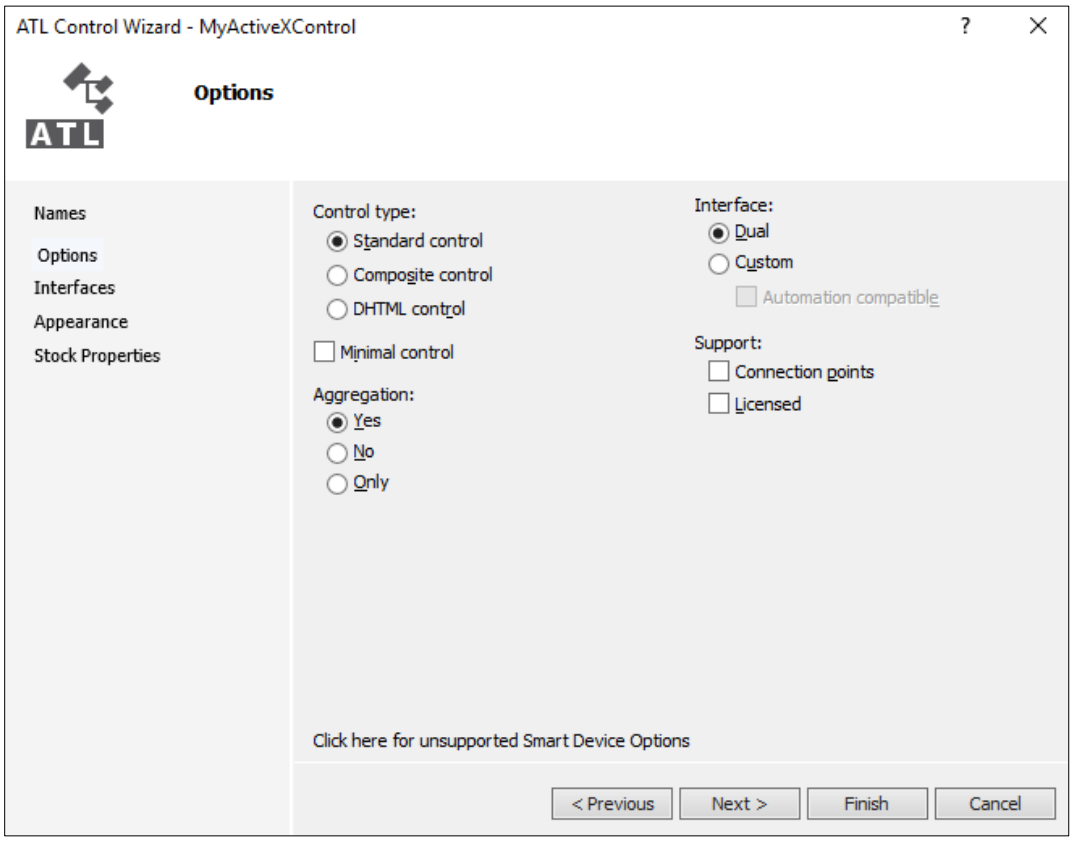


8) On the first tab, enter the name of your display in the “Short Name” field. The wizard will fill out the rest of the tab automatically. For this example, I used “DisplayName” as the short name. The name entered will be combined with your project name and will present the final display name inside of IADS (ProjectName.FunctionName) as explained on page 1. See the

"ProgID" field in your dialog for your final IADS display name. Warning: Newer Visual Studio versions do not automatically populate the ProgID field. Please ensure the ProgID field contains your specific ProjectName.FunctionName text. If not, please type in the appropriate text manually. Press "Next" to continue.



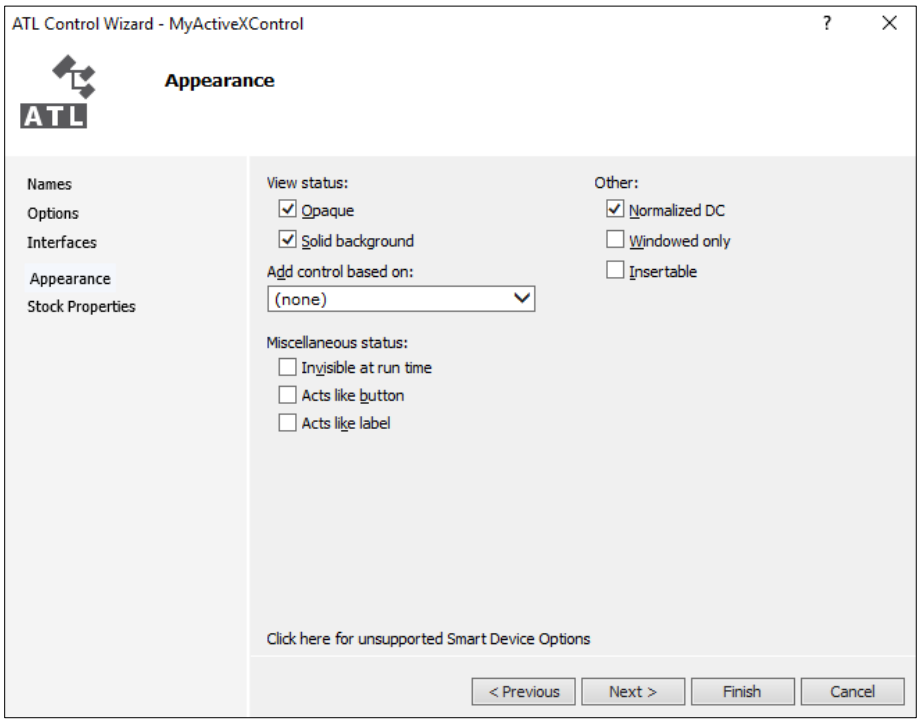
9) On the next tab ("Options"), leave everything as default (Standard control, Apartment, Dual, Yes, and no other options checked). This will allow you to take full control of a "blank canvas" and draw your display using low level graphics libraries such as GDI/GDI+ or OpenGL. On the other hand, if you need to create a "dialog based" display containing typical dialog elements such as text boxes, drop down lists, etc you will need to select the "Composite control" choice.



The remaining options are basically "COM speak". If want understand these options fully, you will have to consult the Microsoft documentation. The most notable remaining option is "Interface". In order to create a real compliant ActiveX "display", you must choose "Dual" interface. This will enable IADS (and other programs) to interface to your control using the "IDispatch" interface, which allows a loosely coupled, "on the fly" communication. This also happens to be the primary (simplistic) way that IADS gets data to your control. More on this subject later.

10) On the next tab ("Interfaces"), leave all of the default choices and select "Next". Again, these options are more "COM speak" and include standard interfaces in which the Wizard will implement for you automatically. If you want more background information, consult the Microsoft documentation.

11) On the next tab ("Appearance"), select the "Windowed Only" checkbox if you plan on using OpenGL; otherwise uncheck it. "Windowed Only" will ensure that we have a window to create an OpenGL context upon. For GDI based displays, we want to attempt to draw "without a window" for speed and resource considerations. Leave the other settings as default (later discuss the speed benefits of de-selecting the "Normalize DC" checkbox). Remember, OpenGL = "Windowed Only". Don't worry, this can be easily changed later if you make a mistake (as can almost anything).



12) On the next tab ("Stock Properties"), leave all the options empty and select "Next". These options are display properties that the Wizard will implement for you automatically. Generally, each property will be added after Wizard is complete. If you want more background information, consult the Microsoft documentation.

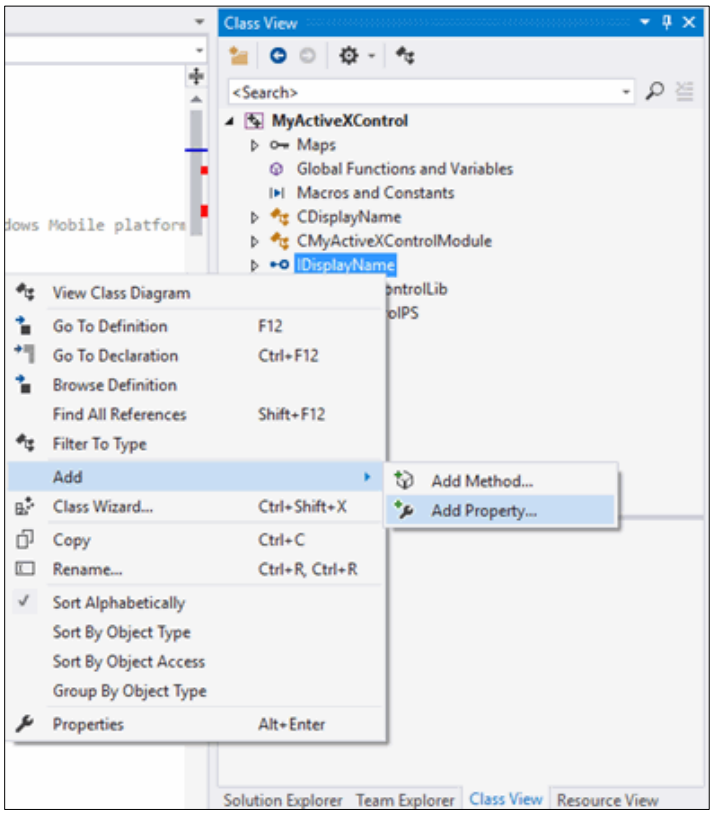
    After clicking "Finish", the Wizard will auto-create most of the code needed for your new display. Examine your "Solution Explorer" view. It should now contain the new display object by name.

2.2.1 Adding properties to your new display using C++ VS2015

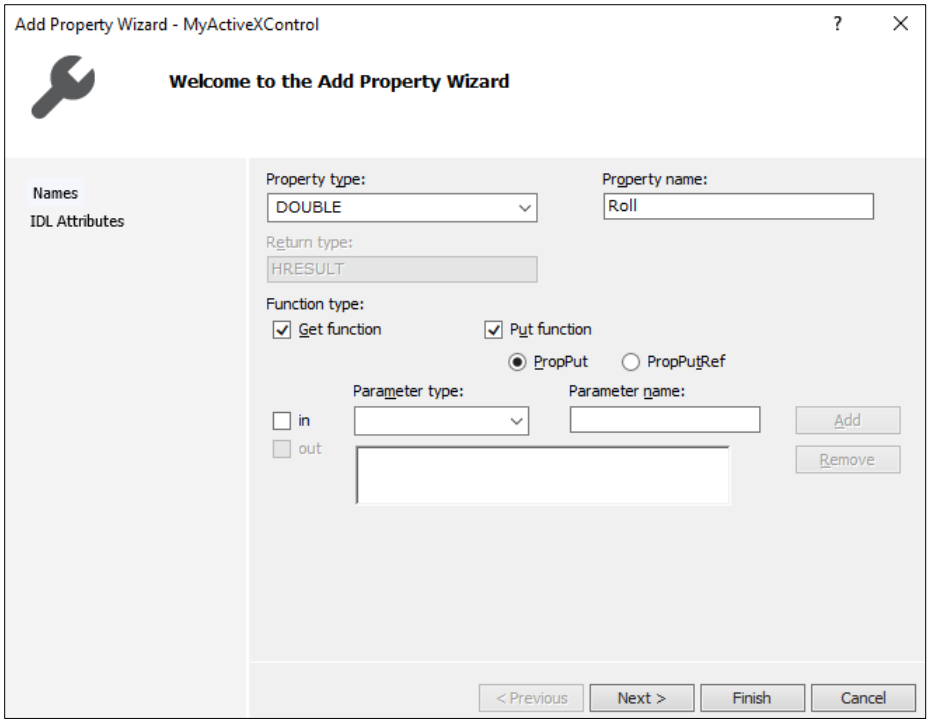
At this step in the process you will add properties to your display. Think of properties as "data injection ports" or "interface plugs". They are attributes of your display, for example, text color, needle angle, scale factor; any feature you want the user to change or animate. To give a concrete example, the included demo project is code for an "Attitude Indicator" that simulates an aircraft dial. It has properties for "Roll", "Pitch" and "Heading" as well as "Sky Color" and "Ground Color".

Any property that you include in your display will be an access point on which the user can modify its contents/characteristics/behavior. Changing the "Pitch" property in my attitude indicator example would, as expected, cause the display to rotate its graphics to indicate the new pitch angle. As so, you need to understand the scope of your display's behavior and provide your users every property that you foresee them changing (within reason); and supply the code that responds to these property values and outputs the appropriate response. When this is complete, the user can drive any of these properties, with data from IADS, simply by dragging and dropping a parameter on the display; or they can set any of these properties to a constant value using the "right-click" properties sheet of the display. The best part is that all you have to do is worry about what properties to add and how to implement them and IADS will take care of ALL of the data related issues.

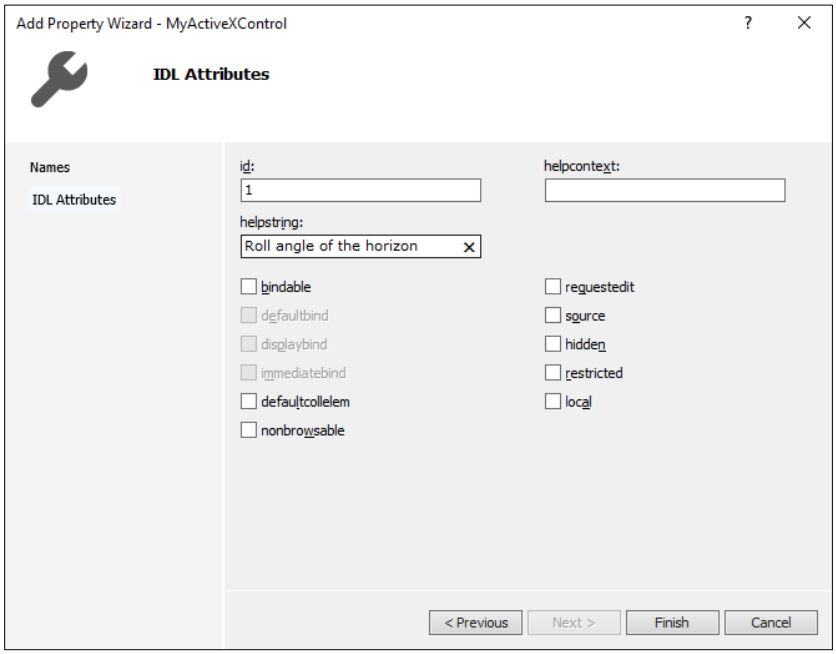
1) Now, make sure that you are in the "ClassView" tab of the VS2015 workspace. To add a "Property" to your new control, Right-click on the "XXXXX" where "XXXXX" is the name of your newly created display (look for the little "magnifying glass" icon). Select "Add Property" from the popup menu.


Class View
<Search>
MyActiveXControl
Maps
Global Functions and Variables
Macros and Constants
CDisplayName
CMyActiveXControlModule
IDisplayName
View Class Diagram
Go To Definition F12
Go To Declaration Ctrl+F12
Browse Definition
Find All References Shift+F12
Filter To Type
Add
Class Wizard... Ctrl+Shift+X
Copy Ctrl+C
Rename... Ctrl+R, Ctrl+R
Sort Alphabetically
Sort By Object Type
Sort By Object Access
Group By Object Type
Properties Alt+Enter
Add Method...
Add Property...
Solution Explorer
Team Explorer
Class View
Resource View

2) In the Add Property Wizard, set the property type to the desired data type (double in this example) and the name of the property (Roll in this example) and click **Next**.



3) In the last page of the Add Property Wizard, leave all the options as default except the helpstring field. This helpstring will be displayed in the IADS properties sheet when the user is setting the property, try to provide a descriptive (but short) sentence for your new property.



4) When you are complete, press the “Finish” button. This will auto-create code to implement a property named “Roll” within your new project. Repeat this process (starting from step 1) for every property that you want to add to the display.

5) Going back to your “Solution Explorer”, you will see the new property code inserted into your display’s .cpp file.

6) The next step is to implement the code created for each new property we added with the Add Property Wizard. Thus, in our example property “Roll”, we will need to focus in on the put_Roll and get_Roll functions. In preparation, we will need to add a class member variable for each new property. For the Roll property, add a new class member variable to the .h display class named “mRoll” with the same data type as the property (double in this example). Do not forget to set this new member variable mRoll to 0.0 in the “FinalConstruct” function which is also part of the .h class.

7) For the put_Roll function, we will need to capture the incoming value and store it in a class member variable. Once that is complete, we will call a function named “FireViewChange” that will let our display know it needs to be redrawn. When the redraw function (OnDraw) is called, we will then use the value contained in the class member variable to draw the display. In addition, we will set a variable called m_bRequiresSave to TRUE, telling IADS that we wish to save our display. This is a vital step to ensure your display is saved within IADS when a property is changed. Notice that we only perform this code if the incoming property value is different than the existing value of the property. The get_Roll function is easy to implement. Simply return the value of our class member variable that we have created for this property.

8) Now, when all of your properties are implemented, add your “drawing code” to the “OnDraw” function. The OnDraw function is where you will take all the values of your class member variables and actually draw the display content. See the sample projects for examples in both GDI and OpenGL.

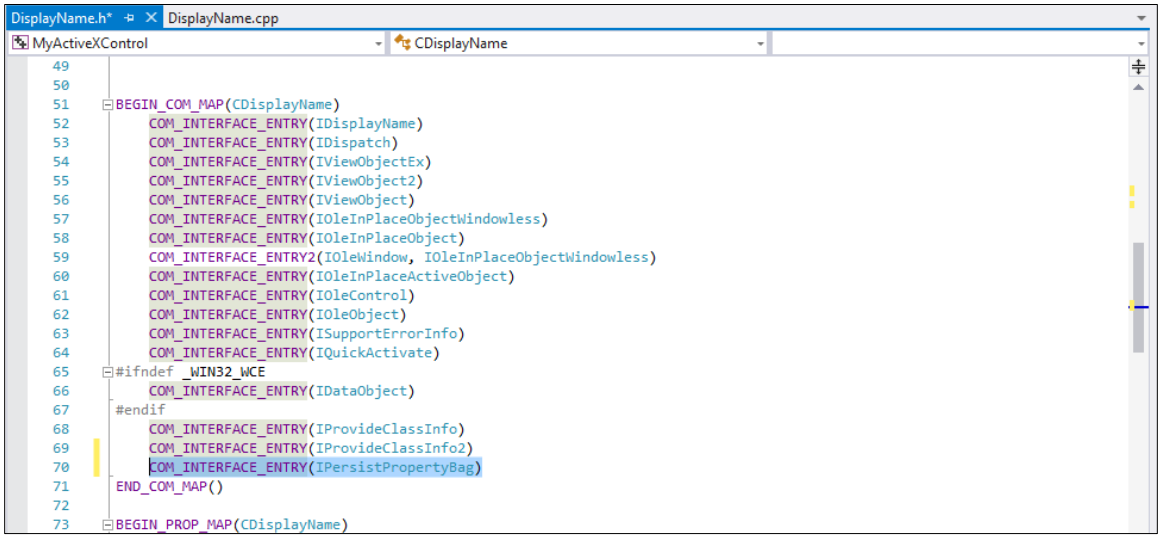
```
DisplayName.cpp | DisplayName.h*
MyActiveXControl | CDisplayName
107
108  // IDisplayName
109  public:
110      HRESULT OnDraw(ATL_DRAWINFO& di)
111      {
112          RECT& rc = *(RECT*)di.prcBounds;
113          // Set Clip region to the rectangle specified by di.prcBounds
114          HRGN hRgnOld = NULL;
115          if (GetClipRgn(di.hdcDraw, hRgnOld) != 1)
116              hRgnOld = NULL;
117          bool bSelectOldRgn = false;
118
119          HRGN hRgnNew = CreateRectRgn(rc.left, rc.top, rc.right, rc.bottom);
120
121          if (hRgnNew != NULL)
122          {
123              bSelectOldRgn = (SelectClipRgn(di.hdcDraw, hRgnNew) != ERROR);
124          }
125
126          Rectangle(di.hdcDraw, rc.left, rc.top, rc.right, rc.bottom);
127          SetTextAlign(di.hdcDraw, TA_CENTER|TA_BASELINE);
128          LPCTSTR pszText = _T("DisplayName");
129  #ifndef _WIN32_WCE
130          TextOut(di.hdcDraw,
131              (rc.left + rc.right) / 2,
132              (rc.top + rc.bottom) / 2,
```

Ensuring your display is saved in IADS

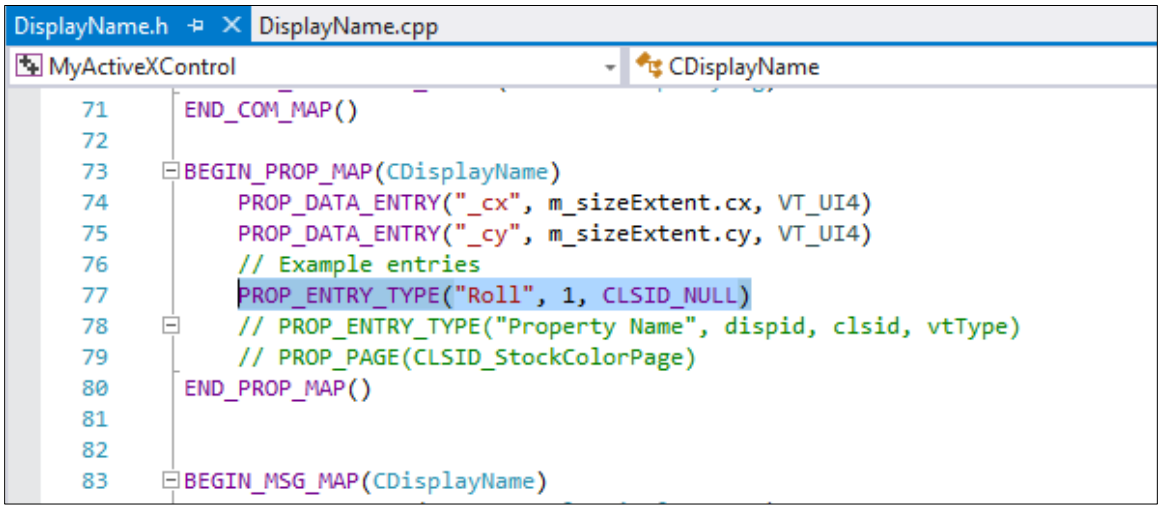
1) Go to your ".h" file (SampleDisplay.h in this example) and insert the code "public IPersistPropertyBagImpl<CYourClassName>" as below. This will allow the control to "save" in IADS properly. Don't forget to add a "comma" to the end of the line above this new line.

```
DisplayName.h* | DisplayName.cpp
MyActiveXControl | CDisplayName
13
14
15    // CDisplayName
16    class ATL_NO_VTABLE CDisplayName :
17        public CComObjectRootEx<CComSingleThreadModel>,
18        public IDispatchImpl<IDisplayName, &IID_IDisplayName, &LIBID_MyActiveXControlLib, /*w
19        public IOleControlImpl<CDisplayName>,
20        public IOleObjectImpl<CDisplayName>,
21        public IOleInPlaceActiveObjectImpl<CDisplayName>,
22        public IViewObjectExImpl<CDisplayName>,
23        public IOleInPlaceObjectWindowlessImpl<CDisplayName>,
24        public ISupportErrorInfo,
25        public IQuickActivateImpl<CDisplayName>,
26    #ifndef _WIN32_WCE
27        public IDataObjectImpl<CDisplayName>,
28    #endif
29        public IProvideClassInfo2Impl<&CLSID_DisplayName, NULL, &LIBID_MyActiveXControlLib>,
30        public CComCoClass<CDisplayName, &CLSID_DisplayName>,
31        public CComControl<CDisplayName>,
32        public IPersistPropertyBagImpl<CDisplayName>
33    {
34    public:
35
36
```

2) Likewise, you need to add “COM_INTERFACE_ENTRY(IPersistPropertyBag)” to the “Com Map” in the “BEGIN_COM_MAP” area of the code as below. This is also needed to allow the control to “save” in IADS.



3) For every property to “save”, add it to the “BEGIN_PROP_MAP” area of the code as below. The number corresponds to the property id that is defined by the wizard in the “.idl” file of the project. Examine your “.idl” file from the “Solution Explorer” tab of the workspace for the correct number. Properties in your “PROP_MAP” will get saved by IADS and reloaded when the display is created in a saved Analysis Window.

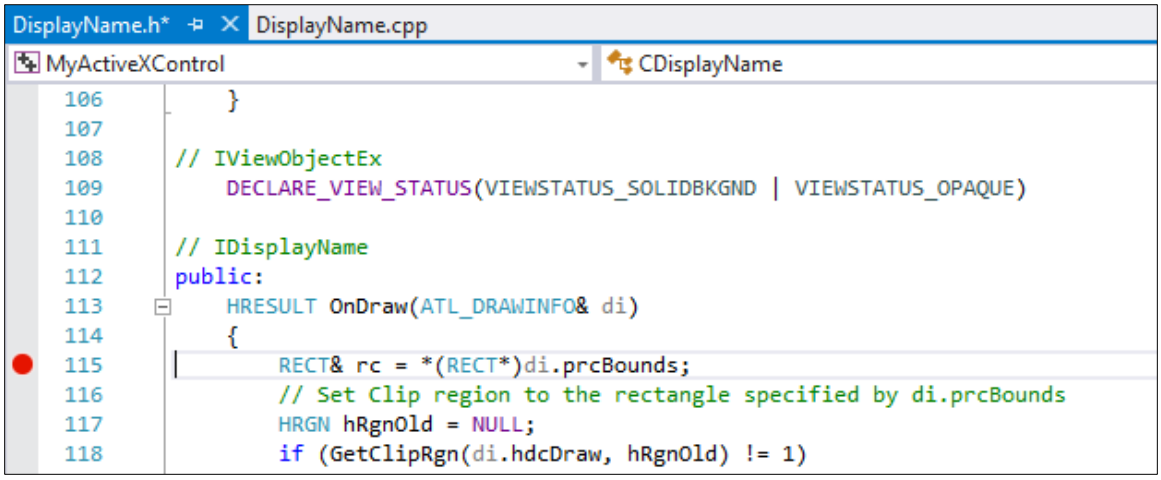


4) At this point, you can begin modifying the code in the display to perform your specific needs.

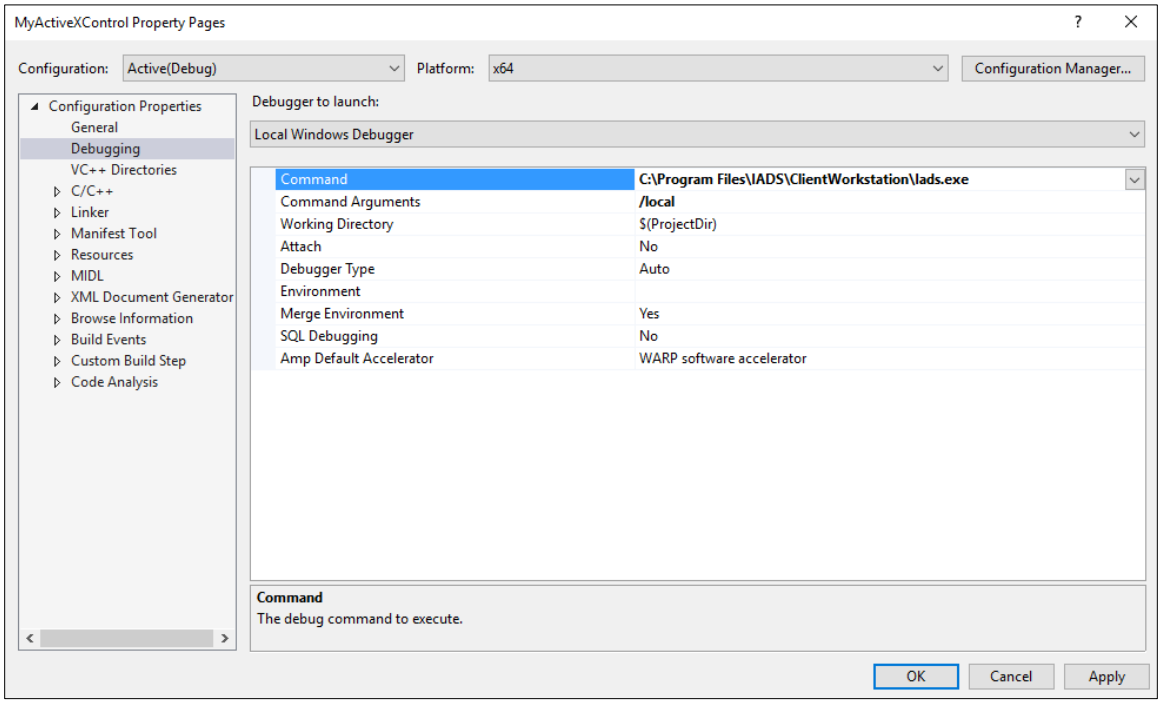
5) See section 2.3 to add your new display to IADS.

2.2.2 Debugging your new display in IADS using C++ VS2015

1) Place a break point in your “OnDraw” method for testing.



2) In Visual Studio, select the **Project** > **Properties** drop down menu. Under the “Configuration Properties > Debugging” tier, pick “Iads.exe” as your “Command”. The Iads.exe is located in your “C:\Program Files\Iads” directory. Build your project and click on the “Go” command. IADS will start.

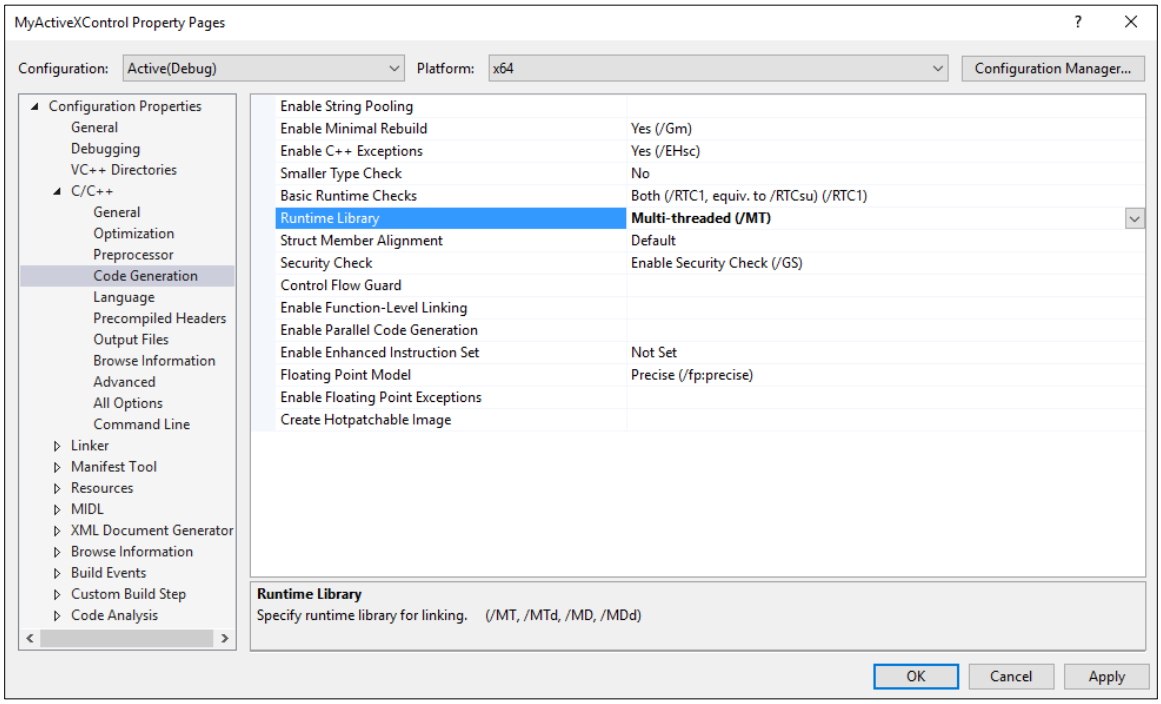


3) Drag-n-Drop your display to the new Analysis Window as in section. Your break point should now hit in the debugger. You can now step through your rendering code if necessary.

Deploying your new display in IADS

When it comes time to deploy your new control to users on other PCs, you need to consider a couple of issues. One issue is that your control may require some auxiliary dlls that are not available on the other systems. If that occurs and the dlls are missing, the control may not operate. To help minimize this possibility, you must always build your new control dll in "Release" mode. You should never distribute a control dll that has been compiled under the "Debug" mode. The debug mode uses libraries that will most certainly be missing on any machine without Visual Studio installed. Beyond that, it is always best to 'statically link' all the runtime libraries. Also, since we've used ATL to build this display, we will need to statically link the ATL library as well.

1) In Visual Studio, select the "Project->Properties" drop down menu. Make sure that the "Configuration" dropdown is set to "Release". Under the "Configuration Properties > C/C++ > Code Generation" tier, set the "Runtime Library" to "**Multi-threaded (/MT)**".



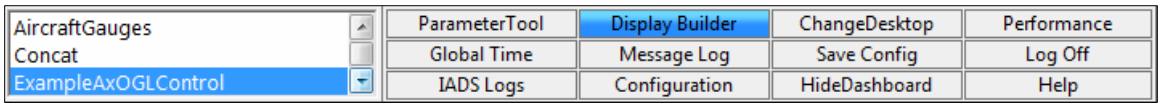
2) Once you have made these changes to your project, you should rebuild your ‘solution’. Make sure once again that your current configuration is set to “Release” and then select the **Build** > **Rebuild Solution** drop down menu option. After this step is complete, your control dll should be in your project “Release” folder. It should now be ready to deploy on another system.

   The control dll will need to be copied to the other PC and ‘registered’. In order to register the dll, you will have to run the ‘regsvr32.exe’ program. One easy way to accomplish this is to double click on the dll in Windows Explorer. When asked what program to execute on the dll, navigate to the Windows\System32 directory and choose the regsvr32.exe file. This procedure may be different if the operating system is a 64 version. Please consult the online documentation for specifics.

   If the dll fails to register at this point, we’ve most likely failed to statically link the needed dlls. We can investigate which dlls are missing by using the “Dependency Walker” tool. The Dependency Walker program is located within the Microsoft Visual Studio\Common\Tools directory and is named “Depends.exe”. Copy Depends.exe from your development PC to the target PC and run the program. From the File drop down menu select **Open** and choose your control dll. Examine the module list in the bottom window pane. Any missing dependent dlls should show up with a question mark. Search for those dll names on the net and find out their purpose. It might help you narrow down what solution setting you have missed. It is also possible that the missing dll is a private library that you are using, in which case you will need to either static link or copy that dll to the target machine as well.

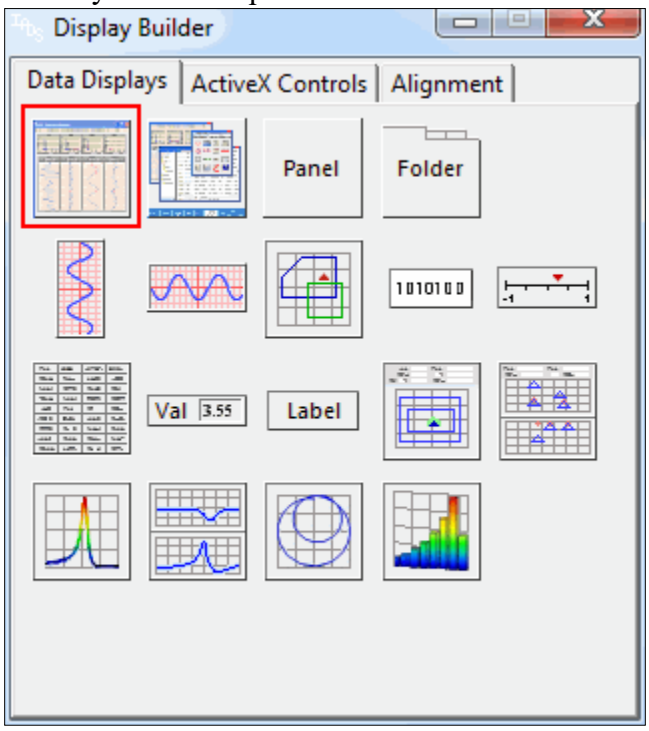
2.3 Adding your new display to IADS

1) Click the **Display Builder** button in the bottom right corner of the IADS Dashboard.

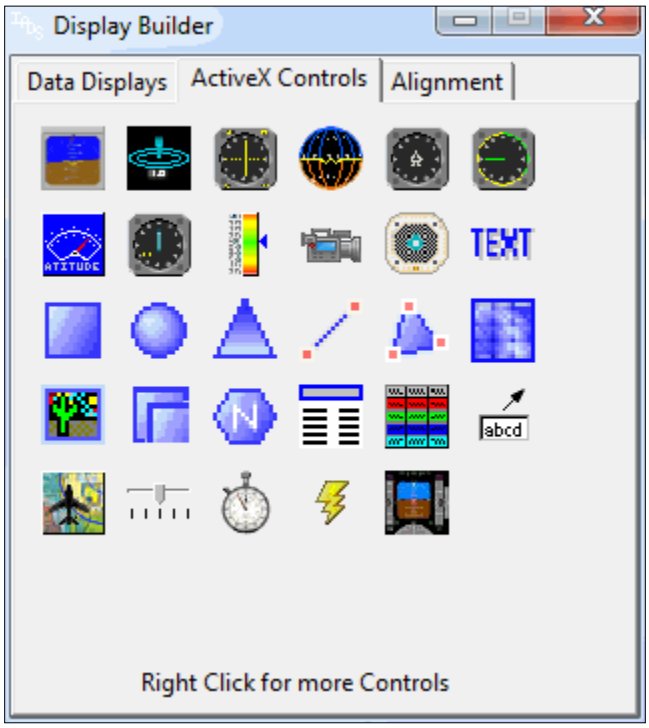


The "Display Builder" dialog will appear with icons of components that you can use to build your displays (including your new ActiveX control).

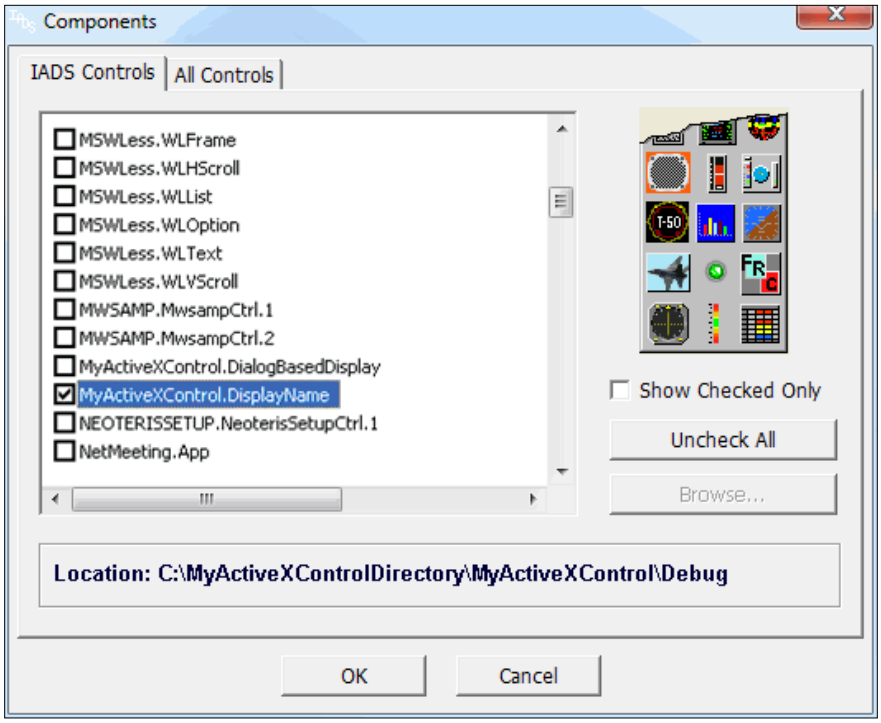
2) Click on the **Analysis Window** icon (upper left) and hold down the left mouse button to drag it onto your Desktop.



3) Now let's add your new control to the "Display Builder". Click on the second tab in the display builder named "ActiveX Controls". This is where all ActiveX displays will reside, ready to be dropped upon your newly created Analysis Window. Notice that there are only a select few ActiveX control icons on this tab of the display builder. If the display builder were to show all of the controls available on your system, the icons would fill several pages of this size. In order to add your new control, you must "Right-click" on tab (somewhere where there are no icons). This will activate yet another dialog containing both the "IADS" supplied ActiveX controls as well an entire list of all the ActiveX controls on your system (including your newly created one!). Click on the **All Controls** tab of this new dialog and find your new control. The name will be ProjectName.ObjectName" as discussed earlier in this tutorial. Click **OK** to add your display to the display builder. This only needs to be done once for each new control that you wish to debug/add in IADS.



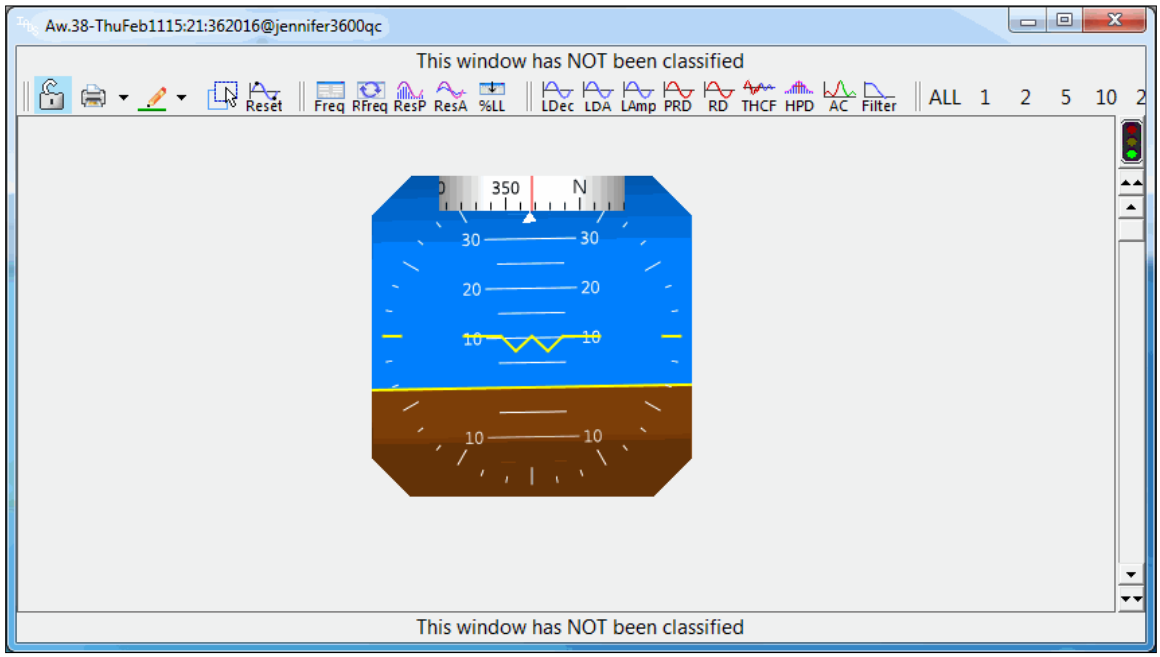
Note: If you cannot locate you display in the “All Controls” list, try checking your “.rgs” file in your VS2015 project in the “workspace” file view. It contains an entry named VersionIndependentProgID. This is where VS2015 stores your “ProgId” (Program ID) for the project.



2.4 IADS demo model control project

The “ActiveX display - 3D Model Demonstration” project is available for download on the Curtiss Wright IADS website at https://iads.symvionics.com/support/programming-examples/

The sample project has code to build an “Attitude Indicator” that simulates an aircraft dial. It has properties for “Roll”, “Pitch”, and “Heading” as well as “SkyColor” and “GroundColor”. Any property that you include in your display will be an “access point” on which the user can modify its contents/characteristics/behavior. Changing the “Pitch” property in my attitude indicator example would, as expected, cause the display to rotate its graphics to indicate the new pitch angle. The magic of building an ActiveX control is then to understand the “scope” of your control’s behavior, and to provide your users every property that you foresee them changing (within reason, don’t go overboard); and also to supply code that responds to these property values and outputs the appropriate response (i.e. draw attitude indicator display at the current value of the roll, pitch, heading, SkyColor and GroundColor properties). When this is complete, the user can drive any of these properties, with data from IADS, simply by dragging/dropping a parameter on your newly created display; or they can set any of these properties to a constant value using the “right-click” properties sheet of the display. The best part is that all you have to do is worry about what properties to add and how to implement them, and IADS will take care of ALL of the data related issues.



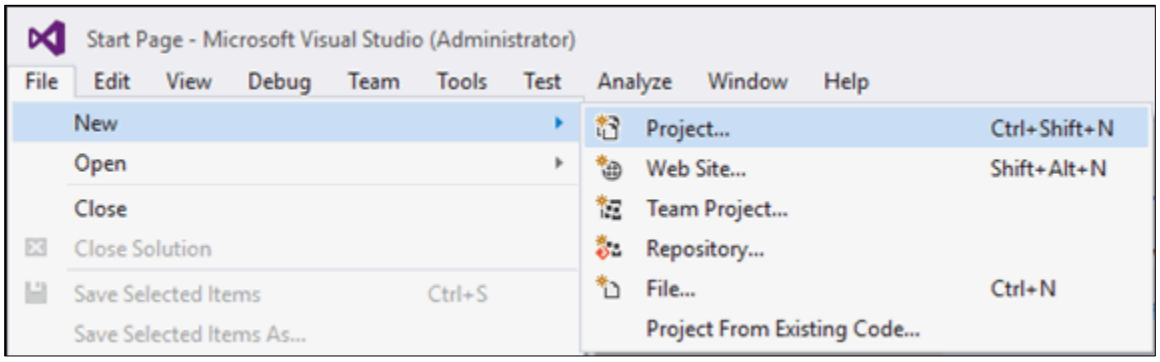
3. Custom Derived Functions

For more background on how to build a custom function, download the tutorial with sample function (zip) on the Curtiss Wright IADS website; and read the comments within the code: https://iads.symvionics.com/support/programming-examples/

Some projects will use files provided in the "Custom Derived Function Helper Classes" download.

3.1 Creating a custom derived function using C# VS2015

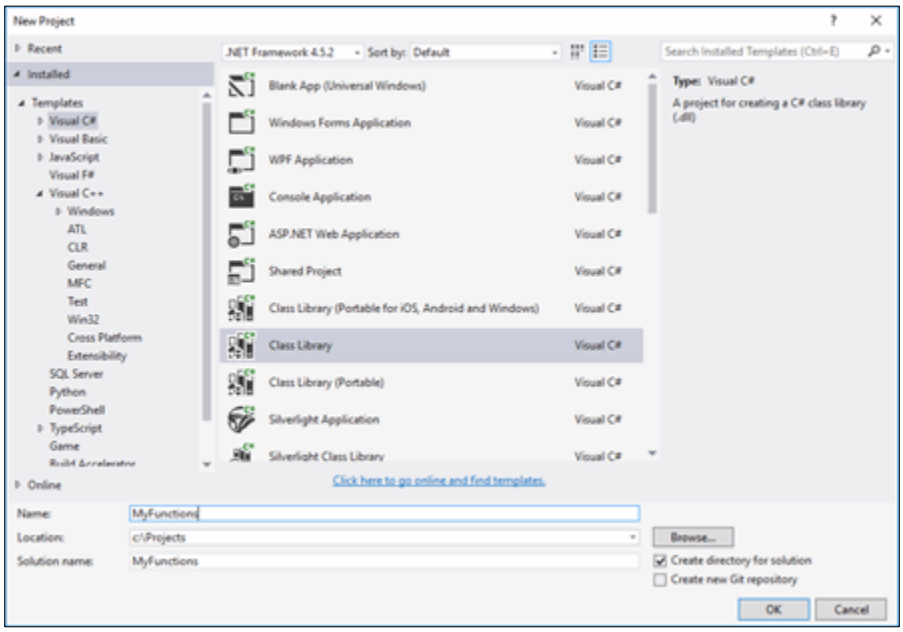
1) Open up VS2015 and Select **File** > **New** > **Project**.



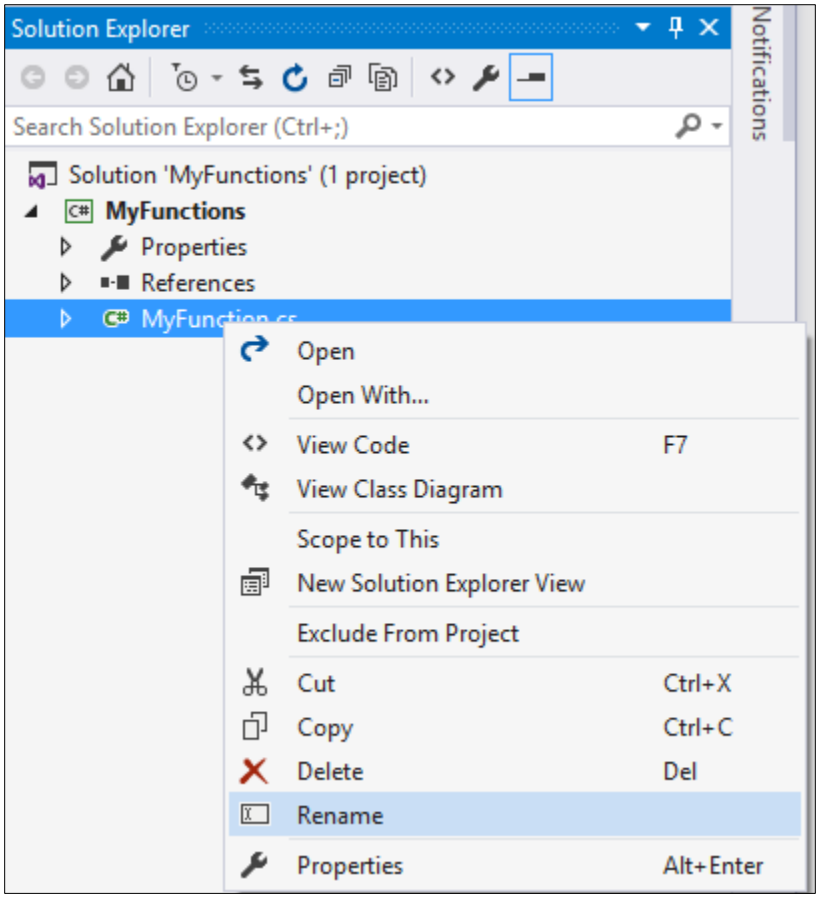
2) In the New Project dialog that appears, choose the "Visual C#" tier and click the Class Library option. At this point, please read the next step before you finish completing the dialog. There are some important considerations when choosing the proper project name.

2) Plan on creating many functions in one "project" (most common and easier to manage the code). The project name should be similar to the "Genus" of your function, so shoot for generality. Consider prefixing the project name with your organization like "NASA" or "Lockheed" and the type of functions you will be adding (example: NasaFluidFuncs).

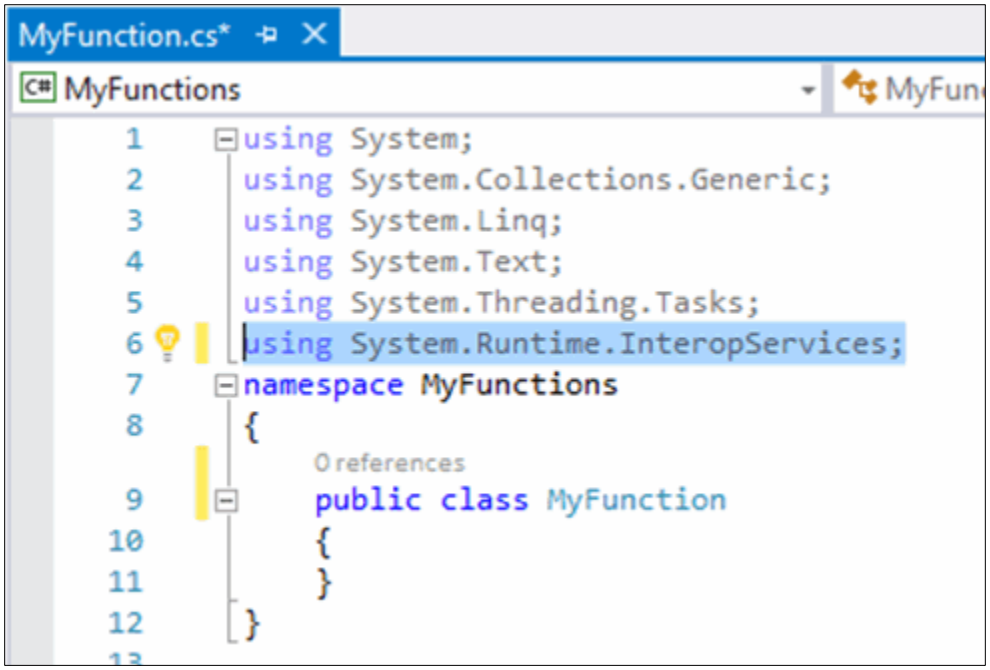
   Now, in the fields at the bottom of the dialog, enter the project name, location, and the solution name.



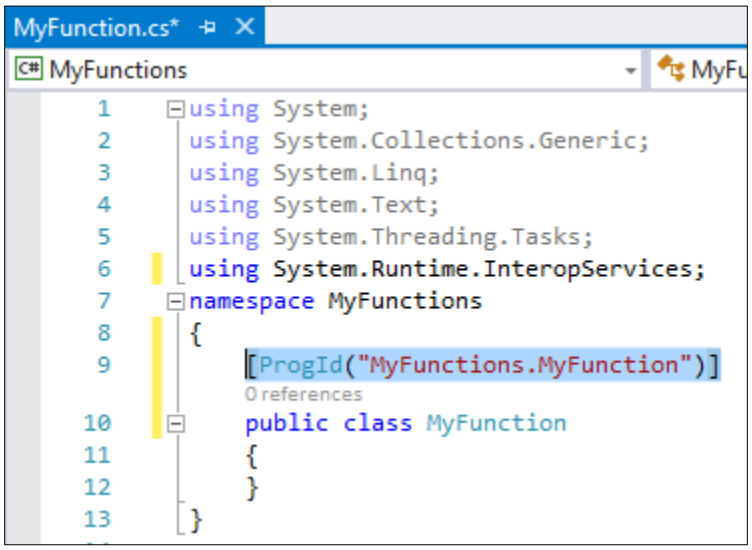
3) After pressing “OK”, the project will be ready for editing. Rename the Class1.cs file to reflect our function name. In this example, we will call it ‘MyFunction.cs’.



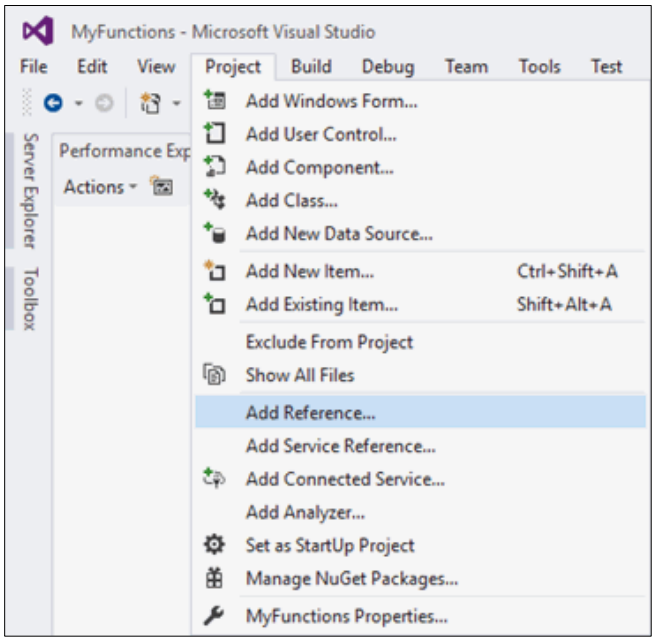
4) In preparation for the next step, we will need to add another “using” directive. In the code view, add a line “using System.Runtime.IteropServices”.



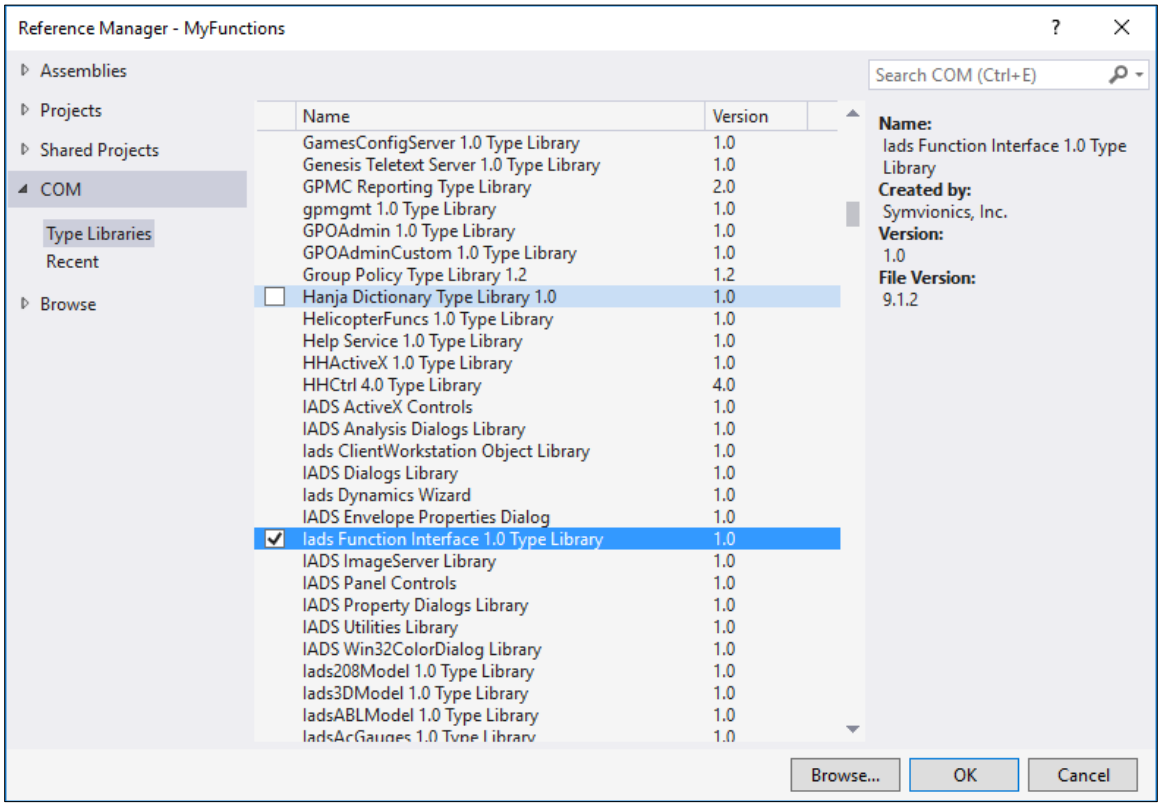
5) Now we need to focus on the entire function name as it will appear to the IADS end user. The format of the function name requires two strings separated by a period (‘.’). It is best practice to use the Visual Studio project name as the first portion of the name (before the period). The portion after the period should be your specific function name. For instance, ProjectName.ClassName, NasaFluidFuncs.FlowRate, or in this tutorial MyFunctions.MyFunction. We will define that name explicitly by using the “ProgId” directive. In the code view, type [ProgId(“MyFunctions.MyFunction”)] above your class definition “public class MyFunction”.



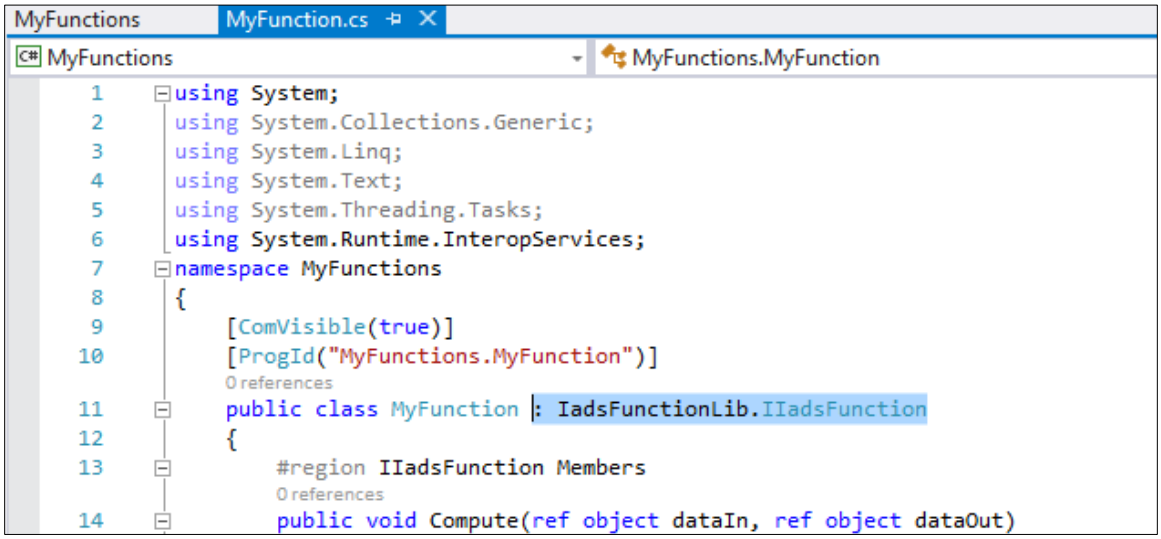
6) The next step is to implement the interface that IADS requires to call your function. To accomplish this task, we will need to add a reference to the definition file of the interface. From the Project menu, select **Add Reference**.



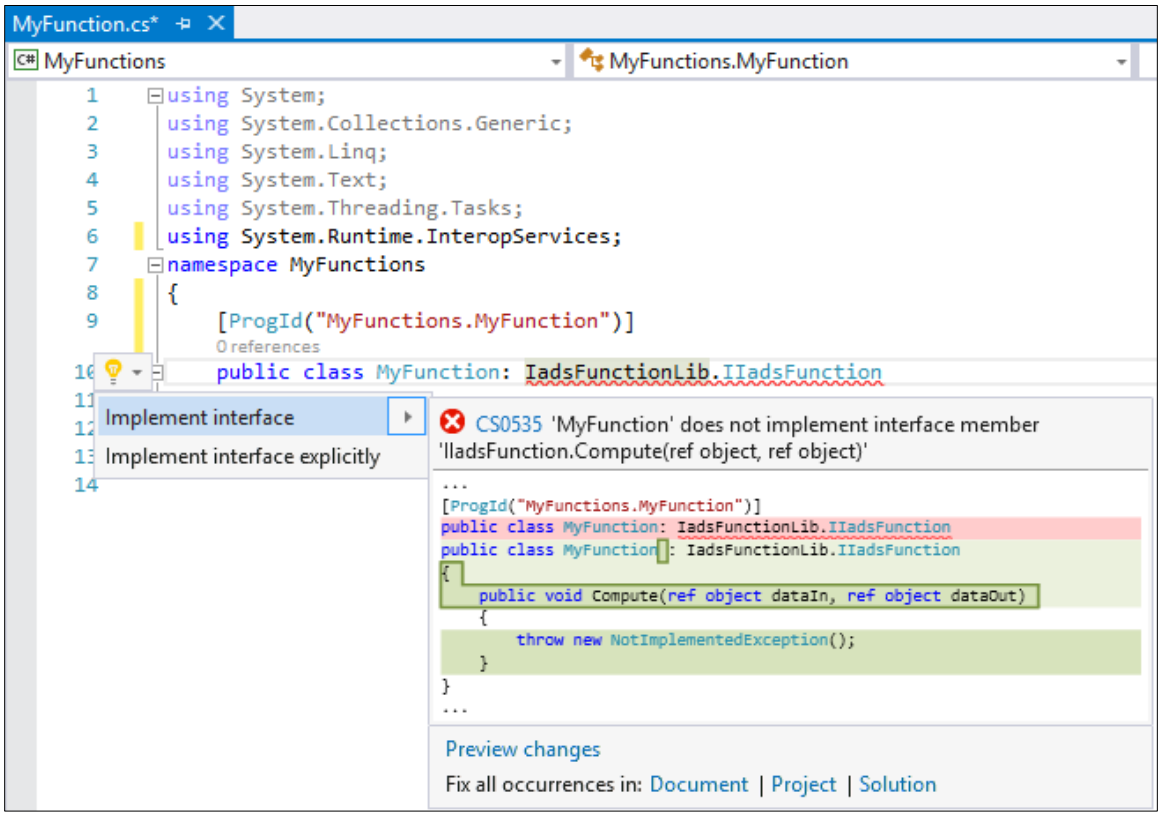
7) In the "Add Reference" dialog, select the **COM** tab. Scroll down until you see "Iads Function Interface 1.0 Type Library". Press **OK** to add the reference to our project.



8) Now, back in the Solution view, notice that the IadsFunctionLib has been added to our References section. This reference contains the definition of the IADS custom function interface. Now, we will need to add the interface to our class definition. First, type a colon ':' after the class name and add the text "IadsFunctionLib.IIadsFunction". In essence we will inherit from the interface definition. After this is complete, we can implement the interface.



9) To implement the interface, click on the "IadsFunctionLib.IIadsFunction" text in the code view window. You will see the 'quick action' (light bulb) icon. Select the icon and select **Implement Interface**.

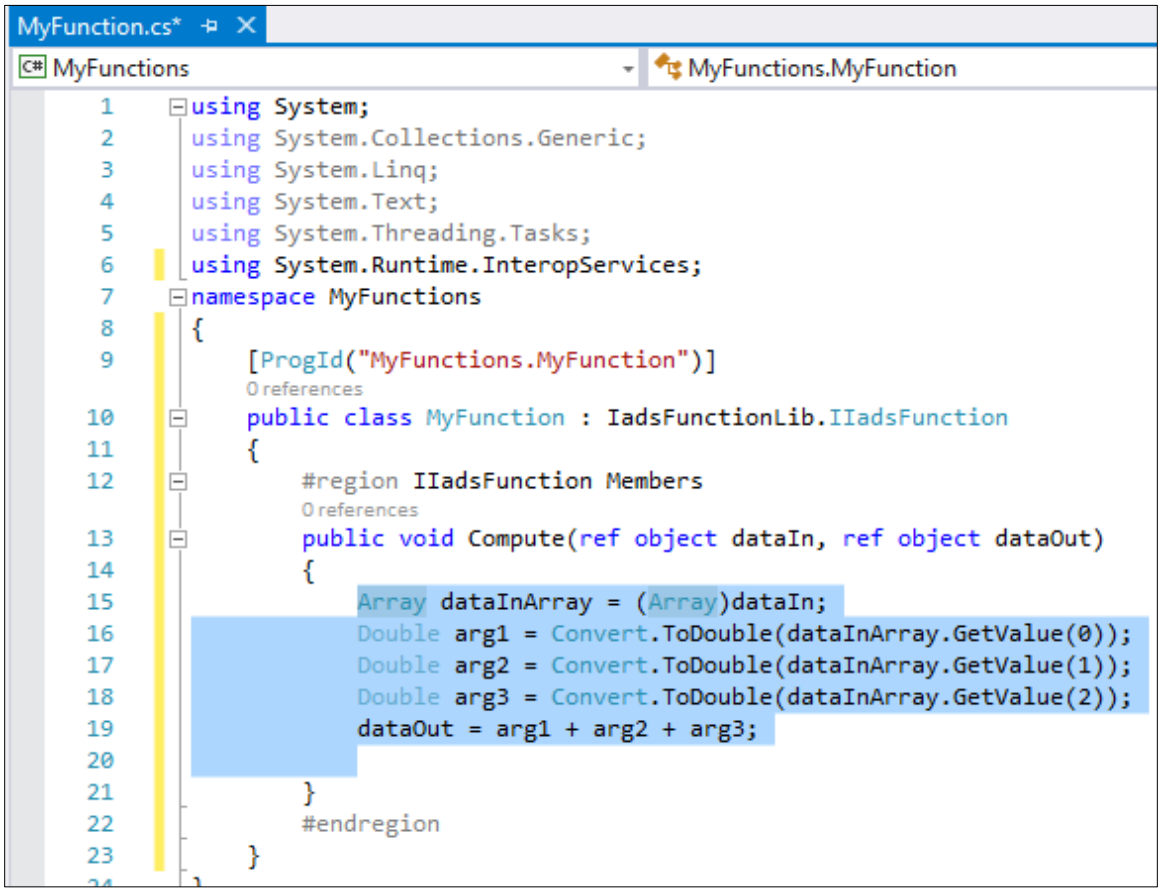


10) After choosing the “Implement Interface” menu item, Visual Studio automatically writes the shell of the function we must implement. Within the function, the “ref object dataIn” argument represents an array of input argument values from the IADS environment. The “ref object dataOut” represents the single return value that we are allowed to return. For example, if a user typed a derived equation MyFunction.MyFunction(1,2,Param1), the dataIn object would be an array of three elements containing the values 1,2, and the value of Param1 respectively. Our job is to take the input values, run some mathematical algorithm, and produce a single object ‘result’. Discussions on returning multiple results from a single function call will be touched upon at a later time. For simplicity sake, let us focus on the multiple in, single out technology.

11) During this next step, we will need to decode the dataIn object and extract the input parameter values. When that is complete, we can perform our calculation and return the result. In the code window remove the line of code containing the “throw” statement. Add the following code it its place:

```
Array dataInArray = (Array)dataIn;
Double arg1 = Convert.ToDouble(dataInArray.GetValue(0));
Double arg2 = Convert.ToDouble(dataInArray.GetValue(1));
Double arg3 = Convert.ToDouble(dataInArray.GetValue(2));
dataOut = arg1 + arg2 + arg3;
```

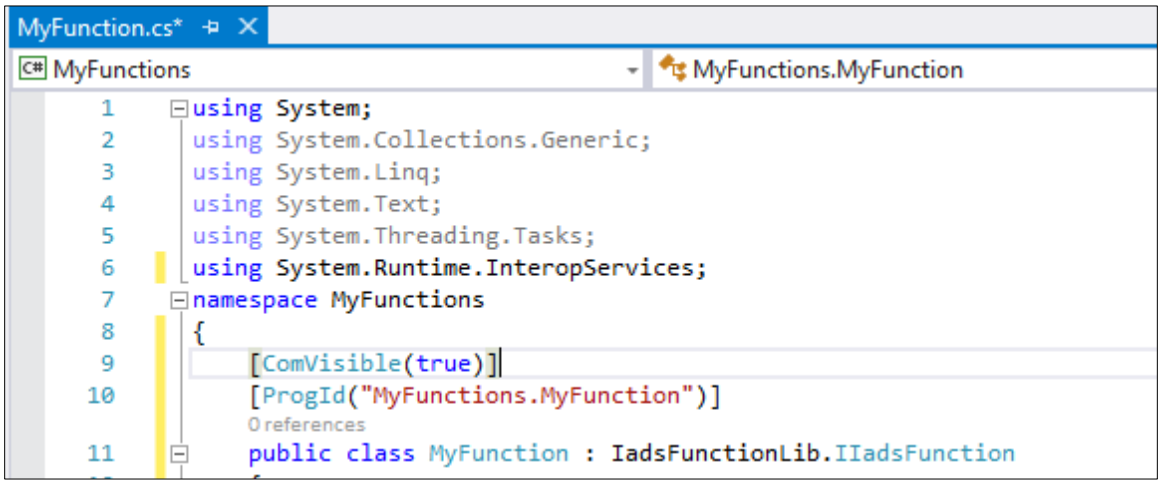
When you complete, your code should appear as follows:



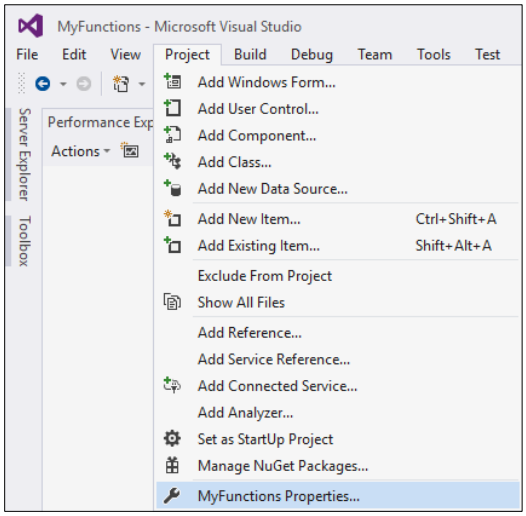
12) Notice in the first line, we cast the 'dataIn' array to a C# array type object. This will allow us to extract each function input argument in the subsequent lines of code. The first argument passed into this function from IADS is array element 0, the second argument is 1, and so on. By using the 'Convert' object, we can assign each input argument to a temporary variable. Once these temporary variables are assigned, we can perform our calculation and return our result. To return a result, simply assign the computed value to the 'dataOut' object.

13) At this point, you can begin modifying the code in the function to perform your specific computation. For more background on how to pass arguments, check their types, and return values, please refer to the SampleFunction project included with this tutorial. Make sure to read the comments in the supplied compute functions.

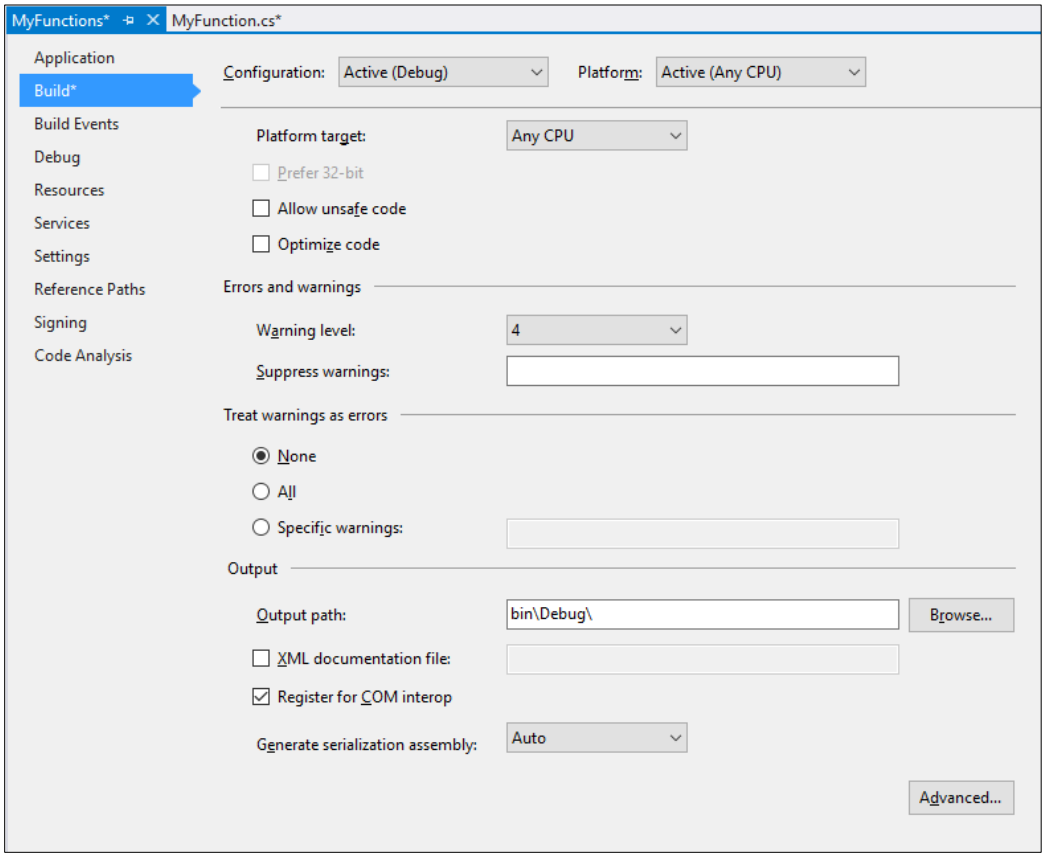
14) Now that your function is basically complete, we must take care of some remaining interop issues. We must ensure that the function is compiled with the necessary COM code so that it can communicate with IADS. In the code view, type `[ComVisible(true)]` above the ProgId definition line.



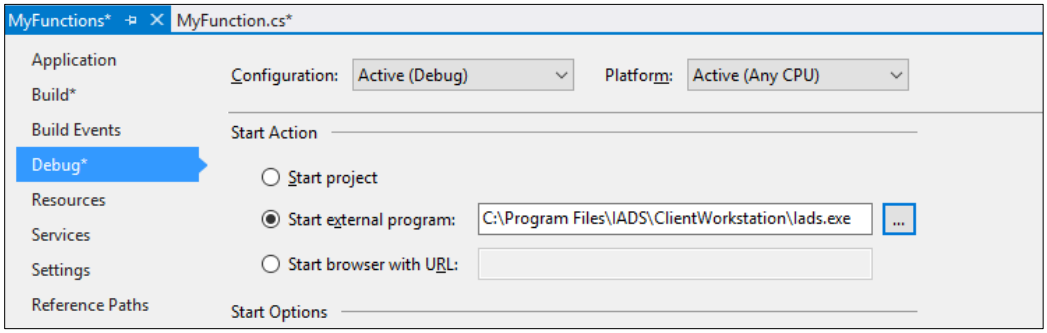
In the "Project" drop down menu, select **Properties**.



15) Under the "Build" tab, scroll down to the bottom and check the **Register for COM interop** option. These two last steps are very important. If you forget this step, or the previous `[ComVisible(true)]` directive step, the function will remain undefined in IADS because it will not properly registered on the system.



To debug the function, go to the "Debug" tab in the same dialog and set the "Start external program" item to the location of the IADS executable, in this case "C:\Program Files\Iads\ClientWorkstation\Iads.exe".

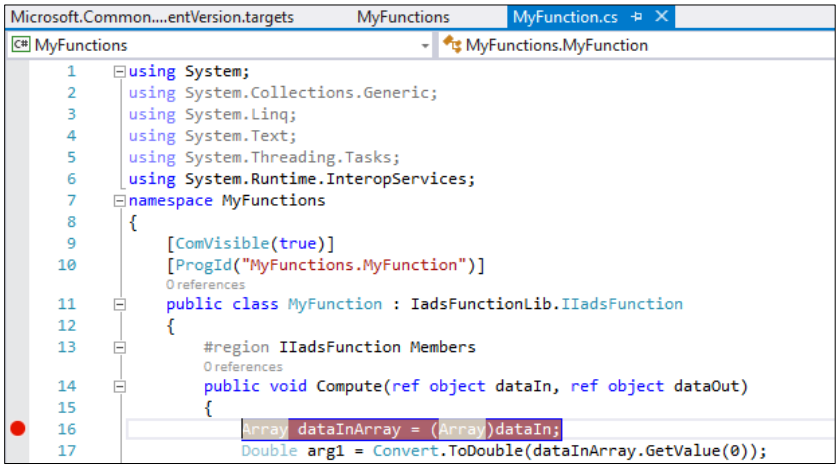


16) When all these steps are complete, compile the project, fix any errors and run. In IADS, build a new derived parameter that calls your new function, and drop the parameter in any display. If you want to debug your calculation step by step, put a break point in your compute function. The code will break for each new data point calculated.

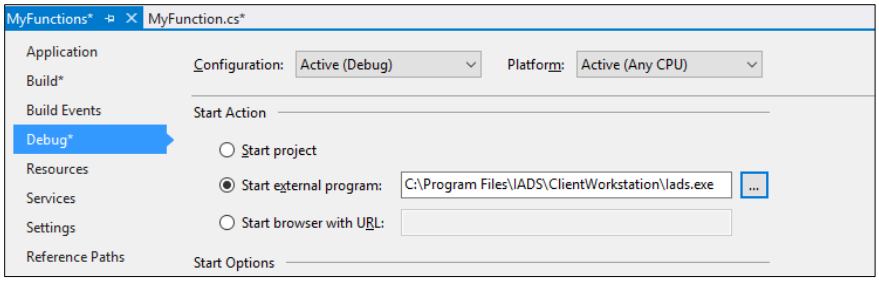
Note: See section 3.3 to access your new function in IADS.

3.1.1 Debugging your new function in IADS

1) Bring up your Visual Studio custom function project, and place a break point in your "Compute" method for testing.



2) To run IADS from the debugger, go to the "Debug" tab in the same dialog and set the "Start external program" item to the location of the IADS executable, in this case "C:\Program Files\IADS\ClientWorkstation\Iads.exe".



3) Build your solution again for good measure and click on the **Start Debugging** command (or the F5 key); IADS will start. When IADS starts, pick the configuration file you wish to use and click **Open**.

4) After IADS initializes, open the Configuration Tool, ParameterDefaults table (PDT) and create a derived parameter. If you have already created a derived parameter referencing your function, click on your equation in the PDT.

   Notice that when you "tab out" or finish the equation in the PDT, your function will be called. At this point you can debug all of the argument types and make sure you are getting the correct items. If you have an argument error and return an error code from your function, you will get an error message inside of IADS and the equation text will turn red in color. After you have checked out the arguments, you can remove the breakpoint and debug the function with real data.

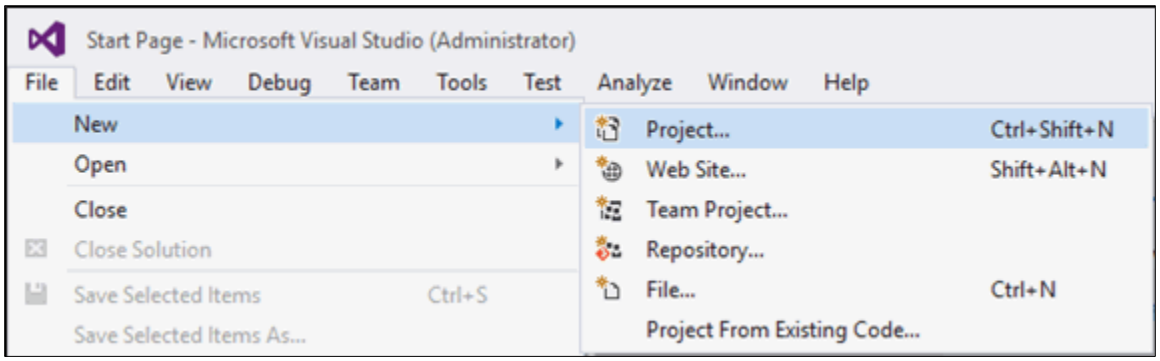
5) Add a display to the new Analysis Window (i.e., Alphanumeric or Stripchart) as described in section 3.3. If your parameter is not already attached to a display, drag and drop our new

function into the display. Your break point should now hit in the debugger. You can now step through your computational code as necessary.

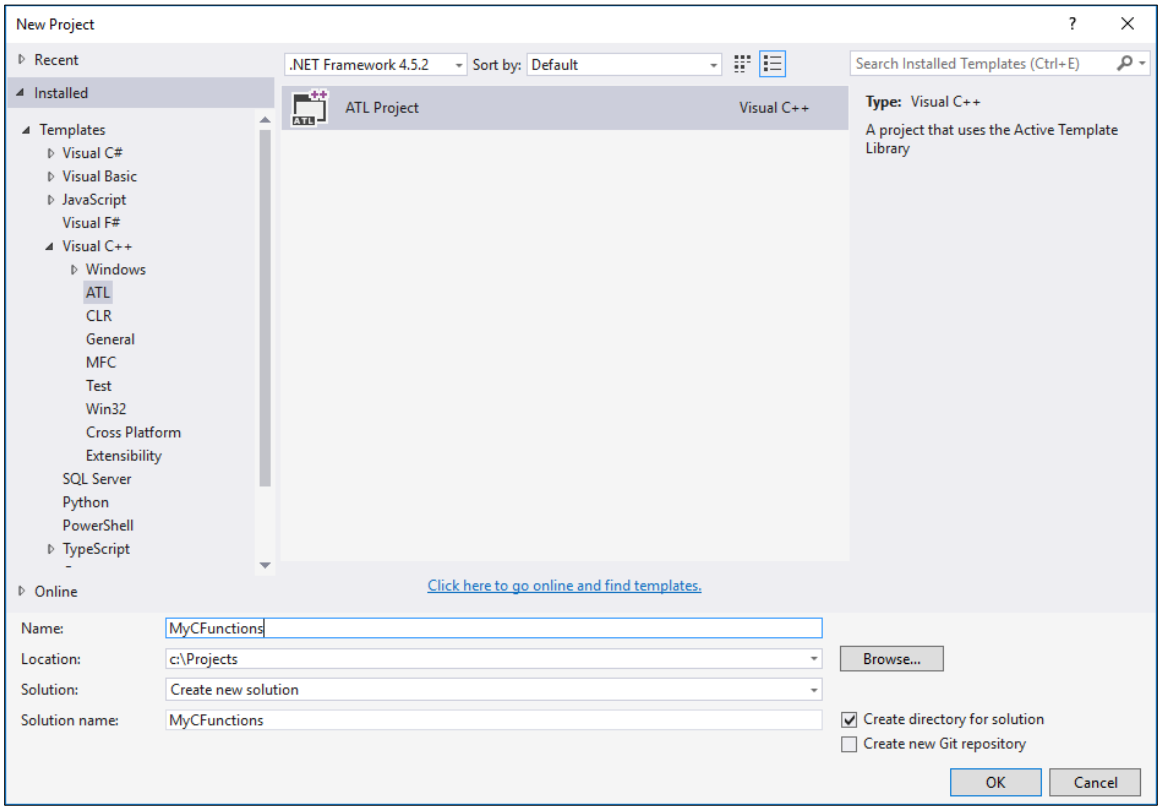
Again, for more background on how to pass arguments, check their types and return values, please read the comments in the example project.

3.2 Creating a custom derived function using C++ VS2015

1) Open up VS2015 and Select **File** > **New** > **Project**.



2) In the New Project dialog that appears, choose the "Visual C++ > ATL" tier and click the **ATL Project** option. At this point, please read the next step before you finish completing the dialog. There are some important considerations when choosing the proper project name.

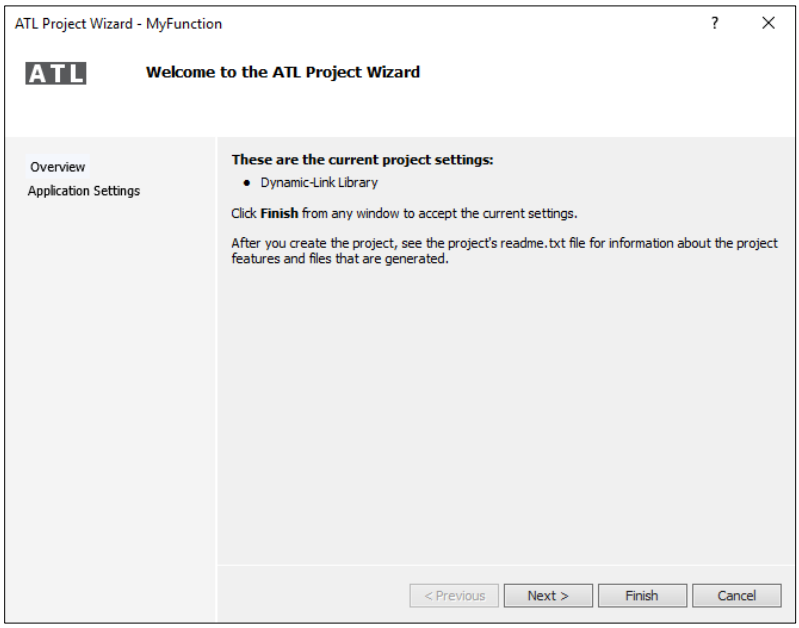


3) The project name you choose will become part of the function identifier name (aka ProgID, see inset). When it comes time to use your function in IADS, users will call your new function in a derived equation based solely upon its ProjectName.ObjectName (we will add the specific object name later). Plan on creating many functions in one "project" (most common and easier to manage the code). One way to look at it is that the project name is akin

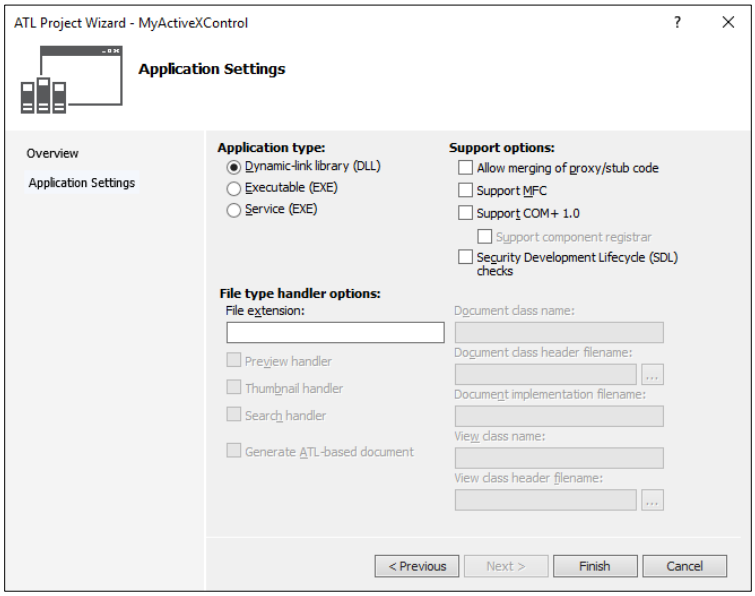
to the “Genus” of your function, so shoot for generality. Consider prefixing the project name with your organization like “NASA” or “Lockheed” and the type of functions you will be adding (example: NasaFluidFuncs).

Now, in the fields at the bottom of the dialog, enter the project name, location, and the solution name.

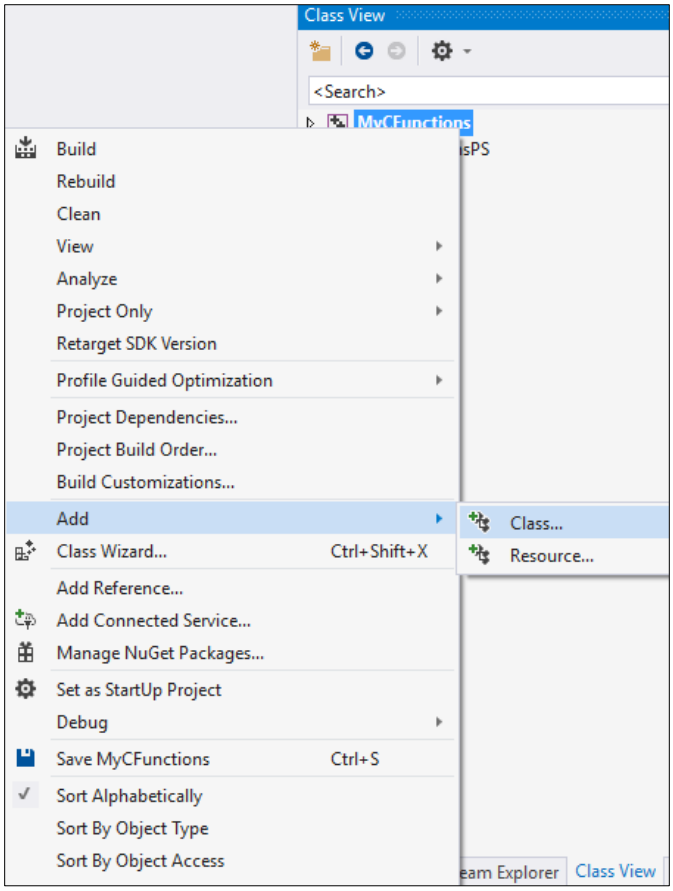
After pressing **OK**, the “ATL Project Wizard” dialog will appear as below.



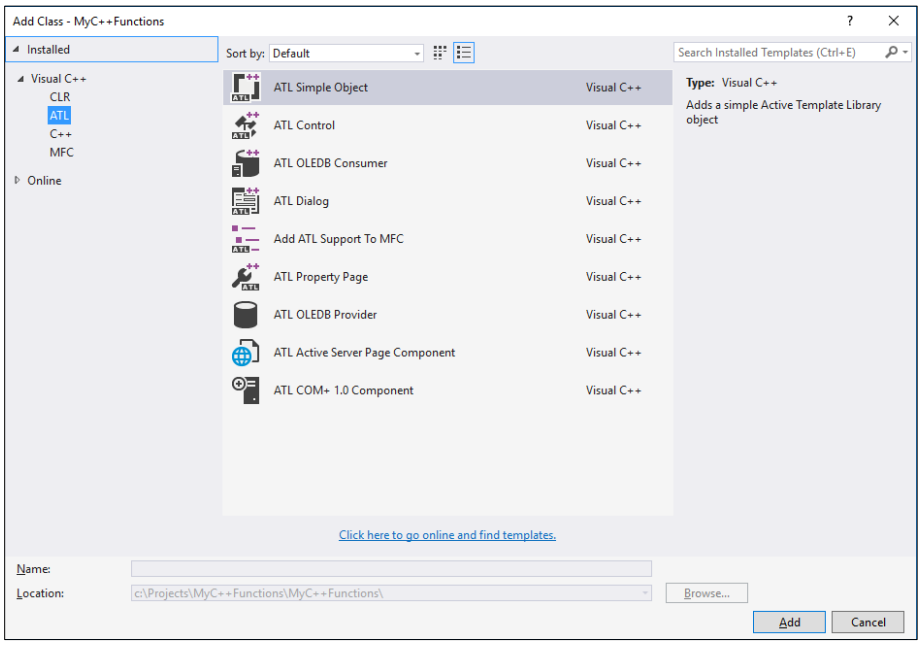
4) Click the **Next** button in the Wizard. On the new wizard page, ensure that the “Dynamic Link Library (DLL)” is checked. Every function that runs in IADS is of type DLL because it allows for maximum speed in computing calculations. Press the “Finish” button and the Wizard will set up your project.



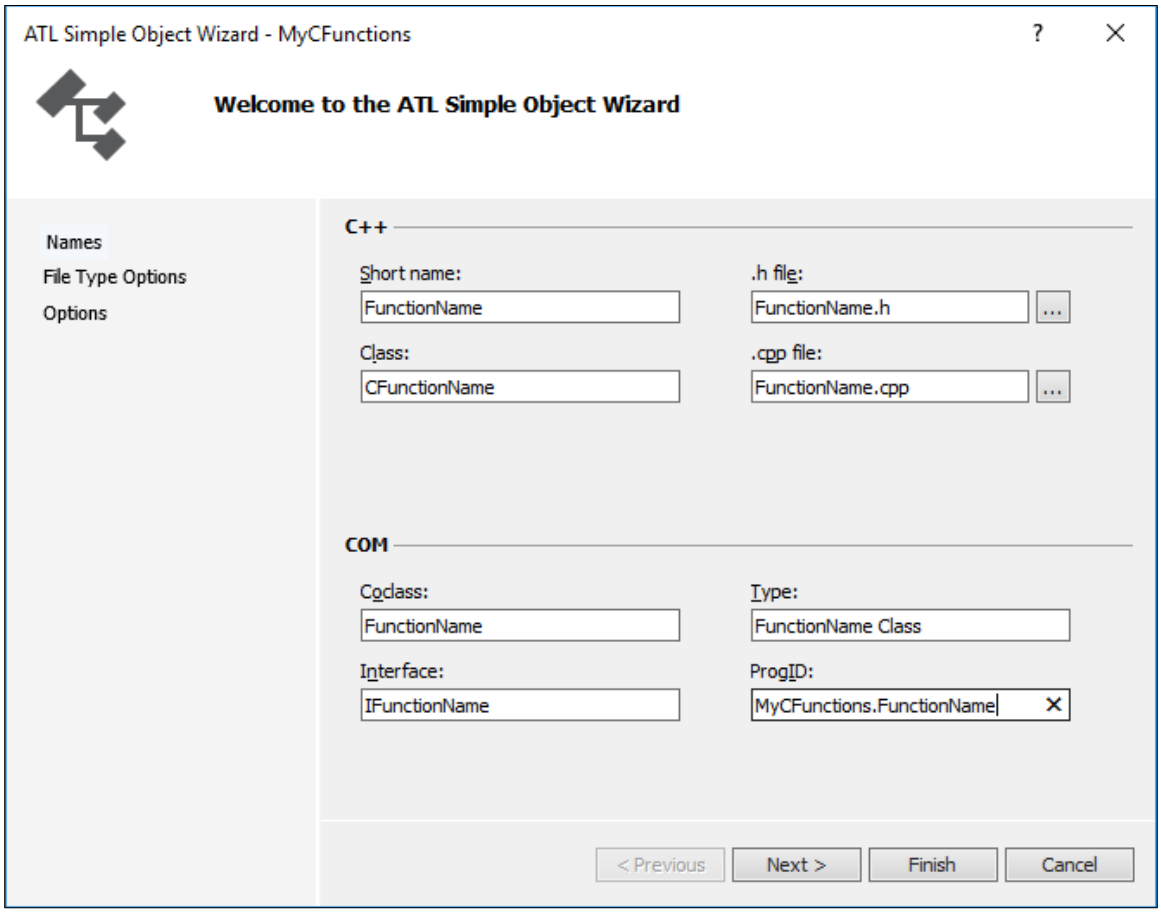
5) Next, go to the “ClassView” tab in Visual Studio’s workspace and right-click on the project name. Choose “Add->Class”.



6) Upon adding a new class you will be presented with a dialog. Click the **ATL** tier and **ATL Simple Object** as shown below. When that is complete, press the **Add** button.


Add Class - MyC++Functions
Installed
Visual C++
CLR
ATL
C++
MFC
Online
Sort by: Default
Search Installed Templates (Ctrl+E)
ATL Simple Object Visual C++
ATL Control Visual C++
ATL OLEDB Consumer Visual C++
ATL Dialog Visual C++
Add ATL Support To MFC Visual C++
ATL Property Page Visual C++
ATL OLEDB Provider Visual C++
ATL Active Server Page Component Visual C++
ATL COM+ 1.0 Component Visual C++
Type: Visual C++
Adds a simple Active Template Library object
Click here to go online and find templates.
Name:
Location:
c:\Projects\MyC++Functions\MyC++Functions\
Browse...
Add
Cancel

7) On the first tab, enter the name of your function in the “Short Name” field. The wizard will fill out the rest of the tab automatically. For this example, I used “FunctionName” as the short name. The name entered will be combined with your project name and will present the final function name inside of IADS (ProjectName.FunctionName) as explained on page 1. See the “ProgID” field in your dialog for your final IADS function name. Warning: Newer VisualStudio versions do not automatically populate the ProgID field. Please ensure the ProgID field contains your specific ProjectName.FunctionName text. If not, please type in the appropriate text manually. Press “Finish” to continue.

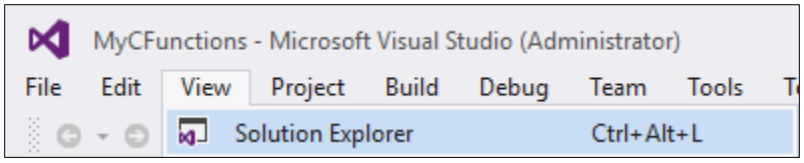


8) At this point, the Wizard will automatically create the shell of your function code. All we need to do now is to take care of the interface portion of the function. Basically, we will need to implement the defined “IIadsFunction” interface so that the function will be compatible with the IADS environment.

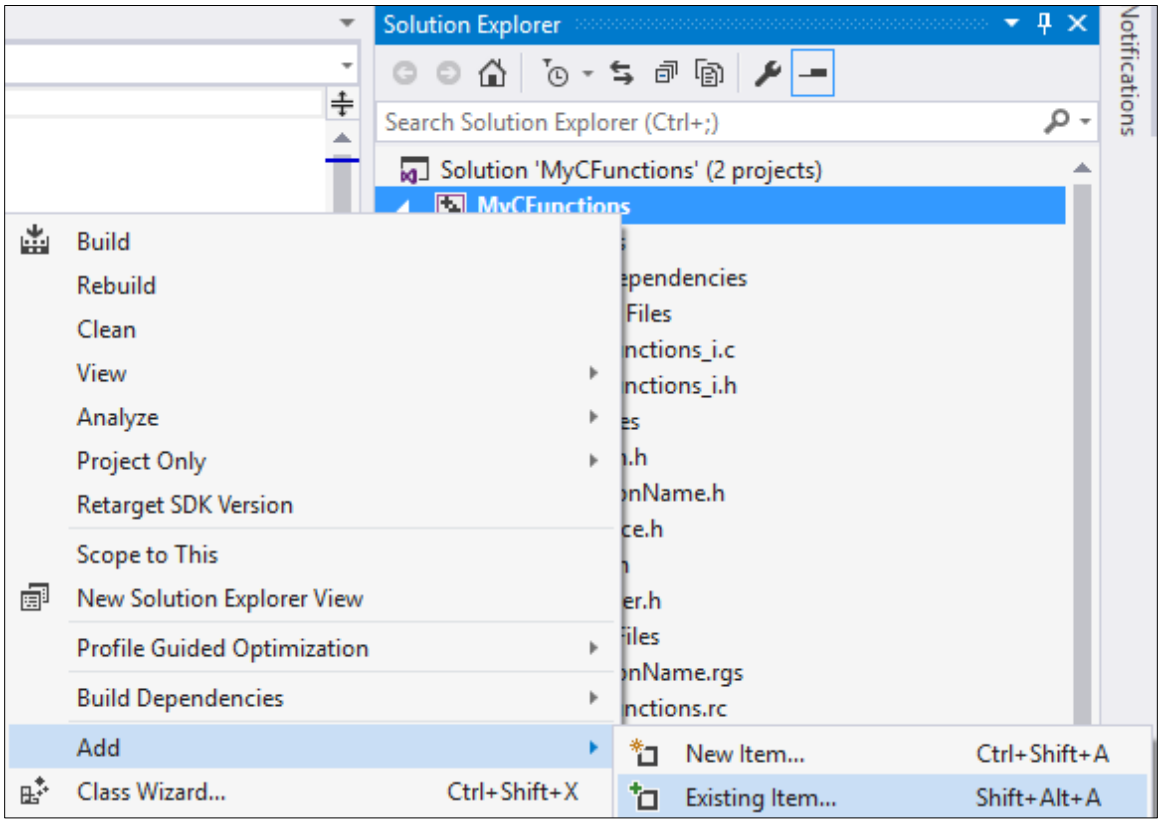
   Download the “Custom Derived Function Helper Classes” on the Curtiss Wright IADS web site: https://iads.symvionics.com/support/programming-examples/

   After you have downloaded the zip file, unzip its contents into your project folder. While unzipping, you will notice a file called “IadsFunction.idl”. That is the file we will use to implement the interface.

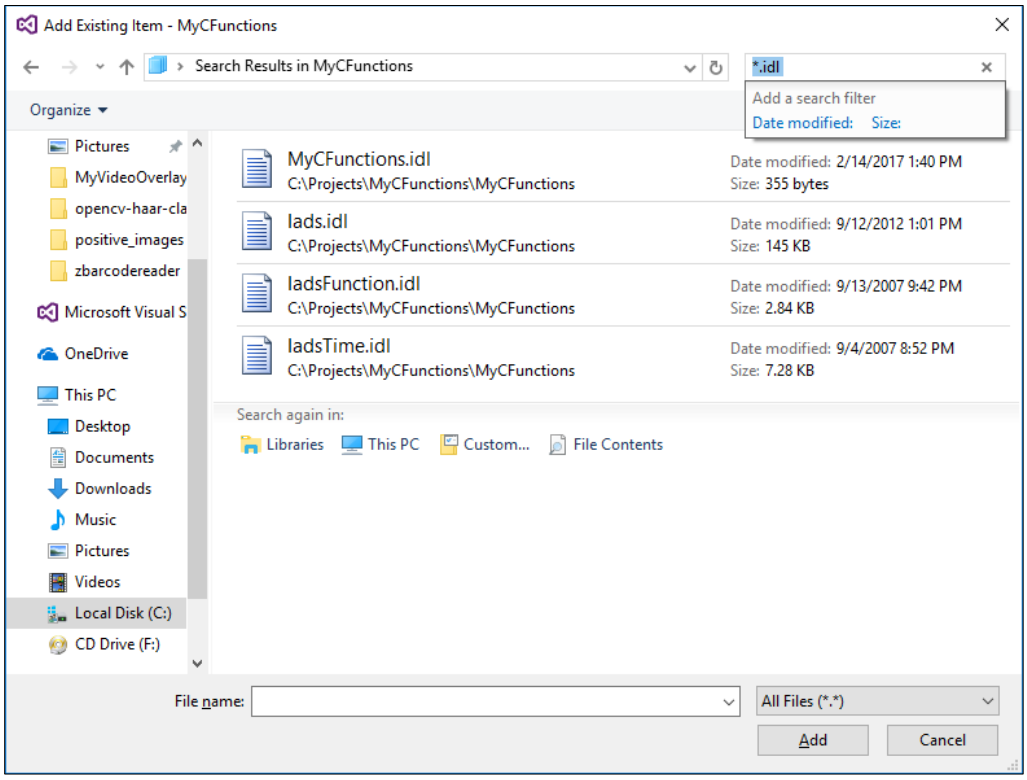
9) Now let's add the IadsFunction.idl to the project. Click the **View** tab at the top menu bar and select **Solution Explorer**.



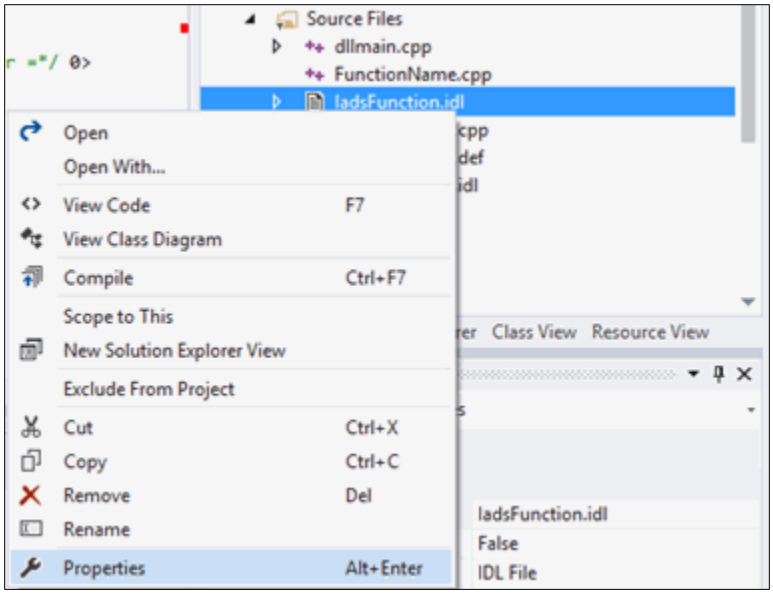
10) Expand you Solution, Right-click on the Project name, and select **Add** > **Existing Item…**



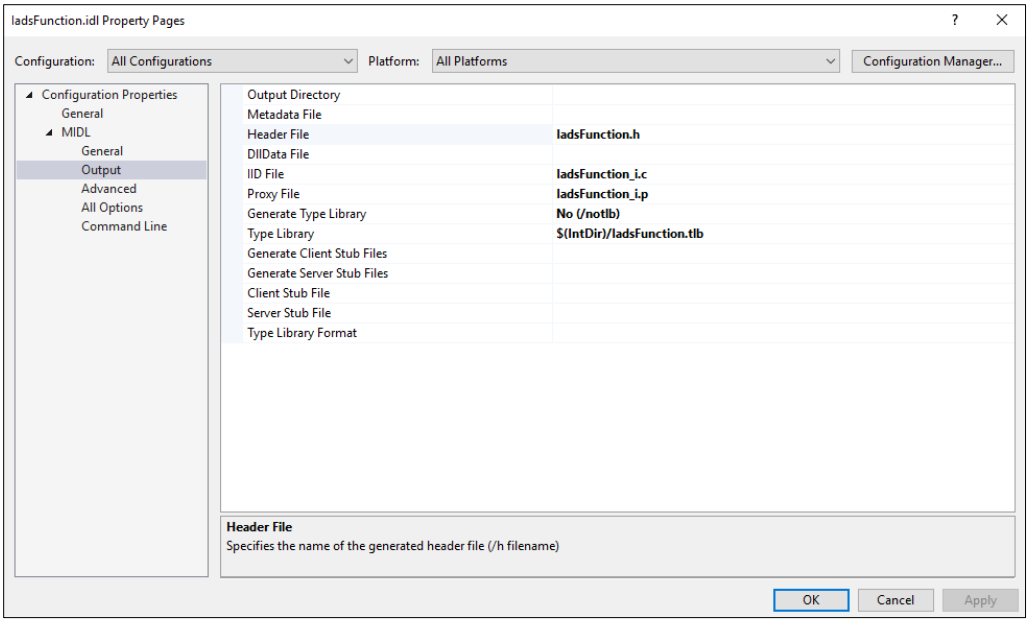
11) In the File Name box, type "*.idl" and then the enter key to show the Interface Definition Language files. Choose "IadsFunction.idl" and then press the **OK** button to add the file into your project. This should be the same IadsFunction.idl file that you unzipped in step 9.



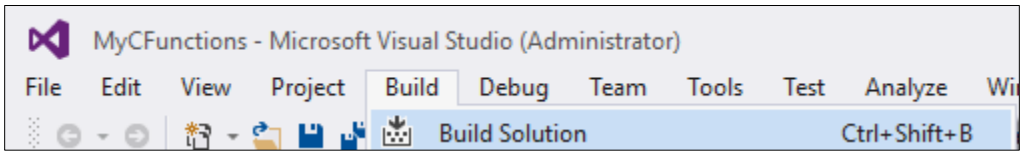
12) Due to an apparent bug in Visual Studio 2015 (and earlier), we will have to manually correct the output of the IadsFunction.idl file. Apparently, Visual Studio attempts to merge this information into the output of your project's idl file, but it does seem to work properly. Right-click on the IadsFunction.idl in your Solution Explorer and select "Properties".



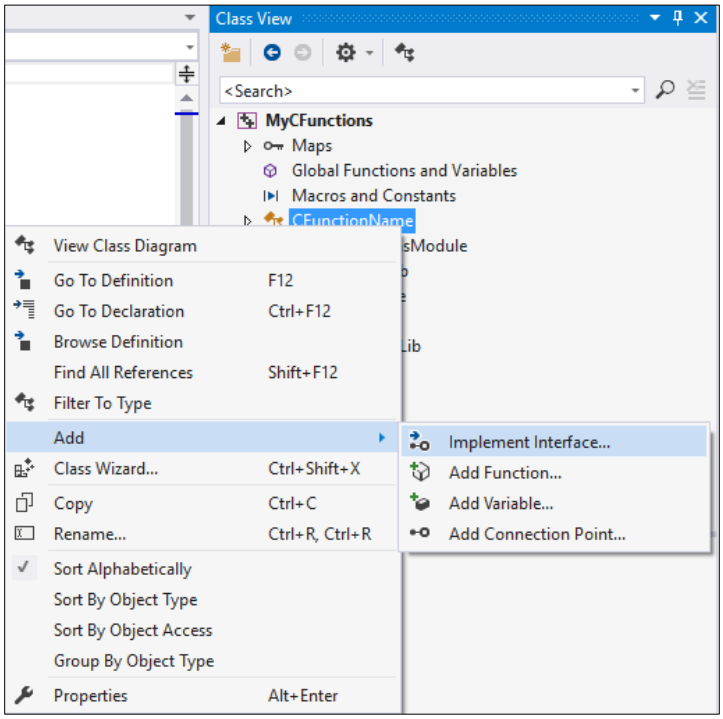
13) In the Property Pages dialog that appears, select the “All Configurations” drop down in the upper left-hand corner of the dialog. It is best to also select “All Platforms” in the Platform drop down. Open the MIDL->Output tier in the left window pane and correct the “Header File”, “IDD File”, “Proxy File”, and “Type Library” fields using the base name “IadsFunction”. When you are complete, the dialog should match the picture below. After confirming the dialog contents, press OK.



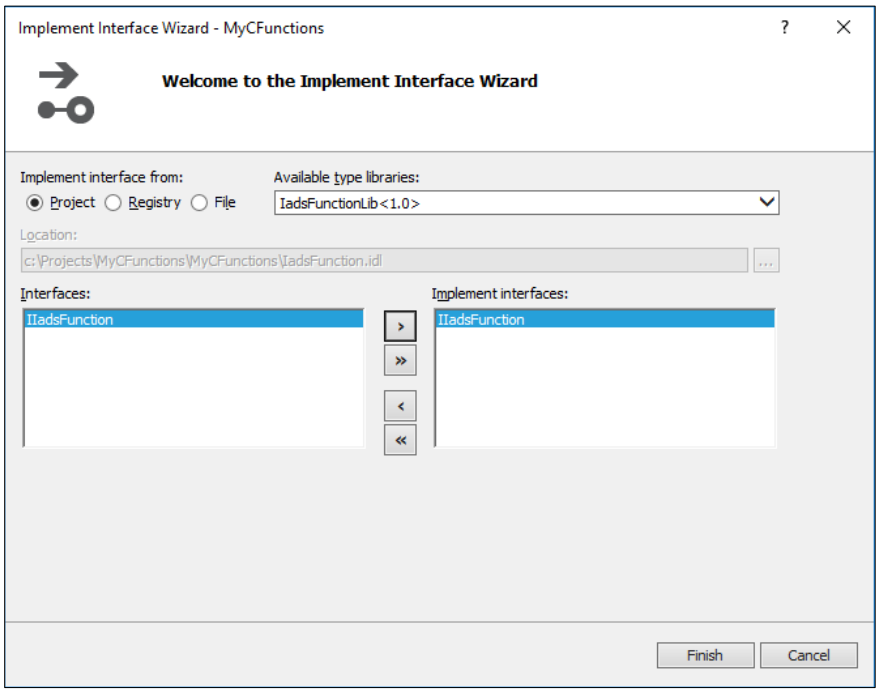
14) Now, build your Solution. After the build process is complete, a "typelib" file will be created. We can use this typelib file to implement the IadsFunction interface. The typelib file is simply a compiled binary version of the IDL file.



15) Go back to the ClassView tab of the Workspace viewer. Right-click on the "C[your function name]" class object and then choose "Add->Implement Interface…"



In the Implement Interface dialog, ensure that the IadsFunctionLib<1.0> library is selected in the "Available type libraries" drop down. When that is complete, you will notice that the IIadsFunction interface appears in the "Interfaces" list. Select IIadsFunction and press the ">" button. When IIadsFunction appears in the "Implement Interfaces" list, press **Finish**.



16) We're almost done now. At this point we can concentrate on the actual function code (at last). In the Solution Explorer tab of the Workspace View, locate your "[Function Name].h" file and click on it to begin edit. Scroll down to almost the end of the source code and locate the wizard generated code:

```
STDMETHOD(Compute)( VARIANT * dataIn,  VARIANT * dataOut)
{
  // Add your function implementation here.
  return E_NOTIMPL;
}
```

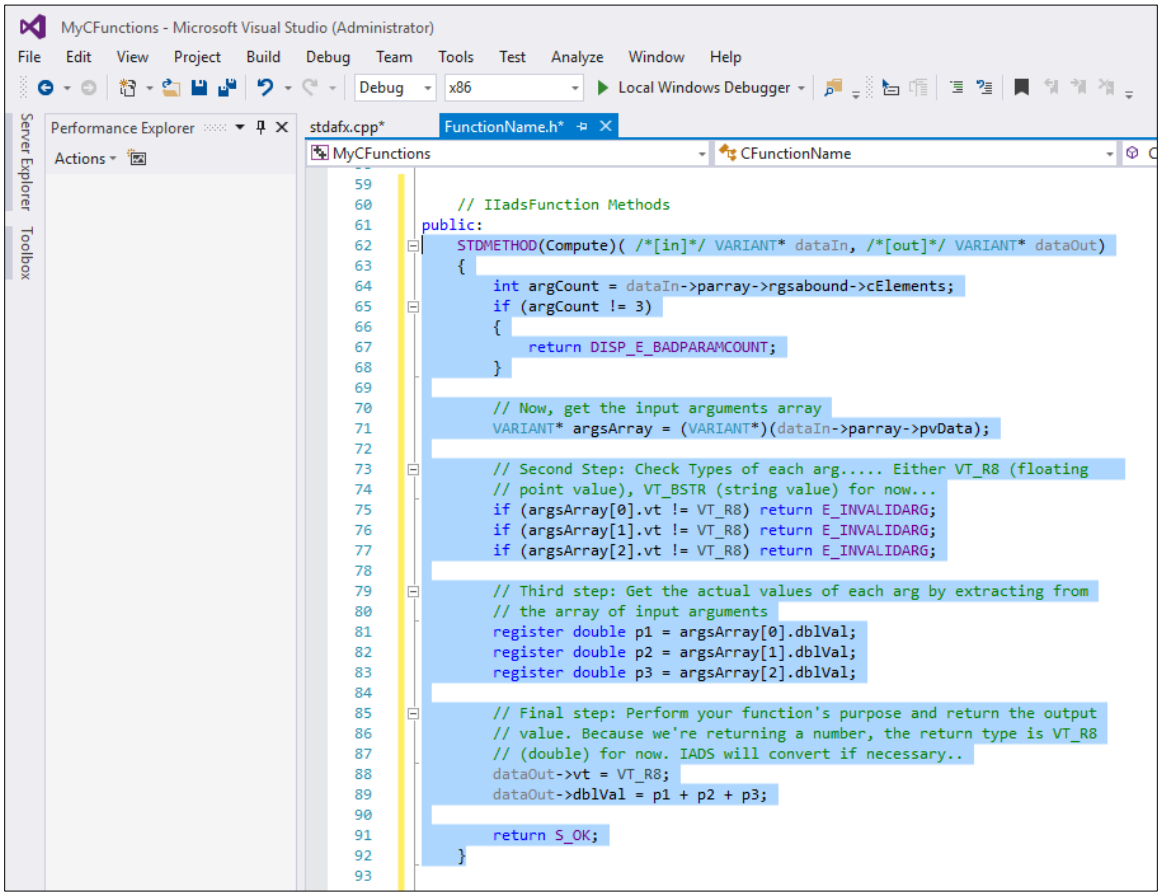
Remove this entire function as we are about to inject some example code.

17) In the place of the code you just removed, insert the following example code:

```
STDMETHOD(Compute)( /*[in]*/ VARIANT* dataIn, /*[out]*/ VARIANT* dataOut )
{
  int argCount = dataIn->parray->rgsabound->cElements;
  if ( argCount != 3 )
  {
    return DISP_E_BADPARAMCOUNT;
  }

  // Now, get the input arguments array
  VARIANT* argsArray = (VARIANT*)(dataIn->parray->pvData);

  // Second Step: Check Types of each arg..... Either VT_R8 (floating
  // point value), VT_BSTR (string value) for now...
  if ( argsArray[0].vt != VT_R8 ) return E_INVALIDARG;
  if ( argsArray[1].vt != VT_R8 ) return E_INVALIDARG;
  if ( argsArray[2].vt != VT_R8 ) return E_INVALIDARG;

     // Third step: Get the actual values of each arg by extracting from
```

```
    // the array of input arguments
  register double p1 = argsArray[0].dblVal;
  register double p2 = argsArray[1].dblVal;
  register double p3 = argsArray[2].dblVal;

  // Final step: Perform your function's purpose and return the output
  // value. Because we're returning a number, the return type is VT_R8
  // (double) for now. IADS will convert if necessary..
  dataOut->vt = VT_R8;
  dataOut->dblVal = p1 + p2 + p3;

  return S_OK;
}
```

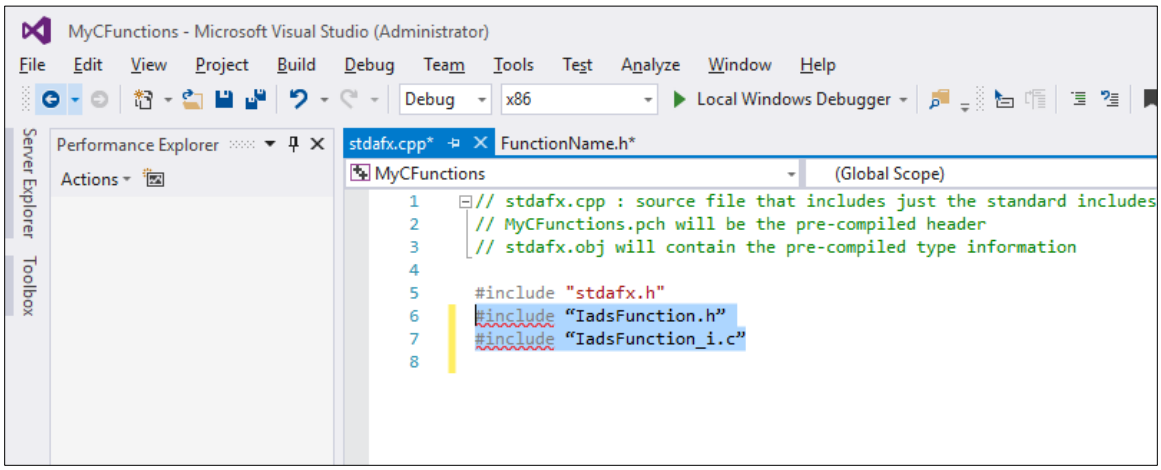
When that step is complete, your code should like something this:



18) Now, build the solution. After the build is complete you will notice that we have link errors. This is a continuation of the Visual Studio bug as noted in steps 11 and 12. To correct the errors, we will need to add the newly created IadsFunction files into the StdAfx.cpp file.

In the Solution Explorer, click on the **StdAfx.cpp** file and add the following lines to the source code:

#include “IadsFunction.h”
#include “IadsFunction_i.c”



19) At this point, you can begin modifying the code in the function to perform your specific computation. For more background on how to pass arguments, check their types, and return values, please refer to the SampleFunction project included with this tutorial. Be sure to read the comments in the supplied compute functions.

20) After you are done modifying the code, build the Solution. By building your Solution, the new dll should be registered so you are ready to run and debug it now inside of IADS. The next section in the tutorial describes how to debug the function.

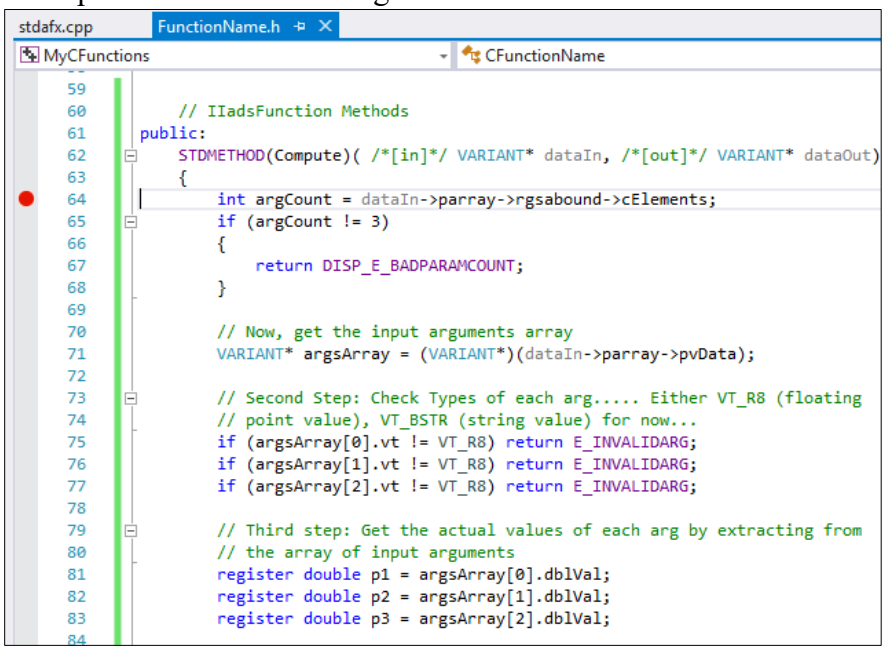
    If you want to use your function on another PC, you will need to register the dll on that specific PC. Please consult the web for documentation on “regsvr32.exe” and how to perform this procedure.

    If you want to add another function, simply repeat steps. You can add as many functions to this project as you would like and they will all be accessible through the same dll (i.e. MyFunction.FunctionName1, …, MyFunction.FunctionNameN). If you wish to create an entirely new dll and set of functions, you will need to repeat this entire tutorial using a unique project name.

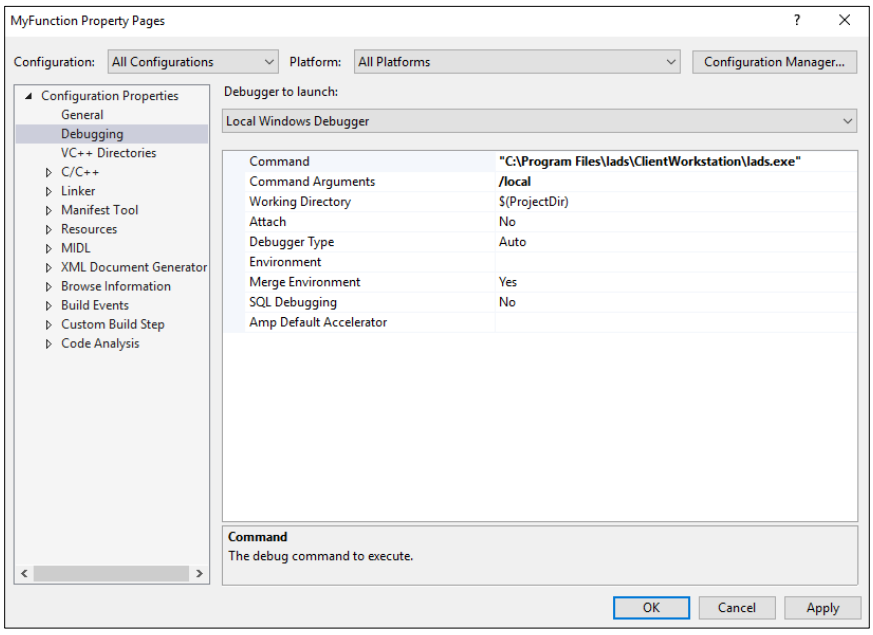
    Note: See section 3.3 to access your new function in IADS.

3.2.1 Debugging your new function in C++ VS2015

1) Bring up your Visual Studio custom function project, and place a break point in your "Compute" method for testing.



2) Go to **Project** > **[ProjectName] Properties** drop down menu in Visual Studio, and in the dialog that appears pick "Iads.exe" as your "Executable for debug session". The Iads.exe file is in your "C:\Program Files\Iads\ClientWorkstation" directory. When you are ready to continue, press the **OK** button.



3) Build your Solution again for good measure and click the **Go** command (or the F5 key). IADS will start. When IADS starts, pick the configuration file you wish to use and click **Open**.

4) After IADS initializes, open up the Configuration Tool and create a derived parameter in the ParameterDefaults table. If you have already created a derived parameter referencing your function, simply click on your equation in the ParameterDefaults table.

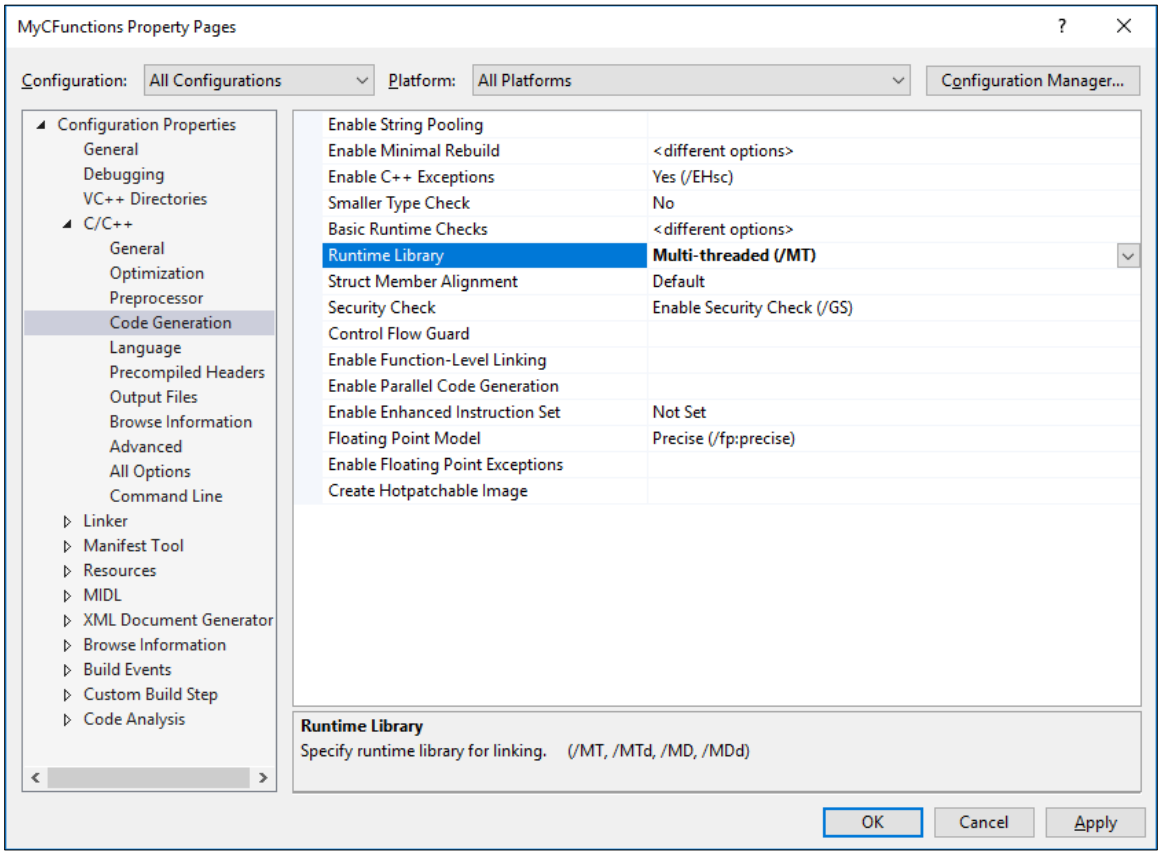
   Notice that when you “tab out” or finish the equation in the Parameter Defaults table, your function will be called. At this point you can debug all of the argument types and make sure you are getting the correct items. If you have an argument error and return an error code from your function, notice that you will get an error message inside of IADS and the equation text will turn red in color. Once you have checked out the arguments, you can remove the breakpoint and debug the function with live data.

5) Add a display to the new Analysis Window (i.e. Alphanumeric or Stripchart) as described in the section 3.3. If your parameter isn’t already attached to a display, simply drag and drop your newly built derived parameter into the display. Your break point should now hit in the debugger. You can now step through your computational code if necessary.

3.2.2 Deploying your new function in C++ VS2015

When it comes time to deploy your new function to users on other PCs, you need to consider a couple of issues. One issue is that your control may require some auxiliary dlls that are not available on the other systems. If that occurs and the dlls are missing, the function may not operate. To help minimize this possibility, you must always build your new function dll in "Release" mode. You should never distribute a function dll that has been compiled under the "Debug" mode. The debug mode uses libraries that will most certainly be missing on any machine without Visual Studio installed. Beyond that, it is always best to 'statically link' all the runtime libraries. Also, since we have used ATL to build this function, we will need to statically link the ATL library as well.

1) In Visual Studio, select the **Project** > **Properties** drop down menu. Make sure that the "Configuration" dropdown is set to "Release". Under the "Configuration Properties > C/C++ > Code Generation" tier, set the "Runtime Library" to **Multi-threaded (/MT)**.



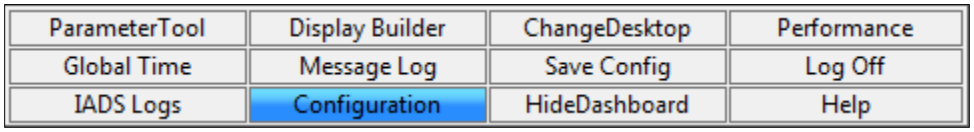
2) Once you have made these changes to your project, you should rebuild your ‘solution’. Make sure once again that your current configuration is set to “Release” and then select the **Build** > **Rebuild Solution** drop down menu option. After this step is complete, your function dll should be in your project “Release” folder. It should now be ready to deploy on another system.

   The function dll will need to be copied to the other PC and ‘registered’. In order to register the dll, you will have to run the ‘regsvr32.exe’ program. One easy way to accomplish this is to double click on the dll in Windows Explorer. When asked what program to execute on the dll, navigate to the Windows\System32 directory and choose the regsvr32.exe file. This procedure may be different if the operating system is a 64-bit version. Please consult the online documentation for specifics.

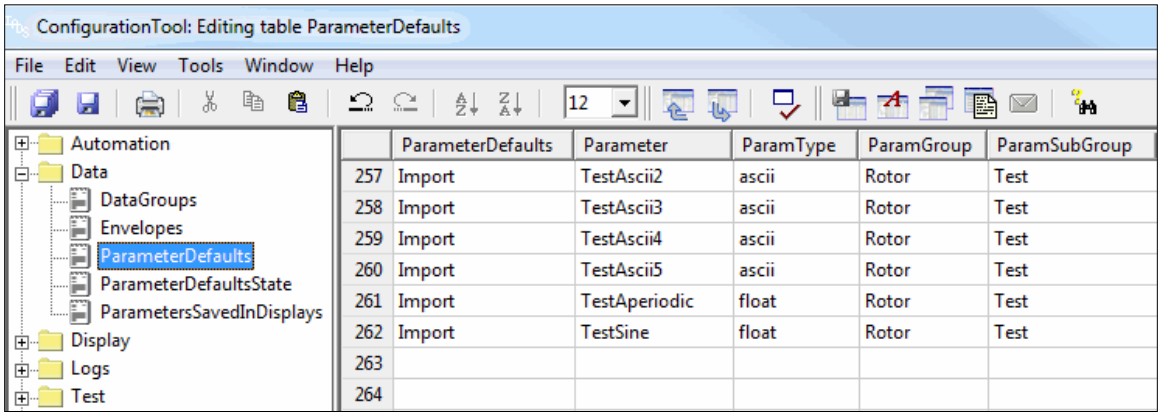
   If the dll fails to register at this point, it most likely failed to statically link the needed dlls. We can investigate which dlls are missing by using the “Dependency Walker” tool. The Dependency Walker program is located within the Microsoft Visual Studio\Common\Tools directory and is named “Depends.exe”. Copy Depends.exe from your development PC to the target PC and run the program. From the File drop down menu select **Open** and choose your function dll. Examine the module list in the bottom window pane. Any missing dependent dlls should show up with a question mark. Search for those dll names on the net and find out their purpose. It might help you narrow down what solution setting you have missed. It is also possible that the missing dll is a private library that you are using, in which case you will need to either static link or copy that dll to the target machine as well.

3.3 Accessing your new function in IADS

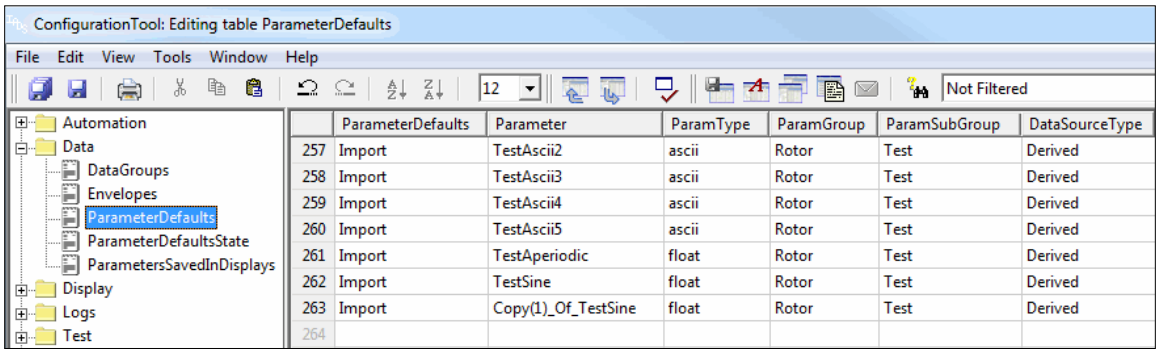
1) Run IADS and login to a test desktop.
2) Press the **Configuration** button on the IADS Dashboard in the lower right-hand corner of the screen.



3) In the Configuration Tool dialog left window pane, click the "+" sign next to the "Data" folder to open it and then select the **ParameterDefaults** table. This is the location in IADS where you will build a new derived parameter to test your function.



4) To add a new derived parameter, for speed, simply copy the last line in the table and then replace our new values as necessary. Select the last row in the table by clicking on the row number. After the row is selected, press **Ctrl+C** to copy and then follow that by a **Ctrl+V** to paste. You should now see a copy of the last line placed into a new row. When you are done the table should look something like this:



5) Click into the first column of the new row. As we go, to proceed to the next cell press the **Tab** key.

   Do not edit the first column, press the **Tab** key to start editing the second column. In the second column, type the name of your test parameter "TestMyFunction". Once you are done press the **Tab** key. Now set the type of the parameter; just leave it "float" (i.e. 4-byte floating

point number). In the future if you are testing an Ascii return value, you will need to set this type to Ascii.

At this point, keep pressing the Tab key until you arrive at the “DataSourceType” column. Make sure that is set to “Derived”.

In the next column (DataSourceArgument) you will write your derived equation. Now, remember from the discussion while creating your function regarding the function name. Enter the function name followed by the arguments:

MyFunctionGroupName.FunctionName(5.0, 10.0, 30.0)

If you want some variety to your test data, you can use something like this:

*MyFunctionGroupName.FunctionName(Rand()*5.0, Rand()*10.0, Rand()*30.0)*

Or if you already have specific input parameters in mind, you can do something like this:

MyFunctionGroupName.FunctionName(Param1, Param2, Param3)

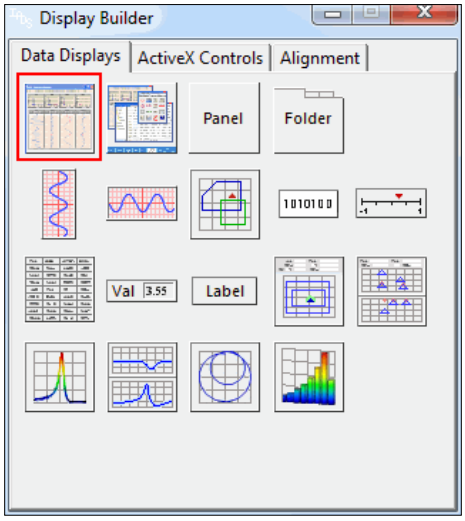
In the next field (UpdateRate), type the sample rate that you desire to update your function. If your equation is based off of other parameters, the sample rate will be automatically computed and placed into this field when you Tab out of the cell.

Just for safe measure, press the Tab key until you get to the “FilterActive” column. Make sure that it is set to “No”. We don’t want a filter to be affecting our output at this time, or it could lead to confusion.

After these steps are complete, click the **Save**  button in the Configuration Tool toolbar. Your new parameter will now be available in the Parameter Tool.

To run the function, drop the parameter into any display.

6) To build a test display, create an empty Analysis Window by dragging the icon from your Display Builder onto your Desktop.

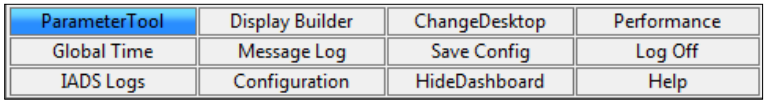


7) Now add the “Alphanumeric” display to the Analysis Window you just created using the same drag-n-drop process; you should see the new display in the Analysis Window when

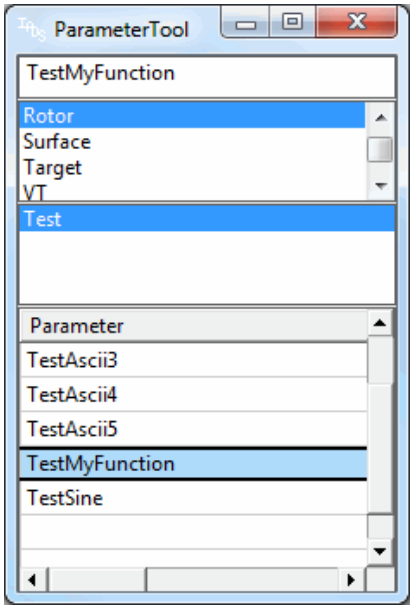
complete. The Alphanumeric is a very simple text display that will be easy to view the equation output results.

Note: Use your cursor to hint on icons on the Display Builder to see which type of display they are.

8) Ok, now for the parameter attachment to the display. Click on the **Parameter Tool** button in the IADS Dashboard (bottom right hand corner of screen). The Parameter Tool dialog will appear. The Parameter Tool dialog contains a list of all your available parameters in the configuration. Now all we need to do is find our parameter.



9) In the top text field (quick find box), start typing the parameter name. I used the name "TestMyParameter", if you have done the same then type "TestMy". You will notice that the window at the bottom opens as soon as it finds your parameter. Keep typing until you see the full parameter appears. Once it is visible, click on the parameter name and drag the parameter into the display on the Analysis Window. As soon as you drop the parameter, data should appear. This is the actual output of your function!



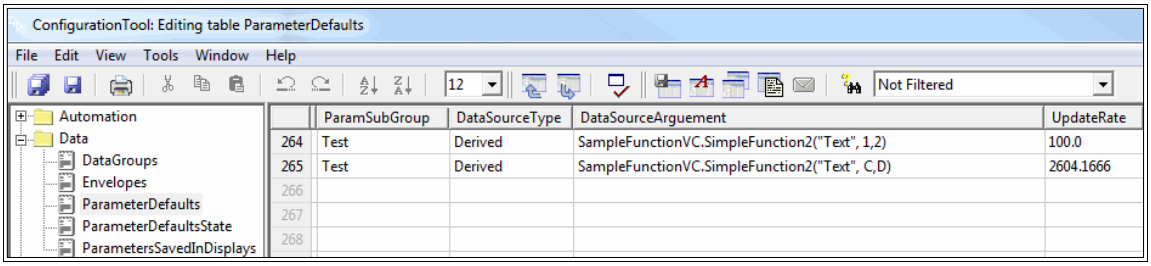
10) After your initial checkout is complete, you can move on to displays such as the Stripchart that will show history and allow you to examine the data point by point for discrepancies. Simply repeat the drag-n-drop process using the Stripchart icon (instead of the Analysis Window icon) in the Display Builder (step 6). Make sure to save the configuration for later.

3.4 Advanced Topics

3.4.1 Initialization and execution of your custom function

In this section, we will review the steps taken during initialization and execution of your custom function. It is important to be aware how IADS creates your function, as well as how it calls your function during both the “initialization stage” and the “computation stage”. This will affect how your Compute function is designed. For reference, you can refer to the SampleFunction2.h file in the SampleFunction project.

First, let’s examine the initialization stage of your function in general. Each and every time a derived parameter is created that references your custom function, an instance of your custom function object is created within the parameter’s computational engine. When the parameter requires data, this object is then used to produce results as described by your specific custom code. As a general rule, your custom function object is created each time a user drops a derived parameter referencing your function into a display, enables and IAP parameter referencing your function, or edits an equation in the ParameterDefaults table referencing your function.



Extending this logic, each “instance” of your function called from within IADS is a completely independent unit of code, akin to a C++ object with member variables and corresponding code. In essence, each derived parameter is running a fully independent object. Obviously, this is necessary if your function maintains states such as “last value” or perhaps a specific “matrix” input file that is required and chosen by the user via the function’s input arguments. In reality, your function can be called from many different derived parameters simultaneously, each with their own unique set of input arguments, and possibly computing at different times within the data. Because of this wide variety of possibilities, be aware that any reference to “static” or “global” variables should be considered carefully. Global variables will allow you to “share” information between multiple instances of your function, but you will have to be very careful about the timing considerations. If you do decide to venture down this path, please do post your scenario to the IADS Google Group. In general, avoid all use of global variables and instead, use member variables within the class to hold any necessary state information.

Now let’s examine the initialization stage in more detail. The function name (i.e. ProgID) within the derived equation is used to call the “CoCreateInstance” function in the Microsoft COM libraries to create your object. Once your object is created within IADS, the “FinalConstruct” method is called. In this method, you can put any initialization needed that is independent of the input values to your function. This most likely would be limited to things such as setting member variables to a known initial value.

```
STDMETHOD(FinalConstruct)( void )
{
  // The ATL "goo" will call this upon construction of your class. It be called once per class creation.
  // Every derived parameter that get's called and uses this function will create it's own "instance" of this
  // class, so if you have 10 derived parameters calling the function, this function will be called 10 times, but
  // each call will be a complete unique copy of this class.
  // If you want to create any "global" resources, that are shared between all the class instances, make sure you
  // create a global static variable (i.e. above this class definition see "example shared variable"). Anything that you
  // want kept separate per instance and not shared, declare in the member variable selection below (see CComBSTR mStrin

  // This variable is for performance and initialization reasons as you will see below in the Compute function.
  // Make sure to add this member variable to your class -> bool mWasInitialized
  mWasInitialized = false;

  // Preparation for an example on how to output a "blob" data
  // For now, we just set our SafeArray pointer to NULL. We'll do the allocation in the Compute function init section
  mSA = NULL;

  return S_OK;
}
```

For instance, say you were building a function allowed a user to specify a number of data points to “buffer” before computing a result. Of course, you will need a member variable in the class to hold this buffer. During the FinalConstruct, you would set your member variable buffer pointer to NULL, but you would not allocate the memory. At this point in the initialization, you don’t have any of the argument values from the user’s equation, thus you don’t know how large to allocate the buffer. In the next paragraph, we will discuss a way to solve this issue.

After your FinalConstruct function is called, IADS then calls the “Compute” function within your object. The main purpose of this **first** call to your Compute function is to validate the equation input variables. Understand that the custom function interface is flexible enough to allow any number of input arguments, and each argument could be a different type (float, ascii, blob, etc). It is at this exact time, the very first call to your Compute function, which you will need to check the number and types of your input arguments. In fact, IADS will only listen to your input argument error return codes on the first call to your function. Since we only want this code to execute on the first call to the Compute function (and never again), a Boolean member variable can be used to solve the problem. Simply add a member variable to your class and initialize it to false in the FinalConstruct.

```
STDMETHOD(FinalConstruct)( void )
{
  // This variable is for performance and initialization reasons as you will see below in the Compute function.
  // Make sure to add this member variable to your class -> bool mWasInitialized
  mWasInitialized = false;
```

The secondary purpose of this first Compute function call is to give you an opportunity to initialize any further variables (such as buffers, etc). Now, inside the Compute function you can check the Boolean member variable’s value, perform your argument checks and buffer initialization, then set the member variable so the code is not triggered again. See the example code snippet below or refer to the SimpleFunction2.

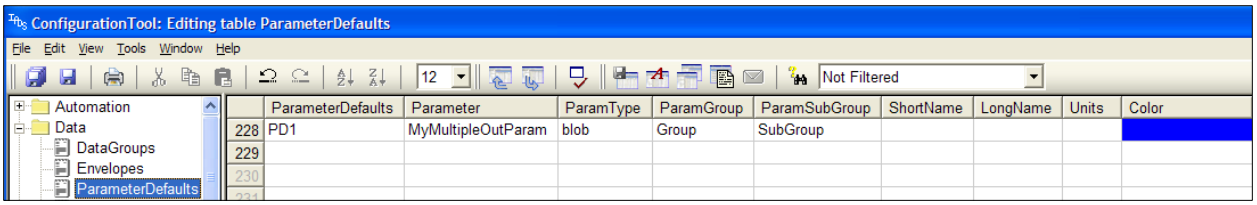
```
STDMETHOD(Compute)( /*[in]*/ VARIANT* dataIn, /*[out]*/ VARIANT* dataOut )
{
  // Get the input arguments array
  register VARIANT* argsArray = (VARIANT*)(dataIn->parray->pvData);  // Could use SafeArrayAccessData, but slower..

  // The very first call to this function is designated as an "initializion call".
  // Check all parameter types here and then never do again. Adding this check to your code will speed up the performance
  if ( !mWasInitialized )
  {
    // First Step: We need to check how many arguments were passed into our function from the user
    // Example, if the user creates a derived parameter with the function -> SampleFunction.SimpleFunction2( "Format", Valu
    // The argument count would be 3
    int argCount = dataIn->parray->rgsabound->cElements;
    if ( argCount != 3 )  return DISP_E_BADPARAMCOUNT;

    // Second Step: Check Types of each arg..... Either VT_R8 (floating point value), VT_BSTR (string value) for now...
    // In IADS, most every type of numerical argument is passed via an 8 byte floating point value VT_R8.
    // If you're in doubt, use VT_R8. You can also break here in the code and examine the argsArray[N].vt value to see the
    // Example, if the user creates a derived parameter with the function -> SampleFunction.SimpleFunction2( "Format", Valu
    // The argument type for arg1 would be VT_BSTR... and arg2/arg3 would be VT_R8
    if ( argsArray[0].vt != VT_BSTR ) return E_INVALIDARG;
    if ( argsArray[1].vt != VT_R8 ) return E_INVALIDARG;
    if ( argsArray[2].vt != VT_R8 ) return E_INVALIDARG;

    // The benefit of initializing here is that you can return an error code back to the user in the event of failure
    // Here's a list of possible return values at this point:
    //  A) E_FAIL or E_UNEXPECTED -> Returns an "unspecified error" to the user
    //  B) E_OUTOFMEMORY -> Returns an "out of memory error" to the user
    //  C) E_INVALIDARG or DISP_E_TYPEMISMATCH -> If one or more of the function arguments were of incorrect type (i.e. nu
    //  D) DISP_E_BADPARAMCOUNT -> If the number of function arguments were incorrect, returns an "invalid number of parame

    // Do whatever kind of alternative initialization you need to here as well.
    // Examples: Connecting to a TCP socket, Serial Port, or any other type of external device
    //           Connecting to an external database or any external file
    //           Preparing computational lookup tables or anything else to prepare for calculation
    //           Allocating memory buffers for this function
    register int lengthOfBufferArg = (int)argsArray[1].dblVal;
    mMyBuffer = new float[ lengthOfBufferArg ];

    mWasInitialized = true;
  }
```

Now that the initialization stage of your function is complete, IADS will call your function as data is required. This we will refer to as the “computation stage”. For each data value needed, IADS will call your Compute function with all the necessary input data. Your custom function will perform the processing and return a single value (the answer). This single answer will then be returned to the derived parameter, buffered to limit redundant computation, and be provided to a display (or other consumer).

The sample code in SampleFunction2.h will show you how to handle the various types of input data (float, string, etc). It will also show you how to return these different types as your custom function result. This will allow you to create custom functions to return data for almost any situation. Again, if you need more help on this subject don’t hesitate to post a question to the IADS Google group. For more advanced topics, such as returning multiple values from your custom function, please continue to the next section.

3.4.2 Returning multiple results from your custom function

One of the apparent limitations regarding the custom function technique described above is that it seems unable to return multiple values. As we learned in the previous section, each input argument that is supplied in the derived equation is sent into the Compute function, the custom code uses these input values to calculate the result, and then the single result is returned to IADS. Suppose you had a function with five input arguments, but instead of only outputting a single result, it outputs five results. This problem can be solved in a simple fairly manner.



When a function needs to output multiple answers in a single computation, we can simply output an "array" of answers. This array type output is referred to in IADS as a BLOB (binary large object). Once the array/blob is output from your custom function, it can then be returned as a blob type parameter and the individual values in the array can be extracted using another derived function called "Decom". In summary, we simply return an array of answers (however many required by the individual function), and then we can extract each value in its own unique derived parameter using the Decom function. Now, let's go more into detail about this technique.

First of all, we will need to create an array to output our five results. IADS requires that this data array be allocated using Microsoft's "SafeArray" mechanism, so we need to add a pointer of type SAFEARRAY to our class. In this case, I used "mSA" as the member variable name.

```
    ////////////////////////////////

    // Member variables of this custom function
    CComBSTR mStringOutput;
    CComBSTR mErrorString;
    SAFEARRAY* mSA;
    SAFEARRAY* mSADouble;
    bool mWasInitialized;

};

#endif //__SIMPLEFUNCTION2_H_
```

Now we will need to allocate the memory for this array. Carrying on the initialization discussion from the last section, we will perform the allocation in the Compute function within the "first time only" portion of the function. To create the array, we will simply call the SafeArrayCreateVector function with the type VT_UI1 (byte) and the number of bytes required.

```
if ( !mWasInitialized )
{
  // First Step: We need to check how many arguments were passed into our function from the user
  // Example, if the user creates a derived parameter with the function -> SampleFunction.SimpleFunction2( "Format", Va
  // The argument count would be 3
  // +

  // At this point in time, all Blobs are "fixed size", so you'll need to determine a constant size in bytes and not ch
  // In this example, we will create a single blob output array that will hold 2 float values.
  // The first 4 byte entity is an unsigned integer which will hold the size of the blob (in bytes)
  // The remaining bytes will hold our 2 floating point numbers
  // The length in bytes would be -> sizeof( unsigned __int32 ) + 2 * sizeof( float )
  const int cNumFloatsInBlob = 2;
  const int cBlobsSizeInBytes = sizeof( unsigned __int32 )/*HeaderSizeInBytes*/ + cNumFloatsInBlob * sizeof( float ) /*

  // Now, let's allocate the SafeArray to contain our blob data
  mSA = ::SafeArrayCreateVector( VT_UI1, 0, cBlobsSizeInBytes );
  if ( mSA == NULL )
  {
    // Example of returning a custom error string to Iads. See GetDescription below for further info
    mErrorString = "SimpleFunction2 failed to allocate memory for Blob output" );
    return E_OUTOFMEMORY;
  }
```

Now let's focus in on the actual "size in bytes" required by the allocation. To do this properly, we have to describe in more detail the actual structure of the blob. In a blob, the first 4 bytes of the array is a number specifying the **total** length of the blob (in bytes).

TotalSizeOfBlobInBytes	Bytes: 1 - 4
DataPortion	Bytes: 5 - N

With this fact in mind, the equation to compute the total length of allocation needed is:

BlobSizeInBytes = sizeof(unsigned __int32) + TotalSizeOfDataPortionInBytes

Or

BlobSizeInBytes = 4 + TotalSizeOfDataPortionInBytes

Or in our example using 5 floating point numbers (4 bytes per number)

BlobSizeInBytes = sizeof(unsigned __int32) + sizeof(float) * 5

At this point, you should have the return blob/array allocated, so now let's examine how to update our values in the array and return the results. First, we will need to access the array pointer within the SAFEARRAY. To do this, we simply call the SafeArrayAccessData function.

```
// Get a pointer to the safeArray data that we allocated in the "initialization stage" above
BYTE* sa;
::SafeArrayAccessData( mSA, (void**)&sa );
```

Second, let's set the blob size into the array. To do this, we simply cast the pointer returned from the SafeArrayAccessData to a type unsigned __int32* and then set the value to the total number of bytes in the blob. The total number of bytes in our example is 24 (4 bytes for the size field + 20 bytes for the 5 float values).

```
// Now access the first 4 byte integer so we can inject the Blob size (in bytes).
unsigned __int32* blobSizeInBytes = (unsigned __int32*)sa;

// Set the blob size in bytes
*blobSizeInBytes = cBlobsSizeInBytes;
```

Now we can inject our computed results into the array. To do this, we need a pointer to the type of variable we are going to store. We also need to make sure that the pointer starts at the proper location in the array (past the BlobSizeInBytes field we just set above).


```
        // Now let's inject the data into the remaining part of the Blob.
        // Get a pointer to the payload portion of the Blob (starting at 5th byte)
        float* payloadValues = (float*)(sa + sizeof(unsigned __int32));

        // Set arg2 and arg3 into the blob payload (array)
        payloadValues[0] = returnValue1;
        payloadValues[1] = returnValue2;
        payloadValues[2] = returnValue3;
        payloadValues[3] = returnValue4;
        payloadValues[4] = returnValue5;
```

Instead of setting each value individually, you may want to simply call another function to compute the results and pass in the output array pointer. You can then set the return values from within that function and also keep all of your "calculation" code separate from the "interface" code. This is a much cleaner approach overall.

```
// Likewise, if you had your own internal function to compute the results, you could simply pass in the input args
// and a reference to this array. Your function would simply write the result directly into the array
// Make sure you maintain a consistent order to your outputs, because we'll have to extract them "one by one" later
CalculateMyResults( arg2, arg3, payloadValues );
```

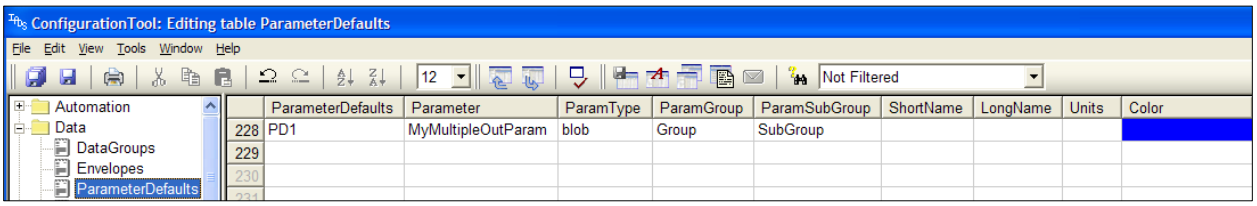
After we are complete, this is how the blob layout will appear in memory (zero based index):

24	Bytes: 0 - 3
ReturnValue1	Bytes: 4 - 7
ReturnValue2	Bytes: 8 - 11
ReturnValue3	Bytes: 12 - 15
ReturnValue4	Bytes: 16 - 19
ReturnValue5	Bytes: 20 - 23

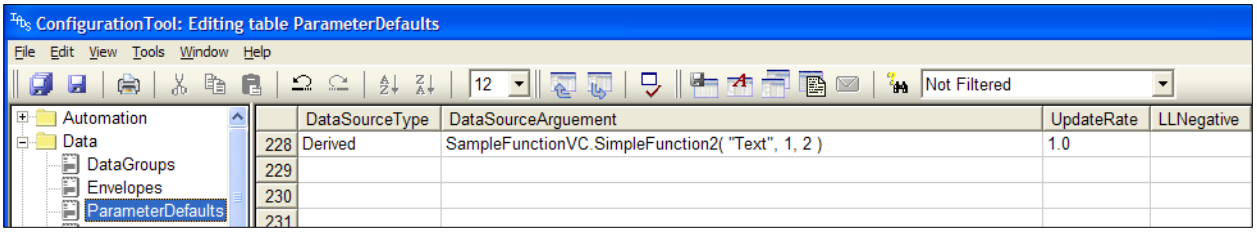
Once you have completed setting the return values into the array, it is now time to return the blob to IADS. All we need to do here is call SafeArrayUnaccessData, set the dataOut->vt to VT_ARRAY|VT_UI1 (i.e. an array of bytes), and assign the dataOut->parray variable to our SafeArray member variable (mSA). To finish the function and return the value to IADS, we simply return S_OK from the Compute function.

```
    SafeArrayUnaccessData( mSA );

    // Finally, set the blob output type and reference the safeArray we just built
    dataOut->vt = VT_ARRAY|VT_UI1;
    dataOut->parray = mSA;
}

return S_OK;
```

At this point in time, we can now test the function. To proceed, we will need to build a derived equation to call your new function. We will also need to build derived functions to extract the results from the blob. Compile your project and clean up any errors. When that is done run IADS, and open up the Configuration Tool. Open the ParameterDefaults table and add a parameter that calls your new function.



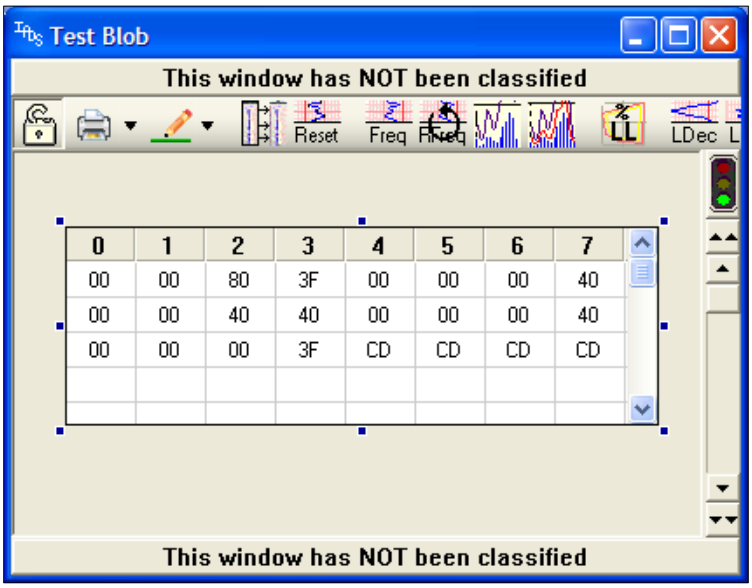
Notice in the figure above that the "ParamType" column is set to "blob". This is an essential step that you can't forget. If the ParamType is not set to "blob" for the derived parameter, you will most likely get random return results or zero while attempting to extract the 5 embedded values.



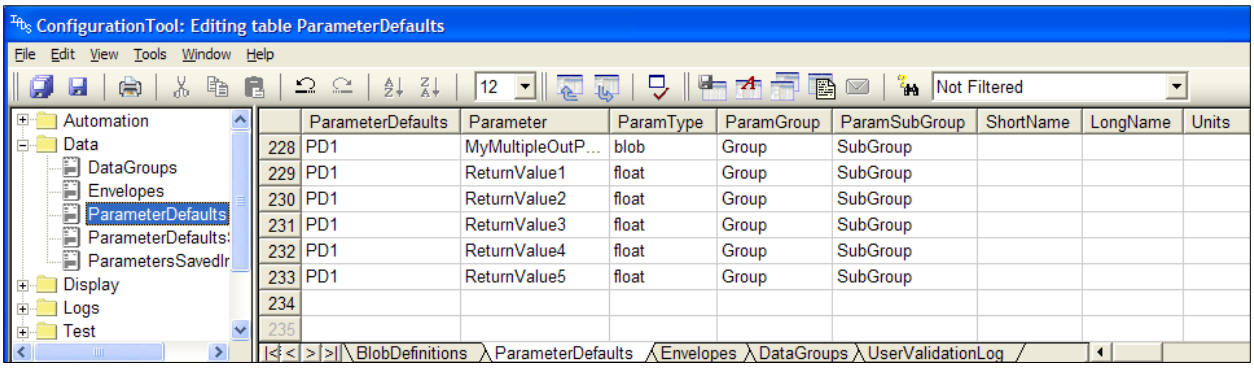
Now, scroll over to the DataSourceType column, and set it to "Derived". In the DataSourceArgument column, type an equation that calls your new function. To debug the equation, you might want to start with a set of known input values (constants). After completing the equation, save your configuration. We can now actually test the raw output of the custom function.

At this time, if you wish to see the raw output of your function you can drop your newly created derived parameter into the "IadsBusMessageDisplays.BlobViewer" display. If you Right-click on the **ActiveX Controls** tab of the Display Builder in IADS, you can add the Blob Viewer to your available displays list. Once that is complete, drag and drop the Blob Viewer display into an Analysis Window. After the display appears, drop your new derived parameter into the display. Notice that the Blob Viewer only shows the "payload" portion of your blob. The size field in the blob has been stripped by IADS. This is to be expected, so don't be alarmed.

Each 4 bytes in the display is a single 32-bit float return value. Bytes 0 .. 3 show the first return value, bytes 4..7 show the second, and so on. Note that since our blob has a total of 5 return values, there is an extra 4-byte field at the end containing all CD values. This is an artifact of the display and not actually in the blob itself. This issue should be fixed in a new version of IADS soon, so you can safely ignore it for now.



Now that we know our blob is successful, we can continue on and actually extract each individual value. When this step is complete, we can drop each individual return value into its own display, or use these return values as an input into another derived equation. After extraction, it will simply be "yet another derived parameter" and you can treat it like any other parameter in the system.



To extract the individual values from the blob, we need to create one derived equation per value. Each derived equation will use the "Decom" function to do the extraction work. Now, return to the Configuration Tool and ParameterDefaults table to add 5 more derived parameters. For each derived parameter, you must set the "ParamType" column to the type of the **extracted** value. In our case, we packed 5 floating point values (32 bits each) into the blob, so the ParamType must be set to "float". If you skip this step you will again most likely get random values or zero.

At this point we're almost done. All we need to do is to write the extraction equations using our blob parameter as the input. Scroll over to the DataSource column and set it to "Derived". In the DataSourceArguement column, add the following equation:

Decom(MyMultipleOutParam, 0, 4, 0, 31, 1, TRUE, FALSE)

The equation looks a little cryptic so, let's go over the Decom function arguments:

FuntionName: Decom
Arguments: 8
ArgumentList: InputDataParam, ByteOffset, NumBytes, StartBit, StopBit, DataTypeToReturn, Signed, ReverseBytes

DataTypeToReturn -> { Integer=0, IEEEFloat=1, 1750Float=2, CharString=3, Array=4 }
Signed -> { False=0, True=1 } or just use TRUE/FALSE
ReverseBytes -> { False=0, True=1 }

Example Usage to extract a 4 byte IEEEFloat: Decom(MyIntParameter, 0, 4, 0, 31, 1, TRUE, FALSE)

Basically, the Decom function is an all-purpose blob field extractor which can convert the bit patterns extracted into any available type in IADS. With this in mind, let's focus back on extracting our values.

Decom(MyMultipleOutParam, 0, 4, 0, 31, 1, TRUE, FALSE)

The first argument of the Decom function is the blob source parameter. In this case, we use the derived parameter that produces packed answers from our custom function. This should be the same parameter we dropped into the Blob Viewer above.

The second argument is the "starting byte offset" of the item we wish to extract within the blob. The byte offset is simply the number of bytes from the start of the payload section of the blob (remember to now ignore the 4 byte size field). Since we are defining the equation for the first return value, the starting byte offset will be zero (all the offsets are zero based in this equation).

The third argument is the number of bytes to extract. In this case, the size of the return value is 4 (4-byte floating point number). If you had chosen to pack double precision floating point values (8 bytes each), this argument would be set to 8.

The fourth argument is the "starting bit offset" of the data within bytes identified in arguments 2 and 3. In this case, we want all the bits so we simply specify bit 0. Likewise, the fifth argument is the "ending bit offset" of the data identified in arguments 2 and 3. Again, we want the full 32 bits, so we will specify 31.

The sixth argument is the actual "data type" that we want to return from the function. In this case it is an IEEE float, so we will specify 1. The seventh and eighth arguments are simply the signed flag and whether we need to reverse the bytes before data type conversion. We will specify TRUE and FALSE respectively.

Now that we understand the Decom function in general, let's simplify our task. Since all of our return values are all of the exact same type and size, we can generalize our equations as such:

Decom(MyMultipleOutParam, index*sizeof(returnValue), sizeof(returnValue), 0, sizeof(returnValue)*8-1, DataType, TRUE, FALSE)

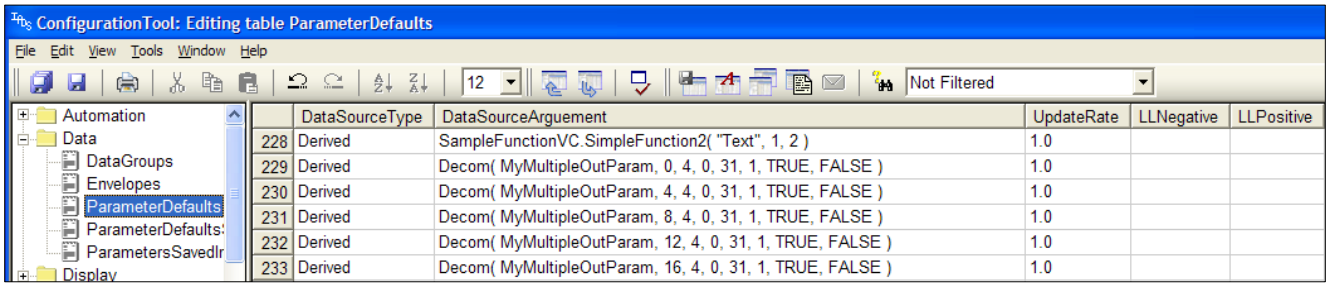
Or for our specific example

Decom(MyMultipleOutParam, index*4, 4, 0, 31, 1, TRUE, FALSE)

Where index goes from 0 to 4 (0 being our first item and 4 being our fifth item)

Using this generalization, we can easily write all of the functions needed:

ReturnValue1 => Decom(MyMultipleOutParam, 0, 4, 0, 31, 1, TRUE, FALSE)
ReturnValue2 => Decom(MyMultipleOutParam, 4, 4, 0, 31, 1, TRUE, FALSE)
ReturnValue3 => Decom(MyMultipleOutParam, 8, 4, 0, 31, 1, TRUE, FALSE)
ReturnValue4 => Decom(MyMultipleOutParam, 12, 4, 0, 31, 1, TRUE, FALSE)
ReturnValue5 => Decom(MyMultipleOutParam, 16, 4, 0, 31, 1, TRUE, FALSE)



	DataSourceType	DataSourceArguement	UpdateRate	LLNegative	LLPositive
228	Derived	SampleFunctionVC.SimpleFunction2("Text", 1, 2)	1.0		
229	Derived	Decom(MyMultipleOutParam, 0, 4, 0, 31, 1, TRUE, FALSE)	1.0		
230	Derived	Decom(MyMultipleOutParam, 4, 4, 0, 31, 1, TRUE, FALSE)	1.0		
231	Derived	Decom(MyMultipleOutParam, 8, 4, 0, 31, 1, TRUE, FALSE)	1.0		
232	Derived	Decom(MyMultipleOutParam, 12, 4, 0, 31, 1, TRUE, FALSE)	1.0		
233	Derived	Decom(MyMultipleOutParam, 16, 4, 0, 31, 1, TRUE, FALSE)	1.0		

When you are finished writing all of the extraction equations, your ParameterDefaults table should look similar to the above figure. Make sure to save your configuration upon completion.

Now, all that is left is to drop the individual parameter into displays and test. If you have any questions, please don't hesitate to post them to the IADS Google group.

4. Custom Plugins

4.1 Creating a custom export plugin using C++ VS2015

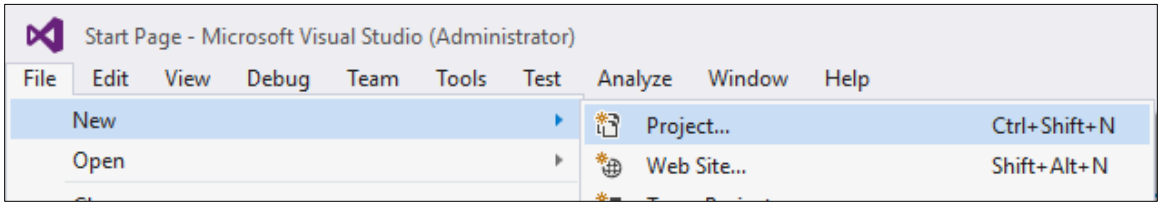
The SampleExportPluginVS2005 demonstration project is available for download from the Curtiss Wright IADS web site at the following location:
https://iads.symvionics.com/support/programming-examples/

It provides the necessary starter code for your new project. Once your plugin is complete and registered on the IADS Client machine, it will appear on the Stripchart's Data Export menu.

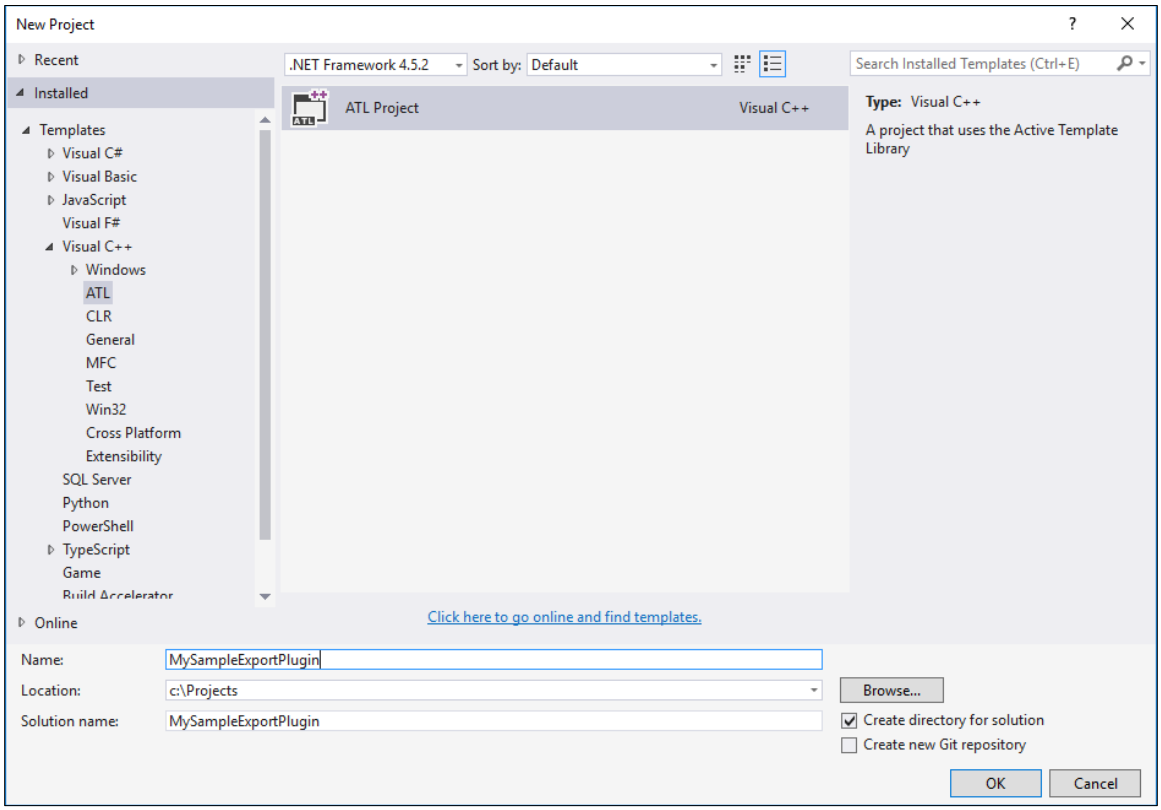
Note- on line 250 of the CSVFile.h in the exportdatagroup function `::OutputDebugString("Failed to create parameter");` you must change to `::OutputDebugString(L"Failed to create parameter");`The L is needed because of the project settings of Unicode vs multibyte characters.

It is important to build a new project rather than simply reusing the sample project developed by SYMVIOINCS. The reason for this is that each export plugin project has its own unique ID called a GUID that is placed in the Windows Registry. If more than one group uses the sample project for their own, they cannot register on the same machine. Therefore, this tutorial is presented as you the user are creating a new DLL project called MyExportPlugin.dll. Further on we will show how to copy and paste code from the sample project so that you can concentrate solely on your export code and not the interfacing between it and the outer IADS Client shell. Remember to ensure that the bitness of your project and therefore the compiled plugin is set to match the bitness of your IADS client (x64, x86).

1) Open up VS2015 and Select “File -> New->Project”



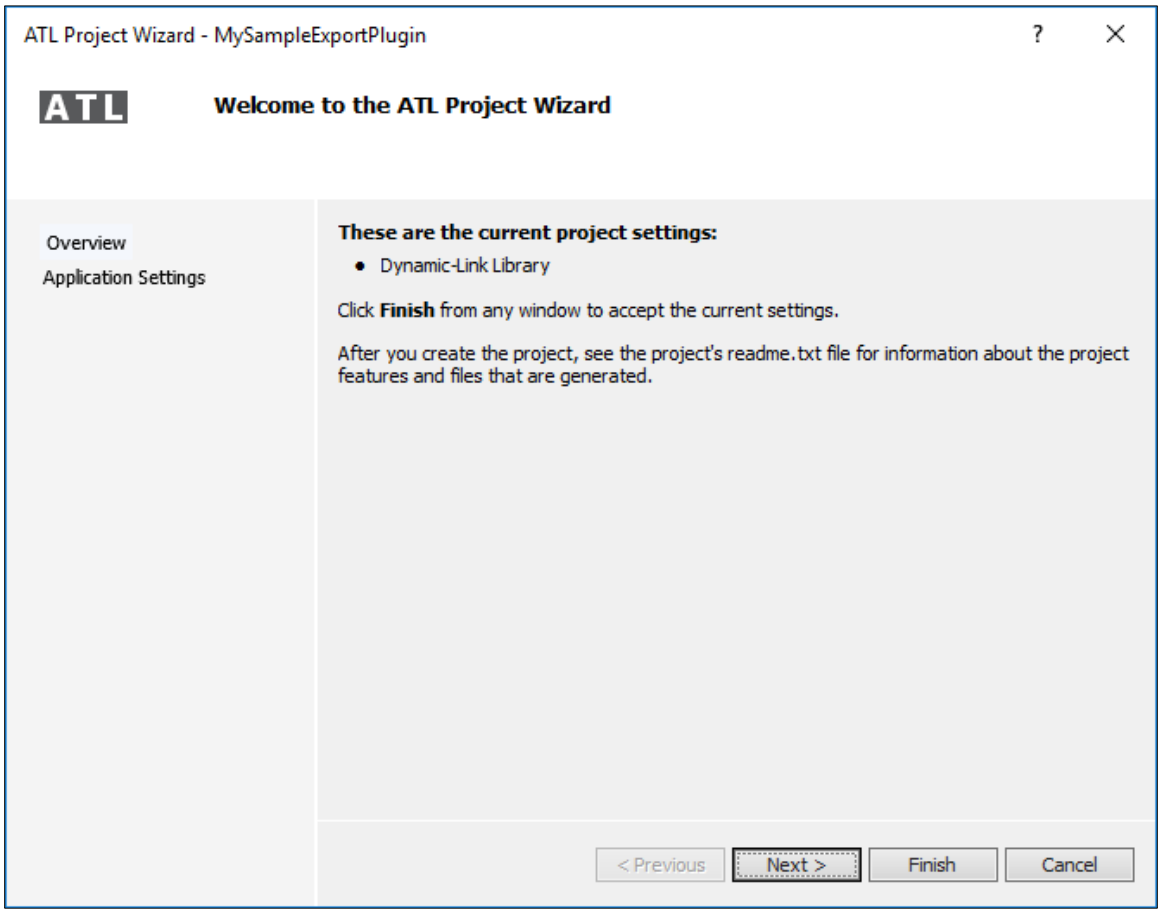
2) In the New Project dialog that appears, choose the “Visual C++ > ATL” tier and click the **ATL Project** option. At this point, please read the next step before you finish completing the dialog. There are some important considerations when choosing the proper project name.



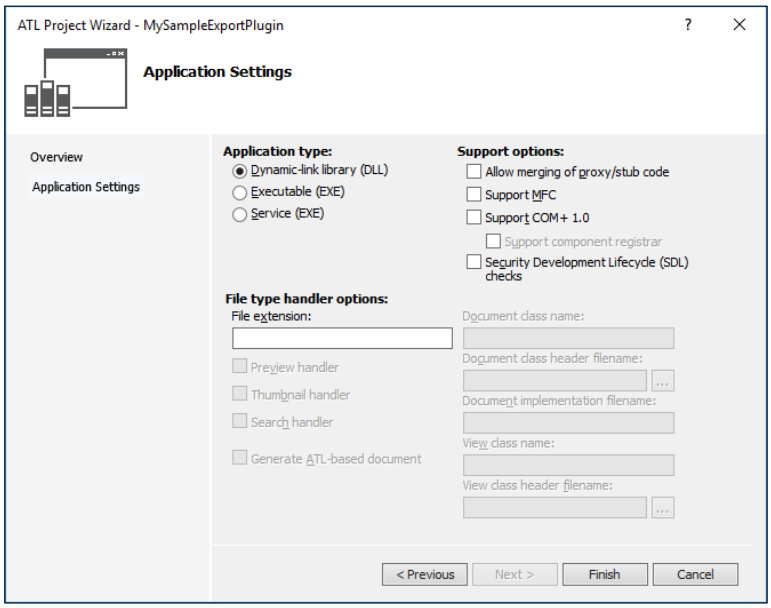
3) The project name you choose will become part of the display identifier name (aka ProgID, see step 10). When it comes time to use your plugin in IADS, users will register your DLL which automatically insert itself into the correct registry position and then be available from the IADS Stripchart’s Right-click menu. The menu display name will come from your plugin (more on this later). Plan on creating many plugins in one “project” (most common and easier to manage the code). Choose a general project name like “NasaExportPlugins” or “BA609ExportPlugins”. Think of the project name like a library name, and your plugins are the books.

   Now, in the fields at the bottom of the dialog, enter the project name, location, and the solution name.

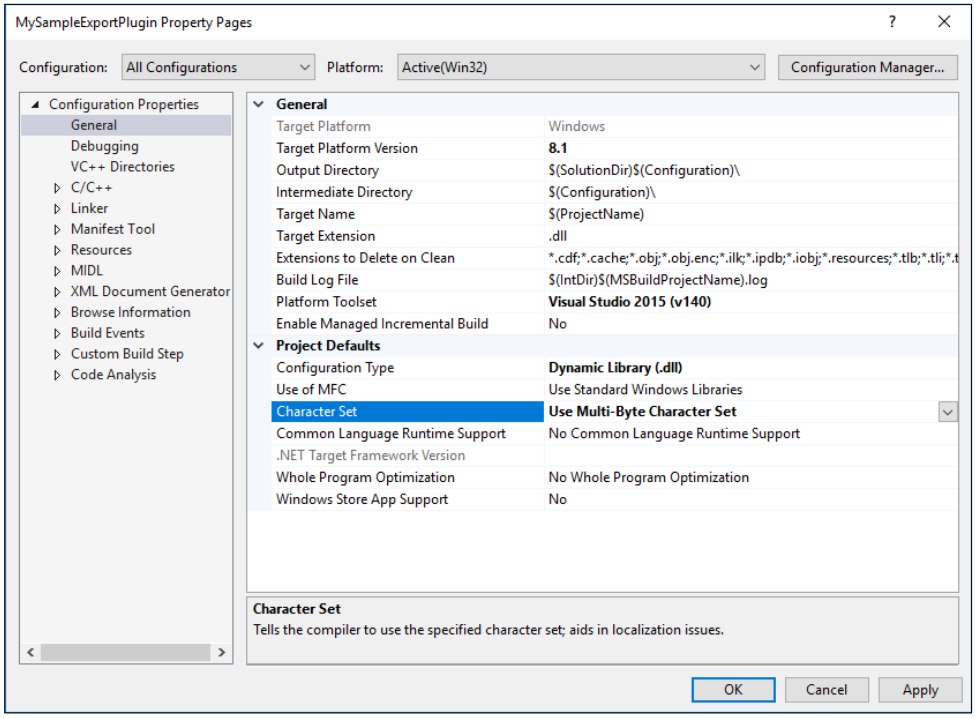
4) After pressing **OK**, the "ATL Project Wizard" dialog will appear as below.



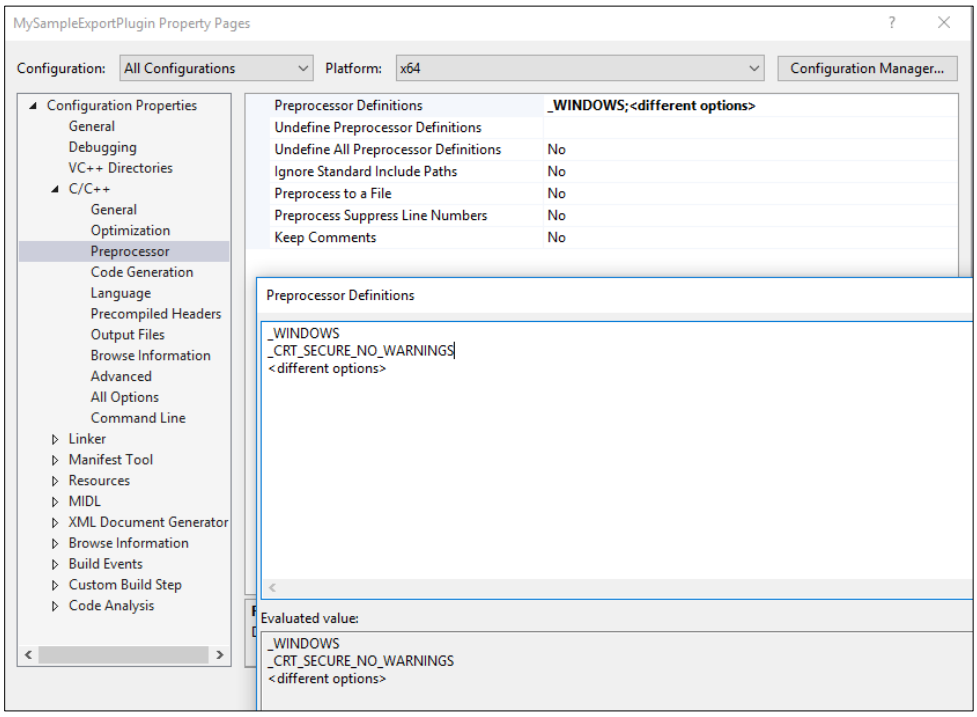
5) Click the **Next** button in the Wizard. On the new wizard page, ensure that the "Dynamic Link Library (DLL)" is checked. Every plugin that runs in IADS is of type DLL. Press the "Finish" button and the Wizard will create your project.


ATL Project Wizard - MySampleExportPlugin
Application Settings
Overview
Application Settings
Application type:
Dynamic-link library (DLL)
Executable (EXE)
Service (EXE)
Support options:
Allow merging of proxy/stub code
Support MFC
Support COM+ 1.0
Support component registrar
Security Development Lifecycle (SDL) checks
File type handler options:
File extension:
Preview handler
Thumbnail handler
Search handler
Generate ATL-based document
Document class name:
Document class header filename:
Document implementation filename:
View class name:
View class header filename:
< Previous
Next >
Finish
Cancel

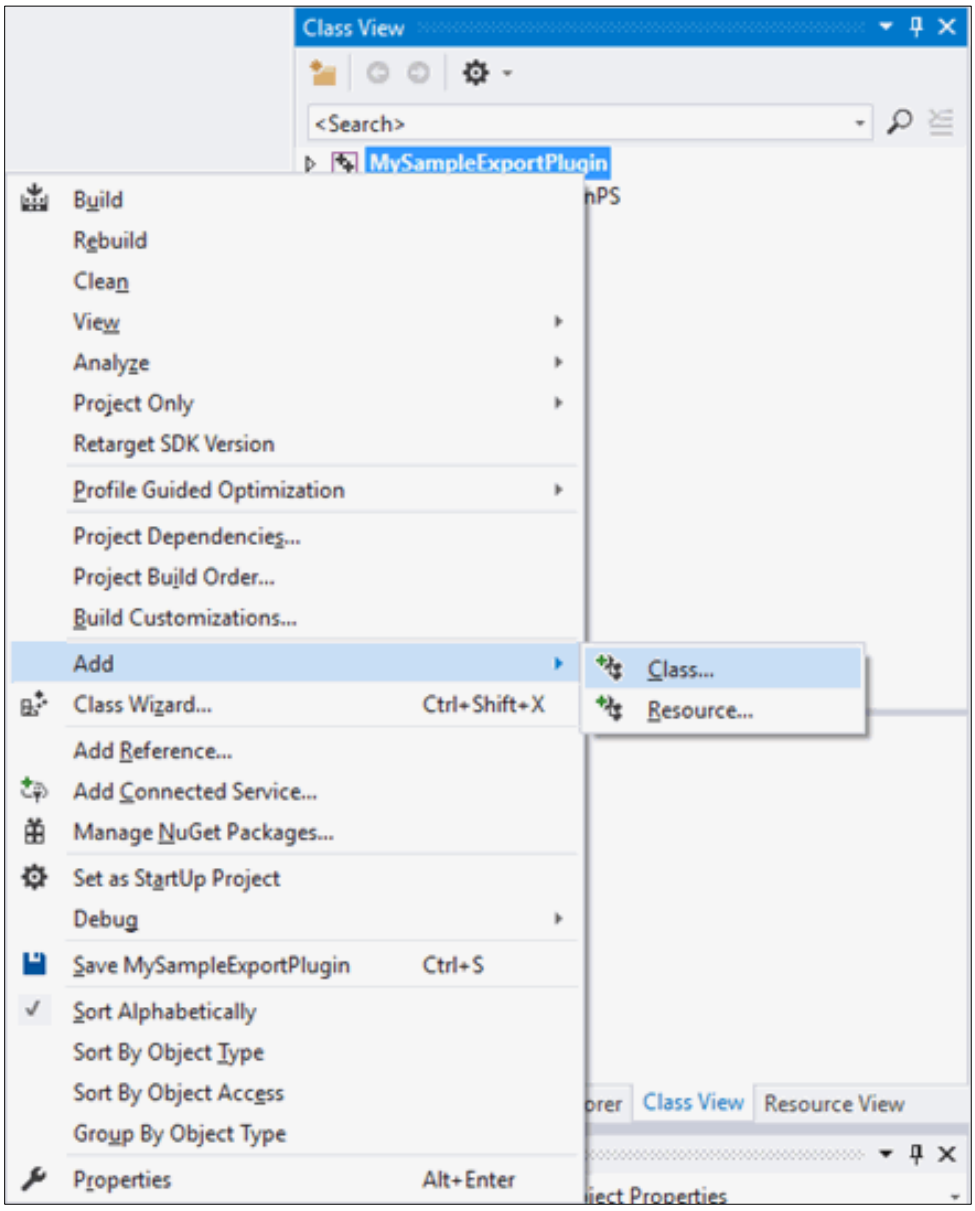
6) In the this step we will setup a couple optional project-wide settings that make it easier to work with provided IADS source code and eliminate warnings. Right-click on the project and select properties from the menu. Click on the **General** tab and change the "Character Set" option to "Use Multi-Byte Character Set".



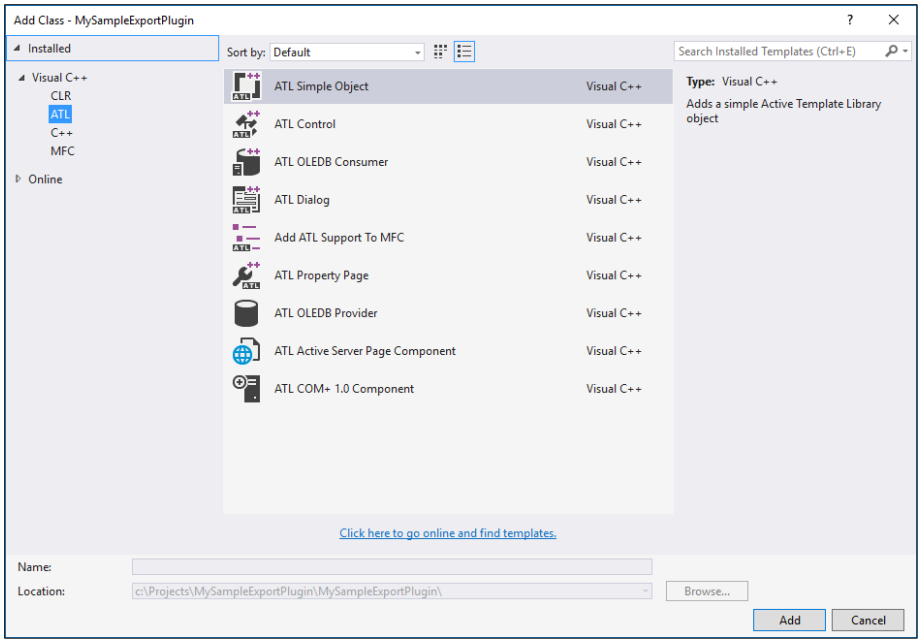
7) Again, from the Project/Properties menu select C/C++, Preprocessor and add the following definition "_CRT_SECURE_NO_WARNINGS"



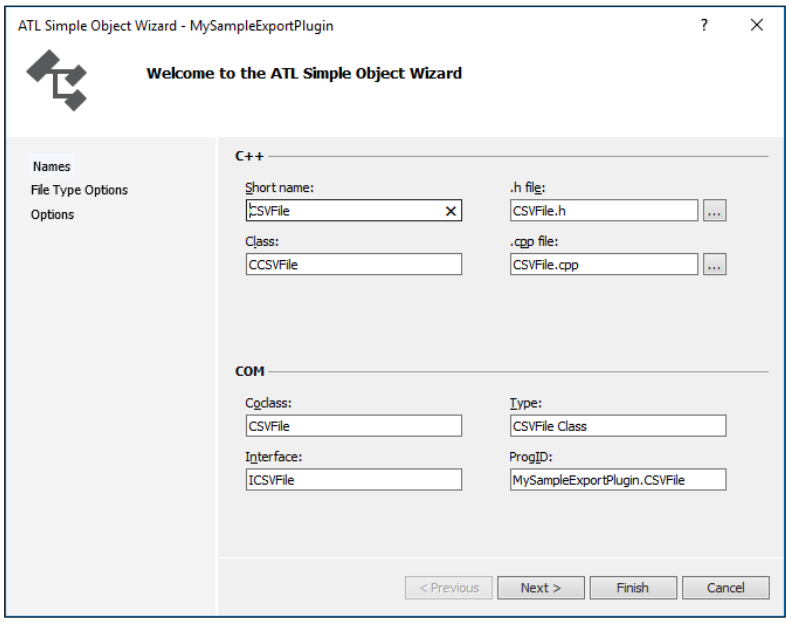
8) Now we will add our actual export object. Go to the "Class View" tab in Visual Studio's workspace and right-click on the project name. Choose **Add** > **Class**. At this point we are adding our first export plugin


Class View
<Search>
MySampleExportPlugin
Build
Rebuild
Clean
View
Analyze
Project Only
Retarget SDK Version
Profile Guided Optimization
Project Dependencies...
Project Build Order...
Build Customizations...
Add
Class Wizard... Ctrl+Shift+X
Add Reference...
Add Connected Service...
Manage NuGet Packages...
Set as StartUp Project
Debug
Save MySampleExportPlugin Ctrl+S
Sort Alphabetically
Sort By Object Type
Sort By Object Access
Group By Object Type
Properties Alt+Enter
Class...
Resource...
Class View
Resource View

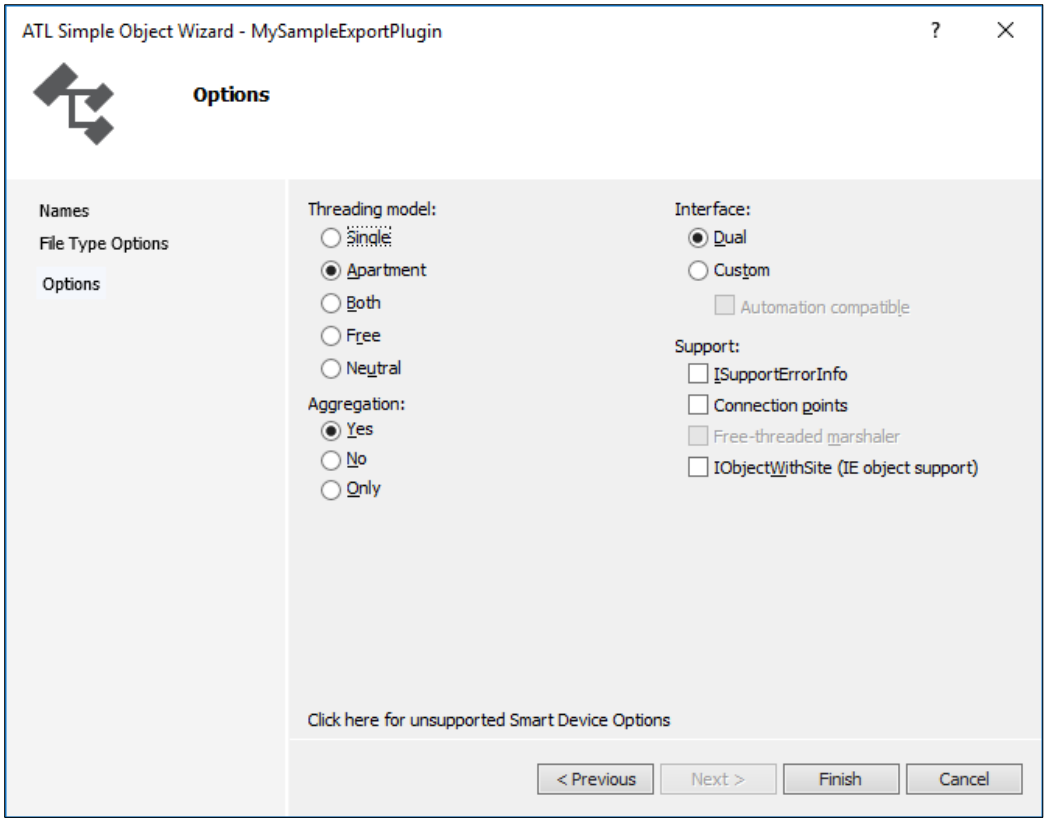
9) Upon adding a new class you will be presented with a dialog. Click the **ATL** tier and **ATL Simple Object** as shown below. When that is complete, press the **Add** button.



10) On the first tab, enter the name of your display in the "Short Name" field. The wizard will fill out the rest of the tab automatically. For this example, I used "CSVFile" as the short name. Notice that CSVFile.h and CSVFile.cpp will be created by the wizard and will be the source code files that you will edit with your own export code. The name entered will be combined with your project name and will form the final "ProgId" as shown. Press "Next" to continue. The ProgID is not populated automatically using in VS2015. You must input this manually. It should be in the form ProjectName.ClassName or in this case MySampleExportPlugin.CSVFile.



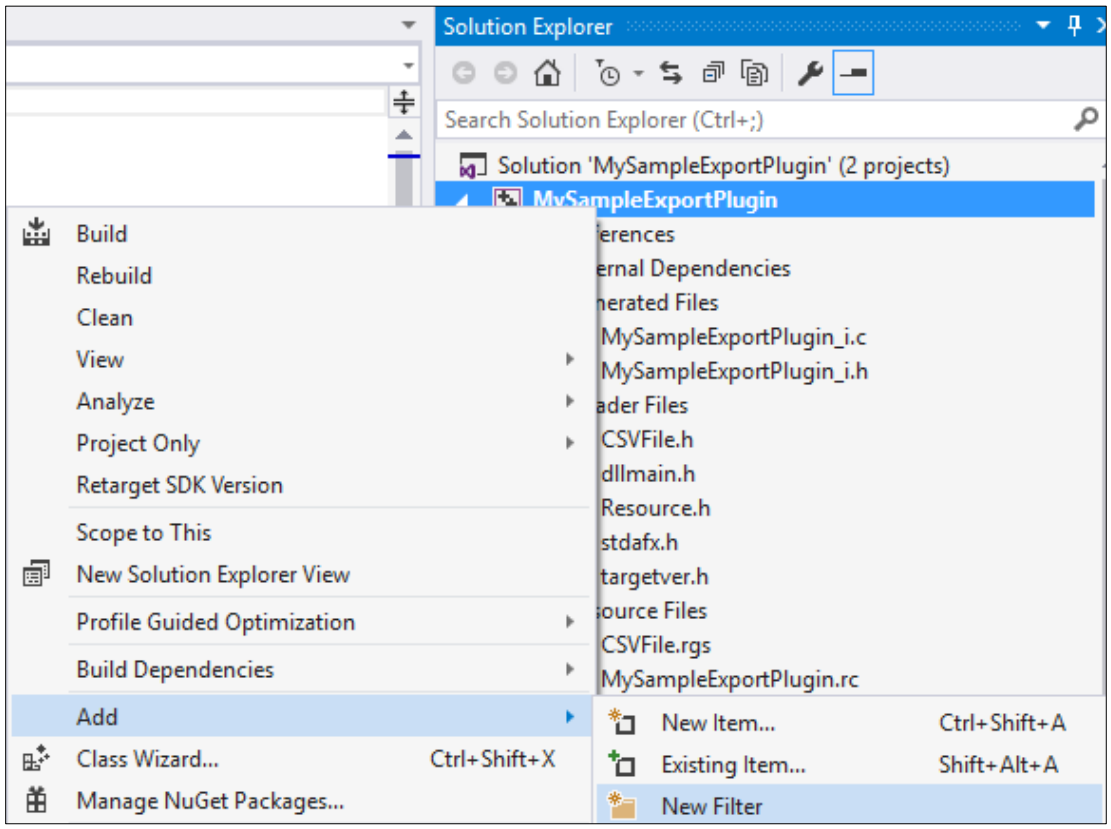
11) On the next tab (“Options”), leave everything as default (Apartment, Dual, Yes, and no other options checked). Any other dialogue boxes can be left in default.

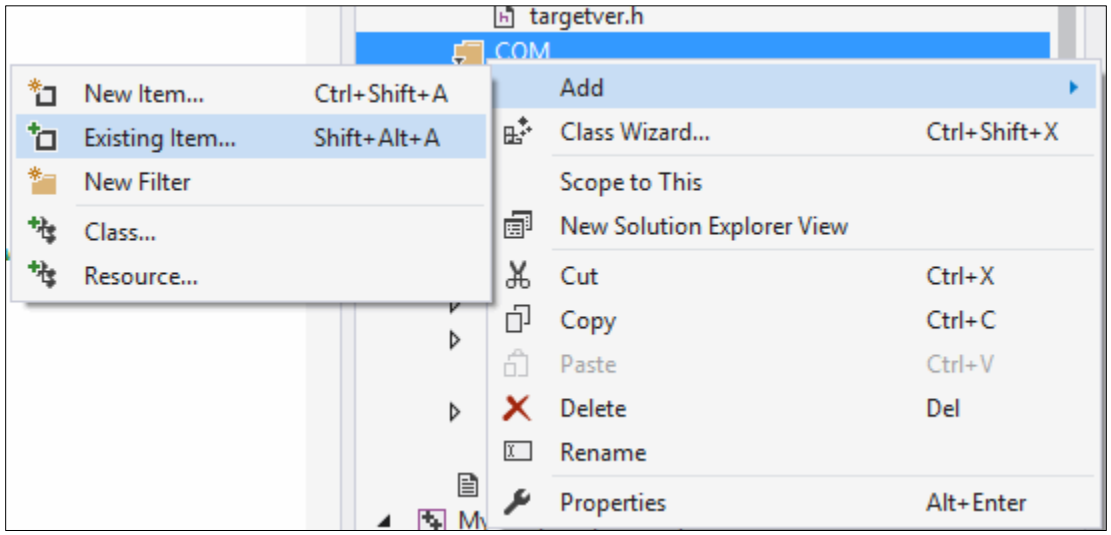


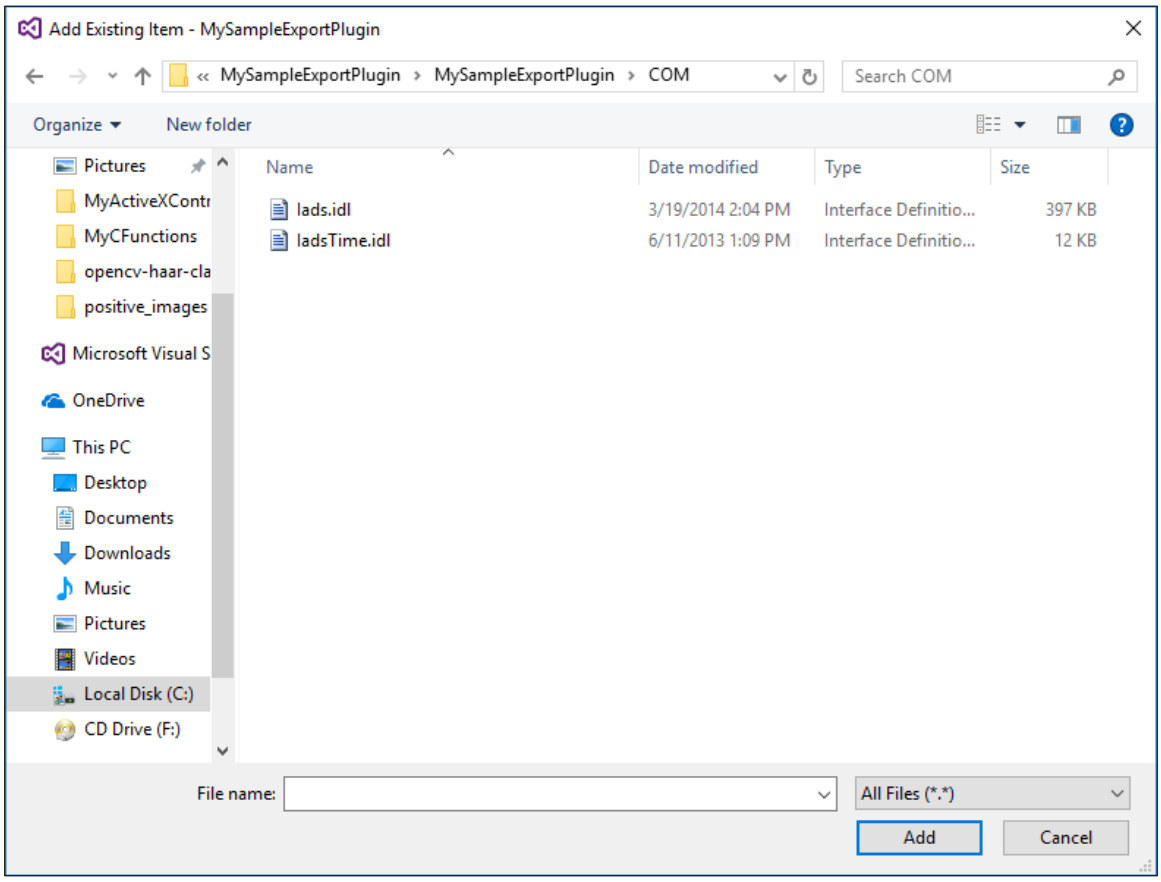
The remaining options are basically “COM speak”. More information about these options can be found in the Microsoft documentation. At this point your project can compile and register your DLL successfully, although it does not do anything yet.

4.1.1 Adding IADS Interface files

The next step is to add the two required IADS interface files, “iads.idl, and “IadsTime.idl”. These files provide the interface between your export plugin and the running IADS Client. They can be added anywhere in your project files but I recommend creating a new folder called “COM” to put them into it. Do not get the .idl files from the Curtiss Wright IADS website in the folder called iadscomhelperfunction. These .idls are old and do not have all of the functions necessary to the project. Instead use the. idls provided with the tutorial project. Then you can create a filter within your project to add the two existing idls.

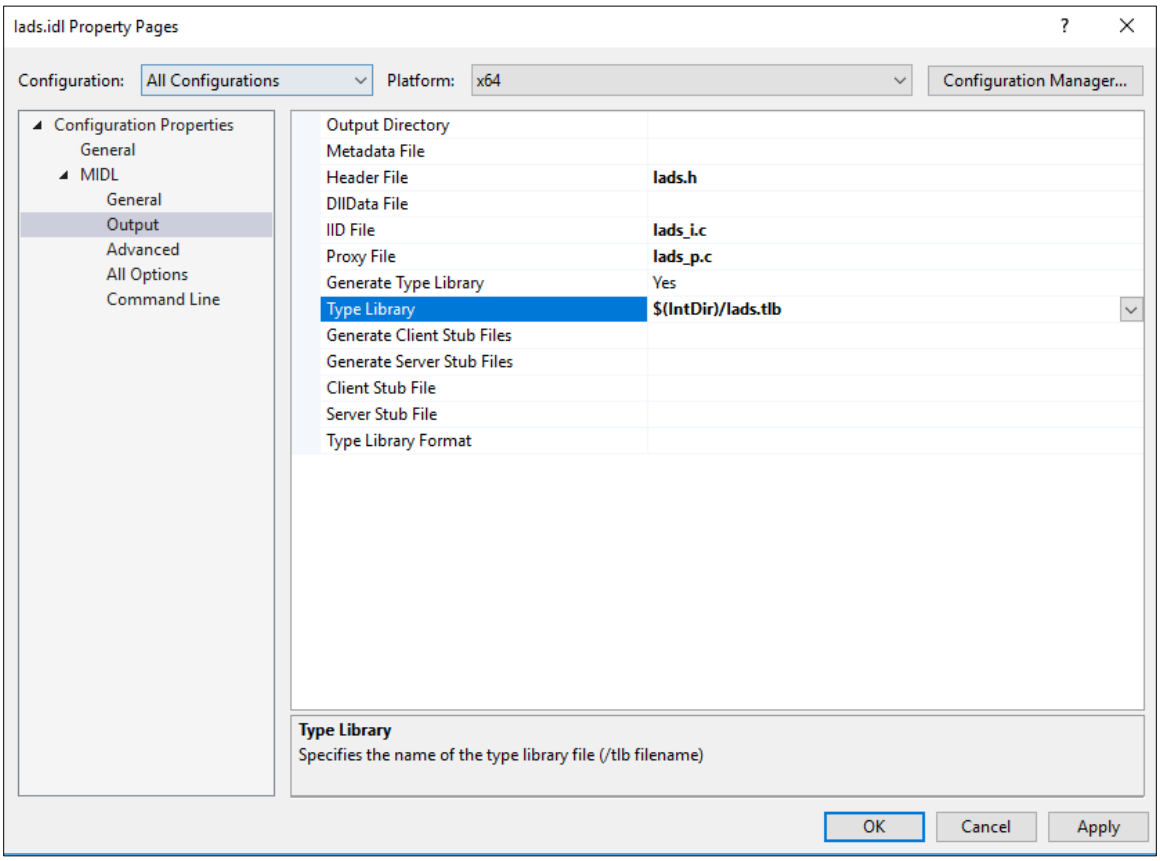




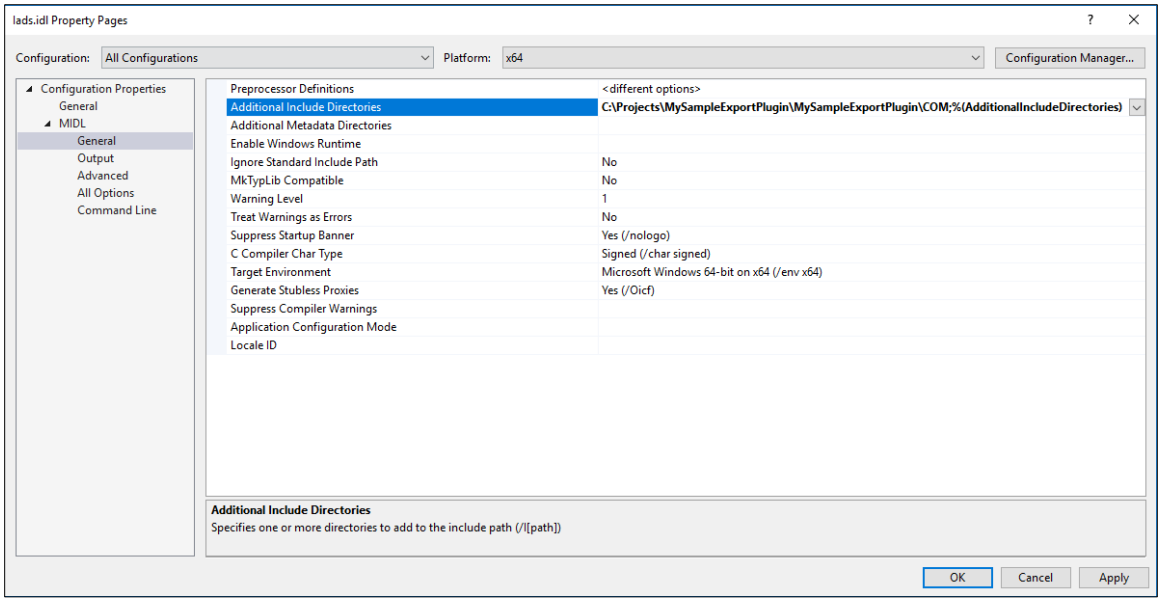


Now we need to compile the newly added IDL files to generate the output files needed in the export source code we will be editing. IDL files are compiled by a program called "MIDL" (Microsoft's IDL compiler). This is accomplished by setting up the configuration of each file in the following manner:

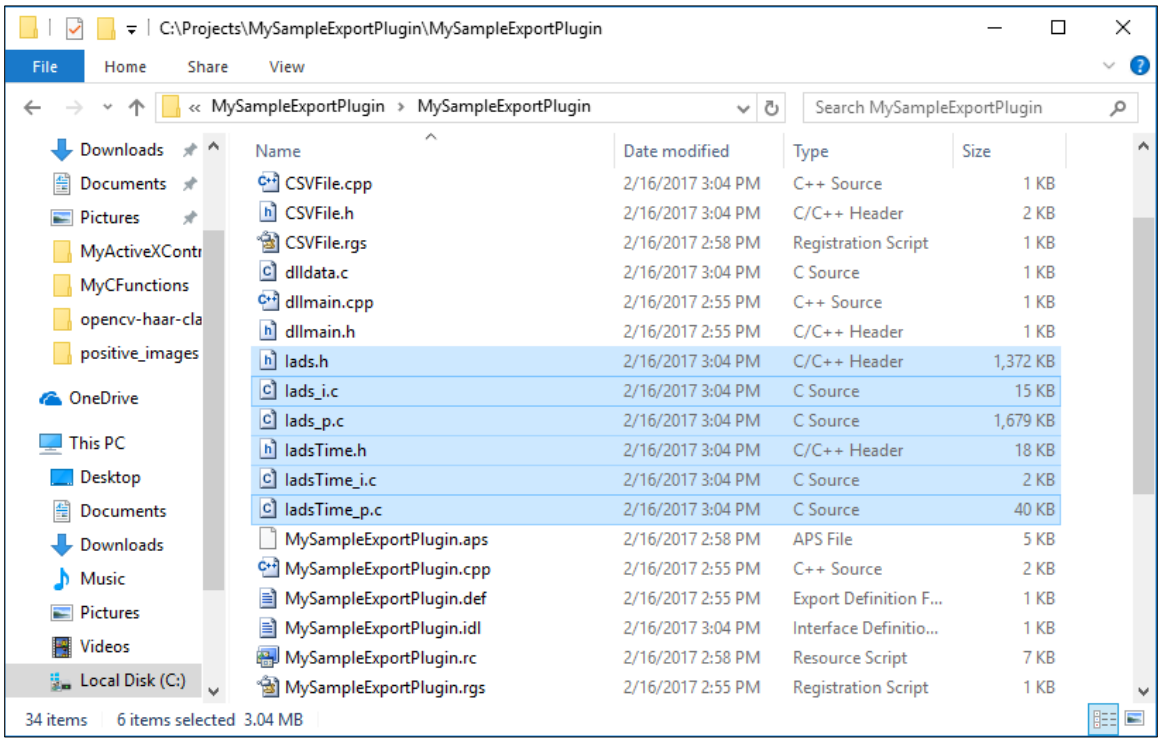
1) Right-click on the "iads.idl" file and select **Properties**.
2) Click on the **MIDL** tier in the dialog and select the **Output** option.
3) Change the default names from MyExportPlugin to the name of the IDL file as shown in the following example.
4) Repeat this step for the "IadsTime.idl file".



5) The “iads.idl” file includes the “IadsTime.idl file” so you may need to set the path for the MIDL compiler as shown:

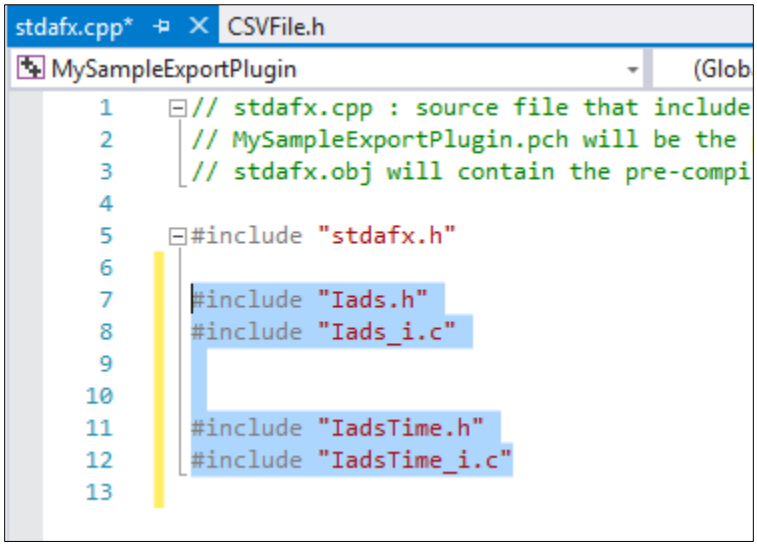


6) Make sure and compile each file individually to run MIDL and create the needed output files:



Note: iads_p.c and iadstime_p.c and are created but are not used.

7) Add the MIDL generate files to the “stdafx.c” source code file. Adding these files here will allow access to the needed global variables in your export source code. Once complete, rebuild to ensure no errors.

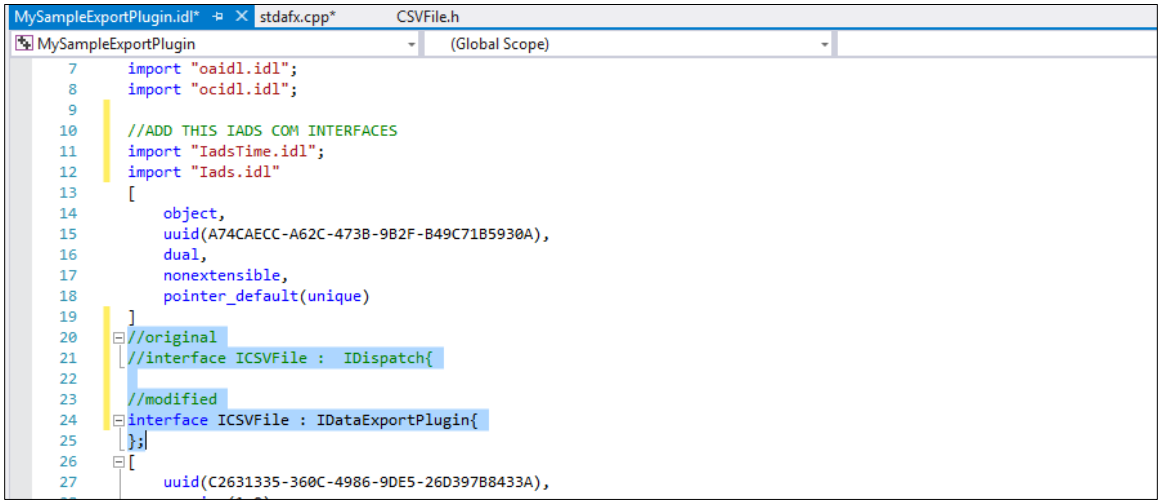


8) Add the IADS IDL interface file includes into your IDL file and change your export interface to derive from IDataExportPlugin instead of IDispatch that was generated by the wizard.

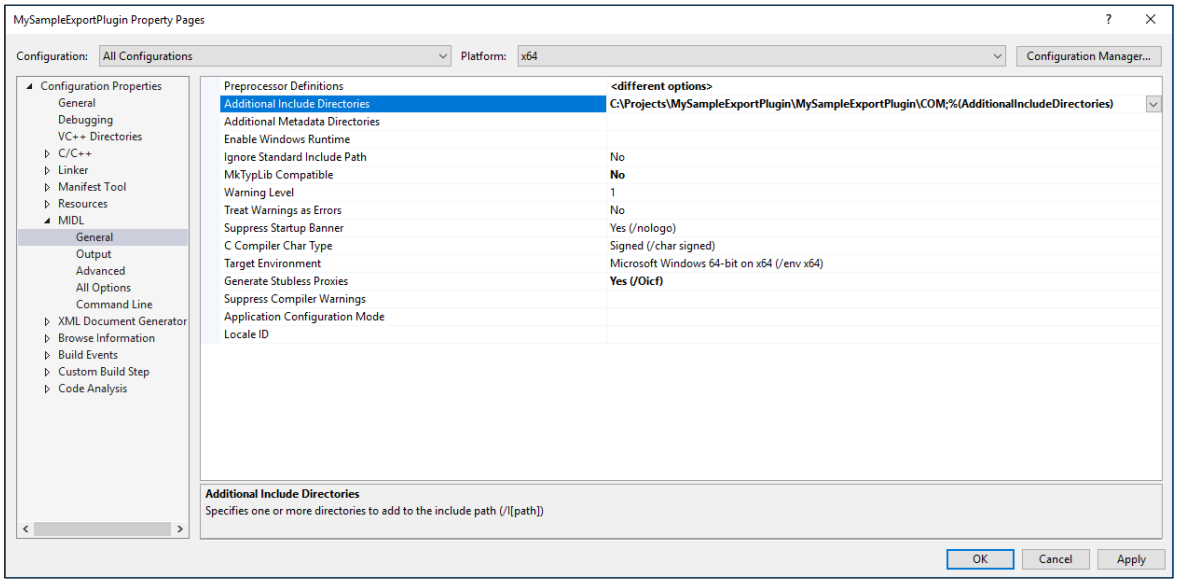
```
import "IadsTime.idl";

import "Iads.idl"
```

and

```
interface ICSVFile : IDataExportPlugin{
```



9) Just as in step 4 you may need to set the MIDL path property so that your IDL knows to the location of the "iads.idl" and IadsTime.idl files.

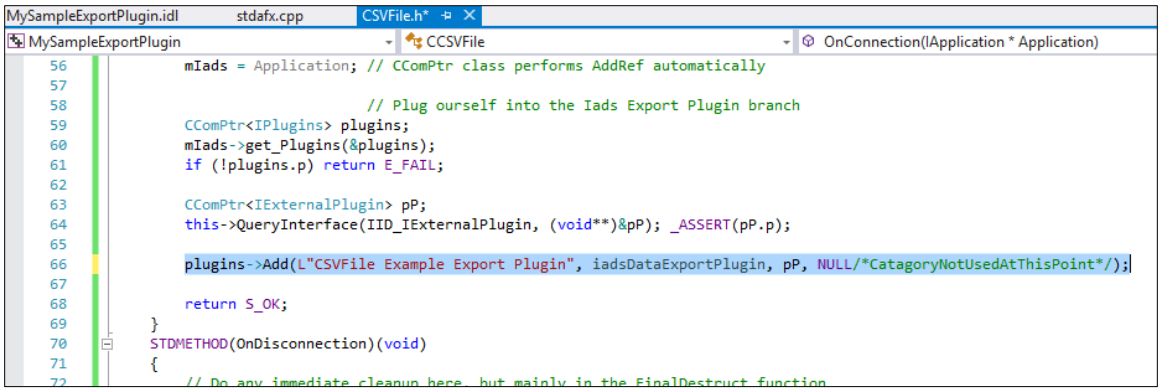


10) At this point all the necessary external files have been added to your project, however it will not compile until we add the routines that are expected by the IDataExportPlugin interface.

4.1.2 Adding IDataExportPlugin code and your export code

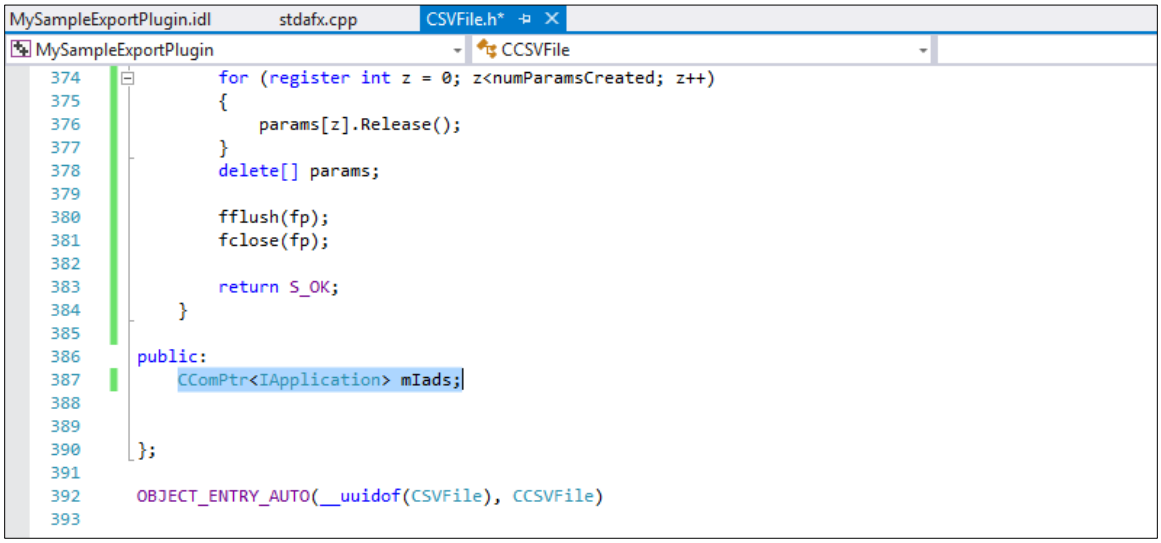
In this section we will add the required routines to that comprise the IDataExportPlugin interface, including the PerformDataExport routine which is where the entirety of your export code will reside.

1) The first step is to cut and paste the IDataExportPlugin interface code from the sample project. The source to copy is in the "CSVFiles.h" file. Cut and Paste the following routines without modification: OnConnection, OnDisconnection, PerformDataExport, ExportSelectedDisplay, ExportDataForSingleParameter, and ExportDataGroup into the CSVFile.h within your project. These routines already include most of the source code you will need to access the running IADS Client for necessary parameter information. All you will need to do is replace the actual code that exports to a CSV File with your own output file type, such as HDF.

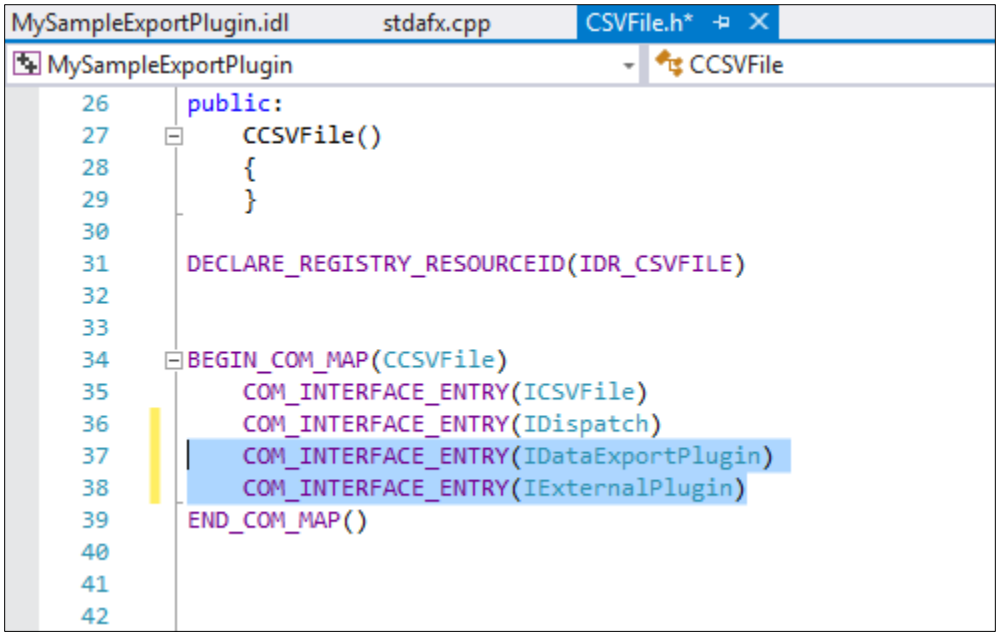


2) Modify the string argument in the plugins-add call. This will become the actual display name that appears in Iads on the Right-click menu.

3) Next add the "mIads" member variable to the public section of your class in CSVFile.h.



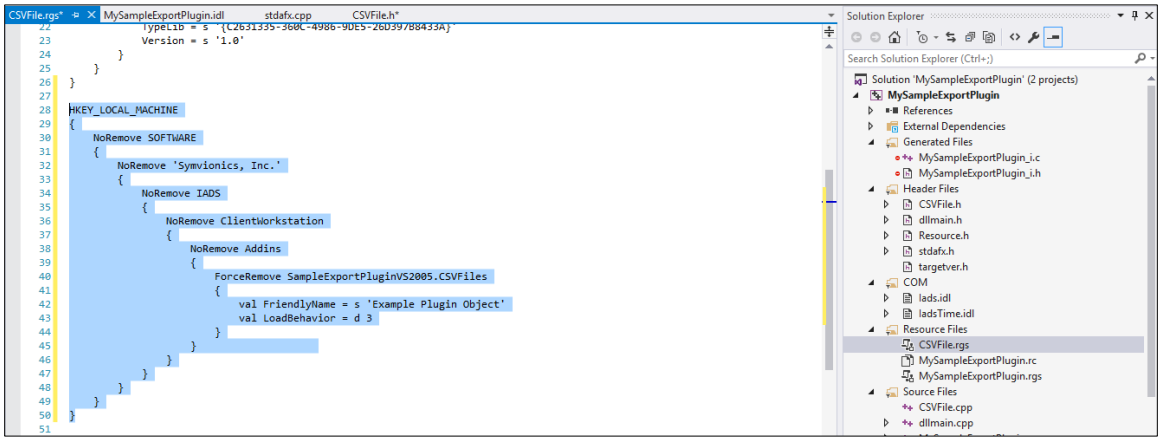
4) Finally add the IDataExportPlugin and the IExternalPlugin COM_INTERFACE_ENTRY entries to the CSVFile.h header file's COM Map as shown



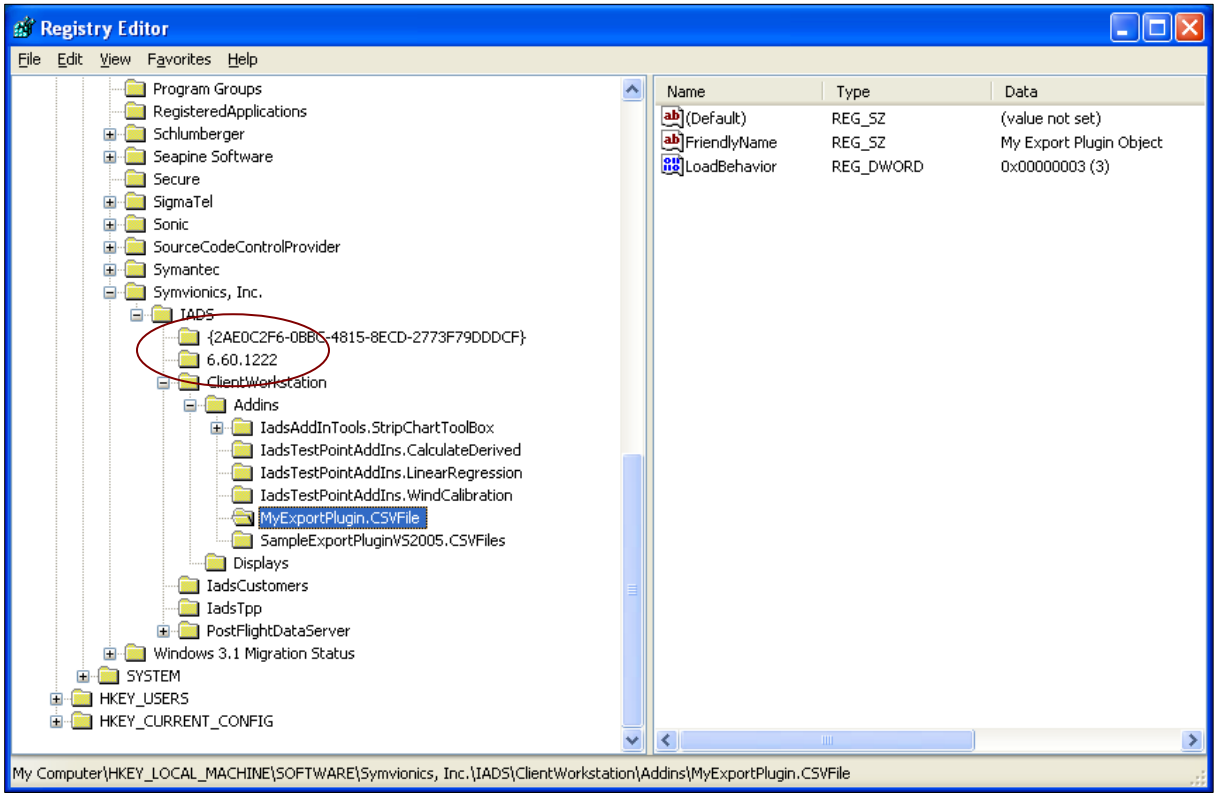
5) At this point your project should compile without errors or warnings.

4.1.3 Make your DLL self-register for use in IADS

1) In order for your export plugin to be shown in the Stripchart's Data Export menu, registration code must be added to your object's registration script. The easiest way to do this is to cut and paste the code from the sample project's CSVFile.rgs into the CSVFile.rgs file from your solution and change the name of the ProgId to the new project's ProgId which should have stayed constant throughout the program, as shown here:

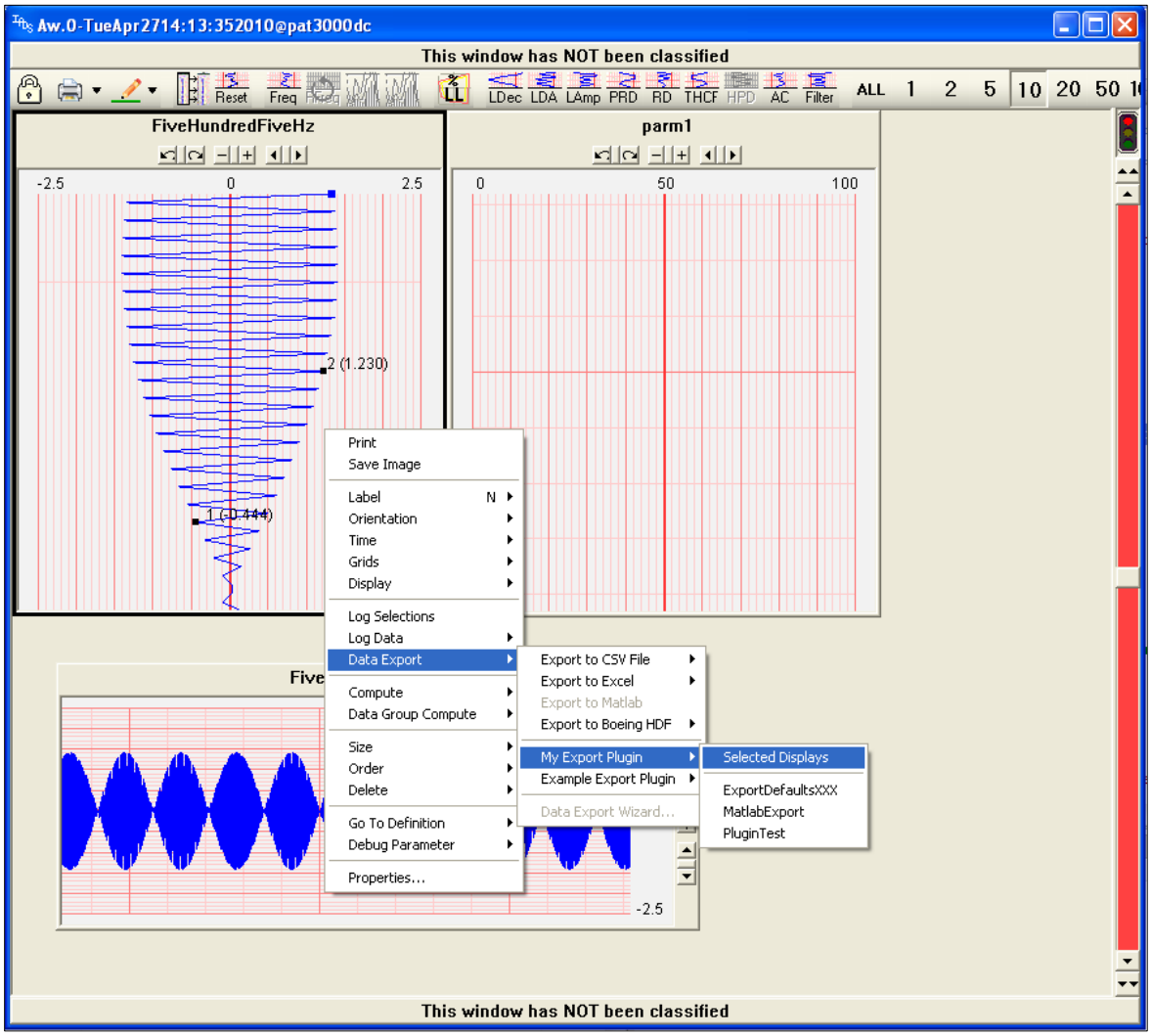


2) After building the solution, you can use regedit to verify that the registration code worked properly by putting the ProgId into the HKEY_LOCAL_MACHINE/Symvionics, Inc./IADS/ClientWorkstation/Addins registry hive.



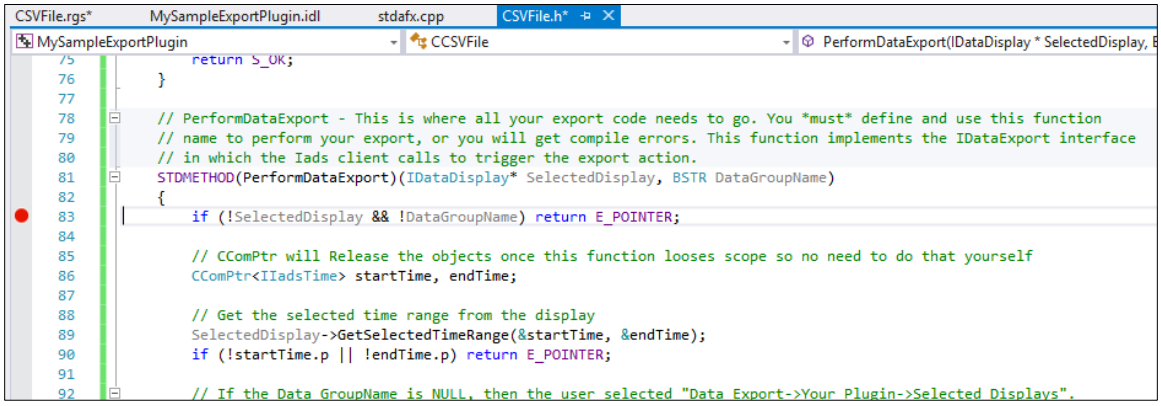
3) Finally, you can run IADS (requires version 7 or greater) and verify that MyExportPlugin was added to the Stripchart Data Export menu as shown here.

   Before you can run Iads in debug mode to verify your plugin was added to the program please complete the next section, 5.1.4.

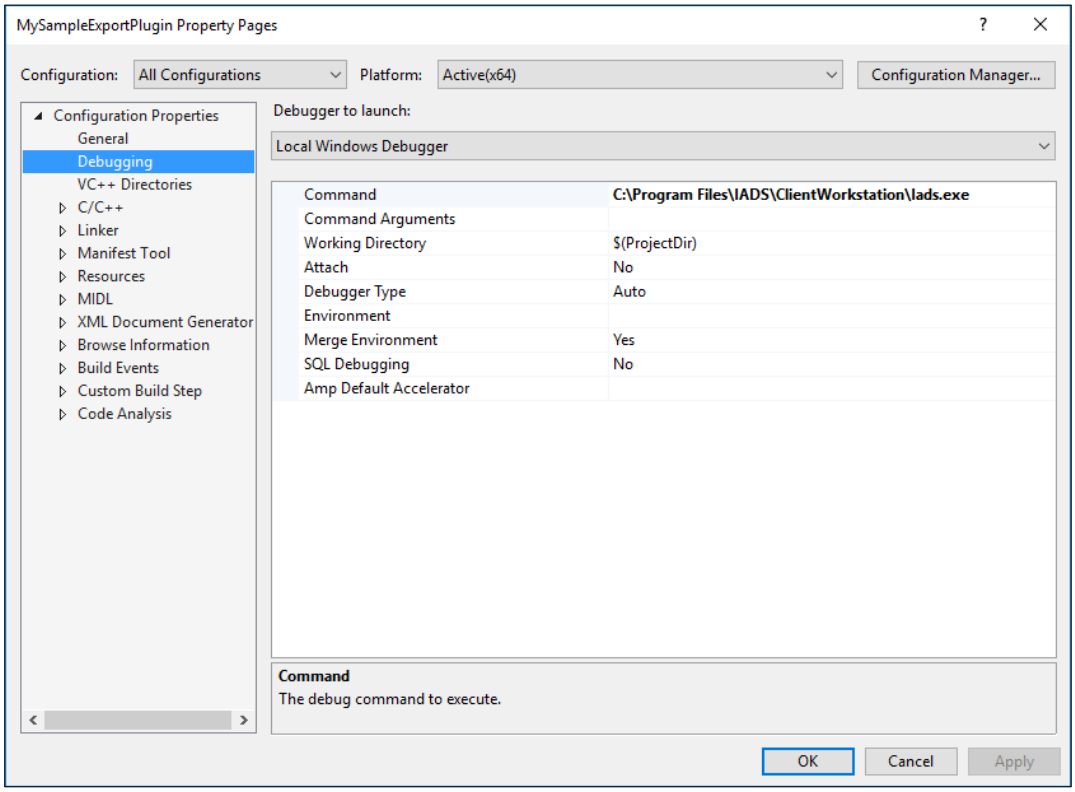


4.1.4 Debugging your new plugin in IADS

1) Place a break point in your “PerformDataExport” method for testing. Remember this function is within CSVFile.h.



In Visual Studio, select **Project** > **Properties** drop down menu. In the “Debugging” tier, pick “Iads.exe” as your “Command”. It is in your “C:\Program Files\Iads\ClientWorkstation” directory. Build your project and click the “Go” command. IADS will start. If you have built a x64 plugin you’re the iads client will be in C:\Program Files\IADS\ClientWorkstation\iads.exe. If you compiled a x86 version you will have to use the 32 bit Iads client which is located within C:\Program Files (x86)\IADS\ClientWorkstation\iads.exe.



2) Right-click on the Stripchart and select the Properties option and then the Data Export option. Select “My Export Plugin” to hit your breakpoint in your code in the “PerformDataExport routine.

5. Application Programming Interfaces

5.1 IADS Configuration File API

The IADS Configuration data base is an ASCII file that is used by all the IADS software components for setup, communication, and archiving of both system and user generated meta-data. It is used and manipulated by both the IADS Server (CDS) and the IADS Client display workstation (the Client).

This document provides the interface specification necessary in order to manipulate the Configuration file using the IADS Configuration File Application Programmers Interface (API). This API was developed to allow outside agencies a programmatic interface in order to create and manipulate the IADS Configuration database file in a non-real time environment.

This document will cover the contents and structure of the IADS Configuration database file. In addition, a general discussion of the API’s Parameter Defaults table and the Event Marker table along with a discussion of the general SQL query interface will be included.

IADS provides a Component Object Model (COM) programming Application Programmers Interface (API) Dynamic Link Library (DLL) interface to manipulate an IADS Configuration file. This API provides methods using two different techniques; a Collection based interface for frequently used tables and a general-purpose SQL interface for complete access to all internal tables (See Appendix A for list of tables).

The COM DLL and test project are available for download on the Curtiss Wright IADS website at: https://iads.symvionics.com/support/programming-examples/

5.1.1 Configuration Interface

The Configuration interface provides all the methods needed to manipulate the Configuration file. This is the API starting-point for all Collection interfaces that are described in section 5.1.2. Following is a table of the methods available:

Method	*Return*	*Argument*	*Description*
Open	In	BSTR	Open an existing configuration file with create backup option
Create	In	BSTR	Create a new configuration file with open option
	In	VARIANT_BOOL	Set to true to Open the file after creation
Save	None	VOID	Commit and save changes from open configuration file
Close	In	VARIANT_BOOL	Close configuration file with save option
VersionFromFile	In	BSTR	Configuration file name and path
	Out	BSTR*	Version number
Version	Out	int*	return version of currently opened configuration file, (-1) indicates no version has been set yet
Version	In	int	put version of configuration file to create
OpenMessageLog	In	BSTR	Log error messages to a file

	In	VARIANT_BOOL	True to delete existing log
ParameterSets	Out	IParameterSets**	Get the ParameterSets collection
ParameterDefaults	Out	IParameterDefaults**	Get the ParameterDefaults collection
Events	Out	IEvents**	Get the Events collection
Thresholds	Out	IThresholds**	Get the Thresholds collection
Testpoints	Out	ITestpoints**	Get the Testpoints collection
Selections	Out	ISelections**	Get the Selections collection
PlannedTestPoints	Out	IPlannedTestPoints**	Get the Planned Test Points collection
Query	In	BSTR	Query interface. Keywords are: Select, Update, Delete, and Create. Returns an array of BSTR results if applicable
	Out	VARIANT*	Array of query results

5.1.2 Collection Interfaces

Seven collection interfaces are provided as a layer on top of the SQL engine due to their frequent use. They are the ParameterSets, ParameterDefaults, EventMarkers, Selections, Thresholds, Testpoints and PlannedTestPoints.

Appendix B has the value for IADS defined data types and enumerations. Following is a description of each of the collection interfaces

ParameterSets Collection

IADS allows multiple Parameter Sets to be defined and enabled for processing. The ParameterSets collection is used to return ParameterSet Item which in turn contains a ParameterDefaults collection (see the "ParameterDefaults Collection" section below) . Following are the functions in the ParameterSets collection:

Method	*Return*	*Arguments*	*Description*
Count	Out	long*	Gets the number of IParameterSet Items within the collection.
Add	In	BSTR	Add a new IParameterSet Item to this collection. Add a ParameterSet by name
	In	BSTR	The group name of the IParameterSet Item to add to this collection.
	In	VARIANT_BOOL	The active flag of the IParameterSet Item to add to this
	Out	IParameterSet**	Return value reference to the newly added IParameterSet Item.
Remove	In	VARIANT	Remove IParameterSet object by index or name from this collection
RemoveAll	None	VOID	Remove all IParameterSet Items from this collection.
Item	In	VARIANT	Return an IParameterSet Item by name (string) or index number (0..Count). IndexOrName
	Out	IParameterSet**	Return value, reference to the specified IParameterSet Item. Returns a Set object by name (string) or index number (0..Count)
SaveTable	*None*	*VOID*	*Save changes to this Collection*

ParameterSet Item

The Parameter Set item contains the Set name, the Group name, and an IsActive boolean if the set is active in IADS. It also contains the ParameterDefaults collection for further processing of Parameter information.

Method	*Return*	*Arguments*	*Description*
SetName	Out	BSTR*	Get the Parameter Set name
SetName	In	BSTR	Set the Parameter Set Name
Group	Out	BSTR*	Get Parameter Group name
Group	In	BSTR	Set Parameter Group name
IsActive	Out	VARIANT_BOOL*	Get Parameter Set Active
IsActive	In	VARIANT_BOOL	Set Parameter Set is active
ParameterDefaults	Out	IParameterDefaults*	Get Parameter Defaults collection

ParameterDefaults Collection

The ParameterDefaults Collection holds all the available Parameter Items. It returns ParameterDefault Item which contains all the available properties for each parameter.

Method	*Return*	*Arguments*	*Description*
Count	Out	long*	Gets the number of IParameterDefault items within the collection.
Add	In	BSTR	Add a new IParameterDefault Item to this collection, this is the Parameter Name
	In	IadsDataType	This is the inherent type of the parameter
	In	Double	This is the sample rate of the parameter
	Out	IParameterDefault**	Return value, reference to the newly added IParameterDefault Item.
Remove	In	VARIANT	IParameterDefault Item by index or name from this collection
RemoveAll	None	VOID	Remove all IParameterDefault items from this collection.
Item	In	VARIANT	Return an IParameterdefault Item by name (string) or index number
SetName	In	BSTR*	Return the IParameterSet name that this parameter belongs to
SaveTable	*None*	*VOID*	*Save changes to this collection*

ParameterDefault Item

The Parameter item is returned from the ParameterDefaults collection. Following are the Methods in the ParameterDefaults collection along with their return value, or input argument and description

Method	*Return*	*Argument Type*	*Description*
Name	Out	BSTR*	Get parameter name
Name	In	BSTR	Set parameter name
DataType	Out	IadsDataType*	Get parameter data type
DataType	In	IadsDataType	Set parameter data type
Group	Out	BSTR*	Get parameter group name
Group	In	BSTR	Set parameter group name
SubGroup	out	BSTR*	Get parameter subgroup name
SubGroup	In	BSTR	Set parameter subgroup name
ShortName	Out	BSTR*	Get parameter short name
ShortName	In	BSTR	Set parameter short name
LongName	Out	SBSTR*	Get parameter long name
LongName	In	BSTR	Set parameter long name

Units	Out	BSTR*	Get parameter units name
Units	In	BSTR	Set parameter units name
Color	Out	OLE_COLOR*	Get color value
Color	In	OLE_COLOR	Set color value
Width	Out	int*	Get pen width
Width	In	int	Set pen width
DataSourceType	Out	IadsDataSourceType*	Get the data source type
DataSourceType	In	IadsDataSourceType	Set data source type
DataSourceArgument	Out	BSTR*	Get the data source argument
DataSourceArgument	In	BSTR	Set data source argument
SampleRate	out	double*	Get the data sample rate
SampleRate	In	double	Set data sample rate
LoadLimitNegative	Out	double*	Get negative design load limit
LoadLimitNegative	In	double	Set negative design load limit
LoadLimitPositive	Out	double*	Get positive design load limit
LoadLimitPositive	In	double	Set positive design load limit
TimeScaleRangeMin	Out	double*	Get time scale range min
TimeScaleRangeMin	In	double	Set time scale range min
TimeScaleRangeMax	Out	double*	Get time scale range max
TimeScaleRangeMax	In	double	Set time scale range max
TimeCaleAuto	in	double	Not currently used in IADS
TimeScaleAuto	Out	IadsOnOrOff*	Get time scale auto (Not currently used by IADS)
TimeScaleAuto	In	IadsOnOrOff	Set time scale auto (not currently used by IADS)
FreqScaleRangeMin	Out	double*	Get frequency scale range Max
FreqScaleRangeMin	In	double	Set frequency scale range Min
FreqScaleRangeMax	Out	double*	Get frequency scale range max
FreqScaleRangeMax	In	double	Set frequency scale range max
FreqScaleAuto	In	double	Not currently used in IADS
FreqScaleAuto	Out	IadsOnOrOff*	Get frequency scale auto (not currently used by IADS)
FreqScaleAuto	In	IadsOnOrOff	Set frequency scale auto (not currently used by IADS)
WarningThreshRangeMin	Out	double*	Get warning threshold min range
WarningThreshRangeMin	In	double	Set warning threshold range min
WarningThreshRangeMax	Out	double*	Get warning thresh range max
WarningThreshRangeMax	In	double	Set Get warning thresh range max
WarningThreshColor	Out	OLE_COLOR*	Get Warning Threshold color
WarningThreshColor	In	OLE_COLOR	Set warning threshold color
WarningThreshLabel	Out	BSTR*	Get warning threshold label
WarningThreshLabel	In	BSTR	Set warning threshold label
WarningThreshLineWidth	Out	int*	Get warning threshold line width
WarningThreshLineWidth	In	int	Set warning threshold line width
AlarmThreshRangeMin	Out	double*	Get alarm threshold range min
AlarmThreshRangeMin	In	double	Set alarm threshold range min
AlarmThreshRangeMax	Out	double*	Get alarm threshold range max
AlarmThreshRangeMax	In	double	Set alarm threshold range max
AlarmThreshColor	Out	OLE_COLOR*	Get alarm threshold color
AlarmThreshColor	In	OLE_COLOR	Set alarm threshold color
AlarmThreshLabel Out	Out	BSTR*	Get alarm threshold label
AlarmThreshLabel	In	BSTR	Set alarm threshold label
AlarmThreshLineWidth	Out	int*	Get alarm threshold line width

AlarmThreshLineWidth	In	int	Set alarm threshold line width
FilterActive	Out	IadsYesOrNo*	Get filter active
FilterActive	In	IadsYesOrNo	Set filter active
FilterAlgorithm	Out	IadsFilterAlgorithm*	Get filter algorithm
FilterAlgorithm	In	IadsFilterAlgorithm	Set filter algorithm
FilterPassType	Out	IadsFilterPassType*	Get filter pass type
FilterPassType	In	IadsFilterPassType	Set filter pass type
FilterLowCutoff	Out	double*	Get filter low cutoff
FilterLowCutoff	In	double	Set filter low cutoff
FilterHighCutoff	Out	double*	Get filter high cutoff
FilterHighCutoff	In	double	Set filter high cutoff
FilterOrder	Out	int*	Get filter order
FilterOrder	In	int	Set filter order (1..8)
WildPointRangeMin	Out	double*	Get wild point range min
WildPointRangeMin	In	double	Set wild point range min
WildPointRangeMax	Out	double*	Get wild point range max
WildPointRangeMax	In	double	Set wild point range max
WildPointCorrectionMethod	Out	IadsDataCorrectionMethod*	Get wild point correction method
WildPointCorrectionMethod	In	IadsDataCorrectionMethod	Set wild point correction method
WildPointCorrectionValue	Out	double*	Get wild point correction value
WildPointCorrectionValue	In	double	Set wild point correction value
SignChange	Out	IadsYesOrNo*	Get sign change
SignChange	In	IadsYesOrNo	Set sign change
NullCorrection	Out	IadsNullCorrection*	Get null correction type
NullCorrection	In	IadsNullCorrection	Set null correction, depends on data source type if TPP(nullOn or nullOff), if derived(equationResult, equationInput
NullBaseline	Out	double*	Get null basline value
NullBaseline	In	double	Set null baseline value
NullBias	Out	double*	Get null base value that is added to the parameter value
NullBias	In	double	Set null baseline value that is added to the parameter value
NullGroup	Out	IadsNullGroup*	Get Parameter Null group
NullGroup	In	IadsNullGroup	Set Parameter Null group
SpikeDetectionMethod	Out	IadsSpikeDetectionMethod*	Get spike detection method
SpikeDetectionMethod	In	IadsSpikeDetectionMethod	Set spike detection method
SpikeCorrectionMethod	Out	IadsDataCorrectionMethod*	Get spike correction method (last good value or none
SpikeCorrectionMethod	In	IadsDataCorrectionMethod	Set spike correction Method (last good value or none)
SpikeChangeLimit	Out	double*	Get spike change limit
SpikeChangeLimit	In	double	Set spike change limit
ComputeType	Out	IadsComputeType*	Get default compute type
ComputeType	In	IadsComputeType	Set default compute type
ExcitationSignal	Out	BSTR*	Get default excitation parameter
ExcitationSignal	In	BSTR	Set default excitation parameter
WindowType	Out	IadsWindowType*	Get window type
WindowType	In	IadsWindowType	Set window type
Alpha	Out	IadsAlpha*	Get alpha value for Kaiser-Bessel Window

Alpha	In	IadsAlpha	Set alpha value for Kaiser-Bessel window type
AveragingMethod	Out	IadsAverageMethod*	Get averaging method, (avgTime not implemented
AveragingMethod	In	IadsAverageMethod	Set averaging method, (avgTime not implemented)
Overlap	Out	double*	Get overlap value (0.0 >= value < 100.0
Overlap	In	double	Set overlap value (0.0 >= value < 100.0
BlocksPerAverage	Out	int*	Get blocks per average (1..5)
BlocksPerAverage	In	int	Set blocks per average (1..5)
BlockSize	Out	IadsBlockSize*	Get block size - (64 bytes..64K bytes
BlockSize	In	IadsBlockSize	Set block size - (64 bytes.. 64K bytes
StringLookupTable	Out	BSTR*	Get string lookup table
StringLookupTable	In	BSTR newVal	Set String Lookup table
Delete	Out	VARIANT_BOOL*	Get Parameter deletion setting
Delete	In	VARIANT_BOOL	Set parameter deletion setting

Events Collection

Method	*Return*	*Arguments*	*Description*
Count	Out	long*	Gets the number of IEvent Items within the collection.
Add	In	BSTR	Add a new IEvent Item to this collection by parameter name.
	In	BSTR	IRIG time at threshold break.
	Out	IEvent**	Return value reference to the newly added IEvent Item.
Remove	In	VARIANT	Remove IEvent Item by index or name from this collection
RemoveAll	None	VOID	Remove all IEvent Items from this collection.
Item	In	VARIANT	Return an IEvent Item by name (string) or index number (0..Count). IndexOrName
	Out	IThreshold**	Return value, reference to the specified IEvent Item. Returns a Set object by name (string) or index number (0..Count)
SaveTable	*None*	*VOID*	*Save changes to this Collection*

Event Item

Method	*Return*	*Argument*	*Description*
Group	Out	BSTR*	Get the group name
Group	In	BSTR	Set the group name
SubGroup	Out	BSTR*	Get subgroup name
SubGroup	In	BSTR	Set subgroup name
User	Out	BSTR*	Get user name
User	In	BSTR	Set user name
IrigTime	Out	BSTR*	Get IRIG time
IrigTime	In	BSTR	Set IRIG time
Name	Out	BSTR*	Get the comment
Name	In	BSTR	Set a comment
PropBag	Out	BSTR*	Get property bag
PropBag	In	BSTR	Set the property bag

Thresholds Collection

Method	Return	Arguments	Description
Count	Out	long*	Gets the number of IThreshold Items within the collection.
Add	In	BSTR	Add a new IThreshold Item to this collection by parameter name.
	In	BSTR	IRIG time at threshold break.
	In	IadsThresholdLevel	Threshold Level at the break
	Out	IThreshold**	Return value reference to the newly added IThreshold Item.
Remove	In	VARIANT	Remove IThreshold Item by index or name from this collection
RemoveAll	None	VOID	Remove all IThreshold Items from this collection.
Item	In	VARIANT	Return an IThreshold Item by name (string) or index number (0..Count). IndexOrName
	Out	IThreshold**	Return value, reference to the specified IThreshold Item. Returns a Set object by name (string) or index number (0..Count)
SaveTable	*None*	*VOID*	*Save changes to this Collection*

Threshold Item

Method	Return	Argument	Description
Group	Out	BSTR*	Get the group name
Group	In	BSTR	Set the group name
SubGroup	Out	BSTR*	Get subgroup name
SubGroup	In	BSTR	Set subgroup name
User	Out	BSTR*	Get user name
User	In	BSTR	Set user name
Level	Out	IadsThresholdLevel*	Get Threshold Level
Level	In	IadsThresholdLevel	Set Threshold Level
AnalysisWindowName	Out	BSTR*	Get property bag
AnalysisWindowName	In	BSTR	Set the property bag
DisplayType	Out	BSTR*	Get property bag
DisplayType	In	BSTR	Set the property bag
ParameterName	Out	BSTR*	Get property bag
ParameterName	In	BSTR	Set the property bag
IrigTimeAtBreak	Out	BSTR*	Get IRIG time at threshold break
IrigTimeAtBreak	In	BSTR	Set IRIG time at threshold break
ValueAtBreak	Out	double*	Get value at time break
ValueAtBreak	In	double	Set IRIG time
DisplayName	Out	BSTR*	Get display name
DisplayName	In	BSTR	Set the display name
Comment	Out	BSTR*	Get the comment
Comment	In	BSTR	Set a comment
PropBag	Out	BSTR*	Get property bag
PropBag	In	BSTR	Set the property bag

Selections Collection

Method	Return	Arguments	Description
Count	Out	long*	Gets the number of ISelection Items within the collection.
Add	In	BSTR	Add a new ISelection Item to this collection by parameter name
	In	BSTR	IRIG time of selection.
	In	Double	Value of selection
	Out	ISelection**	Return value reference to the newly added ISelection Item.

Remove	In	VARIANT	Remove ISelection Item by index or name from this collection
RemoveAll	None	VOID	Remove all ISelection Items from this collection.
Item	In	VARIANT	Return an ISelection Item by name (string) or index number (0..Count). IndexOrName
	Out	ISelection**	Return value, reference to the specified ISelection Item. Returns the Item by name (string) or index number (0..Count)
SaveTable	None	VOID	Save changes to this collection

Selection Item

Method	*Return*	*Argument*	*Description*
Group	Out	BSTR*	Get the group name
Group	In	BSTR	Set the group name
SubGroup	Out	BSTR*	Get subgroup name
SubGroup	In	BSTR	Set subgroup name
User	Out	BSTR*	Get user name
User	In	BSTR	Set user name
IrigTime	Out	BSTR*	Get IRIG Time
IrigTime	In	BSTR	Set IRIG Time
Value	Out	double*	Get Selection Value
Value	In	double	Set Selection Value
Parameter	Out	BSTR*	Get Parameter name
Parameter	In	BSTR	Set Parameter Name
Filter	Out	BSTR*	Get Filter
Filter	In	BSTR	Set Filter
Display	Out	BSTR*	Get Display Name
Display	In	BSTR	Set Display Name
Comment	Out	BSTR*	Get Comment
Comment	In	BSTR	Set Comment
PropBag	Out	BSTR*	Get property bag
PropBag	In	BSTRS	Set the property bag

TestPoints Collection

Method	*Return*	*Arguments*	*Description*
Count	Out	long*	Gets the number of ITestpoint Items within the collection.
Add	In	BSTR	Add a new ITestpoint Item to this collection by test point string.
	In	BSTR	The start time of the test point
	In	BSTR	The stop time of the test point
	Out	ITestpoint**	Return value reference to the newly added ITestpoint Item.
Remove	In	VARIANT	Remove ITestpoint Item by index or name from this collection
RemoveAll	None	VOID	Remove all ITestpoint Items from this collection.
Item	In	VARIANT	Return an IThreshold Item by name (string) or index number (0..Count). IndexOrName
	Out	IThreshold**	Return value, reference to the specified IThreshold Item. Returns the Item by name (string) or index number (0..Count)
SaveTable	*None*	*VOID*	*Save changes to this Collection*

TestPoint Item

Method	*Return*	*Argument*	*Description*
Group	Out	BSTR*	Get the group name

Group	In	BSTR	Set the group name
SubGroup	Out	BSTR*	Get subgroup name
SubGroup	In	BSTR	Set subgroup name
User	Out	BSTR*	Get user name
User	In	BSTR	Set user name
Testpoint	Out	BSTR*	Get testpoint
Testpoint	In	BSTR	Set user name
Description	Out	BSTR*	Get testpoint
Description	In	BSTR	Set user name
Maneuver	Out	BSTR*	Get testpoint
Maneuver	In	BSTR	Set user name
StartTime	Out	BSTR*	Get Start time
StartTime	In	BSTR	Set Start Time
StopTime	Out	BSTR*	Get Stop time
StopTime	In	BSTR	Set Stop Time
PropBag	Out	BSTR*	Get property bag
PropBag	In	BSTR	Set the property bag

PlannedTestPoints Collection

Method	*Return*	*Arguments*	*Description*
Count	Out	long*	Gets the number of IPlannedtestpoints Items within the collection.
Add	In	BSTR	Add a new IPlannedTestpoint Item to this collection by unique testpoint string
	Out	IPlannedTestpoint**	Return value reference to the newly added IPlannedTestpoint Item.
Remove	In	VARIANT	Remove IParameterSet object by index or name from this collection
RemoveAll	None	VOID	Remove all IParameterSet Items from this collection.
Item	In	VARIANT	Return an IParameterSet Item by name (string) or index number (0..Count). IndexOrName
	Out	IParameterSet**	Return value, reference to the specified IParameterSet Item. Returns a Set object by name (string) or index number (0..Count)
SaveTable	None	VOID	*S*ave changes to this Collection

PlannedTestPoint Item

Method	*Return*	*Argument*	*Description*
Group	Out	BSTR*	Get the group name
Group	In	BSTR	Set the group name
SubGroup	Out	BSTR*	Get subgroup name
SubGroup	In	BSTR	Set Subgroup name
User	Out	BSTR*	Get User name
User	In	BSTR	Set User name
Testpoint	Out	BSTR*	Get Testpoint
Testpoint	In	BSTR	Set Testpoint (This is a user defined format)
Description	Out	BSTR*	Get Description
Description	In	BSTR	Set Description
Maneuver	Out	BSTR*	Get Maneuver name
Maneuver	In	BSTR	Set the Maneuver name
AircraftConfig	Out	BSTR*	Get the Aircraft Configuration (This is a user defined setting)

AircraftConfig	In	BSTR	Set the Aircraft Configuration
FlightConditions	Out	BSTR*	Get Flight Conditions (This is a user defined setting)
FlightConditions	In	BSTR	Set the Flight Conditions
PredictedResults	Out	BSTR*	Get the Predicted Results
PredictedResults	In	BSTR	Set the Predicted Results
PropBag	Out	BSTR*	Get Property bag
PropBag	In	BSTR	Set the Property bag

5.1.3 General Purpose Query Interface

From the IadsConfig API level general purpose queries can be made. This allows full access to the Configuration file without using the Collection interfaces as detailed above. Intimate knowledge of the File structure is required. Use caution with this routine, especially with hierarchical tables, as entries made will most likely need to be made in other relational tables as well

Query string construction has the following form:

"keyword <field name> from <table name> where <qualifiers>"

Keywords are:

1. Select - Selected values are returned in BSTR array
2. Update - Query user passes in a string to update in the configuration file.
3. Delete - Deletes one or more rows in a configuration file
4. Create - Create a table in the configuration file.

Field names: These are directly from the Configuration file; therefore, the query user must have knowledge of its table construction: Multiple field names are separated by commas. A wild card of '*' can be used in which case the entire row is returned with individual field values delimited by the '|' vertical pipe character.

Table name: Table names are those that are in the configuration file, therefore the query user must have knowledge of its construction.

Qualifiers: Qualifiers take the form of: "field name = value". All values of type string must be enclosed in single quotes, example: Group = 'Loads'.

Query Examples

1) ***Select * from Desktops*** - Get all fields for all entries in a particular table.
2) ***Select * from Desktops where Group = 'Flutter'*** - Get all fields for selected entries using a "where" clause.
3) ***Select * from Desktops where Group = 'Flutter' && AnalysisWindowName = 'DoubleIntTest'*** - Get all fields for selected entries using a compound "where" clause.
4) ***Select SubGroup from Desktops where Group = 'Flutter' && AnalysisWindowName = 'DoubleIntTest'*** - Get one field for selected entries using a compound "where" clause.
5) ***Select System.RowNumber from Desktops*** - Get one field for selected entries using a compound "where" clause. Note usage of "System.RowNumber. This selects a row based on its unique Id (be careful with this).
6) ***Update BogusDesktops set * = a|b|c|d|e*** - Modify all fields in all rows in a table
7) ***Update BogusDesktops set * = a|b|c|d|e where Group = 'Flutter'*** - Modify all fields in the specified row(s) based on a match in a single field.
8) ***Update BogusDesktops set SubGroup = 'Pat' where Group = 'FQ'*** - Modify a single field in a specified row.
9) ***Create table BogusTable (Group String, X Int, Y Int, AutoScale list(True, False), Classification list(high,medium,low))*** - Creates a table called BogusTable with fields as

shown above. Note usage of the list type. This will create a dropdown list in the configuration tool for easier user entry.

10) ***Delete * from BogusDesktops where Group = 'Flutter'*** - Delete every row from the BogusDesktops table where the field value is Flutter

11) ***Delete * from BogusDesktops where System.RowNumber = 3*** - Delete using "built-in" unique system id or row number

5.2 IADS Data File API

Provides a COM API DLL interface to access IADS data directly from an IADS storage file (.iadsData).

The Data Tester test program was written to test as much of the Data File Interface as possible; it creates all formats of the periodic type and several for the aperiodic and multi-periodic formats. The program will read and display IADS data in a console window and is for demonstration purposes only.

The COM DLL, test project, on-line reference and Data Tester test program are available for download on the Curtiss Wright IADS website at https://iads.symvionics.com/support/programming-examples/.

6. IADS Automation Interfaces

6.1 IADS Data Export Scripts

Provides the following IADS Automation Scripts:

1) Data Export - This is an *event-triggered* data export. You define your event triggers in the code, and the script will attach to an already running IADS client and export parameters that you have specified in the script to a CSV file. No GUI for this one, you have to tweak all your settings in the code.
2) DataGroup Summary - Attaches to a client that is already running; takes the parameters from a specified data group and exports them to an Excel spreadsheet and PowerPoint slides based on start/stop times specified in the Test Point log.
3) IADS Data Move - Moves parameters from a specified input directory to a specified output directory. Command-line driven.
4) Time Slice Export - Connects to an already running IADS client and provides a dialog that allows you to select parameters from the currently running config file; specify a start/stop time, and a destination directory to export them to. Exports the data to a CSV file.

The DataExport.vbs, DataGroup Summary.vbs, IadsDataMove.zip, TimeSliceExport.zip and on-line reference are available for download on the Curtiss Wright IADS website at https://iads.symvionics.com/support/programming-examples/.

6.2 IADS Data File Reader in Visual Basic

Provides the IADSDataFileInterface dialog (Project1.exe) to read IADS data directly from an IADS data file and export to a CSV file.

The IadsDataFileInterface.dll must be registered to use the reader.

The VBDataTester.zip (Project1.exe) and IADSDataFileInterface.dll are available for download on the Curtiss Wright IADS website at https://iads.symvionics.com/support/programming-examples/ .

7. IADS Data Processing

7.1 IADS Real Time Data Source Interface

This documentation describes the interface required of a real time data source to send data to the IADS Server (CDS); the connection, protocol and format requirements.

The test project is available for download on the Curtiss Wright IADS website at: https://iads.symvionics.com/support/programming-examples/ Data Processing Examples: 1. IADS Data Source

The purpose of this instruction is to describe how to develop an interface that feeds data to IADS; and how to test and troubleshoot the development effort. This development kit includes an example data source program with source code.

Section 7.1.1 provides a specification describing the overall data source architecture including connection, protocol and format requirements along with rules on providing data. Section 7.1.2 describes an example data source program that is available as part of the development kit which can be useful for further understanding and guidance. Section 7.1.3 includes instructions on how to use an IADS product named RT Station to check out the interconnections between the data source and IADS Server along with viewing the data on an IADS Client display. Section 7.1.4 gives tips on troubleshooting potential problems that may arise during development and checkout.

7.1.1 Data Source Specification

Data Source Architecture

The real time data source architecture is one that provides data packets to the IADS Server at as close to fixed rates as possible. Since IADS is a data driven architecture the more consistent the rate the packets are fed to the IADS Server the better to provide a smooth data flow. The recommended data packet frequency is 10-20 milliseconds. To compensate for network and/or other system level delays a capability to buffer up data packets is recommended at the data source to provide some flexibility for potential data delivery delays in order to prevent data overflow/loss between the data source and the IADS Server. This buffering architecture has the advantage of allowing for some “rubber-banding” in the downstream processing without losing data at the data source.

Data Source Socket Interface

To communicate properly with the IADS Server, the data source must be set up as a TCP/IP socket server. The data source will first perform a handshake that specifies the byte order and format of the packets then will begin sending data packets. This protocol is a one-way communication going from data source to IADS Server. After the initial handshake the data source will continually send data packets (preferably in a blocked write mode) to the IADS Server. These messages are recommended to be sent at a frequency of 10-20 milliseconds. The message size can vary between data packets so in order to maintain the packet rate you may need to send packets containing only time parameter samples in cases where data parameter rates are low.

Typical usage of the data source is to keep the source running and allow the IADS Server to perform multiple connections over an extended period of time. In order to accomplish this

functionality another feature of the data source should be to allow reconnections from the IADS Server without having to restart the data source application.

Handshake Protocol

Upon initial connection to the data source, the IADS Server expects to receive two handshake messages describing aspects of the data source environment. First a one-byte message is expected that defines the byte order of all subsequent messages. The codes to specify the byte order are as follows:

Little Endian = 1

Big Endian = 2

Secondly a four-byte message is expected defining the code of the format of all subsequent data packets. The data packets must not vary from this specified format and must also conform to formatting specifications as defined in the next two sections. There are currently two supported packet formats with the following codes:

Tag/value pair format = 100

Tag/size/value format = 101

Note: Tag/size/value format only supports Little Endian byte order.

Data Packet Header Format

Each data packet from the data source has a header that contains various record and status information followed by a body that contains tag/value pairs. Each header contains 8 32-bit fields (32 bytes) described as follows:

Field	*Name*	*Description*
Field 0	Message Size	Total size of header and body (Field 0 non-inclusive)
Field 1	Sequence Number	Message sequence counter
Field 2	Packets Sent	Total number of data packets sent to IADS Server
Field 3	Data loss/Overflow	Total number of data loss/overflow occurrences
Field 4-7	Dummy	Currently unused fields

Except for field 0 (Message Size) and field 1 (Sequence Number) all other fields are essentially unused or optional fields used to describe additional data source status.

Calculating the Message Size field consists of adding the remaining portion of the header (28 bytes) to the entire size of the packet body. For example, if the packet body size is 1200 bytes then the Message Size field should contain the value 1228.

See Appendix C for a block diagram of the message format including the header.

Data Packet Body Format

Currently there are two supported data packet body formats. The first packet body format (code 100) contains sets of tag/value pairs consisting of 16-bit tag fields and 32-bit value fields. A tag is an integer that uniquely identifies a particular parameter. The values are the data associated with each tag instance. The format of a single tag/value pair in 32-bit form is as follows:

Tag1 (16-bit)

Tag2 (16-bit)

Value1 (32-bit)

Value2 (32-bit)

The body consists of **(n)** consecutive sets of these tag/value pairs. Therefore, the size of each message will consist of 28 bytes of header (message size field is non-inclusive) plus **(n)** times 12 bytes of body. This also means there are a total of **(n)** times 2 parameters per message.

The second supported packet body format (code 101) contains sets of tag/size/value sets consisting of 32-bit fields containing the integer parameter identifier followed by the data unit size in bytes followed by the data value. The format of a single tag/size/value set is as follows:

Tag (32-bit)

Size (32-bit)

Value (number of bytes specified in Size field)

The body consists of **(n)** consecutive sets of these tag/size/value sets. Therefore, the size of each message will consist of 28 bytes of header (message size field is non-inclusive) plus **(n)** times (8 + size) bytes of body where size is the size in bytes of the data associated with each particular tag. Currently this interface does not support variable size data for a particular tag. This also means there are a total of **(n)** parameters per message.

See Appendix C for a block diagram of the supported message formats.

Packet Content Notes

A primary requirement for parameters contained within the stream of data packets is that the order of the data samples coming out of the data source be chronological (time sequential) for a particular parameter. Each periodic parameter (i.e. parameter with sample rate greater than 0) is expected to have a consistent interval (with allowances for some minor rubber-banding) in the data stream correlating to the sample rate specified in the parameter definition file. No pre-alignment of data across parameters is assumed (i.e. no manipulation of the data is required prior to entering IADS).

Time parameters are required to be part of the data stream. Ideally the sample rate of the time parameters should be greater than or equal to the highest sample rate of the data parameters. The time words should be interleaved in the data packets such that each sample of a particular data parameter has a unique time stamp. The following is an example that illustrates a valid time/data sequence inside a packet where P1 and P2 are samples from 2 different data parameters and P2 is half the rate of P1. T1 and T2 are the upper and lower words respectively from the time parameter (see the section on “Time Parameters” below for more detail on time word format):

T1/T2/P1/T1/T2/P1/P2/T1/T2/P1/T1/T2/P1/P2/T1/T2/P1/T1/T2/P1/P2…

Low sample rates on time parameters can have side effects. Since IADS is basically a data driven system the update rates of the IADS Client displays can be affected by the sample rate of the time parameters. Time parameter rates lower than about 50 samples per second may show a ‘stuttering’ behavior on the client displays. See the section on “Time Parameters” below for more detail on time parameters.

7.1.2 IADS Server Setup

In order for the IADS Server to operate properly a parameter definition file must exist that provides information on how to process the packet contents. The parameter definition file is also known as the PRN file and typically has a .prn extension on the file name but is not required. The details of that file are described as follows:

Parameter Definition File

The IADS Server requires a file to exist prior to system startup that defines information on parameters expected to be received in the data stream from the data source. The set of information includes tag id, parameter name, sample rate and data format (e.g. integer, float, unsigned integer, etc.) See Appendix B for typical entries in the file.

The first field corresponds to an integer tag identifier to uniquely associate a value with a parameter in the data stream. The second field represents the name of the parameter as specified in the IADS configuration file *ParameterDefaults* table. The third field corresponds to the expected sample rate in samples per second that the parameter will be received from the data source. The sample rates can be integer or floating point. Sample rates of 0 or 0.0 denote aperiodic data. The next field identifies the format representation of the value coming from the data source. The format codes currently supported are as follows:

Data Format	*Description*	*Code Value*
32-bit integer	Integer	0
32-bit unsigned integer	Discrete	1
32-bit single precision floating point	Float	2
64-bit integer	Long	3
64-bit unsigned integer	Ulong	4
64-bit double precision floating point	Double	5
Binary objects	Blob	7

BLOB data can be identified as binary data of any size aligned on byte boundaries.

There is also extended information that can be added to parameter definition entries. The general format of these extended entries is as follows:

Key = Value

Key is a reserved keyword that is recognized by the IADS Server to represent a specific piece of parameter information and Value is the value associated with the keyword. Currently there are two supported extended information keywords: “DataSize=n” and “SystemParamType = type” the latter of which is described further in Section 7.1.3.

Currently the IADS Server does not support variable sized samples within a single tag. See Appendix D for an example Parameter Definition file.

System Parameters

Following are parameters needed for the IADS Server to fully operate properly. These are system-based parameters used for processing other data parameters or for presenting status information. There are two types of system parameters, time parameters and decom status parameters. Time parameters are required to be included as part of the overall parameter set but decom status parameters are optional. The following subsections describe these parameter types:

Time Parameters

Time is represented as a 64-bit word in units of nanoseconds consisting of the time offset from the beginning of the year. As an example, if IRIG time is set at 001:01:00:00.000 then the 64-bit time value would be 3600000000000 (i.e. one hour offset from the beginning of the year). The protocol required to transfer time via the data packets consists of splitting the time word into two 32-bit words. The upper 32-bit word must be identified in the parameter definitions file by appending the SystemParamType = MajorTime to the entry of the parameter to be used as the high order time word. The lower 32-bit word must be identified by appending SystemParamType = MinorTime to the parameter definition file entry of the parameter to be used as the low order time word. The sequence of the time words in the packet should be the upper time word followed by the lower time word. See the data flow example in the previous section "Packet Content Notes" where the upper word is represented as T1 and the lower word is represented as T2.

Ideally the sample rate of the time parameters should be greater than or equal to the highest sample rate of the data parameters. The time words should be interleaved in the data packets such that each sample of a particular data parameter has a unique time stamp.

Low sample rates on time parameters can have side effects. Since IADS is basically a data driven system the IADS Client display update rates can be affected by the sample rate of the time parameters. Time parameter rates lower than about 50 samples per second may show a 'stuttering' behavior on the client displays.

Decom Status Parameters

Decom Status is another system parameter used by the IADS Server to obtain data stream information such as sync loss. This parameter is identified in the parameter definition file by using the extended information property "DecomStatus" (see Appendix D for the data format of this parameter). The "DecomStatus" parameter will be defined in the IADS Client display using the default naming convention of _IadsDecomStatus(**n**)_. Where (**n**) is the stream number for multiple PCM stream setups. These parameters are automatically created in the IADS Configuration file upon startup and can be used by the IADS Client along with pre-defined derived functions for display purposes. These parameters are also used for informational purposes by the IADS Operator Console in a real time environment. Even though these parameters are not required in the stream, definition in the parameter definition file (see "IADS Data Source Project" section below) is recommended. The currently supported decom status format is shown in Appendix E.

The following is a parameter definition file excerpt showing both the Time and Decom Status system parameter definitions.

```
1 DecomStatus     50.0    2 SystemParamType = DecomStatus
2 Param1          50.0    2
3 Param2          50.0    2
```

```
4 Param3          50.0    2
5 Param4          50.0    2
6 TimeUpperWord   1000.0  1 SystemParamType = MajorTime
7 TimeLowerWord   1000.0  1 SystemParamType = MinorTime
```

Example Data Source Program

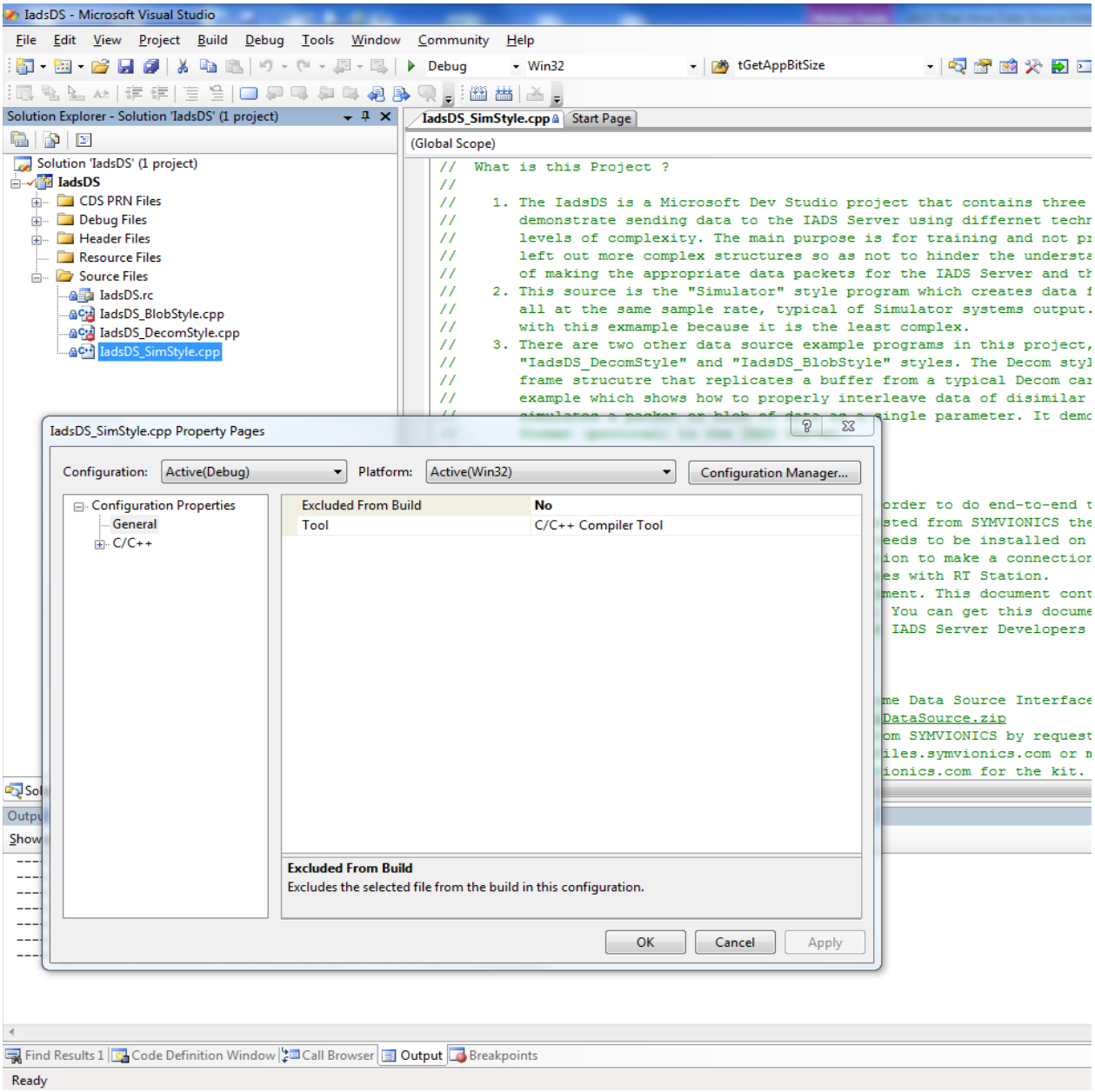
The purpose of the example IADS Data Source program is to provide a better understanding and guidance on the specifics of developing an interface to communicate with the IADS Server. The program initially waits for an IADS Server to connect and then sends data packets containing simulated data. The program also allows for reconnections to the IADS Server without re-launching the application which is a useful feature of the data source.

The IADS Data Source program is available either by requesting the IADS Data Source Developers Kit from Curtiss Wright IADS or downloading from the Programming Examples page on the Curtiss Wright IADS web site (https://iads.symvionics.com/support/programming-examples/) by selecting the IADS Data Source option under the section named “Data Processing Examples”.

IADS Data Source Project

The IADS Data Source program is a Microsoft Visual Studio 2005 project written in C++ that contains three example programs demonstrating how to output the different packet formats available along with various methods of inserting data inside the packets. The project also contains three parameter definition files that describe the parameter specifications for each example. For more background information on packet setup and communication protocol along with details on parameter definition files see Section 7.1.1 of this document.

In order to specify which example program to apply, open up the project in Visual Studio and go to the Solution Explorer then Right-click on one of the source files named IadsDS_SimStyle.cpp, IadsDS_DecomStyle.cpp or IadsDS_BlobStyle.cpp and select Properties. Then in the Property Pages dialog go to the *Excluded From Build* entry located under Configuration Properties > General and specify **No** to include the file or **Yes** to exclude the file. Make sure only one of the cpp files is set to **No** before building the project.



The different example programs are described as follows:

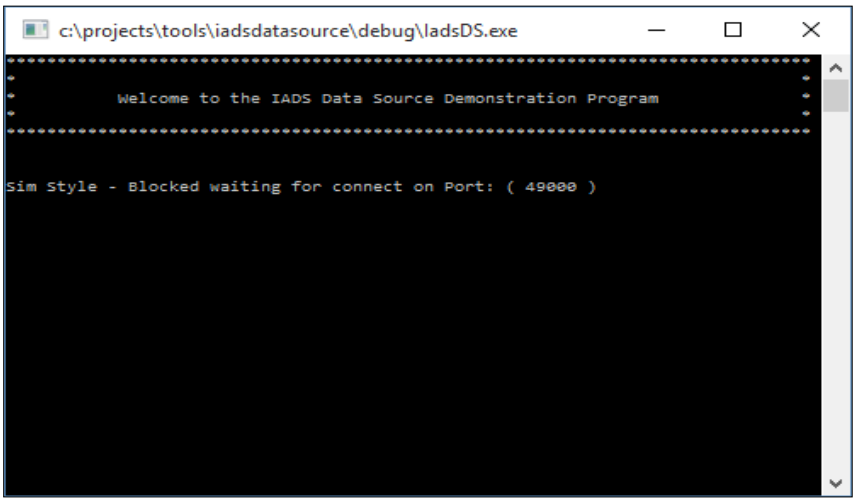
1) Simulator Style (IadsDS_SimStyle.cpp) - This program sends four data parameters along with time words to the IADS Server. The data parameters are all at the same sample rate which is typical of simulator output. Since this is the simplest case, we recommend that you start with this example. To setup a project build, the IadsDS_SimStyle.cpp should be the only cpp file where the *Excluded From Buil*d property is set to **No**. The parameter definition file associated with this program is named IadsDS.prn.SimStyle and located in the main project directory. Note that this program applies packet format 100 (tag/tag/value/value).
2) Decom Style (IadsDS_DecomStyle.cpp) - This program sends five data parameters along with time words to the IADS Server. The data parameters are at different sample rates which

is typical of decom-based systems. This provides a more complex example showing how to populate packets using differing sample rates. To setup a project for build, the IadsDS_DecomStyle.cpp should be the only cpp file where the *Excluded From Build* property is set to **No**. The parameter definition file associated with this program is named IadsDS.prn.DecomStyle and located in the main project directory. Note that this program applies packet format 100 (tag/tag/value/value).

3) Blob Style (IadsDS_BlobStyle.cpp) - This program sends one Blob parameter along with time words to the IADS Server. The Blob parameter contains four floating point parameters. To setup a project for build, the IadsDS_BlobStyle.cpp should be the only cpp file where the *Excluded From Build* property is set to **No**. The parameter definition file associated with this program is named IadsDS.prn.BlobStyle and located in the main project directory. Note that this program applies packet format 101 (tag/size/value).

Running the IADS Data Source Program

The IADS Data Source program can be run either within the Visual Studio environment or by creating a shortcut on the Windows desktop that points to the program's executable file (IadsDS.exe). Each version of the example program launches a command window and then goes into a state that waits for the IADS Server to connect. The next section provides instructions on how to use IADS to test the data source interface.



7.1.3 Testing the data source using IADS Real Time Station

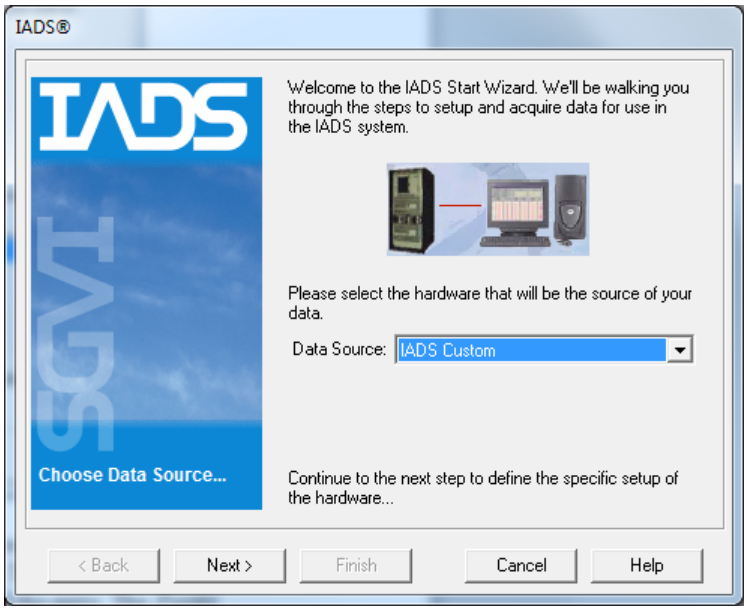
To test the data source interface, we recommend using the IADS Real Time Station (RT Station) product in order to perform communication protocol and data flow verification activities. RT Station is an installable application that includes both the IADS Server and IADS Client display subsystems so that data from the data source program can be delivered and viewed in IADS.

The RT Station installation package is available either by purchasing the product from Curtiss Wright (Part numbers are IADS-TELEM-RTSTATION-1 or IADS-TELEM-BASE-TPP) or by requesting the IADS Data Source Developers Kit (IADS-TELEM-DEV)

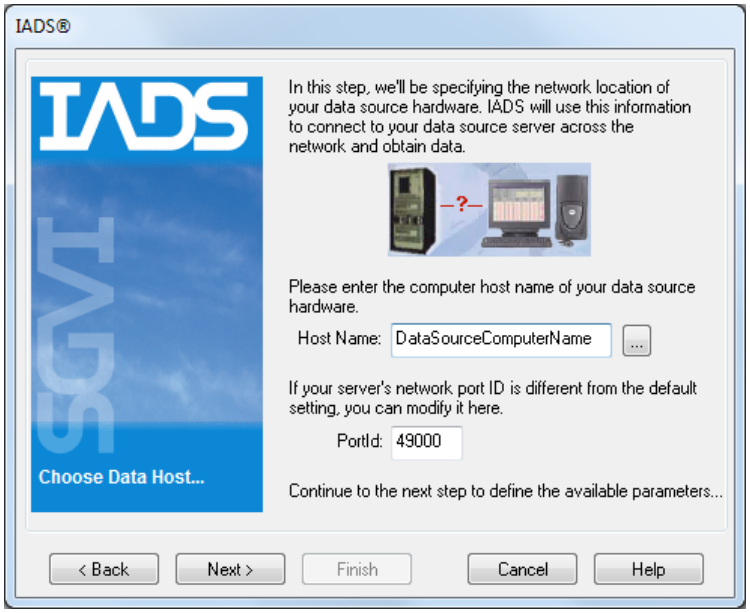
Running RT Station

Running RT Station brings up a start wizard that will guide you through the process of selecting setup information that describes how to connect to the data source program and specify the parameter definition file. The startup steps are as follows:

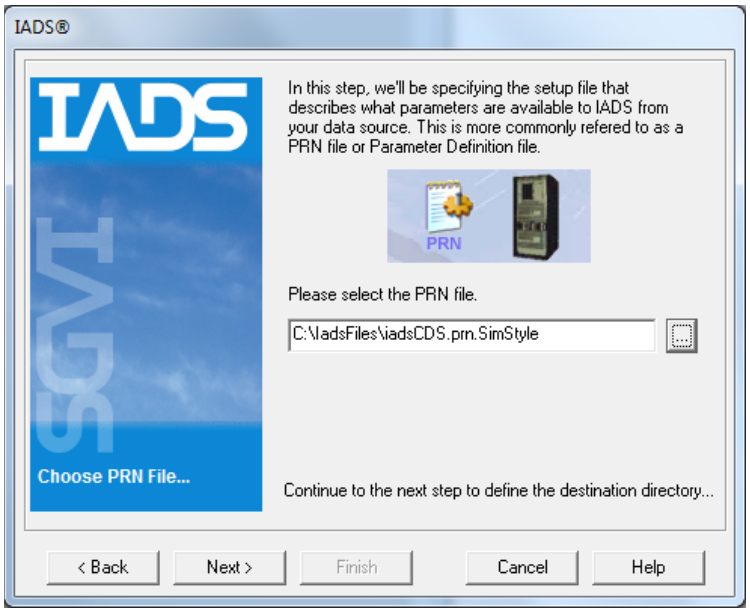
1) Make sure the data source program is running and waiting to connect to IADS.
2) Double click the **IADS Real Time Station** icon on the Windows desktop.
3) On the Choose Data Source page select the **IADS Custom** option from the dropdown menu in the *Data Source* field. Click **Next** to continue.



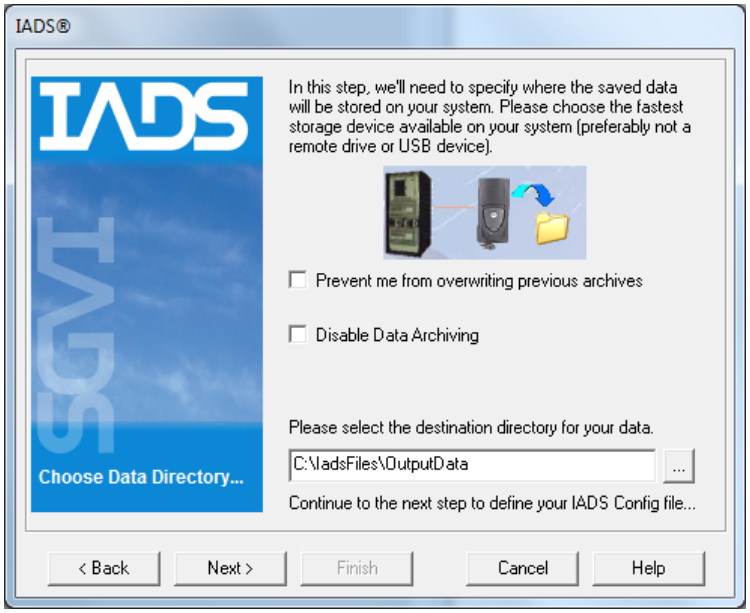
4) On the Choose Data Host page enter the name or IP address of the computer running the data source program in the Host Name entry. This can be entered manually or selected via the browse button on the right of the entry. The PortId entry defaults to 49000 which is the initial setup in the example data source program. This field can be edited to specify the port id that is available on the data source for connection. Click Next to continue.



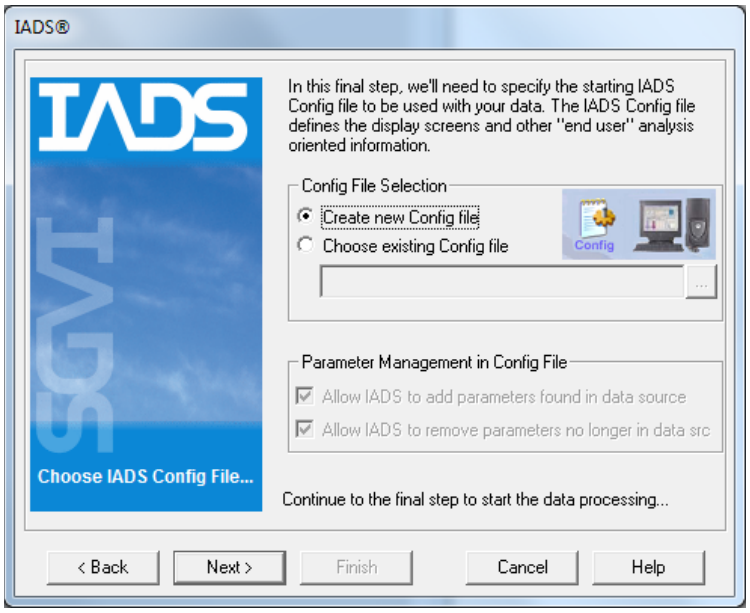
5) On the Choose PRN File page select the parameter definition file that contains the parameter specifications of the data source output. There is a browse button available on the right of the entry to assist in locating the file. If you are running one of the sample programs the matching parameter definition files are in the IADS Data Source project location. Click Next to continue.



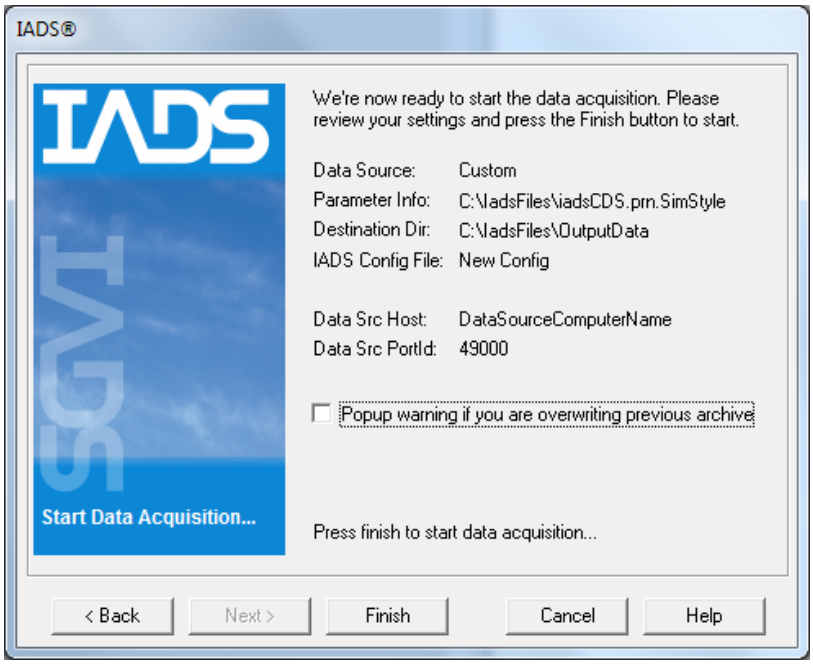
6) On the Choose Data Directory page select the destination folder for your IADS data storage files. A browse button is available on the right of the entry to assist in locating the directory. Click Next to continue.



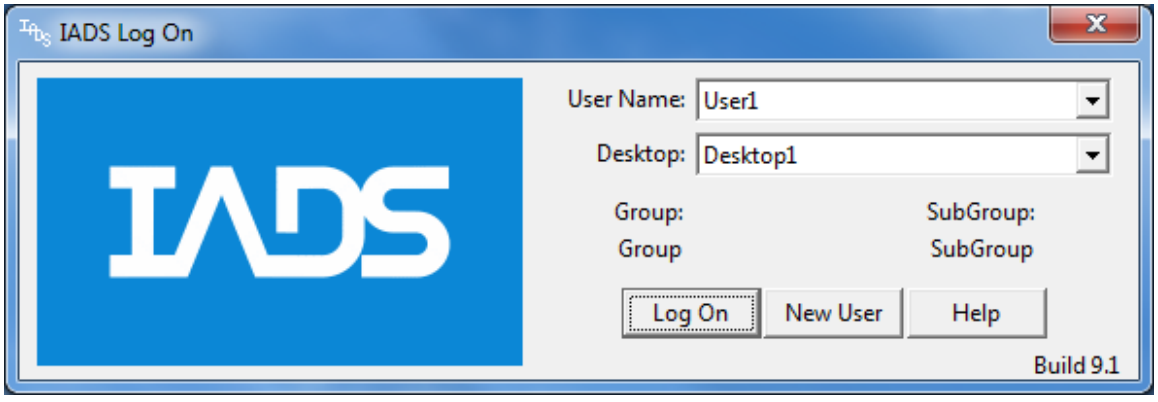
7) On the Choose IADS Config File page select the **Create new Config file** option. This will automatically create a new IADS Configuration File from scratch that contains the parameters specified in the parameter definition file you selected earlier in the wizard. Click Next to continue.



8) On the Start Data Acquisition page review the settings and click Finish to start IADS.



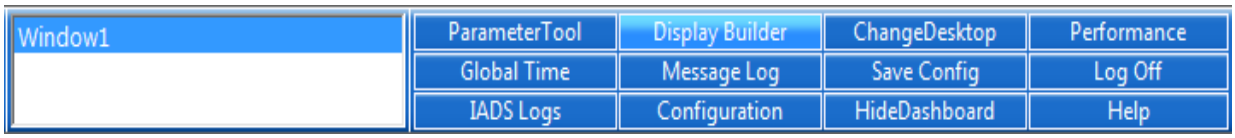
At this point RT Station will connect to the data source and start ingesting and processing data packets. The IADS Client application will then be launched and you will be prompted to startup the client via the IADS Log On dialog. Select the predefined user name User1 and desktop name Desktop1 then click the **Log On** button (This step is automatic if User1 and Desktop1 are the only options.) This user and desktop setup contains a blank Analysis Window (Window1) which acts as a palette for data displays to be added.



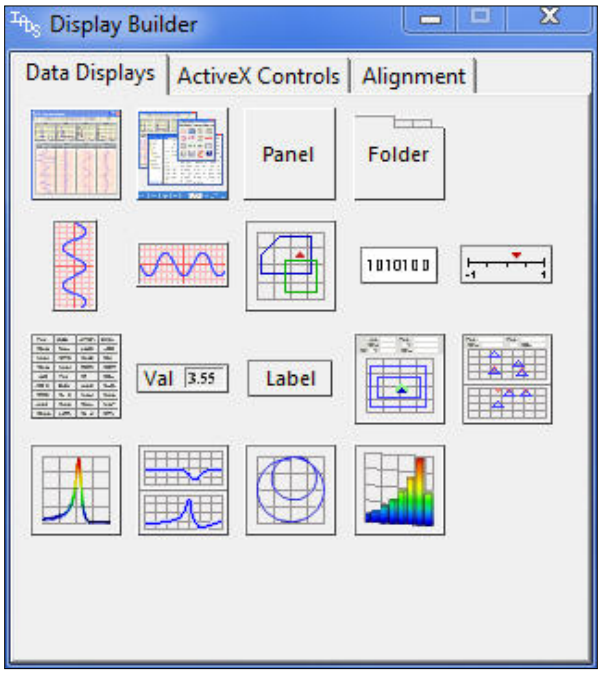
Creating Displays in IADS

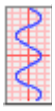
In order to view the data being delivered from the data source you need to create displays within the IADS Client application. IADS displays are created using icons in the Display Builder shown below. The Display Builder button is located in the far-right portion of the IADS Dashboard. Use the Window1 analysis window to house the displays. The steps to build up displays are as follows:

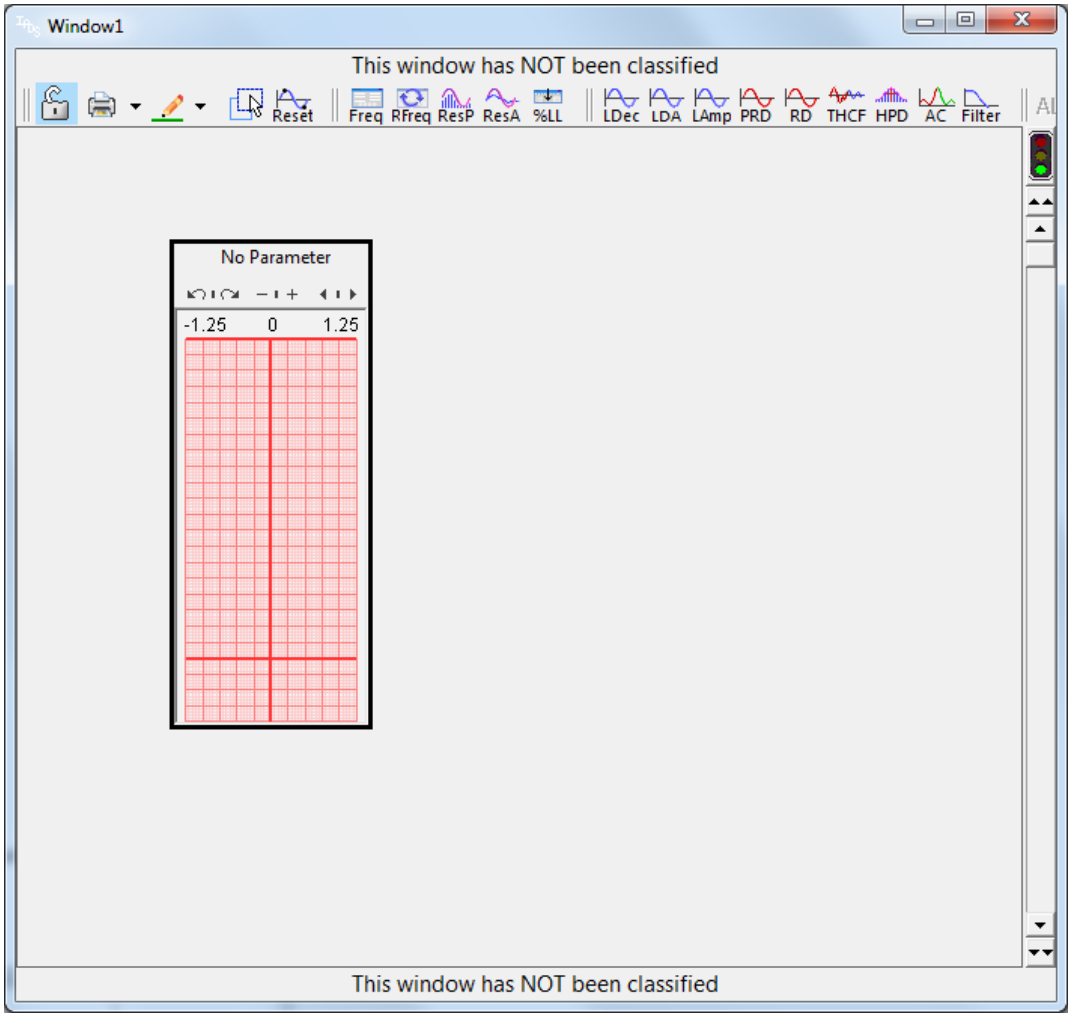
1) On the Dashboard click the **Display Builder** button.



2) On the Display Builder dialog click the **Data Displays** tab. A selection of display types is presented.

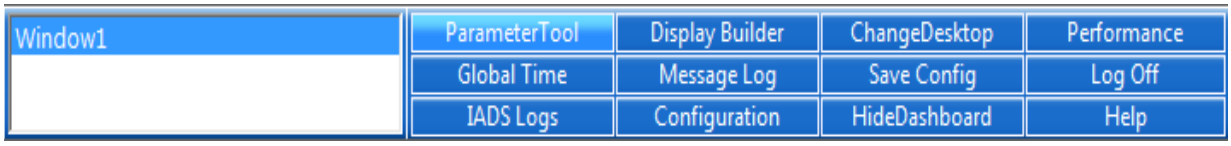


3) On the Data Displays tab click on the **Vertical Stripchart** icon  then hold down the left mouse button and drag onto Window1. This will create a new instance of a Stripchart display inside the window.

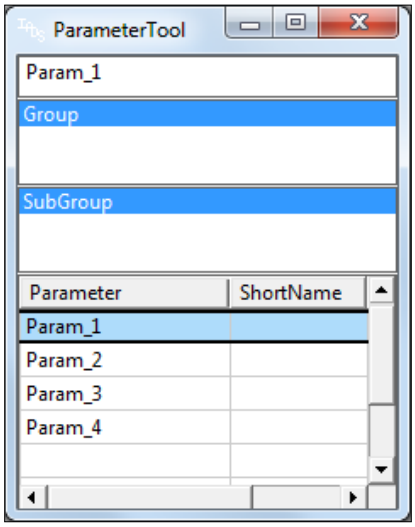


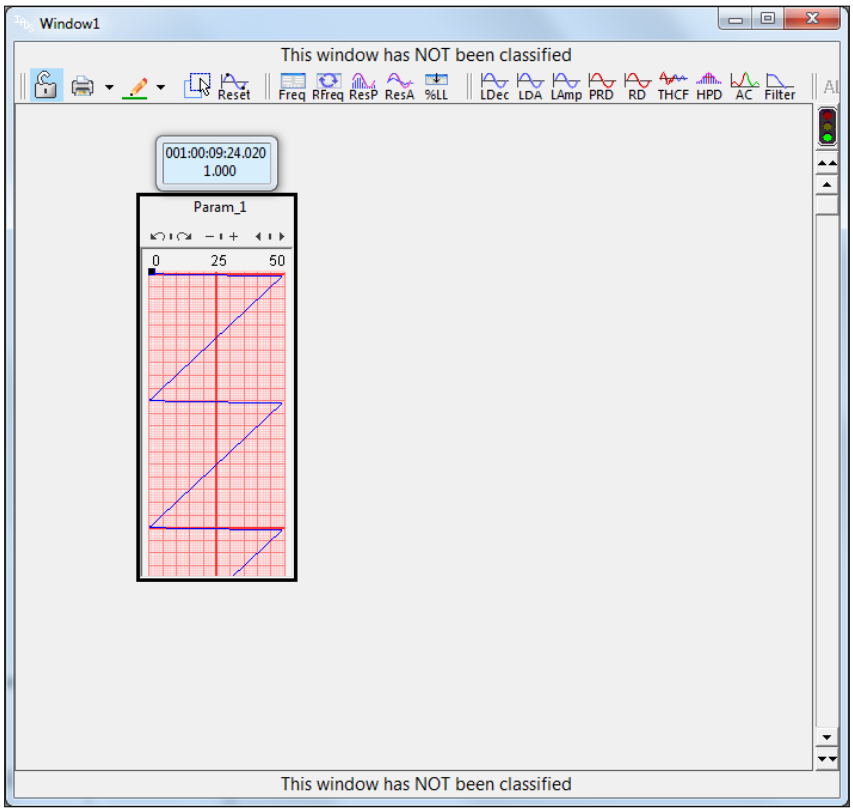
Adding Parameters to an IADS Display

1) On the Dashboard, click the **Parameter Tool** button.



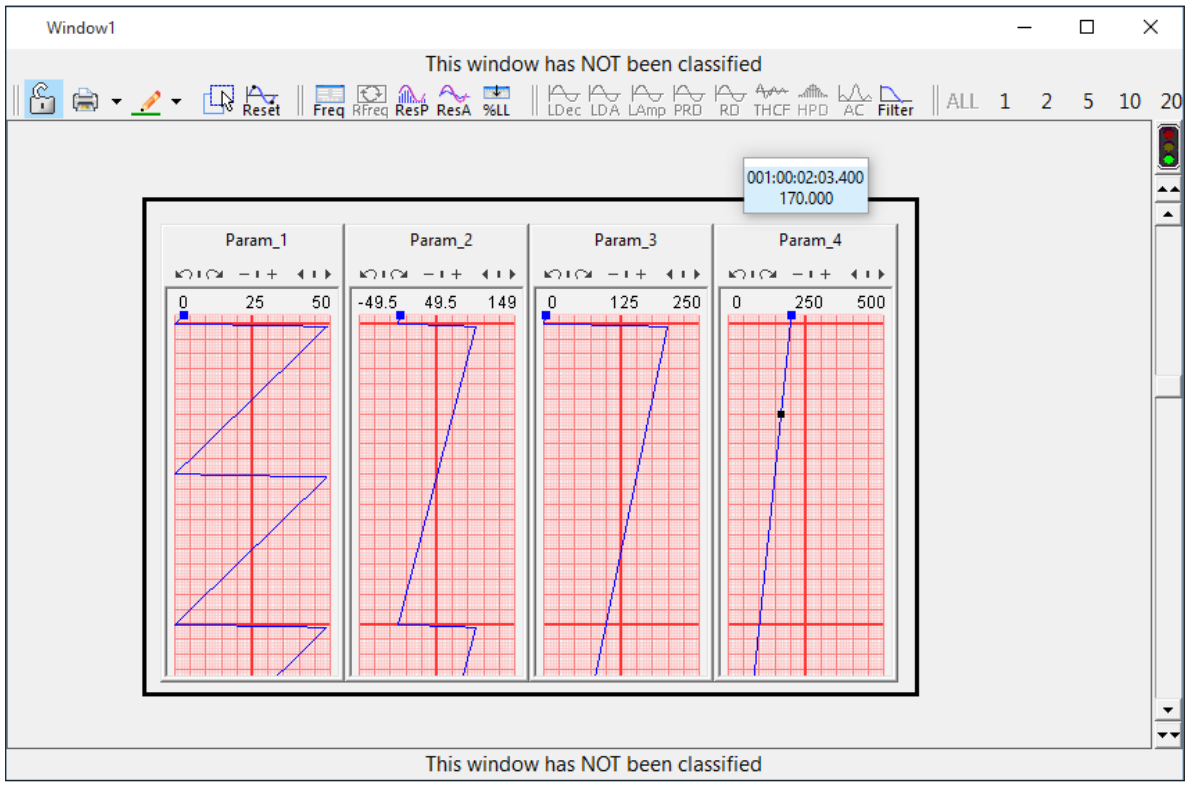
2) On the Parameter Tool select a parameter then hold down the left mouse button and drag onto the Stripchart display. Select Value in the popup options after dropping the parameter onto the display (i.e. releasing the left mouse button).



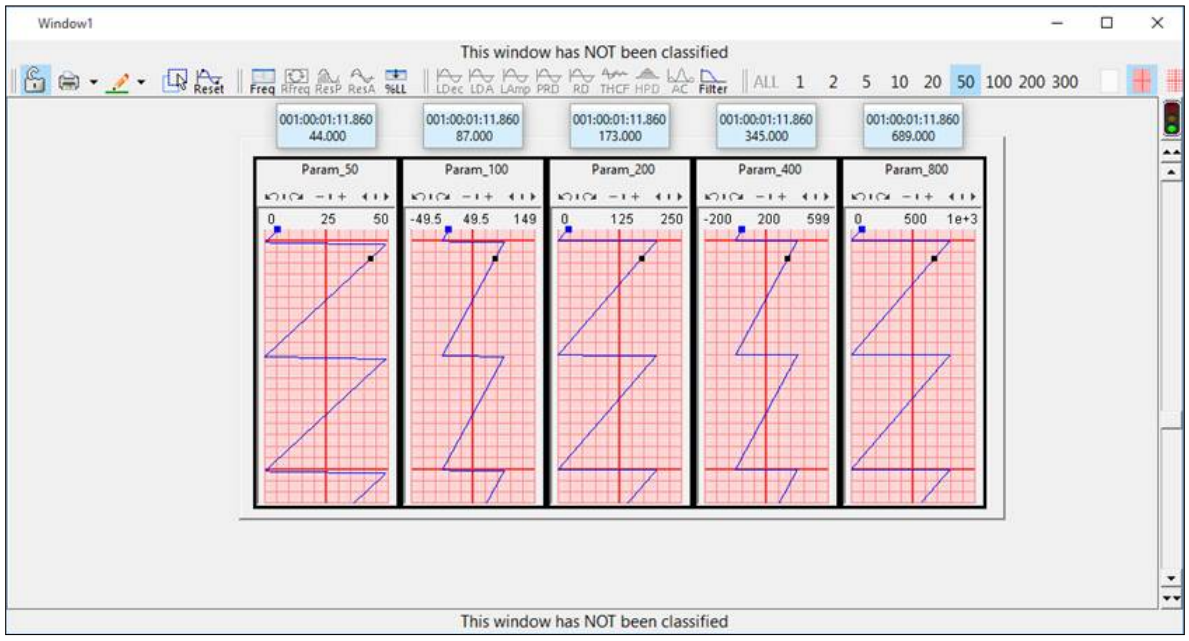


The following are example windows showing data from the three sample programs contained in the IADS Data Source project:

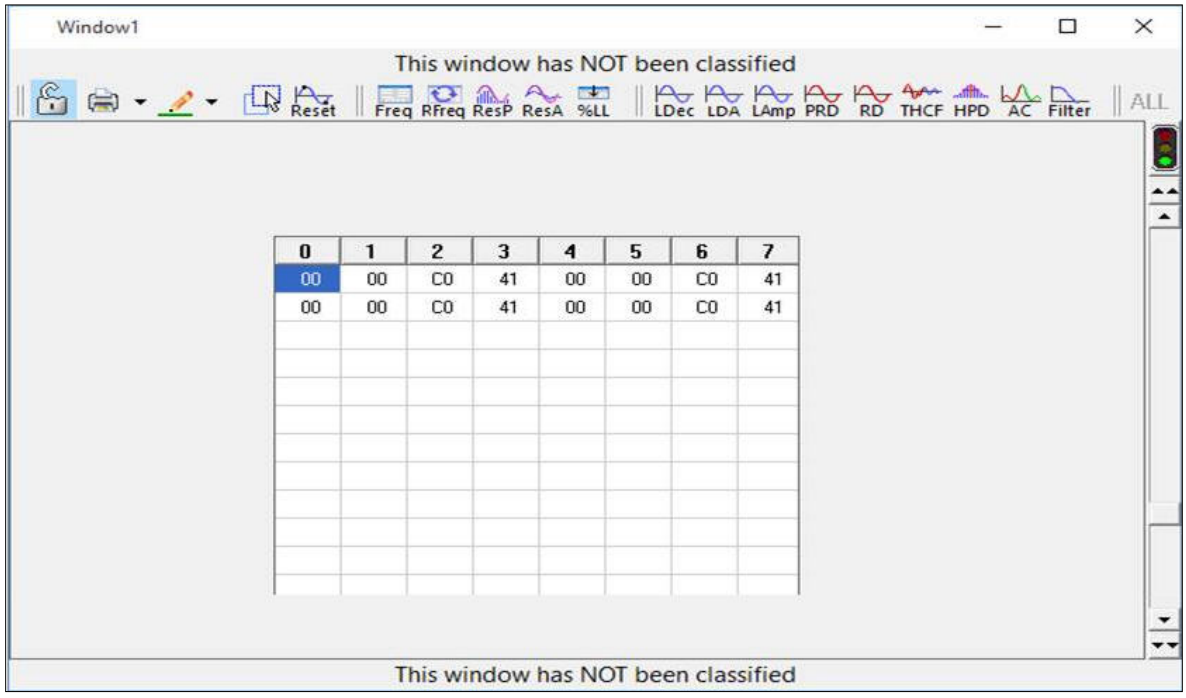
Simulator Style


Window1
This window has NOT been classified
Reset
Freq
RFreq
ResP
ResA
%LL
LDec
LDA
LAmp
PRD
RD
THCF
HPD
AC
Filter
ALL
1
2
5
10
20
001:00:02:03.400
170.000
Param_1
Param_2
Param_3
Param_4
0
25
50
-49.5
49.5
149
0
125
250
0
250
500
This window has NOT been classified

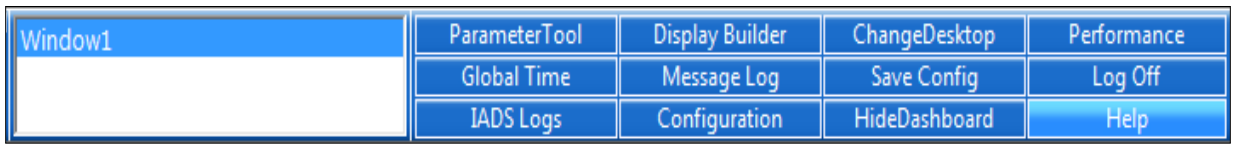
Decom Style


Window1
This window has NOT been classified
Reset
Freq
RFreq
ResP
ResA
%LL
LDec
LDA
LAmp
PRD
RD
THCF
HPD
AC
Filter
ALL
1
2
5
10
20
50
100
200
300
001:00:01:11.860
44.000
001:00:01:11.860
87.000
001:00:01:11.860
173.000
001:00:01:11.860
345.000
001:00:01:11.860
689.000
Param_50
Param_100
Param_200
Param_400
Param_800
0
25
50
-49.5
49.5
149
0
125
250
-200
200
599
0
500
1e+3
This window has NOT been classified

Blob Style



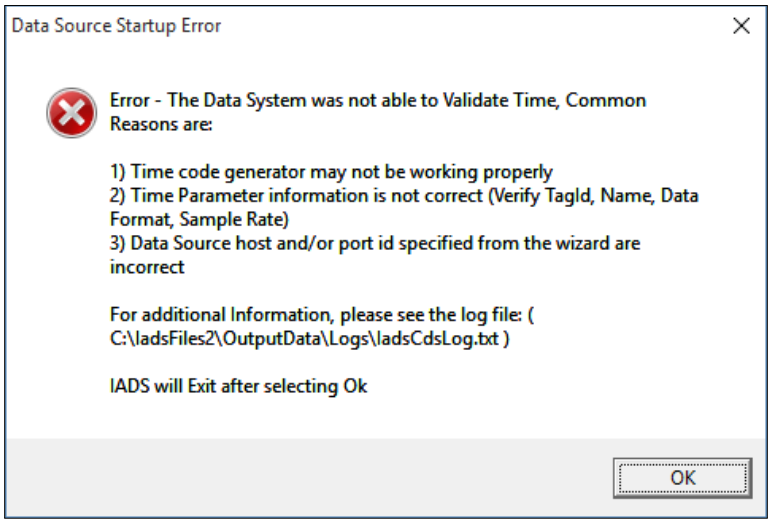
More detail on display types and general client functionality can be found in the IADS Client Help system which can be accessed via the lower right-most button on the Dashboard.



7.1.4 Troubleshooting

Validating Time

When the IADS Server connects to the data source and starts receiving packets it goes through a brief period of verifying that time parameter values are increasing at the expected increment. A common problem with a new data source interface is that the IADS Server may not successfully validate time during startup because the time values are not increasing at an increment based on the time word sample rate. For example, if the rate of the time parameter is 1000 samples per second the IADS Server will expect time value increments of 1 millisecond per sample. Any time decrements or increments over 2x the expected rate will cause a validation failure. If time does not successfully validate a dialog containing possible reasons for failure will pop up as follows:



Typical reasons for time validation failures are as follows:

1) One or both time parameter tag ids are not found in the data. In this case verify that the parameter definition file that was entered during wizard startup contains the correct tag ids for the upper and lower time parameters. Verify that the data source is actually placing time parameter samples within the data packets. Check that the data source is truly sending data packets to the IADS Server.
2) Invalid time sequence errors. In this case verify the sample rate of the time parameters in the parameter definition file are correct. Verify there are no upstream errors occurring from the time source...e.g. Time Code Generator. Time values that seem corrupted (e.g. IRIG day values outside the range of 1-366) could mean that there is a packet misalignment...verify that the packet size specified in the 1st word of the packet header corresponds to the actual size of the remaining portion of the header along with the packet payload size.

A source of information that assists in determining reasons for validation errors is a file named “timeOut0.txt” located in the Logs folder under the IADS data directory that was specified during wizard startup. This file contains time values of each time word received by the IADS Server during the time validation period. An empty file means that one or both time parameter tag ids were not sensed during the validation period (see reason 1 above). The file is

also valuable for obtaining more detail on invalid time sequence errors (see reason 2 above). An example format of the “timeOut0.txt” file is as follows:

001:00:00:00.020 (86400020000000)
001:00:00:00.040 (86400040000000)
001:00:00:00.060 (86400060000000)
001:00:00:00.080 (86400080000000)
001:00:00:00.100 (86400100000000)

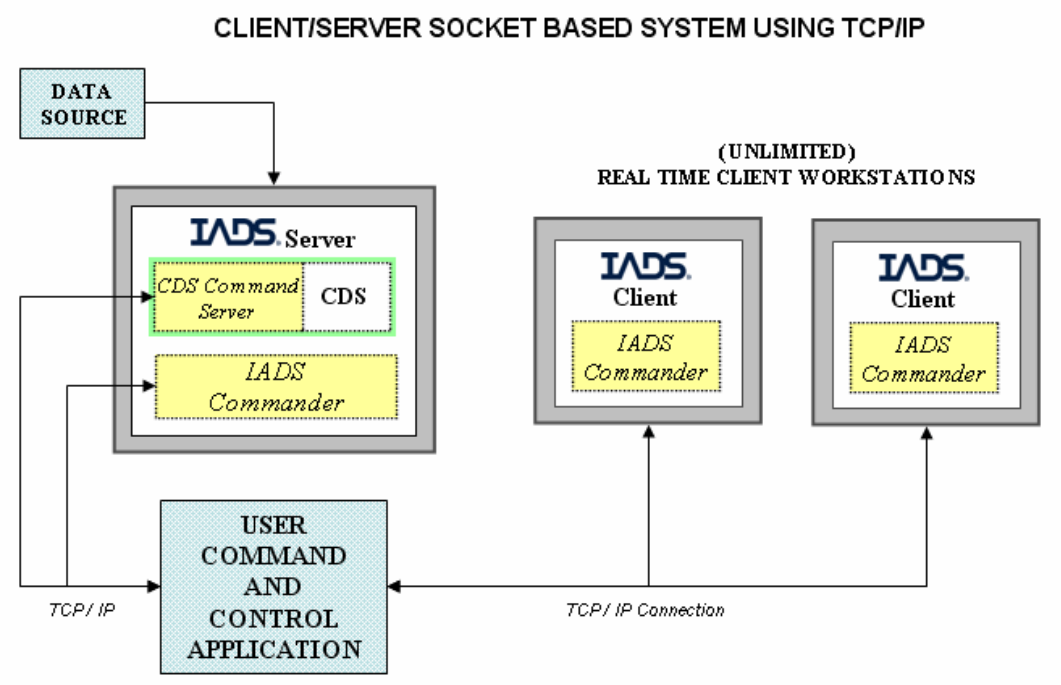
The 1st column is the IRIG representation of the time values and the 2nd column is the 64-bit integer representation of the values (in nanoseconds). In this example the time is incrementing by 20 milliseconds per sample which should correspond to a 50 sample per second rate for the time parameters specified in the parameter definitions file

For further troubleshooting assistance please zip up the entire Logs folder located in the IADS data directory and send to iads-support@curtisswright.com to help in analyzing the issue.

7.2 IADS Command Interface

The IADS Command interface assists users with programming their own command and control application for the IADS real-time system.

In order to command and control the IADS Real-time system the EUCCA must interface to the CDS Command server that is part of the CDS running application. Interfacing to the IADS Commander is optional, but is recommended because the function of this application may increase in future releases. The CDS Command server interface is used to command the CDS application with functions such as validation, startup, resets, and shutdown and to retrieve status. The IADS Commander is typically installed on the target computer to run upon login. Its purpose is to allow remote startup of an IADS application such as the CDS and the IADS display client and to retrieve status on the computer and file transfers. The EUCCA will typically use the IADS Commander to transfer needed startup files for the CDS and the IADS display clients. The IADS commander can also be used to launch the CDS, then, upon successful start of the CDS, start the IADS display clients. During real-time the EUCCA will use the CDS command server to get status and perform a shutdown. Both the CDS and the IADS Commander interfaces are client/server socket-based network applications that utilize TCP/IP on well know Port Ids. Following is a notional diagram of EUCCA and the component server interfaces.



7.2.1 IADS Commander

The IADS Commander is part of the IADS Real Time system installation package and after install will run upon login of the computer. The IADS Commander is installed on the IADS display client workstations and the IADS Server (The CDS). The primary purpose of the IADS Commander is to provide a socket-based interface to start and terminate IADS application components, get status on the machine that it installed on, and performing critical functions such as transferring application startup files. The IADS Commander utilizes the TCP/IP protocol therefore the EUCCA will make an active connection and behave as a client making requests and receiving message replies from the IADS Commander server. Once complete the EUCCA needs to disconnect after which the IADS Commander will be ready for further connections.

Connection Information

Field	*Entry/Selection*
Connection Port	58003
Connection Type	TCP/IP

Protocol Overview

- Fixed length message packet of 2000 ASCII characters
- Client performs a send on the socket of the full 2000 characters regardless of the message type
- The IADS Commander server performs the function and sends back a status string
- Client performs a receive on the socket to retrieve the response message
- Argument strings cannot contain spaces

IADS Commander Message Syntax

Message:<sp>Message:<sp>[Arguments]\n[Message...\n]

IADS Commander Messages Overview

Message	*Description*
CreateProcess	Create a process on a local or remote computer
TerminateProcess	Terminate a process on a local or remote computer
ForwardMessage	Send a message to another Commander
ProcessStatus	Determine if a process is running on a remote machine
ProcessSearch	Find all processes given a process name
TransferFile	Transfer a file from one computer to another
WritePermission	Get the file permission for the given file
TempPath	Get the Windows temporary path for a remote machine
SystemInfo	Get certain Windows system information
ProcessInfo	Get certain Windows process information
PathIsDirectory	Check if the path sent from the Client is an actual path on the remote machine
MakeDirectory	Create the directory given the path sent from the Client
DirectoryHasData	Check if the directory already contains IADS data
GetNextFreeDirectoryName	The Commander returns a directory name
FileExists	The Commander checks if the file already exists

CreateProcess

Description: Create a process on a local or remote computer.

Behavior: The IADS Commander will run the process given the process name (not recommended) or the PID if the process name is empty.

Request Message	
"Message: CreateProcess\nProcessName: \nArguments: \nWorkingDirectory: \n"	
Request Message Arguments	
Arg1	(ProcessName) – Full path and name of the process.
Arg2	(Arguments) – User specified arguments passed to the process.
Arg3	(WorkingDirectory) – Windows will start the process with this as its working directory.
Response Messages	
1	"Message: Acknowledge\nProcessStatus: Stopped\n"
2	"Message: Acknowledge\nProcessStatus: Running\nPID: \n"

CreateProcess message process:

1) Client sends the CreateProcess message with the required arguments.

2) Server receives the message and performs the function.
3) Server responds with a status message.
4) Client receives the status message.

TerminateProcess

Description: Terminate a process on a local or remote computer. Use of the PID argument is recommended.

Behavior: The Commander will hard terminate the process specified by the process name (not recommended) or the PID. Be careful with this call because it kills the process without notifying it.

Request Message	
"Message: TerminateProcess\nPID: \nProcessName: \nExitCode: \n"	
Request Message Arguments	
Arg1	(PID) – Kill process by the process ID.
Arg2	(ProcessName) – Kill Process by Process name (use caution with duplicate running processes).
Arg3	(ExitCode) – User specified arguments passed to the process.
Response Messages	
1	"Message: Acknowledge\nProcessStatus: Stopped\n
2	"Message: Acknowledge\nProcessStatus: Running\nPID: \n

Terminate message process:

1) Client sends the TerminateProcess message with the required arguments.
2) Server receives the message and performs the function.
3) Server responds with a status message.
4) Client receives the status message.

ForwardMessage

Description: Send a message from the currently connected Commander to another running on a different computer.

Behavior: The Commander will forward the message onto the Commander running on the remote computer specified by the Host argument.

Request Message	
"Message: ForwardMessage\nHost: \nRemoteMessage: \n"	
Request Message Arguments	
Arg1	(Host) – Host with running commander to forward the message to.
Arg2	(RemoteMessage) – Message to forward.
Response Messages	
1	"Message: Acknowledge\nForwardMessage: %s\n"
2 - Error	"Message: Acknowledge\nForwardMessage: Unable to Parse Message\n"

3 - Error	"Message: Acknowledge\nForwardMessage: Unable to Connect to RemoteCommander\n"
4 - Error	"Message: Acknowledge\nForwardMessage: Unable to Send Message to RemoteCommander\n"
5 - Error	"Message: Acknowledge\nForwardMessage: Unable to Receive Message from Remote Commander\n"

ForwardMessage message process:

1) Client sends the ForwardMessage message with the required requirements.
2) Server receives the message and performs the function.
3) Server responds with a status message.
4) Client receives the status message.

ProcessStatus

Description: Determine if a process is running on a remote machine. Use of the PID argument is highly recommended.

Behavior: If the ProcessName argument is empty, the PID argument is used to identify the process.

Request Message	
"Message: ProcessStatus\nHost: \nRemoteMessage: \n"	
Request Message Arguments	
Arg1	(ProcessName) – Name of the process to get status on. Use Caution with this argument because multiple versions of the same application may be running.
Arg2	(PID) – Process ID of the process to get status on.
Response Messages	
1	"Message: Acknowledge\nProcessStatus: Running\nVER: 1\n"
2	"Message: Acknowledge\nProcessStatus: Stopped\nVER: 1\n"
3	"Message: Acknowledge\nNumRet: \n"
4 - Error	"Message: Acknowledge\nProcessStatus: Unknown Process"
5 -Error	"Message: Acknowledge\nProcessInfo: PID is invalid\n"
6 - Error	"Message: Acknowledge\nProcessSearch: Unknown ProcessName\n"

ProcessStatus message process:

1) Client sends the ProcessStatus message with the required arguments.
2) Server receives the message and performs the function.
3) Server responds with a status message.
4) Client receives the status message.

ProcessSearch

Description: Find all processes given a process name and return the PID and Command line arguments for each.

Behavior: The Commander will search the process table for all occurrences of the process name.

Request Message	
"Message: ProcessSearch\nProcessName: \n"	
Request Message Arguments	
Arg1	(ProcessName) – Name of the process to get status for. Use Caution with this argument because multiple versions of the same application may be running.
Arg2	(PID) – Process ID of the process to get status on.
Response Messages	
1	"Message: Acknowledge\nNumRet: 0\n"
2	"Message: Acknowledge\nNumRet: \nCmdArgs%d: \n\nPID%d: \n"
3 - Error	"Message: Acknowledge\nProcessSearch: Unknown ProcessName\n"

Process Search message process:

1) Client sends the ProcessSearch message with the required Arguments.
2) Server receives the message and performs the function.
3) Server responds with a response message.
4) Client receives the response message.

TransferFile

Description: Transfer a file from one computer to another.

Behavior: Transfers configuration and various startup files.

Request Message	
"Message: TransferFile\nNameAndPath: \nFileSize: \n"	
Request Message Arguments	
Arg1	(NameAndPath) - The location to write the file on the destination machine.
Arg2	(FileSize) - The total file size that will be transferred.
Response Messages	
1	"Message: Acknowledge\nTransferStatus: Success creating file\n"
2	"Message: Acknowledge\nTransferStatus: Success transferring file\n"
3 - Error	"Message: Acknowledge\nTransferFile: FileName is empty\n"
4 - Error	"Message: Acknowledge\nTransferFile: FileSize is Empty\n"
5 - Error	"Message: Acknowledge\nTransferStatus: File Size is less than or equal to zero\n"
6 - Error	"Message: Acknowledge\nTransferStatus: Error creating file\n"
7 - Error	"Message: Acknowledge\nTransferStatus: Unable to allocate file transfer buffer\n"
8 - Error	"Message: Acknowledge\nTransferStatus: Unable to Recv file transfer buffer\n"
9 - Error	"Message: Acknowledge\nTransferStatus: Error transferring file\n"

TransferFile message process:

1) The Client sends the TransferFile message with the required arguments.
2) The Server receives the message and waits in a socket receive for the file data.

3) The Client receives the file creation message.
4) The Client sends the file.
5) The Server sends the response message.
6) The Client receives the response message.

Warning: Files will fail to transfer if the file size is greater than available contiguous memory. Also, transferring files greater than 1MB is not recommended.

WritePermission

Description: Get the file permission for the given file.

Behavior: The Commander will return the permission of the given file.

Request Message	
"Message: WritePermission\nNameAndPath: \n"	
Request Message Arguments	
Arg1	(NameAndPath) – Absolute file name and path. Please note that the entire message is limited to 2000 characters.
Response Messages	
1	"Message: Acknowledge\nWritePermission: ReadOnly\n"
2	"Message: Acknowledge\nWritePermission: Writeable\n"
3 - Error	"Message: Acknowledge\nWritePermission: Filename is empty\n"

WritePermission message process:

1) The Client sends the WritePermission message with the required arguments.
2) The Commander receives the message and performs the function.
3) The Commander sends the response message.
4) The Client receives the response message.

TempPath

Description: Get the Windows temporary path for a remote machine.

Behavior: The Commander will return the Windows temporary path for the remote computer.

Request Message	
"Message: TempPath\n"	
Request Message Arguments	
None	
Response Messages	
1	"Message: Acknowledge\nTempPath: <temporary path>\n"

TempPath message process:

1) The Client sends the TempPath message.
2) The Commander receives the message and performs the function.

3) The Commander sends the response message.
4) The Client receives the response message.

SystemInfo

Description: Get certain Windows system information.

Behavior: The Commander will return certain system information for the remote computer.

Request Message	
"Message: SystemInfo\n"	
Request Message Arguments	
None	
Response Messages	
1	" Message: Acknowledge\nSystemInfo: <disk size> <disk free> <total memory> <total memory used> <total physical memory> <physical memory used> <number of CPUs> <space separated list of CPU percentages>n"

SystemInfo message process:

1) The Client sends the SystemInfo message.
2) The Commander receives the message and performs the function on the remote machine.
3) The Commander sends the response message.
4) The Client receives the response message.

ProcessInfo

Description: Get certain Windows process information.

Behavior: The Commander will return the percentage CPU used for the process given the process name (not recommended) or the PID if the process name is empty.

Request Message	
"Message: ProcessInfo\n"	
Request Message Arguments	
None	
Response Messages	
1	"Message: Acknowledge\nProcessStatus: Stopped\n"
2	"Message: Acknowledge\nProcessInfo: <percentage cpu used>\n"
3 - Error	"Message: Acknowledge\nProcessInfo: Error with allocation\n"
4 - Error	"Message: Acknowledge\nProcessInfo: Process name is empty\n"
5 - Error	"Message: Acknowledge\nProcessInfo: PID is invalid\n"

ProcessInfo message process:

1) The Client sends the ProcessInfo message.
2) The Commander receives the message and performs the function on the remote machine.
3) The Commander sends the response message.

4) The Client receives the response message.

PathIsDirectory

Description: Check if the path sent from the Client is an actual path on the remote machine.

Behavior: The Commander will return True if the path string sent by the Client is an actual path on the remote machine, otherwise False.

Request Message	
"Message: "Message: PathIsDirectory\nPath: \n"	
Request Message Arguments	
Arg1	(Path) – The path string to verify on the remote machine.
Response Messages	
1	"Message: Acknowledge\nPathIsDirectory: True\n"
2	"Message: Acknowledge\nPathIsDirectory: False\n"
3 - Error	"Message: Acknowledge\nPathIsDirectory: Invalid Path\n"

PathIsDirectory Message Process:

1) The Client sends the PathIsDirectory message.
2) The Commander receives the message and performs the function on the remote machine.
3) The Commander sends the response message.
4) The Client receives the response message.

MakeDirectory

Description: Create the directory given the path sent from the Client.

Behavior: The Commander will return Success if the path was created on the remote machine, otherwise Failure.

Request Message	
"Message: MakeDirectory\nPath: \n"	
Request Message Arguments	
Arg1	(Path) – Path string to of directory to create on the remote machine.
Response Messages	
1	"Message: Acknowledge\nMakeDirectory: SUCCESS\n"
2	"Message: Acknowledge\nMakeDirectory: FAILURE\n"
3 - Error	"Message: Acknowledge\nMakeDirectory: Invalid Path\n"

MakeDirectory message process:

1) The Client sends the MakeDirectory message.
2) The Commander receives the message and performs the function on the remote machine.
3) The Commander sends the response message.
4) The Client receives the response message.

DirectoryHasData

Description: Check if the path sent from the Client already contains IADS data.

Behavior: The Commander will return True if the Path has data contained within in it, otherwise False.

Request Message	
"Message: DirectoryHasData\nPath: \n"	
Request Message Arguments	
Arg1	(Path) – Path string of directory to create on the remote machine.
Response Messages	
1	"Message: Acknowledge\nDirectoryHasData: True\n"
2	"Message: Acknowledge\nDirectoryHasData: False\n"
3 - Error	"Message: Acknowledge\nDirectoryHasData: Invalid Path\n"

DirectoryHasData message process:

1) The Client sends the DirectoryHasData message.
2) The Commander receives the message and performs the function on the remote machine.
3) The Commander sends the response message.
4) The Client receives the response message.

GetNextFreeDirectoryName

Description: The Commander will return a sequential backup directory name.

Behavior: The Commander will check the file system for the next available directory with a number appended to the name.

Request Message	
"Message: GetNextFreeDirectoryName\nPath: \n"	
Request Message Arguments	
Arg1	(Path) – Path string of directory to create on the remote machine.
Response Messages	
1	"Message: Acknowledge\nGetNextFreeDirectoryName: <next free directory name>\n"
2	"Message: Acknowledge\nGetNextFreeDirectoryName: FAILURE\n"
3 - Error	"Message: Acknowledge\nGetNextFreeDirectoryName: Invalid Path\n"

GetNextFreeDirectoryName message process:

1) The Client sends the GetNextFreeDirectoryName message.
2) The Commander receives the message and performs the function on the remote machine.
3) The Commander sends the response message.
4) The Client receives the response message.

FileExists

Description: The Commander will check if the file already exists on the remote machine.

Behavior: The Commander will check the existence of the file sent from the Client on the remote computer.

Request Message	
"Message: FileExists\nPath: \n"	
Request Message Arguments	
Arg1	(File) – File name and path to check for existence on the remote machine.
Response Messages	
1	"Message: Acknowledge\nFileExists: TRUE\n"
2	"Message: Acknowledge\nFileExists: FALSE\n"
3 - Error	"Message: Acknowledge\nFileExists: Unable to determine if file specified\n"

FileExists message process:

1) The Client sends the FileExists message.
2) The Commander receives the message and performs the function on the remote machine.
3) The Commander sends the response message.
4) The Client receives the response message.

C++ Source Code Example

Note: This example is for demonstration purposes only and may not compile as written here. Many languages, such as V, C#, C and C++ provide their own socket routines that will work with the IADS Commander.

```
char inMsg[2000] = {'\0'} ;
char outMsg[2000] = {'\0'} ;

// Creates a socket connection to the IADS Commander
ClientSocket* clientSocket = new ClientSocket( port, host ) ;
if( clientSocket == NULL )
{
   printf("Error: Socket Interface Creation Error...\n" ) ;
   exit( 1 ) ;
}

// Clients make an active connect
printf("- Waiting to Connect to Service\n" ) ;
if( clientSocket->Connect( -1 ) == 0 )
{
    printf("Error Client Unable to make Active Connect...\n" ) ;
```

```
    exit( 1 ) ;
}
printf("- Client Connected...\n\n" ) ;

char file[MAX_PATH] = {"c:\\SomePath\\SomeFile.txt"} ;
sprintf( outMsg, "Message: WritePermission\nNameAndPath: %s\n", file) ;

// Let commander know what's coming...
printf(" Send Message:\n%s", outMsg ) ;
if( clientSocket->Send( outMsg, 2000 ) == 0 )
{
    printf("ClientWorkstation -  Connection Terminted...\n" ) ;
    break ;
}

// Receive Response from the IADS Commander
printf("- Wait to Receive Acknowledge from Server...\n" ) ;
if( clientSocket->Recv( inMsg, 2000 ) == utFailure )
{
    printf("- Client Connection Terminted...\n" ) ;
    break ;
}
// Disconnect from the IADS Commander
delete clientSocket ;
```

7.2.2 The CDS Command Server

The CDS Command server is included as a capability within the CDS process. The CDS is part of the IADS Real-time system to be installed on the computer designated as the IADS Server. The purpose of the CDS Command server is to provide a socket-based interface to control the running operation of the CDS and provide various status information. There is a full set of commands to control the CDS application to perform functions such as; IADS configuration file validation, start data acquisition and stop data acquisition. In addition there is a full set of status commands such as; time validation state, processing status, and information about the upstream data source. The CDS Command server utilizes the TCP/IP protocol therefore the EUCCA will make an active connection and behave as a client making requests and receiving message replies from the Command server.

Setup Information

Field	*Entry/Selection*
Connection Port	58001
Connection Type	TCP/IP

Protocol

- Client performs a send on the socket
- The CDS Command Server performs the function and sends back a response message
- Client performs a receive on the socket to retrieve the response message
- The CDS does not guarantee a response message to be null terminated.

CDS Messages Overview

Message	*Description*
Initialization Commands and Information	
StartConnectTest	Command the CDS to begin the test
IsCdsDataSourceConnectTestCompleted	Check if the test is still running
GetCdsDataSourceConnectTestInfoString	Return information from the test
GetCdsDataSourceConnectTestResult	Return test results
GetCdsInitInfo	State of CDS initialization
GetCdsInitInfoString	Return information from the process
StartConfigValidationFromFile	Start config validation from start file
IsValidationComplete	State of validation processing
GetConfigValidateInfo	State of validation success
GetValidationStatusString	Return information from the process
Data Acquisition Commands and Information	
StartData	Start CDS data acquisition
RestartCds	Perform a CDS recovery
ResetCdsWithAppend	Reset the CDS and append data
ResetCdsWithSave	Reset the CDS and save
ResetCdsWithoutSave	Reset the CDS without saving
ResetCdsWithAppendBlocked	Reset the CDS with append blocked
ResetCdsWithSaveBlocked	Reset the CDS with save blocked
ResetCdsWithoutSaveBlocked	Reset the CDS without save blocked
IsDataStarted	Return data acquisition status
GetCdsDataGatherInfoString	Return information from the process
Stopping Data	
StopData	Stop CDS data acquisition
IsStopDataComplete	Return stop data acquisition status
GetCdsStopInfoString	Return information from the process
Time Information	
IsTimeValidated	Return time validation status
GetCdsTimeValidationInfoString	Return information from the process
GetIrigTime	Return current IRIG time from CDS

Archiving	
StartDataArchiving	Start data archiving
StartDataArchivingWithAppend	Start data archiving with append
StopDataArchiving	Stop data archiving
ResetDataArchiving	Reset data archiving
ResetDataArchivingWithAppend	Reset data archiving with append
IsDataArchiving	Return data archiving status
Nulling	
StartAircraftNulling	Start aircraft nulling
IsAircraftNulling	Return aircraft nulling status
StartWeaponsBayNulling	Start weapons bay nulling
IsWeaponsBayNulling	Return weapons bay nulling status
Data Compression	
StartDataCompression	Start data compression
IsDataCompressionRunning	Return state of process
GetDataCompressionStatus	Return data compression status
GetDataCompressionError	Return information from the process
StopDataCompression	Stop data compression
Shutdown	
Shutdown	Shutdown CDS
Run State	
GetCdsStartInfo	Return status of CDS start command
GetCdsStopInfo	Return status of CDS stop command
GetPredictedAggRate	Return predicted aggregate rate
Data Source Information	
GetSys500Info	Not yet documented
GetSys500DecomStatus	Not yet documented
GetOmegaInfo	Return Omega information
GetOmegaDecomStatus	Return Omega decom status
GetVistaInfo	Not yet documented
GetVistaDecomStatus	Not yet documented
GetS6200	Not yet documented
GetS6200DecomStatus	Not yet documented
GetMCSInfo	Not yet documented
GetMCSDecomStatus	Not yet documented
GetCustomDSInfo	Not yet documented
GetCustomDSDecomStatus	Not yet documented
GetDataInterfaceInfo	Not yet documented
GetDecomStatus	Not yet documented

System Wide Information	
GetCdsSystemInfo	Return certain system information
GetFrontEndType	Return primary front end type
GetNumberDecomStreams	Return the number of decom streams for the primary front end
GetBaseDataSourceInfo	Return active data source information

7.2.3 Initialization Commands and Information

StartConnectTest

Description: Command the CDS to begin a connect test to the upstream data source computer.

Behavior: The CDS will send a response message before the connect test is initiated. A failure condition is triggered if the CDS had already initiated a connect test in the past when the command is received. This message is not necessary for the CDS to successfully run. Subsequently use the "IsCdsDataSourceConnectTestComplete" to determine the state of the connect test.

Request Message	
"StartConnectTest"	
Response Messages	
1	"StartConnectTest ok"
2 - Error	"StartConnectTest failure Connect Test is already running"

StartConnectTest message process:

1) The Client sends the StartConnectTest message.
2) The CDS receives the message and responds back to the Client.
3) The CDS performs the function.
4) The Client receives the response message.

IsCdsDataSourceConnectTestCompleted

Description: Check if a CDS connect test has been completed.

Behavior: The CDS will send a response message of true if the connect test has completed, otherwise false.

Request Message	
"IsCdsDataSourceConnectTestCompleted"	
Response Messages	
1	"IsCdsDataSourceConnectTestCompleted ok <true, false>"

IsCdsDataSourceConnectTestCompleted message process:

1) The Client sends the IsCDSDataSourceConnectTestCompleted message.
2) The CDS receives the message, checks the status of the connect test and responds to the Client.

3) The Client receives the response message.

GetCdsDataSourceConnectTestInfoString

Description: Get information on the results of the CDS connect test.

Behavior: The CDS will send a response message with information about the connect test that was performed. Use this to obtain more detail on the connect test results.

Request Message	
"GetCdsDataSourceConnectTestInfoString"	
Response Messages	
1	"GetCdsDataSourceConnectTestInfoString ok <information string>"
2	"GetCdsDataSourceConnectTestInfoString ok NULL"

GetCdsDataSourceConnectTestInfoString message process:

1) The Client sends the GetCdsDataSourceConnectTestInfoString message.
2) The CDS receives the message and returns the connect test information string.
3) The Client receives the message.

GetCdsDataSourceConnectTestResult

Description: Get information on the results of the CDS connect test.

Behavior: The CDS will send a response message with results on the connect test that was performed. Use this after sensing the connect test has completed to determine whether the connect test succeeded (true) or failed (false) along with additional information upon failure.

Request Message	
"GetCdsDataSourceConnectTestResult"	
Response Messages	
1	"GetCdsDataSourceConnectTestResult ok false <code> <failure result string>"
2	"GetCdsDataSourceConnectTestResult ok <true, false>"

Note: Response message 2 can be false only if command is sent prior to completion of connect test.

GetCdsDataSourceConnectTestResult message process:

1) The Client sends the GetCdsDataSourceConnectTestResult message.
2) The CDS receives the message and returns the connect test result string.
3) The Client receives the message.

GetCdsInitInfo

Description: The Client sends this command to retrieve CDS initialization state information, specifically status on initialization completion and success.

Behavior: The CDS will send an ok token followed by two value strings of either "true" or "false". The first specifies status on initialization completion and the second states the initialization success result.

Request Message	
"GetCdsInitInfo"	
Response Messages	
1	“GetCdsInitInfo ok <true, false> <true, false>” Example: “GetCdsInitInfo ok true true”

GetCdsInitInfo message process:

1) The Client sends the GetCdsInitInfo message.
2) The CDS receives the message and returns the initialization string.
3) The Client receives the message.

GetCdsInitInfoString

Description: The Client sends this command to retrieve the CDS initialization information string. This may provide more detail on the results of the initialization performed.

Behavior: The CDS will send an ok token followed by either a NULL string if nothing is available or the information string.

Request Message	
"GetCdsInitInfoString"	
Response Messages	
1	“GetCdsInitInfoString ok <Information string>”
2	“GetCdsInitInfoString ok NULL”

GetCdsInitInfoString message process:

1) The Client sends the GetCdsInitInfoString message.
2) The CDS receives the message and returns the initialization information string.
3) The Client receives the message.

StartConfigValidationFromFile

Description: Start the configuration validation process.

Behavior: The CDS will send an Ok or Failure Response message immediately upon receipt of this command. A failure condition is triggered when the CDS is already in an active data acquisition state when the command is received. Additional messages will need to be sent to query validation state and results. This step is mandatory in order for the CDS to operate properly.

Request Message	
“StartConfigValidationFromFile”	
Response Messages	
1	“StartConfigValidationFromFile ok”
2 - Error	“StartConfigValidationFromFile failure CDS currently running”

StartConfigValidationFromFile Message Process:

1) The Client sends the StartConfigValidationFromFile message.
2) The CDS receives the message and returns a status before performing the validation.
3) The Client receives the message.

IsValidationComplete

Description: The Client sends this command to check the completion state of the config validation process.

Behavior: The CDS will send an Ok string followed by a true or false string depending if the validation is complete or not.

Request Message	
"IsValidationComplete"	
Response Messages	
1	"IsValidationComplete ok <true, false>"

IsValidationComplete message process:

1) The Client sends the IsValidationComplete message.
2) The CDS receives the message and returns a status of the validation process.
3) The Client receives the message.

GetConfigValidateInfo

Description: The Client sends this command to retrieve status whether the CDS has successfully performed validation.

Behavior: The CDS will send an Ok string followed by a true or false string depending if the config successfully validated or not.

Request Message	
"GetConfigValidateInfo"	
Response Messages	
1	"GetConfigValidateInfo ok <true, false>"

GetConfigValidateInfo message process:

1) The Client sends the GetConfigValidateInfo message.
2) The CDS receives the message and returns a status of the validation success.
3) The Client receives the message.

GetValidationStatusString

Description: The Client sends this command to retrieve the result string from the validation process. Typically, this is used to obtain additional information when validation failures occur.

Behavior: The CDS will send an Ok string followed by either NULL if no status string is available or the validation string.

Request Message	
"GetValidationStatusString"	
Response Messages	
1	"GetValidationStatusString ok <status string>"
2 - Error	"GetValidationStatusString ok NULL"

GetValidationStatusString message process:

1) The Client sends the GetValidationStatusString message.
2) The CDS receives the message and returns the validation string.
3) The Client receives the message.

7.2.4 Data Acquisition Commands and Information

StartData

Description: The client sends this command to start the CDS data acquisition.

Behavior: The CDS will send an "ok" token with nothing following when the CDS is in a state ready to start data acquisition otherwise a failure token is sent if the CDS is already in a data acquisition state. The CDS will send the response before the start is initiated in order not to block the requestor. A failure condition is triggered if the CDS is already in an active data acquisition state when the command is received. This command is required in order to run the CDS and gather data for real time operations.

Request Message	
"StartData"	
Response Messages	
1	"StartData ok"
2 - Error	"StartData failure Data is already running"

StartData message process:

1) The Client sends the "StartData" message.
2) The CDS receives the message and returns a response before initiating the start.
3) The Client receives the message.

RestartCDS

Description: The requestor sends this command to perform a CDS recovery. A recovery is defined as restarting the CDS from scratch but maintaining some states from the previous run such as null bias values along with bypassing certain initialization operations including IADS configuration file validation. All subsequent data archiving is appended to the same archive files created prior to the recovery action. Time validation is performed.

Behavior: The CDS will send an "ok" token upon success otherwise a "failure" token is sent. The CDS will send the response message before the restart is initiated in order not to block the requestor. A failure condition is triggered if the CDS is already in an active data acquisition state when the command is received.

Request Message	
"RestartCds"	
Response Messages	
1	"RestartCds ok"
2 - Error	"RestartCds failure Data is already running"

RestartCDS message process:

1) The Client sends the "RestartCds" message.
2) The CDS receives the message and returns a response before initiating the recovery action.
3) The Client receives the message.

Warning: This command may only be used immediately after execution – NO COMMAND MAY PROCEED THIS COMMAND EXCEPT GetCdsInitInfo. If any other command is used out of order prior to this, the CDS will not be in "recovery" mode.

ResetCdsWithAppend

Description: The Client sends this command to perform a CDS reset. On a reset the CDS stays in a running state but suspends the data acquisition connection while flushing all data to disk following by resuming the data acquisition process and reapplying time validation. Config file validation is not performed. All subsequent data archiving will be appended to the existing archive files.

Behavior: The CDS will send an Ok token with nothing following on success otherwise a failure token is sent. The CDS will send the response message before the start is initiated in order not to block the requestor. A failure condition is triggered if the CDS is not in an active data acquisition state when the command is received.

Request Message	
"ResetCdsWithAppend"	
Response Messages	
1	"ResetCdsWithAppend ok"
2 - Error	"ResetCdsWithAppend failure Data is already running"

ResetCdsWithAppend message process:

1) The Client sends the "ResetCdsWithAppend" message.
2) The CDS receives the message and returns a response before initiating the reset.
3) The Client receives the message.

ResetCdsWithSave

Description: The Client sends this command to perform a CDS reset. On a reset the CDS stays in a running state but suspends the data acquisition connection while flushing all data to disk following by resuming the data acquisition process and reapplying time validation. Config file validation is not performed. The current archive folder is renamed by appending "RestoredN" to the folder name where N is the next unused number starting at 1. A new archive folder will be

created using the original name populated with the archive support files from the original folder and new archive data files will be created.

Behavior: The CDS will send an Ok token with nothing following on success otherwise a failure token is sent. The CDS will send the response message before the reset is initiated in order not to block the requestor. A failure condition is triggered if the CDS is not in an active data acquisition state when the command is received.

Request Message	
"ResetCdsWithSave"	
Response Messages	
1	"ResetCdsWithSave ok"
2 - Error	"ResetCdsWithSave failure Data is already running"

Message Process:

1) The Client sends the "ResetCdsWithSave" message.
2) The CDS receives the message and returns a response before initiating the reset.
3) The Client receives the message.

ResetCdsWithoutSave

Description: The Client sends this command to perform a CDS reset. On a reset the CDS stays in a running state but suspends the data acquisition connection while flushing all data to disk following by resuming the data acquisition process and reapplying time validation. Config file validation is not performed. The currently saved data will be discarded and the archive files will be truncated.

Behavior: The CDS will send an Ok token with nothing following on success otherwise a failure token is sent. The CDS will send the response message before the reset is performed in order to not block the requestor. A failure condition is triggered if the CDS is not in an active data acquisition state when the command is received.

Request Message	
"ResetCdsWithoutSave"	
Response Messages	
1	"ResetCdsWithoutSave ok"
2 - Error	"ResetCdsWithoutSave ok failure Data is already running"

ResetCdsWithoutSave message process:

1) The Client sends the "ResetCdsWithoutSave" message.
2) The CDS receives the message and returns a response before initiating the reset.
3) The Client receives the message.

ResetCdsWithAppendBlocked

Description: The Client sends this command to perform a CDS reset. On a reset the CDS stays in a running state but suspends the data acquisition connection while flushing all data to disk following by resuming the data acquisition process and reapplying time validation. Config file

validation is not performed. All subsequent data archiving will be appended to the existing archive files.

Behavior: The CDS will send an Ok token with nothing following on success otherwise a failure token is sent. The CDS will delay sending the response message until after the reset is completed blocking the requester in the process. A failure condition is triggered if the CDS is not in an active data acquisition state when the command is received.

Request Message	
"ResetCdsWithAppendBlocked" (Reset performed before response)	
Response Messages	
1	"ResetCdsWithAppendBlocked ok"
2 - Error	"ResetCdsWithAppendBlocked failure Data is already running"

ResetCdsWithAppendBlocked message process:

1) The Requestor sends the "ResetCdsWithAppendBlocked" message.
2) The CDS receives the message and returns a response after completing the reset.
3) The Requestor receives the message.

ResetCdsWithSaveBlocked

Description: The Client sends this command to perform a CDS reset. On a reset the CDS stays in a running state but suspends the data acquisition connection while flushing all data to disk following by resuming the data acquisition process and reapplying time validation. Config file validation is not performed. The current archive folder is renamed by appending "RestoredN" to the folder name where N is the next unused number starting at 1. A new archive folder will be created using the original name populated with the archive support files from the original folder and new archive data files will be created.

Behavior: The CDS will send an Ok token with no other tokens following on success otherwise a failure token is sent. The CDS will delay sending the response message until after the reset is completed blocking the requestor in the process. A failure condition is triggered if the CDS is not in an active data acquisition state when the command is received.

Request Message	
"ResetCdsWithSaveBlocked"	
Response Messages	
1	"ResetCdsWithSaveBlocked ok"
2 - Error	"ResetCdsWithSaveBlocked failure Data is already running"

ResetCdsWithSaveBlocked message process:

1) The Client sends the "ResetCdsWithSaveBlocked" message.
2) The CDS receives the message and returns a response after completing the reset.
3) The Client receives the message.

ResetCdsWithoutSaveBlocked

Description: The Client sends this command to perform a CDS reset. On a reset the CDS stays in a running state but suspends the data acquisition connection while flushing all data to disk following by resuming the data acquisition process and reapplying time validation. Config file validation is not performed. The currently saved data will be discarded and the archive files will be truncated.

Behavior: The CDS will send an Ok token with nothing following on success otherwise a failure token is sent. The CDS will send the response message after the reset is performed blocking the requestor in the process. A failure condition is triggered if the CDS is not in an active data acquisition state when the command is received.

Request Message	
"ResetCdsWithoutSaveBlocked"	
Response Messages	
1	"ResetCdsWithoutSaveBlocked ok"
2 - Error	"ResetCdsWithoutSaveBlocked failure Data is already running"

ResetCdsWithoutSaveBlocked message process:

1) The Client sends the "ResetCdsWithoutSaveBlocked" message.
2) The CDS receives the message and returns a response after completing the reset.
3) The Client receives the message.

IsDataStarted

Description: The Client sends the command to determine whether CDS data acquisition has been started.

Behavior: The CDS will send the "ok" token with either a "true" token if data is started or a "false" token if data is not started.

Request Message	
"IsDataStarted"	
Response Messages	
1	"IsDataStarted ok <true, false>"

IsDataStarted message process:

1) The Client sends the "IsDataStarted" message.
2) The CDS receives the message and returns a response.
3) The Client receives the message.

GetCdsDataGatherInfoString

Description: The Requestor sends this command to obtain data gather information from the CDS.

Behavior: The CDS will send the "ok" token followed by the information string or "NULL" token if no information is available.

Request Message
"GetCdsDataGatherInfoString"

Response Messages	
1	"GetCdsDataGatherInfoString ok <information string>"
2	"GetCdsDataGatherInfoString ok NULL"

GetCdsDataGatherInfoString message process:

1) The Client sends the "GetCdsDataGatherInfoString" message.
2) The CDS receives the message and returns a response.
3) The Client receives the message.

7.2.5 Stopping Data Command and Information

StopData

Description: The Client sends this command to stop the CDS data acquisition. All data acquisition and processing activities will be discontinued. All data not currently archived will be flushed to disk.

Behavior: The CDS will send an "ok" token on success otherwise a failure token is sent. The CDS will send the response before the stop is initiated in order not to block the requestor. A failure condition is triggered if the CDS is not in an active data acquisition state when the command is received.

Request Message	
"StopData"	
Response Messages	
1	"StopData ok"
2 - Error	"StopData failure Data is not currently running"

StopData message process:

1) The Requestor sends the "StopData" message.
2) The CDS receives the message and returns a response before initiating the stop.
3) The Requestor receives the message.

IsStopDataComplete

Description: The Client sends this command to check if the CDS has stopped data acquisition.

Behavior: The CDS will send the "ok" token followed by either a "true" token if data is stopped or a "false" token if data is not stopped.

Request Message	
"IsStopDataComplete"	
Response Messages	
1	"IsStopDataComplete ok <true, false>"

IsStopDataComplete message process:

1) The Client sends the "IsStopDataComplete" message.
2) The CDS receives the message and returns a response.

3) The Client receives the message.

GetCdsStopInfoString

Description: The Client sends this command to get the stop data status information string from the CDS.

Behavior: The CDS will send the “ok” token followed by either the information string or “NULL” token if no information is available.

Request Message	
“GetCdsStopInfoString”	
Response Messages	
1	“GetCdsStopInfoString ok NULL”
2	“GetCdsStopInfoString ok <information string>”

GetCdsStopInfoString message process:

1) The Client sends the “GetCdsStopInfoString” message.
2) The CDS receives the message and returns a response.
3) The Client receives the message.

7.2.6 Time Information

IsTimeValidated

Description: The Requestor sends this command to check if the CDS has successfully validated the upstream data source time flow.

Behavior: The CDS will send the “ok” token followed by either a “true” token if the data source time validation succeeded or a “false” token if not.

Request Message	
“IsTimeValidated”	
Response Messages	
1	“IsTimeValidated ok <true, false>”

IsTimeValidated message process:

1) The Client sends the “IsTimeValidated” message.
2) The CDS receives the message and returns a response.
3) The Client receives the message.

GetCdsTimeValidationInfoString

Description: The Client sends this command to time validation information string from the CDS.

Behavior: The CDS will send the “ok” token followed by the information string or “NULL” token if no information is available.

Request Message	
"GetCdsTimeValidationInfoString"	
Response Messages	
1	"GetCdsTimeValidationInfoString ok NULL"
2	"GetCdsTimeValidationInfoString ok <information string>"

GetCdsTimeValidationInfoString message process:

1) The Client sends the "GetCdsTimeValidationInfoString" message.
2) The CDS receives the message and returns a response.
3) The Client receives the message.

GetIRIGTime

Description: The Requestor sends this command to retrieve the current IRIG time from the CDS.

Behavior: The CDS will send the "ok" token followed by the IADS IRIG time string. The returned IRIG string format is DDD:HH:MM:SS.sss.

Request Message	
"GetIRIGTime"	
Response Messages	
1	"GetIRIGTime ok <IRIG time string>"

GetIRIGTime message process:

1) The Client sends the "GetIRIGTime" message.
2) The CDS receives the message and returns a response.
3) The Client receives the message.

7.2.7 Archiving Commands and Information

StartDataArchiving

Description: The Client sends this command to start data archiving. Any previously saved data will be discarded and the archive data files will be truncated.

Behavior: The CDS will send an "ok" token on success otherwise a failure token is sent. The CDS will send the response after the data archiving is initiated. A failure condition is triggered if the CDS is already in an active data archiving state when the command is received.

Request Message	
"StartDataArchiving"	
Response Messages	
1	"StartDataArchiving ok Data archiving is now active"
2 - Error	"StartDataArchiving failure Data archiving is already active"

StartDataArchiving message process:

1) The Client sends the "StartDataArchiving" message.
2) The CDS receives the message and starts data archiving.

3) The CDS sends a response message.
4) The Client receives the message.

StartDataArchivingWithAppend

Description: The Client sends this command to start data archiving. All subsequent data archiving will be appended to the existing archive files.

Behavior: The CDS will send an "ok" token on success otherwise a failure token is sent. The CDS will send the response after the data archiving is initiated. A failure condition is triggered if the CDS is already in an active data archiving state when the command is received.

Request Message	
"StartDataArchivingWithAppend"	
Response Messages	
1	"StartDataArchivingWithAppend ok Data archiving is now active"
2 - Error	"StartDataArchivingWithAppend failure Data archiving is already active"

StartDataArchivingWithAppend message process:

1) The Client sends the "StartDataArchivingWithAppend" message.
2) The CDS receives the message and starts data archiving.
3) The CDS sends a response message.
4) The Client receives the message.

StopDataArchiving

Description: The Client sends this command to stop data archiving. All data not currently archived will be flushed to disk and the CDS will discontinue saving data.

Behavior: The CDS will send an "ok" token on success otherwise a failure token is sent. The CDS will send the response after flushing the data and archiving is stopped. A failure condition is triggered if the CDS is not in an active data archiving state when the command is received.

Request Message	
"StopDataArchiving"	
Response Messages	
1	"StopDataArchiving ok Data archiving is now inactive"
2 - Error	"StopDataArchiving failure Data archiving is already inactive"

StopDataArchiving message process:

1) The Requestor sends the "StopDataArchiving" message.
2) The CDS receives the message and stops data archiving.
3) The CDS sends a response message.
4) The Requestor receives the message.

ResetDataArchiving

Description: This commands the CDS to stop data archiving followed by restarting data archiving. Any saved data will be discarded and the archive data files will be truncated.

Behavior: The CDS will send an “ok” token after data archiving has been fully stopped and the restart of data archiving has been initiated.

Request Message	
“ResetDataArchiving”	
Response Messages	
1	“ResetDataArchiving ok Data archiving is now active”

ResetDataArchiving message process:

1) The Requestor sends the “ResetDataArchiving” message.
2) The CDS receives the message and resets data archiving.
3) The CDS sends a response message.
4) The Requestor receives the message.

ResetDataArchivingWithAppend

Description: This commands the CDS to stop data archiving followed by restarting data archiving. All subsequent data archiving will be appended to the existing archive files.

Behavior: The CDS will send an “ok” token after data archiving has been fully stopped and the restart of data archiving has been initiated.

Request Message	
“ResetDataArchivingWithAppend”	
Response Messages	
1	“ResetDataArchivingWithAppend ok Data archiving is now active”

ResetDataArchivingWithAppend message process:

1) The Requestor sends the “ResetDataArchivingWithAppend” message.
2) The CDS receives the message and resets data archiving.
3) The CDS sends a response message.
4) The Requestor receives the message.

IsDataArchiving

Description: This command will check if the CDS is archiving data.

Behavior: The CDS will send an “ok” token along with a true token if data is currently being archived, otherwise a false token is returned.

Request Message	
“IsDataArchiving”	
Response Messages	
1	“IsDataArchiving ok <true, false>”

Message Process:

1) The Requestor sends the “IsDataArchiving” message.
2) The CDS sends a response message.
3) The Requestor receives the message.

7.2.8 Nulling Commands and Information

StartAircraftNulling

Description: This commands the CDS to start Aircraft group nulling. The CDS will collect 15 seconds worth of data for all parameters within the Aircraft nulling group, compute the average value over that time span then calculate the bias based on the difference between the baseline and average values and update the config with the results.

Behavior: The CDS will send an “ok” token on success otherwise a failure token is sent. The CDS will send the response after initiating the nulling process in order to not block the requestor. A failure condition is triggered if the CDS is unable to initiate the nulling process due to system error.

Request Message	
“StartAircraftNulling”	
Response Messages	
1	“StartAircraftNulling ok Nulling sequence is now started”
2 - Error	“StartAircraftNulling failure Nulling sequence could not be started”

StartAircraftNulling Message Process:

1) The Requestor sends the “StartAircraftNulling” message.
2) The CDS sends a response message.
3) The Requestor receives the message.

IsAircraftNulling

Description: The Client sends this command to check whether the CDS has completed Aircraft group nulling.

Behavior: The CDS will send an “ok” token along with a true token if Aircraft group nulling has been completed, otherwise a false token is returned.

Request Message	
“IsAircraftNulling”	
Response Messages	
1	“IsAircraftNulling ok <true, false>”

IsAircraftNulling message process:

1) The Requestor sends the “IsAircraftNulling” message.
2) The CDS sends a response message.
3) The Requestor receives the message.

StartWeaponsBayNulling

Description: This commands the CDS to start Weapons Bay nulling. The CDS will collect 15 seconds worth of data for all parameters within the Weapons Bay nulling group, compute the average value over that time span then calculate the bias based on the difference between the baseline and average values and update the config with the results.

Behavior: The CDS will send an "ok" token on success otherwise a failure token is sent. The CDS will send the response after initiating the nulling process in order to not block the requestor. A failure condition is triggered if the CDS is unable to initiate the nulling process due to system error.

Request Message	
"StartWeaponsBayNulling"	
Response Messages	
1	"StartWeaponsBayNulling ok Nulling sequence is now started"
2 - Error	"StartWeaponsBayNulling failure Nulling sequence could not be started"

StartWeaponsBayNulling message process:

1) The Requestor sends the "StartWeaponsBayNulling" message.
2) The CDS sends a response message.
3) The Requestor receives the message.

IsWeaponsBayNulling

Description: The Client sends this command to check whether the CDS has completed Weapons Bay group nulling.

Behavior: The CDS will send an "ok" token along with a true token if Weapons Bay group nulling has been completed, otherwise a false token is returned.

Request Message	
"IsWeaponsBayNulling"	
Response Messages	
1	"IsWeaponsBayNulling ok <true, false>"

IsWeaponsBayNulling message process:

1) The Requestor sends the "IsWeaponsBayNulling" message.
2) The CDS sends a response message.
3) The Requestor receives the message.

7.2.9 Data Compression Commands and Information

StartDataCompression

Description: This command will cause the CDS to start building a zip file of the IADS data archive.

Behavior: The CDS will return a "Starting" message to the requestor.

Request Message
"StartDataCompression"

Response Messages	
1	"StartDataCompression ok Starting"
2 - Error	"StartDataCompression failure Already Running"

StartDataCompression message process:

1) The Requestor sends the "StartDataCompression" message.
2) The CDS sends a response message after starting the compression.
3) The Requestor receives the message.

IsDataCompressionRunning

Description: This command will cause the CDS to return if the data compression is running and at what state.

Behavior: The CDS will return a true token and the particular process status or a False token of the compression is finished.

Request Message	
"IsDataCompressionRunning"	
Response Messages	
1	"IsDataCompressionRunning True Compressing"
2	"IsDataCompressionRunning True Building"
3	"IsDataCompressionRunning True Copying"
4	"IsDataCompressionRunning False Finished"

IsDataCompressionRunning message process:

1) The Requestor sends the "IsDataCompressionRunning" message.
2) The CDS sends a response message.
3) The Requestor receives the message.

GetDataCompressionStatus

Description: This command will cause the CDS to return the data compression processing status.

Behavior: The CDS will return a true token and the particular process status or a False token of the compression is finished.

Request Message	
"GetDataCompressionStatus"	
Response Messages	
1	"GetDataCompressionStatus Ok <TotalFilesToCompress,TotalFilesCompressed>"

GetDataCompressionStatus Message Process:

1) The Requestor sends the "GetDataCompressionStatus" message.
2) The CDS sends a response message.
3) The Requestor receives the message.

GetDataCompressionError

Description: This command will cause the CDS to return the data compression error string.

Behavior: The CDS will return a “True” token followed by the error string if an error did occur; otherwise a “False” token is sent indicating no error.

Request Message	
“GetDataCompressionError”	
Response Messages	
1	“GetDataCompressionError ok True <Error String>”
2	“GetDataCompressionError ok False”

GetDataCompressionError message process:

1) The Requestor sends the “GetDataCompressionError” message.
2) The CDS sends a response message.
3) The Requestor receives the message.

StopDataCompression

Description: This command will cause the CDS to start building a zip file of the IADS data archive.

Behavior: The CDS will return a “Stopping” message back to the requestor.

Request Message	
“StopDataCompression”	
Response Messages	
1	“StopDataCompression ok Stopping”
2 - Error	“StopDataCompression failure Already Stopped”
3 - Error	“StopDataCompression failure Stop command rejected”

StopDataCompression message process:

1) The Requestor sends the “StopDataCompression” message.
2) The CDS sends a response message after stopping the compression.
3) The Requestor receives the message.

ShutDown

Description: This commands the CDS to gracefully shutdown and exit the application.

Behavior: The CDS will return an “ok” token back to the requestor prior to initiating the shutdown.

Request Message	
“ShutDown”	
Response Messages	
1	“ShutDown ok”

ShutDown message process:

1) The Requestor sends the “ShutDown” message.

2) The CDS sends a response message before shutting down and exiting.
3) The Requestor receives the message.

7.2.10 Run State Information

GetCdsStartInfo

Description: This commands the CDS to return the completion and success status of the CDS data acquisition startup process initiated by the "StartData" command.

Behavior: The CDS will return an "ok" token followed by two status tokens representing CDS data acquisition startup process completion and success with the former stating "true" if the CDS has completed the data start process, otherwise "false" and the latter stating "true" if the CDS data start process was successful, otherwise "false".

Request Message	
"GetCdsStartInfo"	
Response Messages	
1	"GetCdsStartInfo ok <true, false> <true, false>"

GetCdsStartInfo message process:

1) The Requestor sends the "GetCdsStartInfo" message.
2) The CDS sends a response message.
3) The Requestor receives the message.

GetCdsStopInfo

Description: This commands the CDS to return the completion status of the CDS stop data acquisition process initiated by the "StopData" or "ShutDown" commands.

Behavior: The CDS will return an "ok" token followed by "true" if the CDS has completed the data stop process, otherwise "false".

Request Message	
"GetCdsStopInfo"	
Response Messages	
1	"GetCdsStopInfo ok <true, false>"

GetCdsStopInfo message process:

1) The Requestor sends the "GetCdsStopInfo" message.
2) The CDS sends a response message.
3) The Requestor receives the message.

GetPredictedAggRate

Description: This commands the CDS to return the predicted aggregate sample rate of the active data source parameters being processed by the CDS based on the ParameterDefaults table of the configuration file.

Behavior: The CDS will return an "ok" token followed by the predicted aggregate rate.

Request Message	
"GetPredictedAggRate"	
Response Messages	
1	"GetPredictedAggRate ok <Predicted Aggregate Rate>"

GetPredictedAggRate Message Process:

1) The Requestor sends the "GetPredictedAggRate" message.
2) The CDS sends a response message.
3) The Requestor receives the message.

7.2.11 Data Source Information

GetOmegaInfo

Description: This commands the CDS to return data flow information when attached to an Omega 3000 data source.

Behavior: The CDS will return an "ok" token followed by four tokens: the overflow count returned from the data source to the Omega IOM during data packet reads, the aggregate sample count from all data packets received up to this point, the aggregate data packet count and the actual sample rate aggregate from the data source.

Request Message	
"GetOmegaInfo"	
Response Messages	
1	"GetOmegaInfo ok <Overflow count> <Sample count> <Buffer Count> <Aggregate sample rate>"

GetOmegaInfo message process:

1) The Requestor sends the "GetOmegaInfo" message.
2) The CDS sends a response message.
3) The Requestor receives the message.

GetOmegaDecomStatus

Description: This commands the CDS to return the values of all decom status words defined in the active parameter list of the upstream Omega project.

Behavior: The CDS will return an "ok" token along with the total number of decom status words followed by a series of stream names and raw status words for each instance separated by "|".

Request Message	
"GetOmegaInfo"	
Response Messages	
1	"GetOmegaDecomStatus ok <Total status words> <Status number> <Stream name> <Status word> | <Status number> <Stream name> <Status word> | …"

GetOmegaDecomStatus Message Process:

1) The Requestor sends the "GetOmegaDecomStatus" message.

2) The CDS sends a response message.
3) The Requestor receives the message.

7.2.12 System-wide Information

GetCdsSystemInfo

Description: This commands the CDS to return the various system information.

Behavior: The CDS will return an “ok” token followed by various system information tokens separated by spaces.

Data Source Specific: No

Request Message	
“GetCdsSystemInfo”	
Response Messages	
1	“GetCdsSystemInfo ok <CPU 1 Utilization> <CPU 2 Utilization> <CPU 3 Utilization> <CPU 4 Utilization> <Total primary disk size in bytes> <Free primary disk size in bytes> <Total auxiliary disk size in bytes> <Free auxiliary disk size in bytes> <Total virtual memory> <Used page memory> <Total physical memory> <Used physical memory> <Current IRIG time> <Data source control type (file, network)> <CDS version> <Series of CPU number and utilization for all CPUs> <Data acquisition internal queue length> <Archive request internal queue length> <Archive write internal queue length> <OA IAP internal queue length> <Config update internal queue length>”

GetCdsSystemInfo Message Process:

1) The Requestor sends the “GetCdsSystemInfo” message.
2) The CDS sends a response message.
3) The Requestor receives the message.

GetFrontEndType (Primary source only)

Description: This commands the CDS to return the data source type. This currently only specifies the primary data source.

Behavior: The CDS will return an “ok” token followed by the primary data source type.

Data Source Specific: Not currently

Request Message	
“GetFrontEndType”	
Response Messages	
1	“GetFrontEndType ok <OS90,MCS,SYS500,OMEGA,VISTA,S6200,CUSTOM,Unknown>”

GetFrontEndType message process:

1) The Requestor sends the “GetFrontEndType” message.
2) The CDS sends a response message.
3) The Requestor receives the message.

GetNumDecomStreams

Description: This commands the CDS to return the total number of decom status streams as defined by the upstream data sources.

Behavior: The CDS will return an “ok” token followed by the number of decom status streams.

Data Source Specific: Not currently

Request Message	
“GetNumDecomStreams”	
Response Messages	
1	“GetNumDecomStreams ok <Total decom status streams>”

GetNumDecomStreams message process:

1) The Requestor sends the “GetNumDecomStreams” message.
2) The CDS sends a response message.
3) The Requestor receives the message.

GetBaseDataSourceInfo (All data sources)

Description: This commands the CDS to provide various information for all active data sources.

Behavior: The CDS will return an “ok” token along with the total number of active data sources followed by a series of data source types and number of decom status streams for each instance separated by “|”.

Data Source Specific: Yes

Request Message	
“GetBaseDataSourceInfo”	
Response Messages	
1	“GetBaseDataSourceInfo ok <Total data sources> | <Data source 1 type, Number decom status streams> | <Data source 2 type, Number decom status streams>…”

GetBaseDataSourceInfo Message Process:

1) The Requestor sends the “GetBaseDataSourceInfo” message.
2) The CDS sends a response message.
3) The Requestor receives the message.

C++ Source Code Example

Note: This example is for demonstration purposes only and may not to compile as written.

Note: Many languages, such as V, C#, C and C++ provide their own socket routines that will work with the IADS CDS Command interface.

```
// Creates a socket connection to the IADS Commander
ClientSocket* clientSocket = new ClientSocket( port, host ) ;
if( clientSocket == NULL )
{
    printf("Error: Socket Interface Creation Error...\n" ) ;
```

```
    exit( 1 ) ;
}

// Clients make an active connect
printf("- Waiting to Connect to Service\n" ) ;
if( clientSocket->Connect( -1 ) == 0 )
{
    printf("Error Client Unable to make Active Connect...\n" ) ;
    exit( 1 ) ;
}
printf("- Client Connected...\n\n" ) ;

// Construct and send the message
char message[500];
strcpy( message, "IsTimeValidated") ;

if( clientSocket->send(message, 500) == 0 )
{
    return 0 ;
}

char responsee[500] ;

// CDS doesn't guarantee the message to be null terminated
response[499] = '\0';

if( clientSocket->recv(response, 500) == 0 )
{
    return 0;
}

// Token 1 is the message name
// Token 2 is the Ok or Failure token
// Token 3 is the response
char token1[500], token2[500], token3[500];

// Replace with your own get token function
GetToken( response, token1, 0, ' ');
GetToken( response, token2, 1, ' ');
GetToken( response, token3, 2, ' ');
```

```
// Check if we got an Ok or Failure
if( strcmpi(token1, "failure")==0)
{
    return 0 ;
}

// output the response to the console
// true means the CDS has validated time, false means it has not.
printf("Response Message: %s\n", token3);

delete clientSocket;
```

Undocumented Messages

Message	*Description*
StartStatistics	Not released
StopStatistics	Not released
IsStaticsOn	Not released
IsStaticsConnected	Not released
GetParameterStatOverflows	Not released
GetBeloBoxInfo	Obsolete
GetBeloBoxDecomStatus	Obsolete

7.2.13 Startup IADS Command Line Options

The following command line options are necessary if using the IADS Commander to start the IADS CDS or IADS Client applications in a real time environment. A complete list of all IADS command line options is available in the IADS Help System.

IADS Client: Use only one of the following startup arguments, either /server or /startupFile.

Argument
/server HOSTNAME *For example:* /server IADS-CDS
/startupFile FILEPATH *For example:* /startupFile C:\ProgramFiles\IADS\ClientWorkstation\Client.iads.iadsStartupFile

IADS CDS:

Argument
/startupFile FILEPATH *For example:* /startupFile ….\IADS\ComputeDataServer\CDS.ComputeDataServer.iadsStartupFile

7.3 IADS Server (CDS) Data Throughput Performance Testing

This tutorial provides instruction on setting up the CdsStress program in order to test data throughput performance, including total CPU, memory access and the archiving system of the CDS on your server PC. This program does NOT test the complete system performance including functions such as nulling, client data access and database updates in real time.

The CdsStress program was written to help end users determine their system's performance capabilities for the CDS. Currently on machines such as Dell's 2950 the CDS can process 64000 parameters at 1 Mega sample aggregate data rate. Performance may vary depending on the Server's capabilities and the network performance to the data source sender.

7.3.1 Overview

The CdsStress kit is available for download on the Curtiss Wright IADS website at https://iads.symvionics.com/support/programming-examples/ Data Processing Examples: 3. CDS Performance Analysis Program and includes:

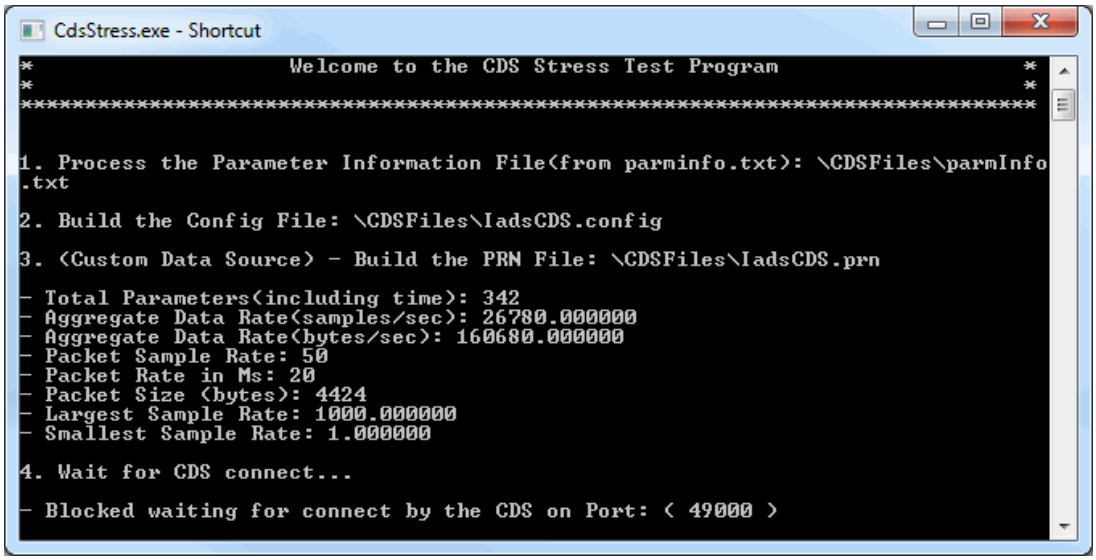
1) CdsStress.exe - The simulated data source program. It easily allows increasing the data throughput, the number of parameters and the sample rate mix to efficiently simulate a real-world data scenario; it automatically creates the IADS Configuration file (the IADS database) and the CDS parameter definitions (PRN) files used by the CDS.
2) parmInfo.txt -The input file to the CdsStress program for setting up the various data output rates. This file is edited by the user to set the data throughput rate of the CdsStress program.
3) iadsCDS.init - The input file to set the CDS run-time properties settings.

Note: This kit does not include the IADS Server (ComputeDataServer.exe) or IADS Client (Iads.exe) executables.

7.3.2 To run the data throughput test

1) Download the CdsStress program kit referenced above and unzip all the files onto your system. If you put all files in a directory you create at **C:\CdsFiles** the IadsCds.init will not need to be modified greatly.
2) Right-click on the CdsStress.exe > **Create Shortcut**.
3) Right-click On the CdsStress.exe - Shortcut > **Properties**. At the end of the Target line enter: **CDS/StartupFile c:\CdsFiles\IadsCDS.init**

4) Double-click on the shortcut to run the CdsStress program. The IADS Configuration file and PRN file will output to the CdsFiles directory.



5) Modify the “LOCATION1” property in the iadsCDS.init file to point to the correct data archive file path.

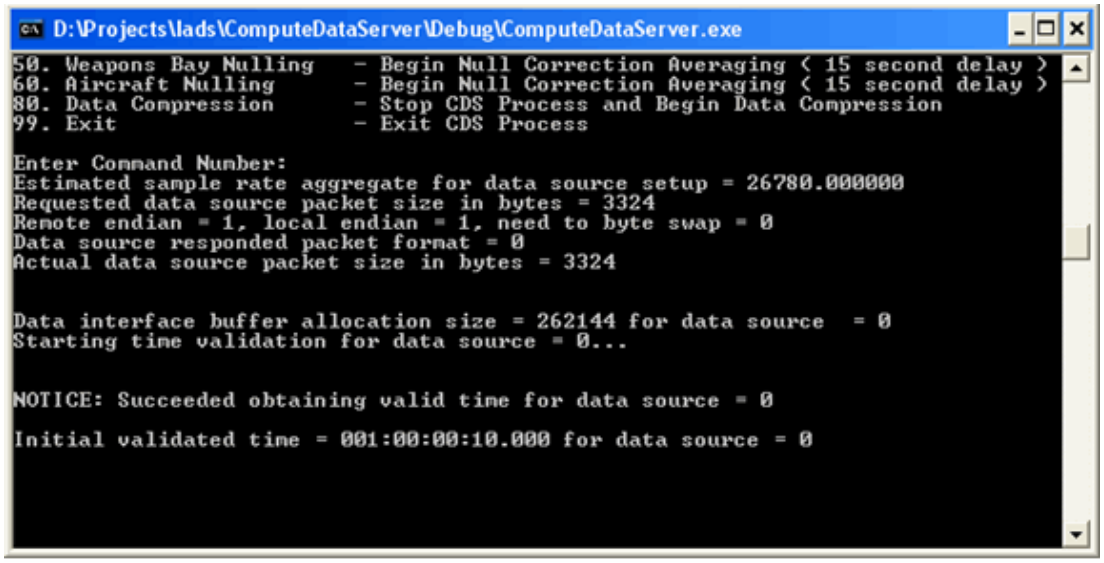
LOCATION1 = C:/CdsFiles/IadsOutputFiles

POSTFLIGHTCONFIG = C:/CdsFiles/IadsOutputFiles/pfConfig

6) Modify the “DATALOCATION” property to point to the PC that the CdsStress program is running on (The ”Port Id” does not need to be modified).

DATALOCATION = Pat3600 49000

7) Run the CDS. The CDS uses the property settings in the IadsCds.init file to locate the prn and config files; and connect to the CdsStress data source program. Enter a “20” on the CDS screen to validate the IADS Configuration. Once complete, enter a “30” to start real time. If everything is setup correctly then the CDS will start receiving data and validate time.



The primary test is to run the system and monitor CDS memory usage. Because the CDS is designed to use memory on a demand basis any overrun conditions are determined by an ever increasing memory usage on the Server PC which can be monitored by using the Windows Task Manager. Each test may take up to an hour before memory usage stabilizes.

Another test is to connect an IADS Client to the CDS and examine the IRIG time on the dashboard with time output of the CdsStress program and verify that they continue to match.

8) Enter a “99” on the CDS menu and the application will shut down. This may take a few minutes to close the archive files. The CdsStress program will then be ready for another connect, therefore it can continue to run. However it will need to be re-launched if another data throughput set is setup in the parmInfo.txt file.

8. Other

8.1 Iadsread Matlab Extension

The iadsread.mexw32 and iadsread.mexw64 MEX-files are included as part of the IADS installation at \Program Files (x86)\Iads\MatlabExtention. The iadsread function allows you to programmatically access your IADS archive data so you can write Matlab programs to read in and process the IADS flight data.

To set the path in Matlab to your IADS Matlab Extension directory:

1) Run Matlab.
2) Click the **File** drop down > **Set Path...**
3) In the Set Path dialog, click the **Add Folder** button.
4) Navigate to C:\ProgramFiles\Iads\MatlabExtention and click **OK**.
5) Click the **Save** button.
6) Click the **Close** button.

To verify the iadsread function is available in Matlab:

In Matlab, enter **iadsread** in the Command Window. It should respond: ??? iadsread: Minimum four inputs required. This is correct! The error occurs because the function call arguments are not complete; follow the instructions below to setup the iadsread function. If ??? Undefined function or variable 'iadsread' is returned, verify the path you have set and saved in Matlab is the MatlabExtention directory that contains your iadsread.mexw32/64 or the iadsread.dll. If the error still occurs, the version of Matlab you are using (pre 7.1) does not recognize the iadsread.mexw32 file. Rename the iadsread.mexw32 to iadsread.dll. For more information on this subject go to: http://www.mathworks.com/access/helpdesk/help/techdoc/rn/f26-998197.html

To use the iadsread function:

In Matlab, enter the iadsread function with a required minimum of four inputs (with the exception of the iadsread('DataDirectory')

Syntax

Variable = iadsread('DataDirectory or ServerName$PortId', 'IrigStartTime', 'IrigEndTime' or NumSeconds, 'ParameterNameList (Comma Separated)', [optional arguments..])

Examples

Data = iadsread('D:\PostTestData\TestSet','001:00:05:05',5,'AB1001X,AB1002X,AB1003X')

Data = iadsread('D:\PostTestData\TestSet','001:00:05:05',5,'AB1001X,AB1002X,AB1003X','DecimationFactor',4,'ReturnTimeVector',1)

Notice that all 5 parameters are combined into 1 matrix called Data. That is because only 1 variable was assigned to the result of iadsread, Data = iadsread(...). To create 3 separate vectors, define the left hand side of the equation as such: [AB1001X,AB1001X,AB1001X] = iadsread(...)

Syntax	*Example*	*Result*
Note: If you assign the output to a variable, for example, **Data = iadsread(...)** it will return the results in a structure ('struct array'). You can then use the Plot function in Matlab to plot the data, for example, plot (Data)		
iadsread ('DataDirectory')	Data=iadsread ('D:\PostTestData\ TestSet')	Returns test data such as Start/Stop Time, Test, Date, etc...
Iadsread ('DataDirectory', ' ', 0, '?')	Data=iadsread ('D:\PostTestData\ TestSet', ' ', 0, '?')	Returns a list of the parameters in the archive by putting a '?' question mark in argument 4. iadsread ignores the contents of arguments 2 & 3.
Iadsread ('DataDirectory|ConfigFile'), ' ', 0, '?')	Data=iadsread ('D:\PostTestData\Fol der1|D:\PostTestData\ Folder12\fpConfig1',", 0,'?')	Returns a list of the parameters from the specified config file by putting a '?' question mark in argument 4. iadsread ignores the contents of arguments 2 & 3.
iadsread('DataDirectory', ' ', 0, 'Parameter')	Data=iadsread('D:\ PostTestData\TestSet', ' ', 0, AB1001X')	Returns all the information for the parameter in argument 4.
iadsread('DataDirectory', ' ', 0, 'Select *Value1* from Table/Log where [Optional] *Value2* = *Value3*')	Data=iadsread('D:\PostTestData\Test Set', ' ', 0, 'Select *Time* from EventMarkerLog where *Comment* = *Takeoff*')	Returns any piece of information in the configuration file through the use of an SQL statement. To use a wild card match, place asterisks around the wild card.

Note: Incorrect spacing can cause errors.

Input arguments:

Argument 1 - 'DataDirectory or ServerName$PortId (required)'

This string defines the directory of the IADS data archive. Use your 'Explorer' to locate the directory of your choice. Copy the directory from the top of explorer into Matlab.

Another option is to specify a server name and port id in the format 'ServerName$PortId' to connect iadsread directly to a real time data stream in the IADS Server. If you wanted to stream through the entire flight while connected to the IADS Server. Leave the 'IrigStartTime' field as an empty string and you will set the 1st argument (DataDirectory or ServerName$PortId) to the IADS Server machine name and portId. Don't forget to separate the ServerName and PortId by a $ (dollar sign). The default portId of the IADS Server is 58000 (unless this setting has been modified this should work).

For example: **Alt = iadsread('IADSServer$58000', '', 20, 'SineWave0-250')**

Argument 2 - 'IrigStartTime' (required)

This string defines the start time of the data that you want to import. The Irig time string format is DDD:HH:MM:SS.MS That is a 3 digit day (0-364), a two digit hour (0-23), a two digit minute (0-59), a two digit second (0-59), and a partial second (MS) up to 9 digits long. This time will most likely be obtained from the IADS Event Marker or Test Points Logs.

Argument 3 - 'IrigEndTime' or NumberOfSeconds (required)

This string defines the end time of the data that you are interested in. The Irig time string format is DDD:HH:MM:SS.MS Another alternative is to specify a "scalar" number of seconds from the start time.

Argument 4 - 'Parameter(s) or SQL statement (required)'

This string defines a list of parameter(s), comma separated (with no spaces between the commas) that you want to import data from. The Parameter name is that defined in the Parameter Defaults Table.

SQL Statement - 'select <ColumnName or Comma Separated ColumnNames> from <TableName> where <Conditional Statement>'

The "where <Conditional Statement>" statement is Optional. In this format, the <ColumnName> and <TableName> refers to the name of the column in any IADS log or table in the Configuration Tool.

Optional Arguments - Start of Matlab Style (optional settings)

1. 'DecimationFactor', factor 1..N (Defaults to 1 which denotes no decimation). This gives you the ability to reduce the amount of data from the actual parameter's update rate. If not defined, it defaults to 1 (no decimation). The decimation is always based on the largest sample rate of the parameters defined in Argument 4. For example, if you wanted a matrix of data that represents half of the original data, you would enter 2. Decimation only removes data points using a "Decimal Sub-Sample' of your original data (i.e. skips every N points). No other interpolation method (such as linear or bspline interpolation) is currently used. Be aware, if you use this option, you do have the possibility of removing data that is important to your analysis. This argument used in the example is: **'DecimationFactor',4**
2. 'OutputSampleRate', sampleRate (Defaults to highest sample rate of parameters chosen. Trumps DecimationFactor). Similar to Decimation factor above, but specifies the exact output sample rate desired. For example, **'OutputSampleRate',50**
3. 'ReturnDataAtSameSR', 0=False 1=True (Defaults to True) Controls whether the data is interpreted to same sample rate as defined by DecimationFactor or OutputSampleRate. By default, the iadsread function "squares off" the data to same sample rate making it easier to analyze. If this option is set to 0 (False) then each vector is output at its native sample rate and thus the lengths of each vector may vary. In this state, the interpolation/correlation is left to the user code.
4. 'ReturnTimeVector', 0=False 1=True (Defaults to False) Controls whether a time vector is returned along with the data vector(s). The vector contains current time for each element of the corresponding data vector elements.
5. 'ExceptionOnNoData', 0=False 1=True (Defaults to True). Determines whether iadsread throws an error/exception if it is unable to get data for a given parameter. If False, returns empty Vector or if Matrix fills column with NaN.

Note: For additional information on the iadsread function see the Howto.m file at \ProgramFiles\Iads\MatlabExtention.

8.2 Iadsread for Python

This section describes the installation and usage of the IADS data interface iadsread for Python (modified version of the one for VB Script, VB.Net, C#, and C++). The library contains the code to interface to your existing IADS data archives (files) is contained within the "IadsDataInterfaces.dll". For those familiar with the Matlab version of iadsread, this is almost identical. Only a few changes have been made to stay compatible with the new languages. Pay close attention to the return data format and also the change in the 'Optional' param value/pair argument. Proper installation of these files will add a function to the system called "IadsDataInterfaces.iadsread", allowing you to programmatically access your IADS archive data. You will then be able to write scripts or other programs to read in and process the store of flight data you have saved with IADS. Just to note, the iadsread function can get data from any parameter within an IADS archive, including derived parameters. If you have already installed IADS on this machine, both the 32 and 64-bit versions of IadsDataInterfaces.dll may be present in your C:\Program Files\IADS\Common directory and "registered". Confirm that you have the IadsDataInterfaces.dll in this location and you should be ready to proceed. If IADS is not already installed, simply create a folder on your PC (i.e. C:\Program Files\Iads\Common) and copy the IadsDataInterfaces.dll to this directory. Once the file is copied you will need to "register" this dll on your PC. To do this, double click on the file in Windows Explorer. When asked what executable to run, browse to the C:\Windows\System32 directory and choose the "regsvr32.exe" file. Once this is complete, you should get a confirmation dialog that it is properly registered. If this does not work, you may have to contact your IT department to have them run regsvr32.exe as Administrator from a command prompt.

To test that the 'iadsread' function is ready to go, open up a script editor (such as PythonWin) and type:

```
import win32com.client
IadsDataInterfaces = win32com.client.Dispatch("IadsDataInterfaces.iadsread")
UsageString = IadsDataInterfaces.iadsread()
```

Upon execution, the value of "UsageString" should be something like:

iadsread: Minimum 4 inputs required. iadsread: Minimum 4 inputs required. Iadsread ("DataDirectoryOrServerName", "IrigStartTime", "IrigEndTime" or NumSeconds, "ParameterNameList (Comma Separated)", [optional]param/value pairs as described below)

Actually, this is correct! This means that your dll file is properly hooked up and will error because the function call arguments are not quite complete. If the function returns with "Invalid function or procedure" your dll is not correctly registered. Make sure you complete the dll registration process above. If you get an error about variable type mismatch, note that the iadsread function can return strings or arrays and needs a variable of type "variant" to receive it (not a variant array).

iadsread("DataDirectoryOrServerName", "IrigStartTime", "IrigEndTime" or NumSeconds, "ParameterNameList (Comma Separated)", [optional]":Param/Value Pairs" as described below)

Input arguments:

Argument 1 - DataDirectory or ServerName$PortId or DataDirectory|ConfigFile

This string defines the source data directory of the IADS archive data for your Flight. Most flight data is arranged in a system of directories by flight/test/tail or data on a server within your local network. The specific location is group dependent. Use your 'Explorer' to locate the directory of your choice. Then just simply copy the directory from the top of explorer into your function (be sure to put quotes around the string). The value should be your FULL directory path (with drive letter) to your IADS archive data. My data directory for this example is: "D:\PostFlightData\Demo"

Another option is to specify a server name and port id in the format "ServerName$PortId" to connect iadsread directly to a real time data stream in the IADS Server (CDS) or Post Test Data Server. There is example code on this subject below.

A final option is to specify a DataDirectory and a separate config file in the format 'DataDirectory|ConfigFile' This will allow you to use a config file other than the default pfConfig within the DataDirectory. An example of this would be D:\PostFlightData\Demo|D:\SomeOtherDirectory\pfConfig

Use this option with care. You should be aware that the derived parameters and meta data for a given archive may not be valid or relevant for another archive.

Argument 2 - IrigStartTime

This string argument is the start time of the data that you want to import. The format of the string is IRIG time in the format “DDD:HH:MM:SS.MS” this is a 3 digit day (0-364), a two digit hour (0-23), a two digit minute (0-59), a two digit second (0-59), and a partial second (MS) up to 9 digits long. This time will most likely be obtained from your flight notes or the IADS EventMarker Log. This interface also has the ability to supply you with your TestPoint/Maneuver start/end times for each flight. Let me know if you have any other ideas.

Argument 2 - IrigEndTime or NumSeconds

This string argument is the end time of the data that you are interested in. The Format of the string is an IRIG time in the format DDD:HH:MM:SS.MS again. Your other alternative to is just to specify an integer number of seconds. You could, for example, get 10 seconds of data from a given start time.

Argument 4 - ParameterNameList

This string argument is a list of comma separated parameter names that you want to import data from. For example, you could import some aircraft "Wing" parameters by defining a list like: "AW0001X,AW0002X,AW0003X". Notice the name is the "Parameter" name defined in the config file's "ParameterDefaults Table" (usually the parameter code)

Note: All filtering and nulling that was set in the ParameterDefaults entry for the specified parameter is applied before the data is returned to Matlab. Spike detection and wild point corrections are *not* applied as of this date. We may consider having this as an option.

One more option is being considered for next build: *The DataGroupName option will allow you to access a group of parameters defined in your config file under the DataGroup table.

Argument 5) - [optional]Start of comma separated "Param/Value Pairs"

Optional Arguments

1. DecimationFactor, factor 1..N (Defaults to 1 which denotes no decimation). This gives you the ability to reduce the amount of data from the actual parameter's update rate. If not defined, it defaults to 1 (no decimation). The decimation is always based on the largest sample rate of the parameters defined in Argument 4. For example, if you wanted a matrix of data that represents half of the original data, you would enter 2. Decimation only removes data points using a "Decimal Sub-Sample' of your original data (i.e. skips every N points). No other interpolation method (such as linear or bspline interpolation) is currently used. Be aware, if you use this option, you do have the possibility of removing data that is important to your analysis.
2. OutputSampleRate, sampleRate (Defaults to highest sample rate of parameters chosen. Trumps DecimationFactor) Similar to Decimation factor above, but specifies the exact output sample rate desired.
3. ReturnDataAtSameSR, 0=False 1=True (Defaults to True) - Controls whether the data is interpreted to same sample rate as defined by DecimationFactor or OutputSampleRate. By default, the iadsread function "squares off" the data to same sample rate making it easier to analyze. If this option is set to 0 (False) then each vector is output at its native sample rate and thus the lengths of each vector may vary. In this state, the interpolation/correlation is left to the user code.
4. ReturnTimeVector, 0=False 1=True (Defaults to False) - Controls whether a time vector is returned along with the data vector(s). The vector contains current time for each element of the corresponding data vector elements. If ReturnDataAtSameSR=False then returns time/data in a struct (not implemented yet).
5. TimeFormat, 0=SecondsSinceNewYear 1=IRIGTimeString (Defaults to SecondsSinceNewYear) - Controls format of the time vector returned along with the data vector(s). The values are either a count of total seconds since New Year or an IRIG String formatted as DDD:HH:MM:SS.MS. Note that the IRIGTimeString option is only available in vector format, so you must supply an output variable for time as well as each item in the parameter name list.
6. ExceptionOnNoData, 0=False 1=True (Defaults to True) - Determines whether iadsread throws an error/exception if it's unable to get data for a given parameter. If False, returns missing values with Empty (VT_EMPTY). An example of using optional args is as follows: IadsDataInterfaces.iadsread(directory, "001:01:01.000", 5, "Param1,Param2,Param3", "DecimationFactor,4,ReturnTimeVector,1")

Ok, let's proceed to a concrete example... Let's get some information on data from an IADS archive using 'iadsread'. Find a directory on your system with IADS data using Microsoft Explorer. Copy the directory name using <Ctrl C> and paste it into the <Insert Your Data Directory Here> below:

ArchiveInfo = IadsDataInterfaces.iadsread("<Paste Your Data Directory Here>") The system should respond with some information about the data within this directory including its

StartTime, StopTime, DataDir, FlightId, TestId, TailId. The results inside of the ArchiveInfo variable is a double dimension array. The first row contains all of the property names (as described in the last sentence). The second row contains all the values of these properties. Please recall the VB accesses array using the Array (Row, Column) format, with zero based index values.

Example:

ArchiveInfo[0][0] is the first row first value (which in this case is the property name "StartTime")

ArchiveInfo[0][1] is the first row second value (which in this case is the property name "StopTime")

ArchiveInfo[0][2] is the first row third value (which in this case is the property name "DataDirectory") etc.. and now for the actual values of these properties for your specific flight...

ArchiveInfo[1][0] is the second row first value (which in my case is the value "001:00:00:00.000")

ArchiveInfo[1][1] is the second row second value (which in my case is the value "001:02:00:00.000")

ArchiveInfo[1][2] is the second row third value (which in my case is the value "D:\PostFlightData\Demo") and so on..

Now, let's say you want to know what parameters are available in an archive... To achieve this, we put a '?' (Question Mark) in the ParameterList (argument number 4)

My line looks like this > IadsDataInterfaces.iadsread("D:\PostFlightData\Demo", "", 0, "?") (iadsread ignores contents of arguments 2 & 3)

Type the line below into your script editor inserting your own dir into the <Insert Your Data Directory Here> text

ParameterList = IadsDataInterfaces.iadsread("<Paste Your Data Directory Here>", "", "", "?")

The system should respond with the list of parameters defined in your ParameterDefaults table ParameterList will be an array of Strings containing all the parameters. In my case:

ParameterList[0] = "DV1"
ParameterList[1] = "IABALT"
ParameterList[2] = "IIIALT"
ParameterList[3] = "IATASP", etc
You can use the UBound and a for loop to iterate though the parameters:
For index = 0 To UBound(ParameterList)
ParamName = ParameterList(index)
Next
Or by using the For Each statement:
For Each Param In ParameterList
ParamName = Param
Next

Another helpful tool is the ability to look at the settings of an individual parameter. It's just a small difference from the last line. Put a "?" (Question Mark) in the ParameterList (argument

number 4) then a <space> then the parameter name you want more information about... My line looks like this -> InfoOnIABALT = IadsDataInterfaces.iadsread("D:\PostFlightData\Demo", "", 0, "? IABALT")

iadsread returns an array much like the previous 'ArchiveInfo' above with the first row containing the 'ColumnName' and the second row containing the actual value of the column. Again, the array is accessed InfoOnIABALT(Row, Column) with zero based indices

InfoOnIABALT[0][0] = "ParameterDefaults" InfoOnIABALT[1][0] = "STRUCTURES"

InfoOnIABALT[0][1] = "Parameter" InfoOnIABALT[1][1] = "PF5032"

InfoOnIABALT[0][2] = "ParamType" InfoOnIABALT[1][2] = "float"

InfoOnIABALT[0][3] = "ParamGroup" InfoOnIABALT[1][3] = "LOADS"

InfoOnIABALT[0][4] = "ParamSubGroup" InfoOnIABALT[1][4] = "Door - Misc"

InfoOnIABALT[0][5] = "ShortName" InfoOnIABALT[1][5] = "LIRCM Bay Pressure"

InfoOnIABALT[0][6] = "LongName" InfoOnIABALT[1][6] = "LIRCM Bay Pressure"

InfoOnIABALT[0][7] = "Units" InfoOnIABALT[1][7] = "psi"

InfoOnIABALT[0][8] = "Color" InfoOnIABALT[1][8] = 16711680

InfoOnIABALT[0][9] = "Width" InfoOnIABALT[1][9] = 1

InfoOnIABALT[0][10] = "DataSourceType" InfoOnIABALT[1][10] = "Tpp"

InfoOnIABALT[0][11] = "DataSourceArguement" InfoOnIABALT[1][11] = "1"

InfoOnIABALT[0][12] = "UpdateRate" InfoOnIABALT[1][12] = "49.3213"

InfoOnIABALT[0][13] = "LLNegative" InfoOnIABALT[1][13] = "-1000"

InfoOnIABALT[0][14] = "LLPositive" InfoOnIABALT[1][14] = "1686"

(values continue...)

Okay, now to extract actual flight data from an IADS archive. Type in the following code. We need the output from this statement for a reasonable StartTime

ArchiveInfo = IadsDataInterfaces.iadsread("<Paste Your Data Directory Here>")

iadsread returns the values in ArchiveInfo. Recall from a previous time that ArchiveInfo is a double dimension array with the column names in the first row and values in the second. My data returns:

ArchiveInfo[0][0] = "StartTime" ArchiveInfo[1][0] = "318:17:37:52.393"

ArchiveInfo[0][1] = "StopTime" ArchiveInfo[1][1] = "318:21:58:51.518"

ArchiveInfo[0][2] = "DataDir" ArchiveInfo[1][2] = "D:\PostFlightData\Demo"

ArchiveInfo[0][3] = "Flight" ArchiveInfo[1][3] = "100"

ArchiveInfo[0][4] = "Test" ArchiveInfo[1][4] = "100-ABC"

ArchiveInfo[0][5] = "Tail" ArchiveInfo[1][5] = "001"

ArchiveInfo[0][6] = "Date" ArchiveInfo[1][6] = "11/14/1998"

Just as an example, let's read the first 5.5 seconds of data from a couple of parameters. Use the "StartTime" string of "318:17:37:52.393" obtained from ArchiveInfo[1][0] above as the value of

argument 1. Your second argument should be 5.5 (or any number of seconds). Your third argument should be a list of comma separated parameter names. Pick parameter names from the list returned above...My line looks something like this ->

Data = IadsDataInterfaces.iadsread("D:\PostFlightData\Demo", "318:17:37:52.393", 5.5, "Sweep,SineWave10Hz,SineWave20Hz,SineWave30Hz,SineWave40Hz")

Type the line below into your script editor inserting your own dir, StartTime, and parameter list.

Data = IadsDataInterfaces.iadsread("<Insert Your Data Directory Here>", "<Insert StartTime String Here>", 5.5, "<Your Parameter List Comma Separated>")

Hint: If you have an error getting data at this StartTime, add a couple of minutes to your start time. Sometimes data for a given parameter starts later than others...Alternatively, if you knew the actual StartTime and EndTime of your data, you could use it to define your data of interest.

My line looks something like this:

Data = IadsDataInterfaces.iadsread("D:\PostFlightData\Demo", "318:17:40:53.393", "318:17:40:54.393", "Sweep,SineWave10Hz,SineWave20Hz,SineWave30Hz,SineWave40Hz", "DecimationFactor,2")

I requested one second of data from the parameters defined in my list, and I wanted the data at DecimationFactor of 2 (giving me every other point from the data).

Data = IadsDataInterfaces.iadsread("<Insert Your Data Dir Here>", "<Insert StartTime Here>", "<Insert EndTime Here>", "<Your Parameter List Comma Separated>, "DecimationFactor,2")

If all is well, you will get the data requested in matrix form (double dimension array row,column) with the number of columns matching the number of parameters you have requested and each row being the set of data values.

In my example:

Data[0][0] = FirstValueOfSweepParam

Data[0][1] = FirstValueOfSineWave10HzParam

Data[0][2] = FirstValueOfSineWave20HzParam

Data[0][3] = FirstValueOfSineWave30HzParam

Data[0][4] = FirstValueOfSineWave40HzParam

Data[1][0] = SecondValueOfSweepParam

Data[1][1] = SecondValueOfSineWave10HzParam

Data[1][2] = SecondValueOfSineWave20HzParam

Data[1][3] = SecondValueOfSineWave30HzParam

Data[1][4] = SecondValueOfSineWave40HzParam

And so on....

Let's say that along with data you also want the current value of time. You could do that with the "ReturnTimeVector" optional argument as follows:

DataWithTimeAsFirstVector = IadsDataInterfaces.iadsread("<Insert Your Data Dir Here>", "<Insert StartTime Here>", "<Insert EndTime Here>", "<Your Parameter List Comma Separated>", "DecimationFactor,2,ReturnTimeVector,1")

The first column DataWithTimeAsFirstVector(0..N,0) will be filled with the actual time stamp of elapsed seconds since midnight. If you want an ascii IRIG time representation simply add "TimeFormat,1" to the optional last argument like so:

DataWithTimeAsFirstVector = IadsDataInterfaces.iadsread("<Insert Your Data Dir Here>", "<Insert StartTime Here>", "<Insert EndTime Here>", "<Your Parameter List Comma Separated>", "DecimationFactor,2,ReturnTimeVector,1,TimeFormat,1")

Notice that all parameters are combined into 1 matrix called 'Data'. That's because the interface only allows one variable to the result of iadsread Data = iadsread(..). If you want the data in separate vectors, you would have to call the iadsread function 3 times. Here is an example below how to create 3 separate vectors

a = IadsDataInterfaces.iadsread("D:\PostFlightData\Demo", "318:17:40:53.393", 20, "Sweep")

b = IadsDataInterfaces.iadsread("D:\PostFlightData\Demo", "318:17:40:53.393", 20, "SineWave10Hz")

c = IadsDataInterfaces.iadsread("D:\PostFlightData\Demo", "318:17:40:53.393", 20, "SineWave20Hz")

The vector 'a' will contain data from the 'Sweep' parameter, 'b' from the 'SineWave10Hz' parameter, and 'c' from 'SineWave20Hz'

Unlike the multiple parameter requests, these single parameter requests return a single dimensional array. The data is accessed as:

a(0), a(1), a(2), etc...

To loop through all the values would be similar to the ParameterList above:

For index = 0 To UBound(a)

Value = a(index)

Next

In other words, we don't have to worry about the double dimension (row,column) anymore, but there are other things to consider.

If for example a, b, and c had different sample rates, we would probably get different amounts of data in each case. Lining up the values to perform computations will be a difficult task, so be aware of this fact. When parameters are combined into a matrix, IADS handles this issue by up-sampling all the parameter to the highest rate in the list (unless overridden by the 'OutputSampleRate' or 'DecimationFactor' optional argument. So, if you want separate vectors, try using the "OutputSampleRate" option to force matching rates (recommend to upsample to highest rate).

Writing a program to analyze data should be fairly simple, maybe something like this:

Read in the data from IADS

Data = iadsread("D:\PostFlightData\Demo", "318:17:40:53.393", "318:17:40:54.393", "Sweep,SineWave10Hz,SineWave20Hz,SineWave30Hz,SineWave40Hz")

Call my analysis function with the data matrix obtained from IADS

b = myAnalysisFunction(Data)

Now output the results

That should be it for the basics; let's continue on with more advanced subjects.

Here is another example of accessing data sequentially. Say you just wanted to stream through the entire flight worth of data for a number of parameters and plot them. What you have to do it call iadsread at least once with a valid start time (and you must use the NumSeconds option in argument 3). In you next call to iadsread, you leave the StartTime string (argument 2) blank. This will tell iadsread that you wish to continue reading at the point you left off last. In the example below, the first call will read 10.0 seconds at time 318:17:40:53. Each sequential call with " as the argument 3 value will return the next sequential 10.0 seconds of data.

In order to make this work on your system, you'll have to modify arguments 1,2 & 4 to the correct values for your archive.

```
SweepData = IadsDataInterfaces.iadsread("D:\PostFlightData\Demo", "318:17:40:53", 10.0, "Sweep")
For index = 0 to 100
   plot( SweepData )
   MsgBox "Press ok for next Plot"
   ' Notice that the second argument "IrigStartTime" is a blank string. This will read the next
   SweepData = IadsDataInterfaces.iadsread("D:\PostFlightData\Demo", "", 10.0, "Sweep")
Next
```

Here is yet another example of accessing data sequentially. Say you just wanted to stream through the entire flight while connected to the IADS CDS or a Post Test Data Server. It is very similar to the last example except you'll leave the 'StartTime' field as an empty string and set the 1st argument (DataDirectory or ServerName$PortId) to the CDS or Post Test Data Server machine name and portId. Don't forget to separate the ServerName and PortId by a $ (dollar sign). The default portId of the IADS CDS is 58000, so unless you have modified it in the setup bag this should work.

In order to make this work on your system, you'll have to modify arguments 1 & 4 to the correct values for your system setup.

```
Sub TestRealTime2()
   For j = 1 to 100
      Press_Alt = IadsDataInterfaces.iadsread( "IADS-CDS$58000", "", 2.0, "SineWave0-250" )
      plot( Press_Alt )
      WScript.Sleep 10 ' Allow a little time for the new data. This line may need to be modified for the scripting environment used
   end
End Sub
```

New and advanced stuff to query info from the config file

There is now a new capability to query any piece of information in configuration file through the use of an SQL statement. To achieve this, we must first explain how write a simple SQL statement.

The basic format is: 'select <ColumnName or Comma Seperated ColumnNames> from <TableName> where <Conditional Statement> (The 'where' statement is optional). In this format, the <ColumnName> refers to the name of the column in any IADS log or in any ConfigTool window... Likewise, the <TableName> refers to the actual name of the log name or 'Table' name in the ConfigTool. This is a simple and very useful example that extracts every column from every row in the 'EventMarkerLog' table. If we want to get every row and column, we must use a 'wildcard' (an '*') for the column name, as well as no 'where' clause.... EventMarkerLogContents = IadsDataInterfaces.iadsread("D:\PostFlightData\Demo", "", 0, "? select * from EventMarkerLog")

iadsread returns:

5x2 array with field names in row 0 (EventMarkerLogContents[0][0) .. EventMarkerLogContents[0][5)):

Group
SubGroup
User
Time
Comment
PropertyBag

Examining the second row of the matrix (i.e. the first row of the EventMarkerLog) EventMarkerLogContents[1][0) .. EventMarkerLogContents[1][5)

Shows:

Group: "LOADS"
SubGroup: "Maneuver Quality"
User: "IadsUser2"
Time: "001:12:40:29.011"
Comment: "E1 APU Start"
PropertyBag: "Each field can be accessed using the proper array index. EventMarkerLogContents[1][3) would return the time of the event or "010:12:40:29.011"

What if we were only concerned with the 'Time' column information? Let's limit our output to only the "Time" column like so:

TimeOfEvent = IadsDataInterfaces.iadsread("D:\PostFlightData\Demo", "", 0, "? select Time from EventMarkerLog")

ColumnName = TimeOfEvent(0,0)

Time = TimeOfEvent(1,0)

iadsread returns:

Time
012:17:49:29.011

Now even more useful... What if we were only concerned with the 'Time' column information when a certain comment occurred?

Let's assume that we had a Comment in our 'EventMarkerLog' table that always contained the word 'Takeoff' and corresponded to the takeoff time of the aircraft. Let's limit our output to only the 'Time' column when the 'Comment' contained the word 'Takeoff'. Ok, this is where the 'where' clause comes into play. It is a conditional statement that will allow you to filter through the many rows of a table/log and find the specific row that you need. The query would look something like:

TimeOfSpecificEvent = IadsDataInterfaces.iadsread("D:\PostFlightData\Demo", "", 0, "? select Time from EventMarkerLog where Comment = '*Takeoff*' ")

ColumnName = TimeOfSpecificEvent(0,0)

Time = TimeOfSpecificEvent(1,0)

Look at the 'where' clause above.... where Comment = '*Takeoff*'.... Confusing, isn't it? First of all, the IADS SQL query statement requires that all strings in the 'where' clause be single quoted... so we need two single quotes around the where clause 'Takeoff' string. We also need to use a 'Wildcard' match, placing asterisks '*' around the word 'Takeoff'. This tells iadsread to match any comment that has 'Takeoff' anywhere in it... (i.e. 'E10 Takeoff …' matches)

iadsread returns:

Time
012:18:41:19.247

Thus, the takeoff time of the aircraft is "012:18:41:19.247"

Just remember, you can get to any piece of data in the config file with the proper query.

If you need help to write your SQL statement or explain further, email iads-support@curtisswright.com.

More examples.....

Test Streaming data for the entire flight into a processing function

IadsDataInfo = IadsDataInterfaces.iadsread("D:\PostFlightData\MyIadsDataDirectory")

This returns a struct array where each field can be accessed by the proper index

Choices are StartTime, StopTime, DataDir, Flight, Test, Tail, and Date

IadsDataInfo(1,0) would return for example: "318:17:37:52.393"

Set block size to read... Basically, the number of seconds to read for each computation
BlockSizeInSecondsToRead = 10

Process all the data for a given set of parameters. Fetch the next Matrix of data using iadsread until the end it reached

MyParameterList = "Param1,Param2,Param3,Param4"

Done = False
On Error Resume Next
While (Not(Done))

  Err.Clear()

```
    Data = IadsDataInterfaces.iadsread( "D:\PostFlightData\MyIadsDataDirectory",  StartTime,
    BlockSizeInSecondsToRead, MyParameterList )
    If ( Err.Number = 0 ) Then
                    ProcessTheData( Data )
        StartTime = ""
    Else
        Done = True
    End If

End
Err.Clear()
On Error Goto 0
```

APPENDIX A

IADS Configuration Table Reference

Name	Description	Type
AircraftReferences	Used by Parameter Identification	Flat
Aircraft Properties	Aircraft properties table	Flat
ActualFlutterTestPointsLog	Completed flutter test points	Flat
ActualLoadsTestPointsLog	Completed Loads test points	Flat
AnalysisLog	Location of saved analysis results	Flat
AnalysisWindows	User created Analysis Windows	Hierarchical
AttachedDataDisplays	List of displays attached to AWs	Hierarchical
FlutterSummaryLog	Ongoing collection of flutter results	Flat
DataStorageInformation	Data Information from a real-time test	Flat
CurrentFlightInformation	Table of the current flight test	Flat
Classifications	List of available classifications	Flat
GroupDefinitions	List of available classification strings	Flat
PredefinedComments	Pre-defined event marker strings	Flat
Constants	User defined constants	Flat
DataDisplays	List of used data displays	Hierarchical
DataStorageLog	Information on data archive set	Flat
DataDropOutLog	Not Currently used	Flat
DisplayDefaults	Data display defaults	Flat
Desktops	User defined Desktops	Hierarchical
ExtendedDesktopInfo	Additional Desktop information	Flat
Envelopes	User defined envelopes	Flat
ReferenceCurves	User defined reference envelopes	Flat
EventMarkerLog	User created event markers	Flat
HardCopyLog	<blank>	Flat
HardCopyBanners	<blank>	Flat
LogBehavior	Log behavior settings	Flat
LoadsSummaryLog	User created loads information	Flat
TableUpdateBehavior	Table update behavior properties	Flat
ModalDefinitions	User defined mode ranges and titles	Flat
ParameterDefaultsState	List of parameter default sets	Hierarchical
ParameterDefaults	List of all user defined parameters	Flat
ParametersSavedInDisplays	Parameters saved in defined displays	Hierarchical
PlannedLoadsTestPoints	User defined planned loads test points	Flat
PlannedFlutterTestPoints	User defined planned flutter points	Flat
AlphaNumeric	List of Alphanumeric displays	Flat
AlphaNumericTable	List of AlphaNumericTable displays	Flat
Annunciator	List of Annunciator displays	Flat
FrequencyResponsePlot	List of Frequency response displays	Flat
DisplayLabel	List of Display Label displays	Flat
DisplayFolder	List of Display Folder displays	Flat
CrossPlot	List of Cross Plot displays	Flat
DisplayTab	List of Display Tab displays	Flat
FlutterSummaryPlot	List of Flutter Summary Plot displays	Flat
FrequencyPlot	List of Frequency Plot displays	Flat
LoadsSummaryPlot	List of Loads Summary Plot displays	Flat
NyquistPlot	List of Nyquist Plot displays	Flat
Slider	List of Slider displays	Flat
Stripchart	List of Stripchart displays	Flat

TppDefinitions	List of TPP parameters validated	Flat
Users	List of user defined Users	Flat
Lists	<blank>	Flat
PeaksLog	User selected peak values	Flat
SystemValues	User defined System values	Flat
ToolPositions	Internal table used for positions	Flat
ThresholdLog	Calculated thresholds	Flat
ViewQueries	Internal table of view queries	Flat
SelectionsLog	User data selections	Flat
SystemParameterDefaults	List of System Parameter Defaults	Flat
ValidationLog	Results of TPP parameter validation	Flat
DataEditLog	List of data edits performed	Flat
NullCorrections	System calculated Null corrections	Flat
FESParameters	Parameters used for the FES automation	Flat
ActiveXControlsTab	ActiveX displays on display builder tab	Flat
ActiveXDisplay	List of ActiveX displays	Flat
DataViewsDisplay	List of Data Views displays	Flat
DataGroups	User defined Data Groups	Flat
DerivativeSummaryLog	List of pEst calculated derivatives	Flat
OctaveBandDisplay	List of Octave Band displays	Flat
TestPointLog	List of completed test points	Flat
PlannedTestPoints	List of Planned test points	Flat
Maneuvers	List of pEst required maneuvers	Flat
FlightConditions	List of pEst required Flight conditions	Flat
PredictedResults	List of pEst required Predicted Results	Flat
CurrentFlightInformation2	Additional Flight information	Flat

APPENDIX B

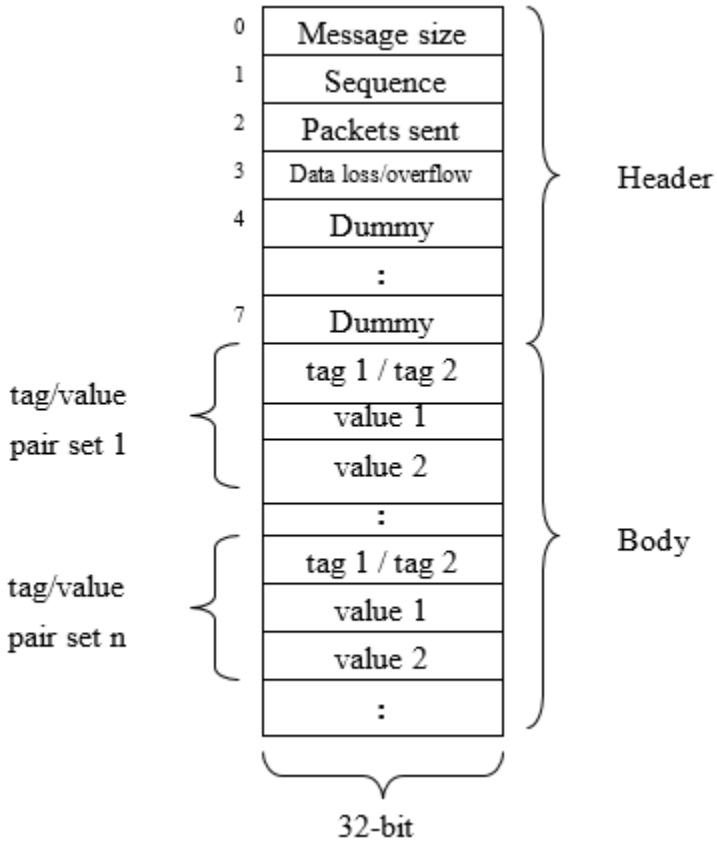
IADS Data Types Enumerated

IADS Data Types		
iadsInteger	0	Integer data type.
iadsDiscrete	1	Discrete data type.
iadsFloatingPoint	2	Floating point data type.
iadsLong	3	Long data type.
iadsUnsignedLong	4	Unsigned long data type.
iadsDouble	5	Double data type.
iadsAscii	6	ASCII data type.
iadsBlob	7	Binary data type
IADS Data Source Type		
iadsTpp	1	TPP data source
iadsDerived	2	Derived data source
iadsIap	3	Derived data source.
On/Off enumeration		
iadsOn	0	On setting
iadsOff	1	Off setting
Yes/No enumeration		
iadsYes	0	Yes setting
iadsNo	1	No setting
Filter algorithms		
iadsFilterNone	0	No filter algorithm
iadsButterworthFilter	1	Butterworth filter
iadsEllipticFilter	2	Elliptic filter
Filter pass types		
iadsLowPass	1	Low pass filter
iadsHighPass	2	High pass filter
iadsBandPass	3	Band pass filter
Data correction methods		
iadsDataCorrectionNone	0	No data correction
iadsDefaultValue	1	Default value
iadsLastValue	2	Last value
Null corrections		
iadsNullCorrectionNo	0	No null correction
iadsNullCorrectionYes	1	Use null correction
iadsNullEquationInput	2	Equation input correction
iadsNullEquationResult	3	Equation result correction
Null group enumeration		
iadsAircraftGroup	1	Aircraft group
iadsWeaponsGroup	2	Weapons group
Spike detection method		
iadsSpikeDetectionMehodNone	0	No spike detection
iadsSlopeChange	1	Slope change detection
iadsAbsoluteChange	2	Absolute change detection
IADS compute types	Enum Value	**Description**
iadsAutoSpectrum	0	Auto spectrum compute type
iadsPsd	1	PSD compute type
iadsPhaseMagnitude	2	Phase magnitude compute type
iadsPhaseReal	3	Phase real compute type
iadsPhaseImaginary	4	Phase imaginary compute type

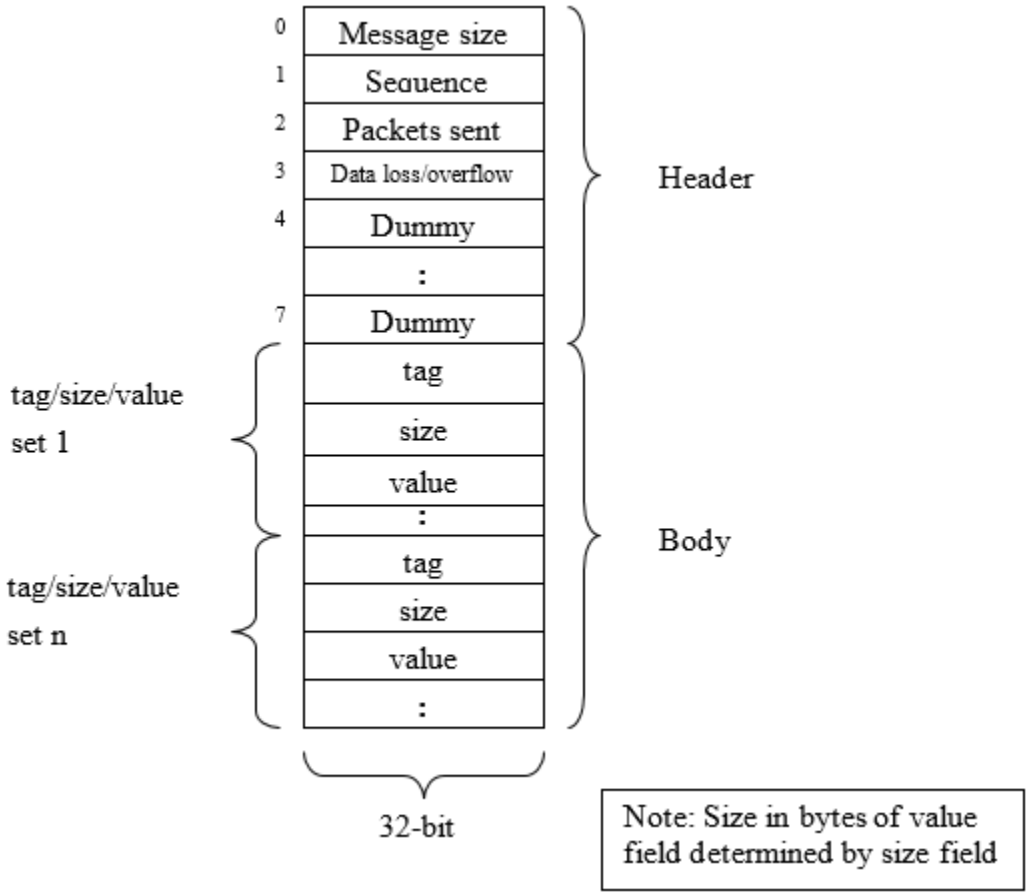
iadsPhaseGain	5	Phase gain compute type
iadsBode	6	Bode compute type
iadsNyquist	7	Nyquist compute type
IADS Window types	Value	**Description**
iadsWindowTypeNone	0	Default window type (none)
iadsHanning	1	Hanning window
iadsHamming	2	Hamming window
iadsBlackman	3	Blackman window
iadsKaiserBessel	4	Kaiser Bessel window
iadsRectangular	5	Rectangular window
iadsFlatTop	6	Flat Top Window
IADS Alpha	Value	**Description**
iadsAlphaNone	0	No alpha
iadsAlphaTwoPointZero	1	2.0 alpha
iadsAlphaTwoPointFive	2	2.5 alpha
iadsAlphaThreePointZero	3	3.0 alpha
iadsAlphaThreePointFive	4	3.5 alpha
IADS Averaging Methods		
iadsAverageMethodNone	0	No averaging method
iadsAverageTime	1	Time averaging method
iadsAverageFrequency	2	Frequency averaging method
IADS Block Sizes (in bytes)		
iadsBlock64	64	64 byte block
iadsBlock128	128	128 byte block
iadsBlock256	256	256 byte block
iadsBlock512	512	512 byte block
iadsBlock1024	1024	1024 byte block
iadsBlock2048	2048	2048 byte block
iadsBlock4096	4096	4096 byte block
iadsBlock8192	8192	8192 byte block
iadsBlock16384	16384	16384 byte block
iadsBlock32768	32768	32768 byte block
iadsBlock65536	65536	65536 byte block
IADS Threshold levels	Value	**Description**
iadsNoThreshold	0	
iadsWarning	1	
iadsAlarm	2	

APPENDIX C

IADS Interface Message Format 1



IADS Interface Message Format 2



APPENDIX D

Sample Parameter Definition File

```
1    TimeUpperWord  1000.0           1 SystemParamType = MajorTime
2    TimeLowerWord  1000.0           1 SystemParamType = MinorTime
3    PARAMETER1     12.330334596     2
4    PARAMETER2     24.6606691919    2
5    PARAMETER3     49.3213383838    2
6    PARAMETER4     98.6426767677    2
7    PARAMETER5     197.285353535    2
8    PARAMETER6     394.570707071    2
9    PARAMETER7     789.141414141    2
10   PARAMETERBLOB  10.0             7 DataSize = 92
100  DECOMSTATUS    789.141414141    1 SystemParamType = DecomStatus
```


APPENDIX E

IADS 32-bit Decom Status Parameter Format

31-6	5	4	3	2	1	0
SF(n+1)STAT- SF(n)STAT	SF2STAT	SF2STAT	SF1STAT	SF1STAT	FSTAT	FSTAT

Bits	Signal	Description
0-1	FSTAT	Frame Status bits are decoded as follows: **1 0** 0 0 Lock 0 1 Check 1 0 Verify 1 1 Search
2-3	SF1STAT	Subframe 1 Status Bits are decoded as follows: **3 2** 0 0 Lock 0 1 Check 1 0 Verify 1 1 Search
4-5	SF2STAT	Subframe 2 Status Bits are decoded as follows: **5 4** 0 0 Lock 0 1 Check 1 0 Verify 1 1 Search

