Creating an IADS
Custom Derived Function
using Visual C++ 6.0

April 2010
SYMVIONICS Document SSD-IADS-045

SYMVIONICS, Inc.
Telemetry Systems

Table of Contents

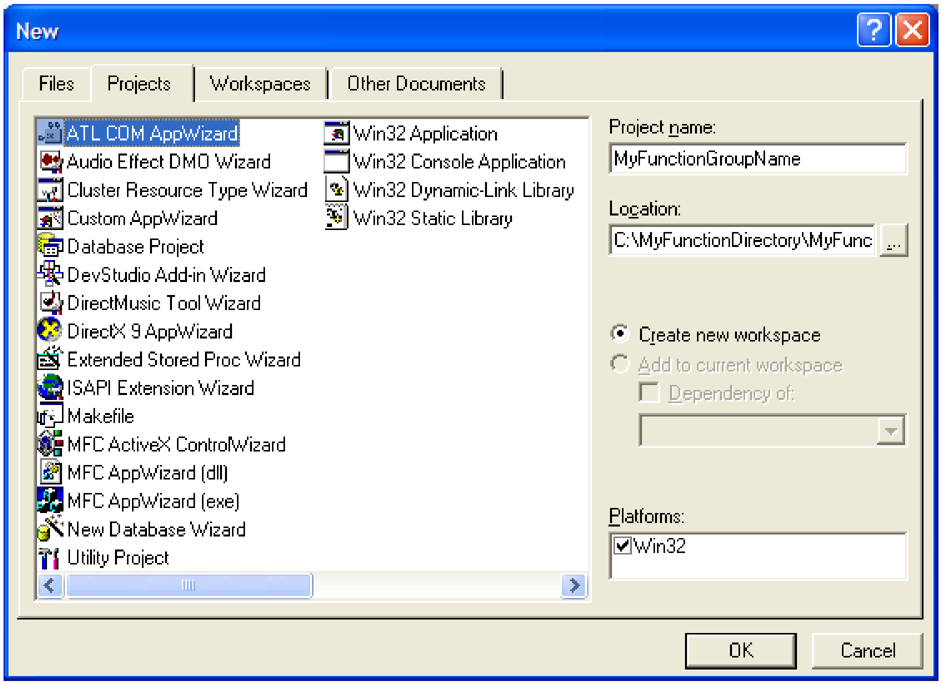
1. Introduction

This document assumes you are using Microsoft Visual C++ 6.0. The tutorial has not yet been attempted on a newer version, although it may still apply. This instruction guide will cover: creating a new function using the ATL COM Wizard, how to access your new function in IADS, and how to debug your new function in IADS.

2. Creating Your Function Using the ATL COM Wizard

1) Open up Visual C++ and Select “File -> New”.

2) Choose “ATL COM AppWizard”, and pick your project name and location.



New

Projects: ATL COM AppWizard

Project name: MyFunctionGroupName

Location: C:\MyFunctionDirectory\MyFunc

● Create new workspace

○ Add to current workspace

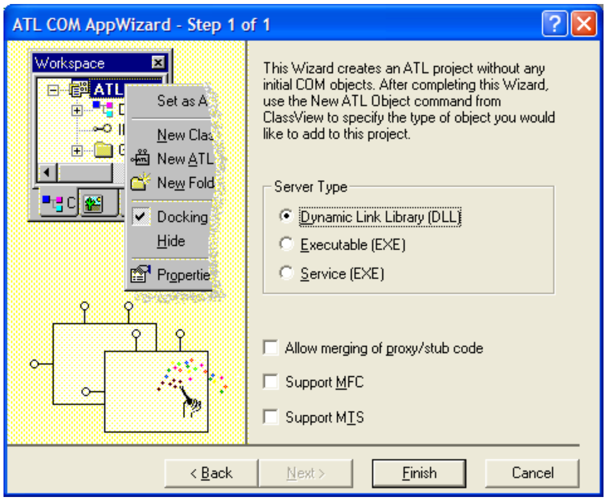
☐ Dependency of:

Platforms: ☑ Win32

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’ll 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’ll be adding (example: NasaFluidFuncs). Choose wisely and press OK when you are ready to continue.

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).*

3) After pressing "Ok", the "ATL COM AppWizard" dialog will appear as below. In step 1, choose "Dynamic Link Library (DLL)" and click Finish. Every function that runs in IADS is DLL. This allows for maximum speed computing calculations. Press the "Finish" button on this wizard and then press "Ok" on the next dialog that appears.



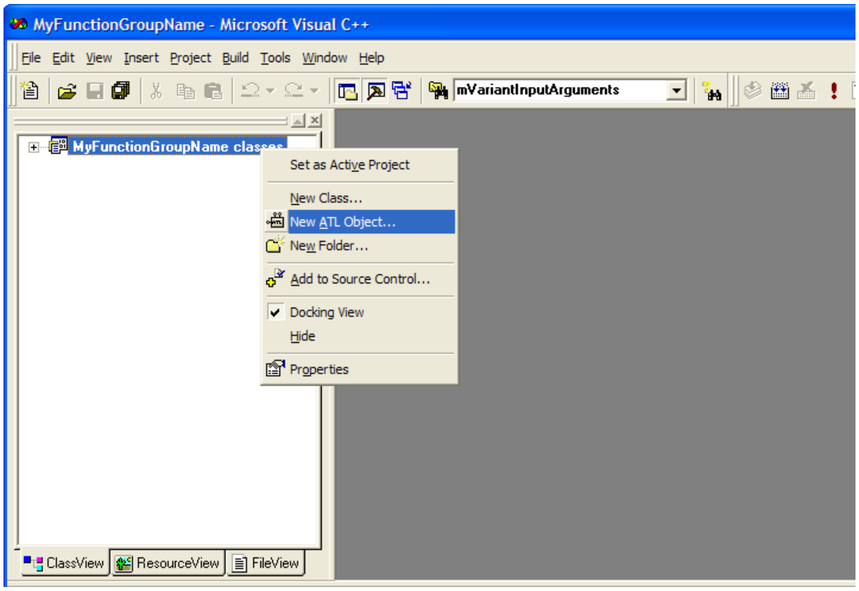
This Wizard creates an ATL project without any initial COM objects. After completing this Wizard, use the New ATL Object command from ClassView to specify the type of object you would like to add to this project.

Server Type

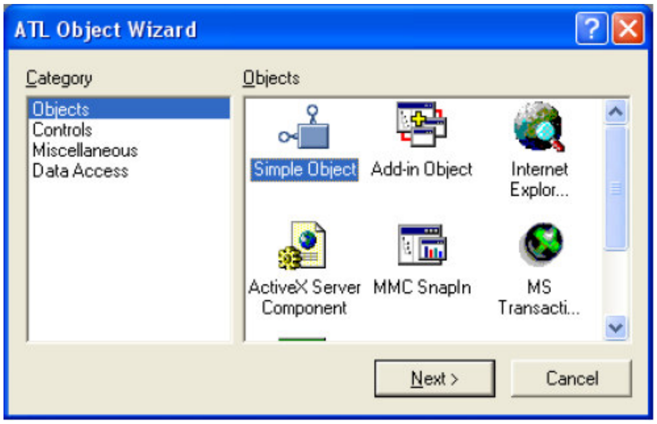
● Dynamic Link Library (DLL)
○ Executable (EXE)
○ Service (EXE)

☐ Allow merging of proxy/stub code
☐ Support MFC
☐ Support MTS

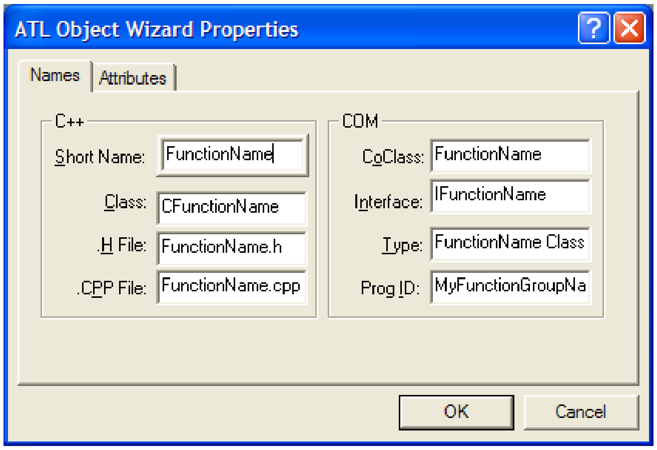
4) Next, go to the "ClassView" tab in DevStudio's workspace view and right-click on the project name. Choose "New ATL Object".



5) When the ATL Object Wizard dialog appears, choose “Simple Object” and then click “Next”.



6) 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 function name. Press Ok to continue.



Short Name: FunctionName
Class: CFunctionName
.H File: FunctionName.h
.CPP File: FunctionName.cpp
CoClass: FunctionName
Interface: IFunctionName
Type: FunctionName Class
Prog ID: MyFunctionGroupNa

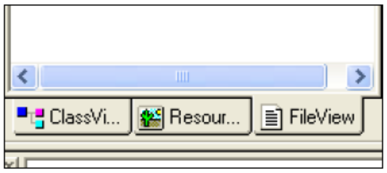
7) At this point, we need to take care of the interface portion of the function. Basically, we’ll need to implement the defined “IIadsFunction” interface so that the function will be compatible with the Iads environment.

Download the ComFunctionHelper files on the Symvionics web site:

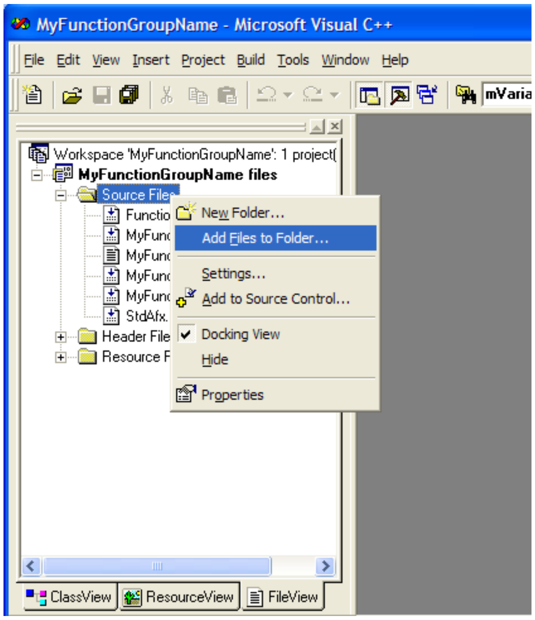
http://iads.symvionics.com/Downloads/SampleFunctionVC.zip

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

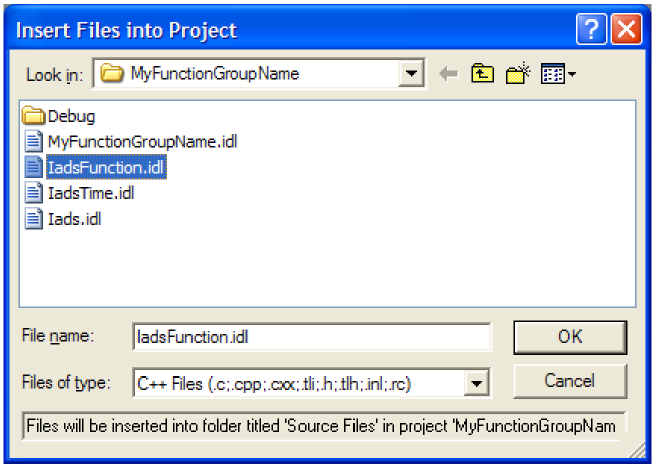
8) Now we'll add the IadsFunction.idl to the project. Click the FileView tab at the bottom of the Workspace viewer.



9) Expand your project files and right click the "Source Files" folder then click the "Add Files to Folder"



10) 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.



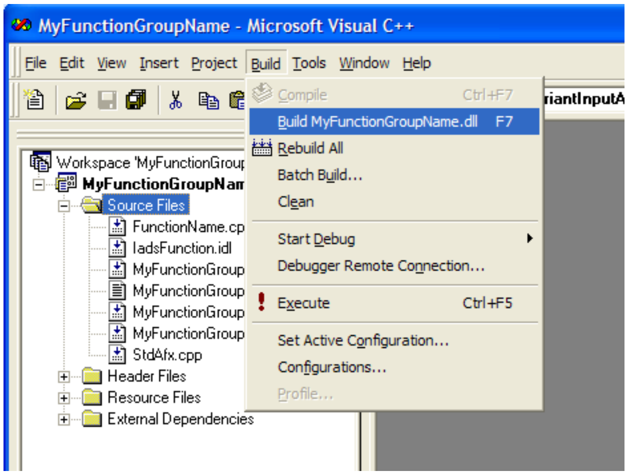
Look in: MyFunctionGroupName

File name: IadsFunction.idl

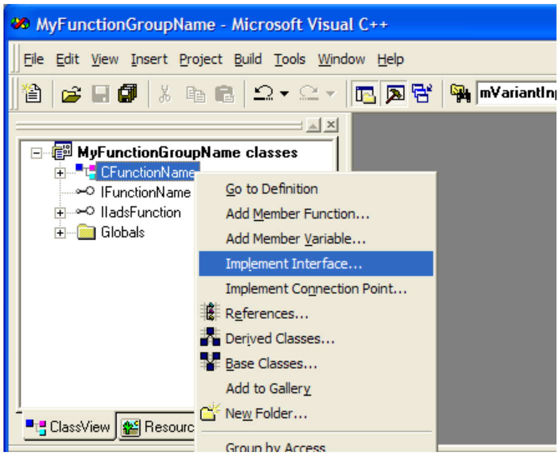
Files of type: C++ Files (.c;.cpp;.cxx;.tli;.h;.tlh;.inl;.rc)

Files will be inserted into folder titled 'Source Files' in project 'MyFunctionGroupNam

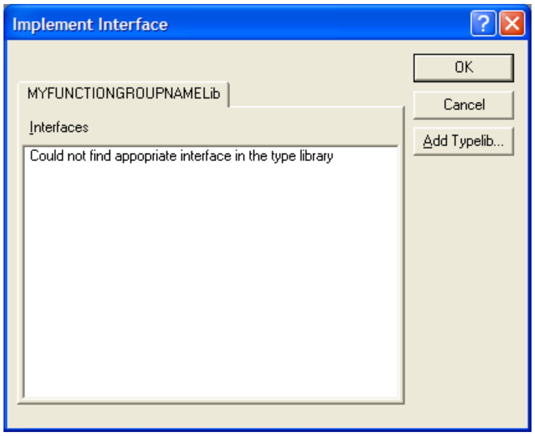
11) Build your project. After building the project, a "typelib" file will be created. We can use this typelib file to implement the interface. The typelib file is simply a compiled binary version of the IDL file.



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

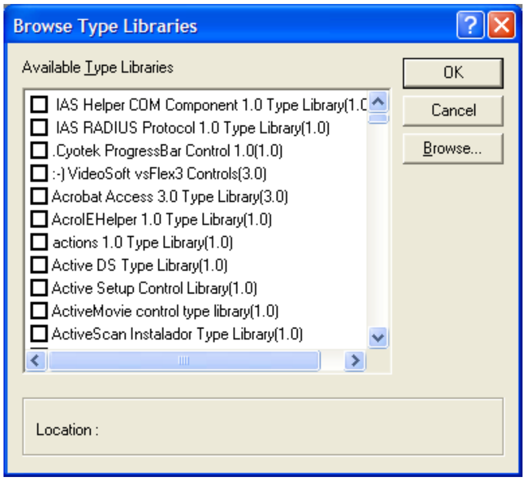


13) In the Implement Interface dialog, click the "Add Typelib…" button.



Could not find appropriate interface in the type library

14) Now click the “Browse…” button

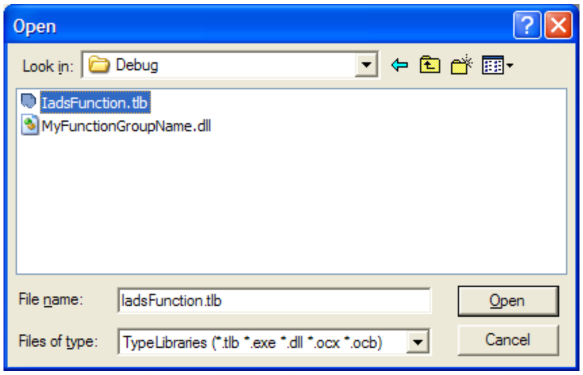


Available Type Libraries

☐ IAS Helper COM Component 1.0 Type Library(1.
☐ IAS RADIUS Protocol 1.0 Type Library(1.0)
☐ .Cyotek ProgressBar Control 1.0(1.0)
☐ :-) VideoSoft vsFlex3 Controls(3.0)
☐ Acrobat Access 3.0 Type Library(3.0)
☐ AcroIEHelper 1.0 Type Library(1.0)
☐ actions 1.0 Type Library(1.0)
☐ Active DS Type Library(1.0)
☐ Active Setup Control Library(1.0)
☐ ActiveMovie control type library(1.0)
☐ ActiveScan Instalador Type Library(1.0)

Location :

15) Browse to the “Debug” folder within your project (or whichever configuration you just compiled, default is Debug) and choose the “IadsFunction.tlb” file.



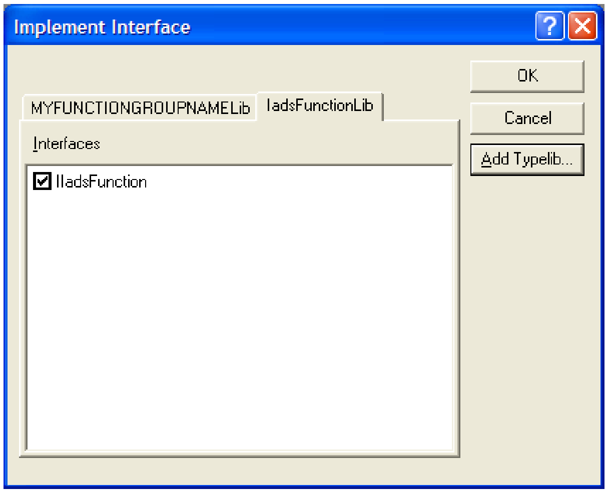
Look in: Debug

IadsFunction.tlb
MyFunctionGroupName.dll

File name: IadsFunction.tlb

Files of type: TypeLibraries (*.tlb *.exe *.dll *.ocx *.ocb)

16) Check the “IIadsFunction” interface and then press the Ok button.



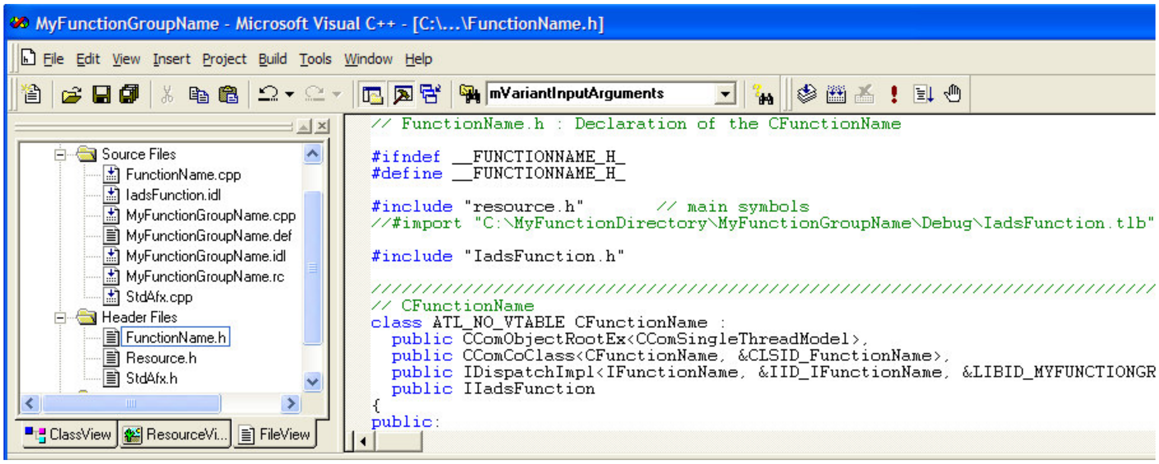
Interfaces

☑ IIadsFunction

17) Go back to the FileView tab in the Workspace viewer and open the "[Function Name].h" file. Comment out the line towards the top of the file that resembles:

#import "C:\MyFunctionDirectory\MyFunctionGroupName\Debug\IadsFunction.tlb" raw_interfaces_only, raw_native_types, no_namespace, named_guids

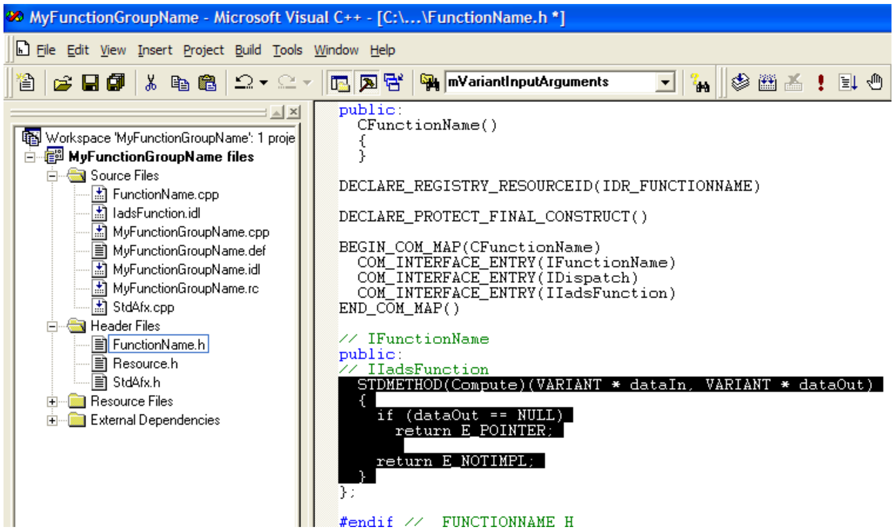
Then add: #include "IadsFunction.h"



```
// FunctionName.h : Declaration of the CFunctionName

#ifndef __FUNCTIONNAME_H_
#define __FUNCTIONNAME_H_

#include "resource.h"       // main symbols
//#import "C:\MyFunctionDirectory\MyFunctionGroupName\Debug\IadsFunction.tlb"

#include "IadsFunction.h"

/////////////////////////////////////////////////////////////////////////////
// CFunctionName
class ATL_NO_VTABLE CFunctionName :
  public CComObjectRootEx<CComSingleThreadModel>,
  public CComCoClass<CFunctionName, &CLSID_FunctionName>,
  public IDispatchImpl<IFunctionName, &IID_IFunctionName, &LIBID_MYFUNCTIONGR
  public IIadsFunction
{
public:
```

18) We're almost done now. At this point we can concentrate on the code. In the "[Function Name].h" file, scroll down to almost the end of the source code. Locate the wizard generated code below and Remove this entire function as we are about to inject some example code.

```
STDMETHOD(Compute)(VARIANT * dataIn, VARIANT * dataOut)
{
        if (dataOut == NULL)
                return E_POINTER;

        return E_NOTIMPL;


}
```

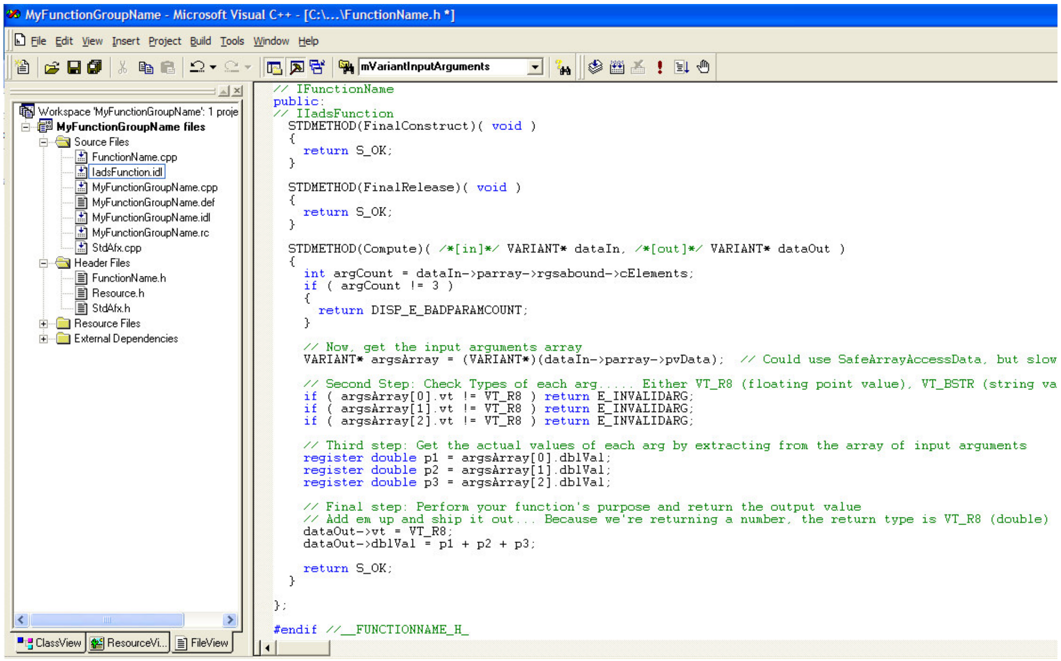


```
public:
  CFunctionName()
  {
  }

DECLARE_REGISTRY_RESOURCEID(IDR_FUNCTIONNAME)

DECLARE_PROTECT_FINAL_CONSTRUCT()

BEGIN_COM_MAP(CFunctionName)
  COM_INTERFACE_ENTRY(IFunctionName)
  COM_INTERFACE_ENTRY(IDispatch)
  COM_INTERFACE_ENTRY(IIadsFunction)
END_COM_MAP()

// IFunctionName
public:
// IIadsFunction
  STDMETHOD(Compute)(VARIANT * dataIn, VARIANT * dataOut)
  {
    if (dataOut == NULL)
      return E_POINTER;

    return E_NOTIMPL;
  }
};

#endif //  FUNCTIONNAME H
```

19) In the place of the code you just removed, insert the following example code.

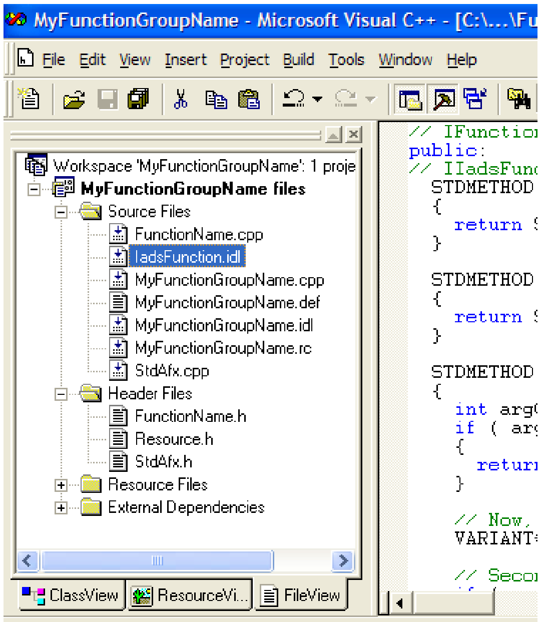
```
STDMETHOD(FinalConstruct)( void )
{
 return S_OK;
}

STDMETHOD(FinalRelease)( void )
{
 return S_OK;
}

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);  // Could use SafeArrayAccessData, but slow..

 // 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
 // Add em up and ship it out... 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:



```
// IFunctionName
public:
// IIadsFunction
  STDMETHOD(FinalConstruct)( void )
  {
    return S_OK;
  }

  STDMETHOD(FinalRelease)( void )
  {
    return S_OK;
  }

  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);  // Could use SafeArrayAccessData, but slow

    // Second Step: Check Types of each arg..... Either VT_R8 (floating point value), VT_BSTR (string va
    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
    // Add em up and ship it out... Because we're returning a number, the return type is VT_R8 (double)
    dataOut->vt = VT_R8;
    dataOut->dblVal = p1 + p2 + p3;

    return S_OK;
  }

};

#endif //__FUNCTIONNAME_H_
```

20) Because of a Visual C++ 6.0 bug, you'll need to remove the "IadsFunction.idl" file from your project or you will receive an error if you attempt to add another ATL object (function). Simply click on the IadsFunction.idl file in the File Workspace view and hit the Del (delete) key.



21) Build the project again (as shown in step 11). By building your project, the new dll should be registered so you're ready to run and debug it now. If you want to use your function on another PC, you'll need to register the dll. Please consult the web for documentation on "regsvr32.exe" and how to perform this procedure.

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.

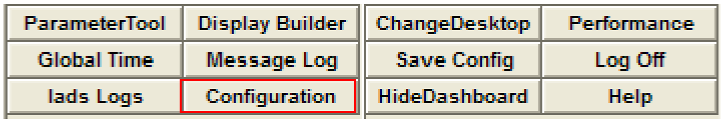
If you have any further questions, you can search the Iads Google Group or post a question:

http://groups.google.com/group/iads

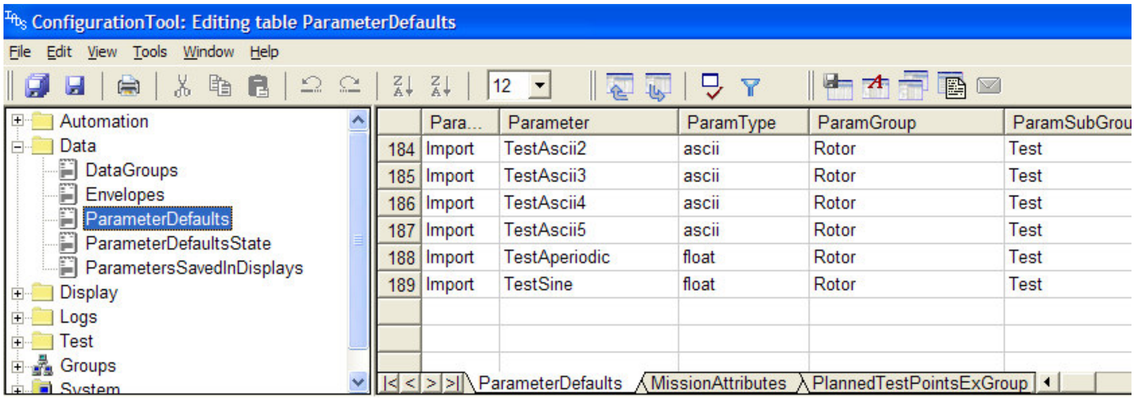
3. How to Access Your New Function in IADS

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.



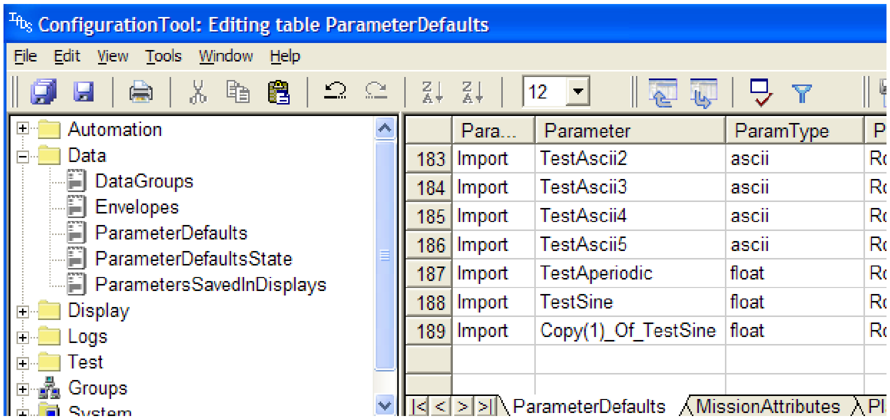
3) After pressing the button, the Configuration Tool dialog will appear. In the left window pane, click the "Data" folder and then finally the "ParameterDefaults" table. This is the location in Iads where you will build a new derived parameter to test your function.



	Para...	Parameter	ParamType	ParamGroup	ParamSubGrou
184	Import	TestAscii2	ascii	Rotor	Test
185	Import	TestAscii3	ascii	Rotor	Test
186	Import	TestAscii4	ascii	Rotor	Test
187	Import	TestAscii5	ascii	Rotor	Test
188	Import	TestAperiodic	float	Rotor	Test
189	Import	TestSine	float	Rotor	Test

4) Ok, let's add a new derived parameter. For speed, we'll simply copy the last line in the table and then replace our new values as needed. Select the last row in the table by pressing the row button (in the picture, button #189). 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're done, the table should look something like this:



	Para...	Parameter	ParamType	P
183	Import	TestAscii2	ascii	R
184	Import	TestAscii3	ascii	R
185	Import	TestAscii4	ascii	R
186	Import	TestAscii5	ascii	R
187	Import	TestAperiodic	float	R
188	Import	TestSine	float	R
189	Import	Copy(1)_Of_TestSine	float	R

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

Leave the first column alone and simply press the Tab key to start editing the second column. In the second column, type the name of your test parameter. Let's call it "TestMyFunction". Once you are done, press the Tab key as always. Now let's set the type of the parameter. Just leave it "float" (i.e. 4 byte floating point number). In the future, if you're testing an Ascii return value, you'll need to set this to Ascii.

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

In the next column (DataSourceArgument) you'll 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)*

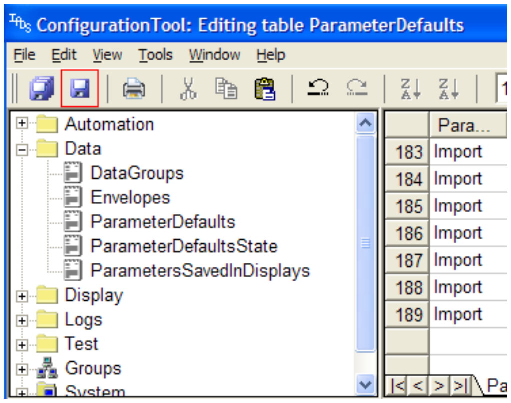
Or if you already have specific input parameters in mind, you can do something like this:

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.

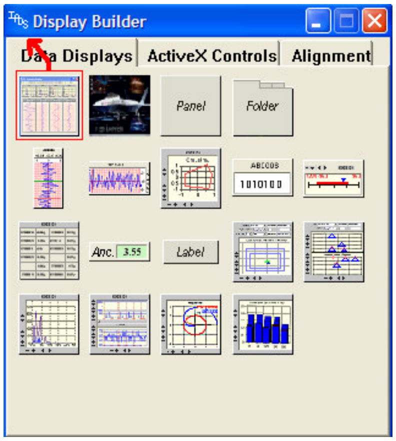
After these steps are complete, press the “Save” toolbar icon in the Configuration Tool.



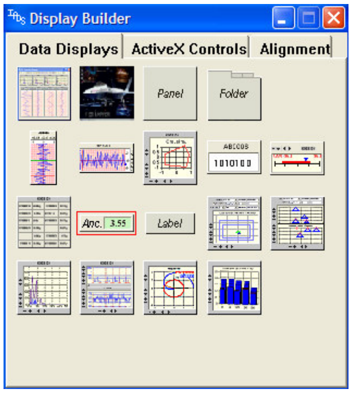
	Para…
183	Import
184	Import
185	Import
186	Import
187	Import
188	Import
189	Import

6) After you’ve clicked the Save button, your new parameter will appear in the Parameter Tool. To run the function, simply drop the parameter into any display. If you’re not familiar with building a test display and attaching a parameter, continue on the tutorial.

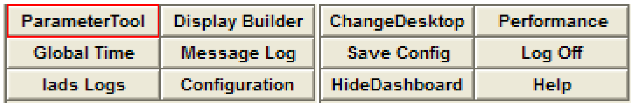
7) To build a test display, simply create an empty Analysis Window by dragging the icon from your Display Builder tool and on to your Microsoft Windows desktop and dropping it. After you’ve dropped the Analysis Window, you have a choice to name the window.



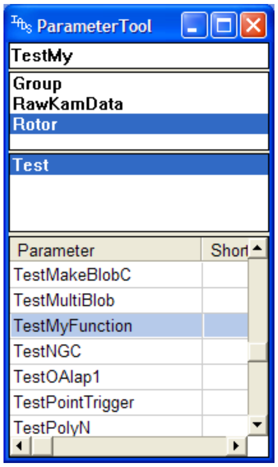
8) Now simply repeat the process, but drop the “AlphaNumeric” display into the Analysis Window (be sure drop the new display into the Analysis Window you’ve just created and not on to the Microsoft desktop). After the drop is complete, you should see the new display in the Analysis Window. The AlphaNumeric is a very simple text display that will be easy to view our equation output results.



9) 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.



10) In the top text field (quick find box), start typing the parameter name. I used the name “TestMyParameter”, so if you’ve done the same then simply type “TestMy”. You’ll 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’s 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! See that wasn’t too bad ;)



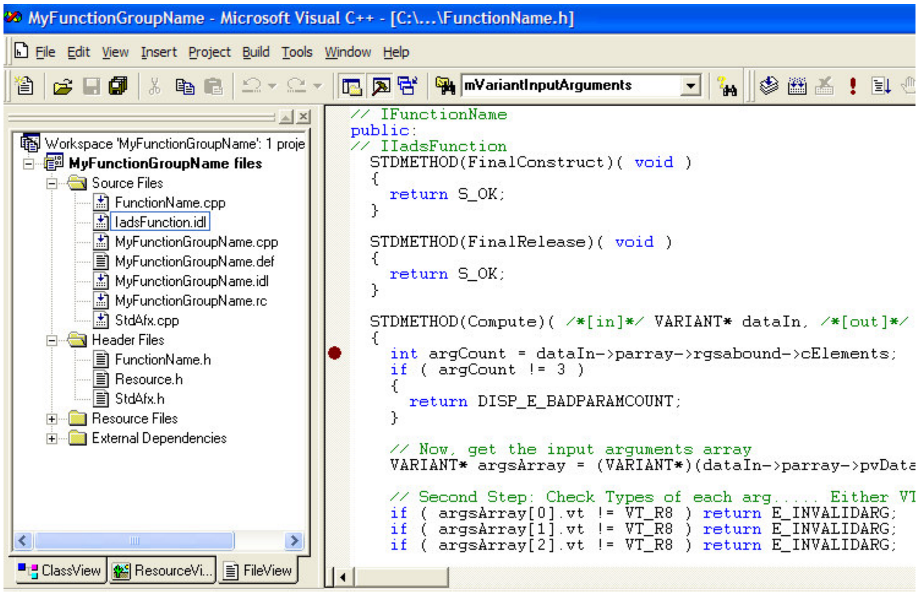
Parameter	Short
TestMakeBlobC	
TestMultiBlob	
TestMyFunction	
TestNGC	
TestOAlap1	
TestPointTrigger	
TestPolyN	

11) After your initial checkout is complete, you can move on to displays such as the Strip Chart that will show history and allow you to examine the data point by point for discrepancies. Simply repeat the process above at step 8, but in this instance use the icon just under the Analysis Window icon (first column second row). Make sure to save the configuration for later.

12) If you want to debug the function using the Visual C++ 6.0 debugger, continue to the next section.

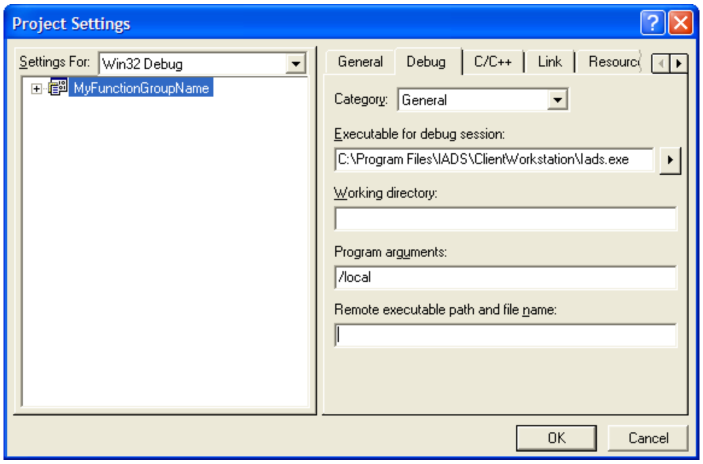
4. How to Debug Your New Function in IADS

1) Bring up your project, and place a break point in your “Compute” method for testing.



```
// IFunctionName
public:
// IIadsFunction
  STDMETHOD(FinalConstruct)( void )
  {
    return S_OK;
  }

  STDMETHOD(FinalRelease)( void )
  {
    return S_OK;
  }

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

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

    // Second Step: Check Types of each arg..... Either VT
    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;
```

2) Go to “Project->Settings” drop down menu in Visual C++ 6.0, 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. Add “/local” to your “Program arguments”.



Settings For: Win32 Debug

Category: General

Executable for debug session:
C:\Program Files\IADS\ClientWorkstation\Iads.exe

Program arguments:
/local

3) Build your project again for good measure and click on the “Go” command (or the F5 key). Iads will start.

4) In the Configuration Tool, create a derived parameter in the ParameterDefaults table. If you need more background info on how to do this, consult the last section. If you’ve 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 ParameterDefaults table, your function will be called. At this point you can debug all of the argument types and make sure you’re getting the correct items. If you have an argument error and return an error code from your function, notice that you’ll get an error message inside of Iads and the equation text will turn red in color. Once you’ve 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 Strip Chart) as described in the last section. 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.

   Again, 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.

   If you have any further questions, you can search the Iads Google Group or post a question:

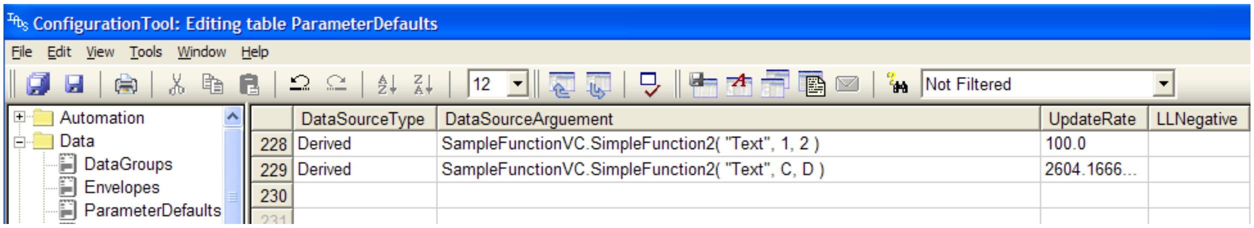
   http://groups.google.com/group/iads

5. Advanced Topics

5.1. 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 SampleFunctionVC project listed above.

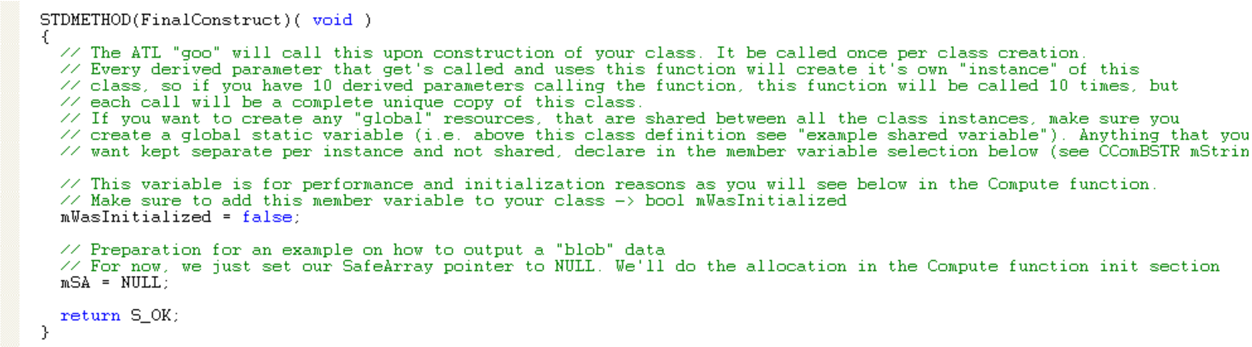
First, let’s examine the initialization stage of your function in general. Each and every time a derived parameter is created that references your custom function, an instance of your custom function object is created within the parameter’s computational engine. When the parameter requires data, this 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.



	DataSourceType	DataSourceArguement	UpdateRate	LLNegative
228	Derived	SampleFunctionVC.SimpleFunction2("Text", 1, 2)	100.0	
229	Derived	SampleFunctionVC.SimpleFunction2("Text", C, D)	2604.1666...	
230				

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'll 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.

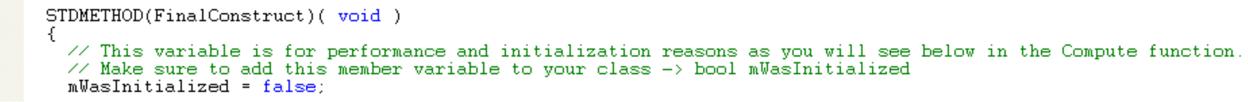
Now let's examine the initialization stage in more detail. The function name (i.e. ProgID) within the derived equation is used to call the "CoCreateInstance" function in the Microsoft COM libraries to create your object. Once your object is created within Iads, the "FinalConstruct" method is called. 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;
}
```

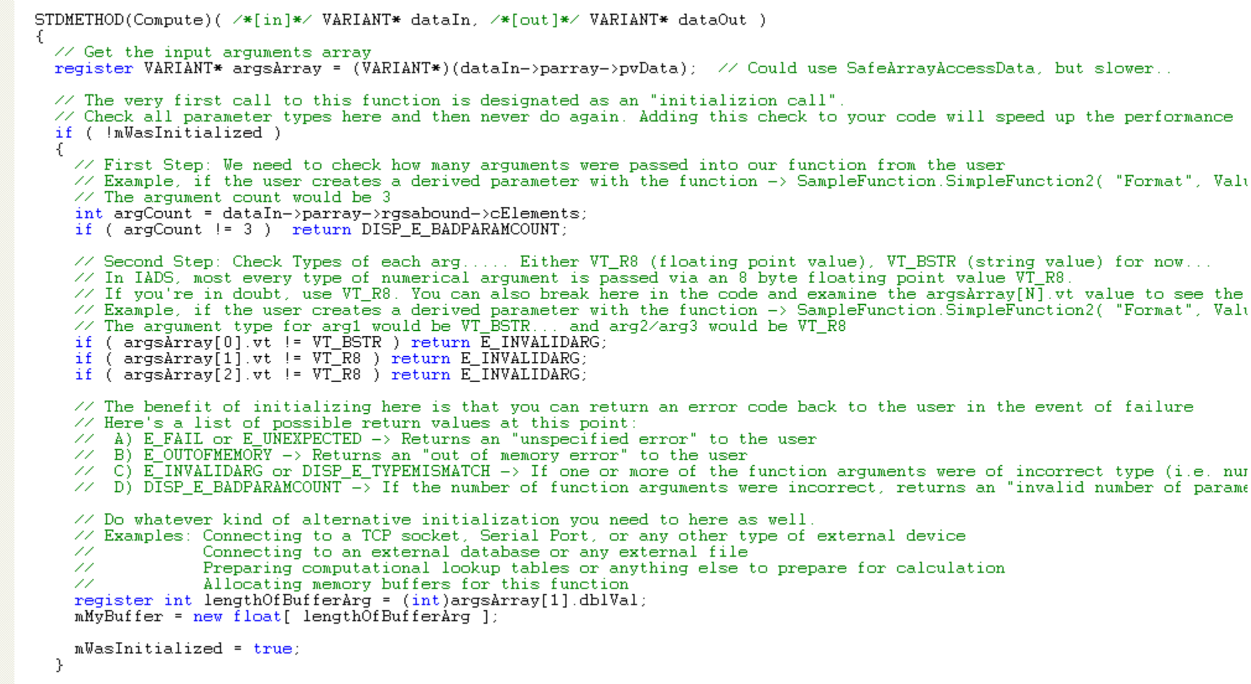
For instance, say you were building a function allowed a user to specify a number of data points to "buffer" before computing a results. Of course you'll need a member variable in the class to hold this buffer. During the FinalConstruct, you would set your member variable buffer pointer to NULL, but 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.



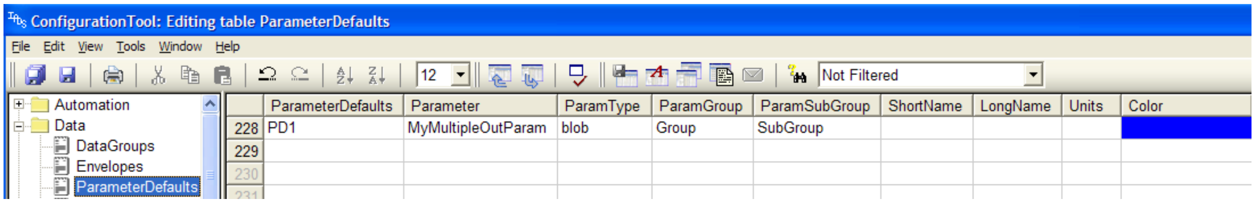
```
STDMETHOD(Compute)( /*[in]*/ VARIANT* dataIn, /*[out]*/ VARIANT* dataOut )
{
  // Get the input arguments array
  register VARIANT* argsArray = (VARIANT*)(dataIn->parray->pvData);  // Could use SafeArrayAccessData, but slower..

  // 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", Val
    // 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", Val
    // 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.

5.2. Returning Multiple Results from your Custom Function

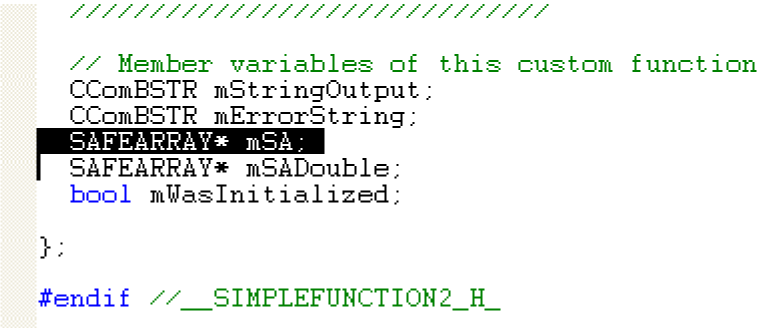
One of the apparent limitations regarding the custom function technique described above is that it seems unable to return multiple values. As we have learned in the previous section, each input argument that is supplied in the derived equation is sent into the Compute function, the custom code uses these input values to calculate the result, and then the single result is returned to Iads. Suppose you had a function with 5 input arguments, but instead of only outputting a single result, it outputs 5 results. This problem can be solved in a simple fairly manner.


ConfigurationTool: Editing table ParameterDefaults
File Edit View Tools Window Help
12
Not Filtered
Automation
Data
DataGroups
Envelopes
ParameterDefaults

	ParameterDefaults	Parameter	ParamType	ParamGroup	ParamSubGroup	ShortName	LongName	Units	Color
228	PD1	MyMultipleOutParam	blob	Group	SubGroup				🟦
229									
230									
231									

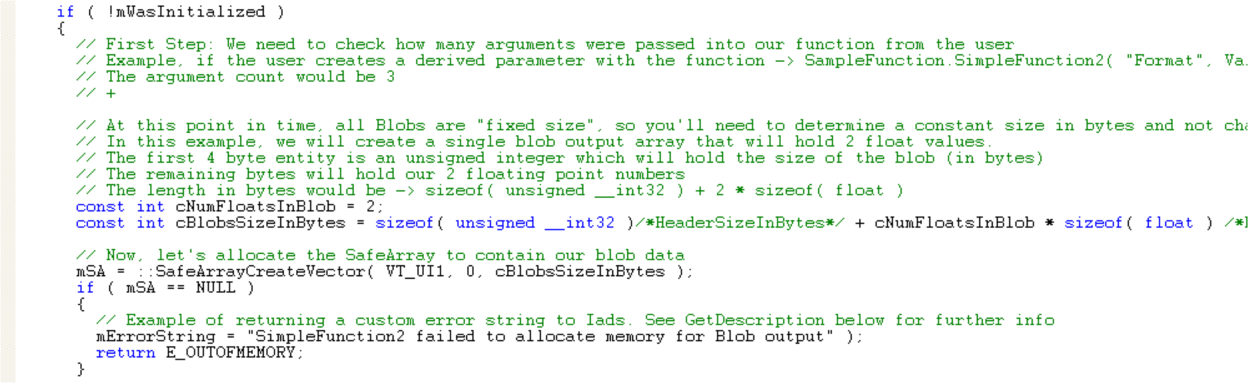
When a function needs to output multiple answers in a single computation, we can simply output an “array” of answers. This array type output is referred to in Iads as a BLOB (binary large object). Once the array/blob is output from your custom function, it can then be returned as a blob type parameter 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’ll need to create an array to output our 5 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.



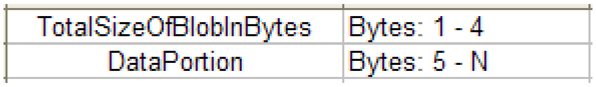
```
  //////////////////////////////

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

};

#endif //__SIMPLEFUNCTION2_H_
```

Now we’ll need to allocate the memory for this array. Carrying on the initialization discussion from the last section, we’ll perform the allocation in the Compute function within the “first time only" portion of the function. To create the array, we'll 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

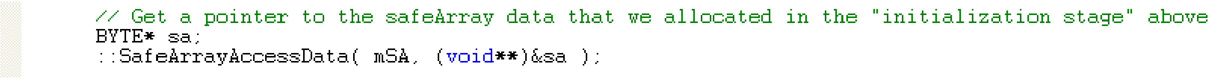
Or

BlobSizeInBytes = 4 + TotalSizeOfDataPortionInBytes

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

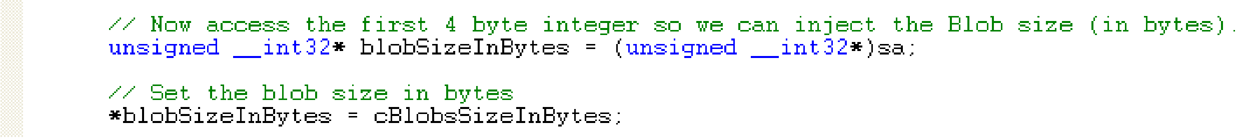
BlobSizeInBytes = sizeof(unsigned __int32) + sizeof(float) * 5

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'll need to access the array pointer within the SAFEARRAY. To do this, we simply call the SafeArrayAccessData function.



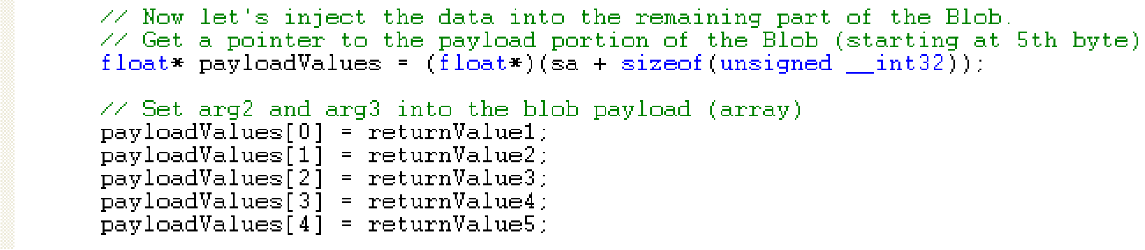
```
// Get a pointer to the safeArray data that we allocated in the "initialization stage" above
BYTE* sa;
::SafeArrayAccessData( mSA, (void**)&sa );
```

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).



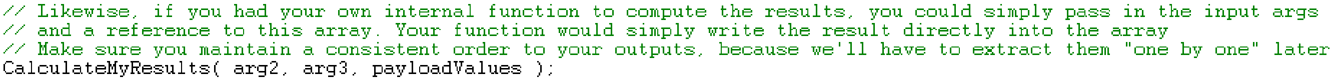
```
// Now access the first 4 byte integer so we can inject the Blob size (in bytes).
unsigned __int32* blobSizeInBytes = (unsigned __int32*)sa;

// 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 fieldwe just set above).



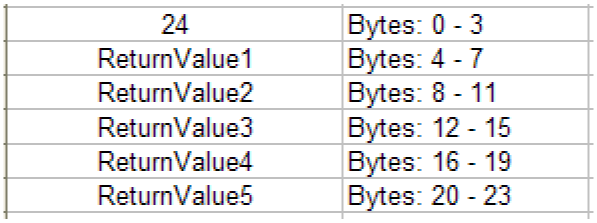
```
// Now let's inject the data into the remaining part of the Blob.
// Get a pointer to the payload portion of the Blob (starting at 5th byte)
float* payloadValues = (float*)(sa + sizeof(unsigned __int32));

// 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.



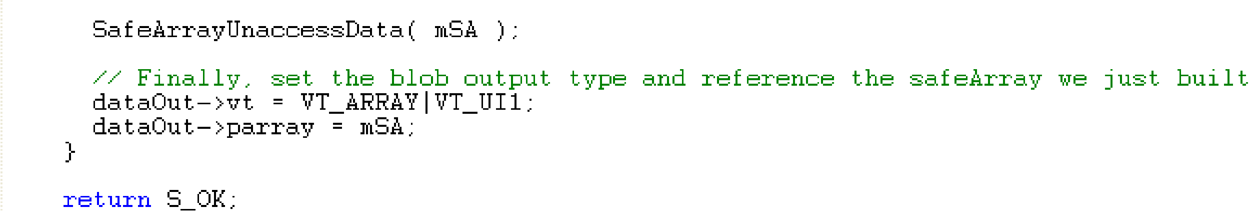
```
// Likewise, if you had your own internal function to compute the results, you could simply pass in the input args
// and a reference to this array. Your function would simply write the result directly into the array
// Make sure you maintain a consistent order to your outputs, because we'll have 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):



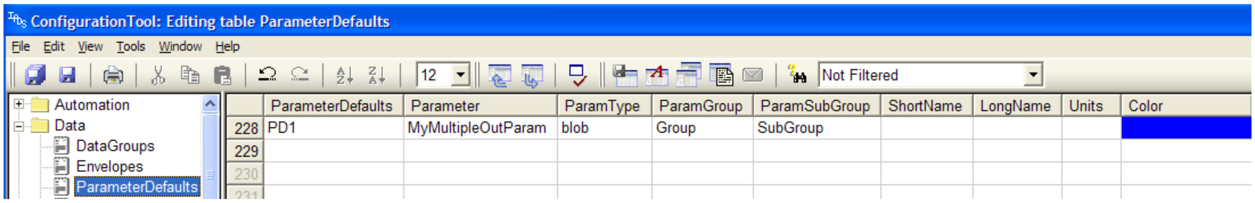
24	Bytes: 0 - 3
ReturnValue1	Bytes: 4 - 7
ReturnValue2	Bytes: 8 - 11
ReturnValue3	Bytes: 12 - 15
ReturnValue4	Bytes: 16 - 19
ReturnValue5	Bytes: 20 - 23

Once you have completed setting the return values into the array, it's 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.



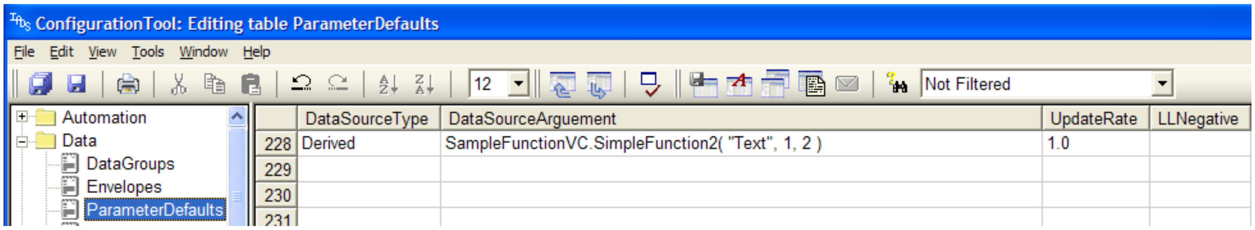
```
        SafeArrayUnaccessData( mSA );

        // 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'll 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.


ConfigurationTool: Editing table ParameterDefaults
File Edit View Tools Window Help
12
Not Filtered
Automation
Data
DataGroups
Envelopes
ParameterDefaults

	ParameterDefaults	Parameter	ParamType	ParamGroup	ParamSubGroup	ShortName	LongName	Units	Color
228	PD1	MyMultipleOutParam	blob	Group	SubGroup				🟦
229									
230									
231									

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.

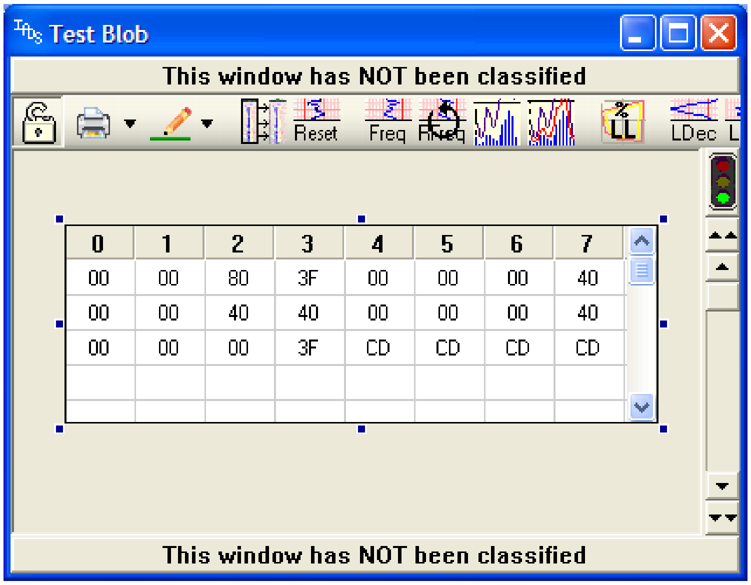

ConfigurationTool: Editing table ParameterDefaults
File Edit View Tools Window Help
12
Not Filtered
Automation
Data
DataGroups
Envelopes
ParameterDefaults

	DataSourceType	DataSourceArguement	UpdateRate	LLNegative
228	Derived	SampleFunctionVC.SimpleFunction2("Text", 1, 2)	1.0	
229				
230				
231				

Now, scroll over to the DataSourceType column, and set it to "Derived". In the DataSourceArguement 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.

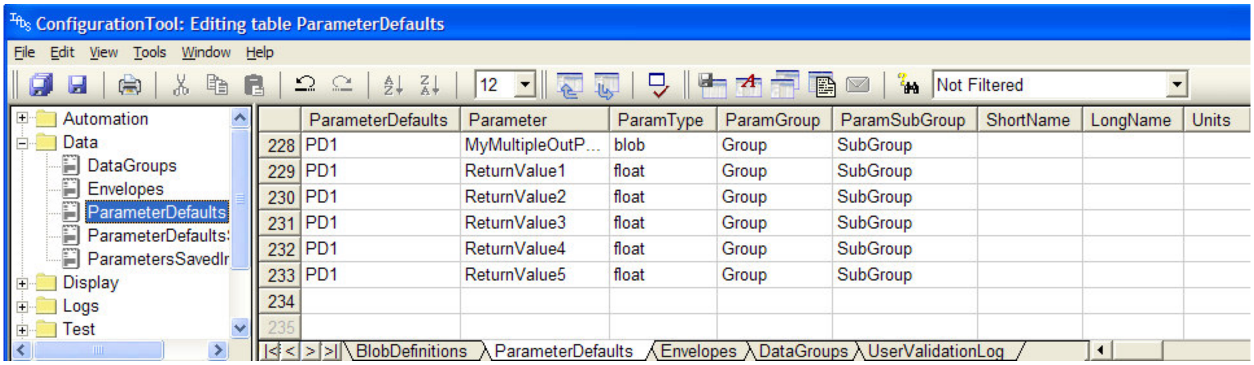
At this time, if you wish to see the raw output of your function you can drop your newly created derived parameter into the "IadsBusMessageDisplays.BlobViewer" display. If you right click on the ActiveX tab of the Display Builder in Iads, you can add the Blob Viewer to your available displays list. Once that is complete, drag and drop the Blob Viewer display into an AnalysisWindow. 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.


Test Blob
This window has NOT been classified
Reset
Freq
RFreq
LDec
This window has NOT been classified

0	1	2	3	4	5	6	7
00	00	80	3F	00	00	00	40
00	00	40	40	00	00	00	40
00	00	00	3F	CD	CD	CD	CD

Now that we know our blob is alive (no pun intended), 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.



	ParameterDefaults	Parameter	ParamType	ParamGroup	ParamSubGroup	ShortName	LongName	Units
228	PD1	MyMultipleOutP...	blob	Group	SubGroup			
229	PD1	ReturnValue1	float	Group	SubGroup			
230	PD1	ReturnValue2	float	Group	SubGroup			
231	PD1	ReturnValue3	float	Group	SubGroup			
232	PD1	ReturnValue4	float	Group	SubGroup			
233	PD1	ReturnValue5	float	Group	SubGroup			
234								
235								

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'll specify 31.

The sixth argument is the actual "data type" that we want to return from the function. In this case it's an IEEE float, so we'll 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'll 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:

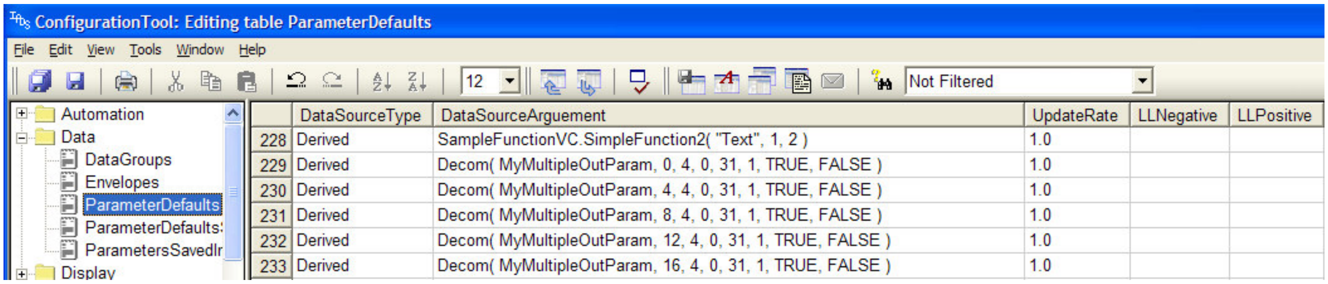
ReturnValue1 => Decom(MyMultipleOutParam, 0, 4, 0, 31, 1, TRUE, FALSE)

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.