Intelligent Machine Architecture 1.0
Programmer's Guide – Mechanism Walkthrough

ROBERTO OLIVARES*

*Vanderbilt University, Department Of Electrical Engineering & Computer Science, Nashville, TN.

Intelligent Robotics Laboratory – Internal Technical Document, June 10th, 2003.

1. INTRODUCTION

This document is meant to provide a walkthrough for creating IMA 1.0 mechanisms. We discuss how to create an IMA project, configure the project, insert a mechanism into the project, and how to add code to the resulting mechanism. We also address how to implement bindings and override basic IMA functions. This document does not address basic IMA principles, how to use AgentBuilder, or how to program in Visual C++ and COM. For background in these areas, we refer readers to the *IMA 1.0 Introduction & System Overview,* the *IMA 1.0 AgentBuilder Reference*, and the technology-specific books referenced there.

2. OVERVIEW

The process of creating a blank IMA mechanism is relatively straightforward. The majority of the work involves implementing the mechanism's algorithm and linking it to other existing components. This document guides developers through the easy part and provides and overview of the harder areas. An important thing to remember when creating mechanisms is that they are designed to be a passive function libraries—repositories for functions that will be called by engines such as the *StateMachineEngine*. Additionally, mechanisms must provide COM properties and methods so that they can be configured from AgentBuilder and interact with other components.

The first step of this mechanism creation process is using Visual C++ to create a new IMA 1.0 mechanism project. Then, the project must be configured and components inserted into it. The developer will then have to add properties, methods, and bindings to the component, as well as mechanism-specific functionality. The project is then compiled into a DLL that will be used by AgentBuilder to insert the component into an agent.

In IMA 1.0, all component development is performed in the Microsoft Visual C++ 6.0 development environment. This is due to limitations imposed by COM when IMA was first designed. This situation provides both advantages and disadvantages. Developers benefit from having a C++ level of control over their code performance, Windows API, and hardware drivers. On the other hand, C++ developers must contend with the complexity of COM's C++ implementation, as well as the pointer, complexity, and resource allocation problems C++ is infamous for.

3. CREATING A NEW IMA PROJECT

Starting Visual C++. To create a new IMA 1.0 project, the Visual C++ application (also referred to as VC++ or Visual Studio) must first be started. This is accomplished by selecting *Start→Program Files→Microsoft Visual C++ 6.0* (see Figure 1).

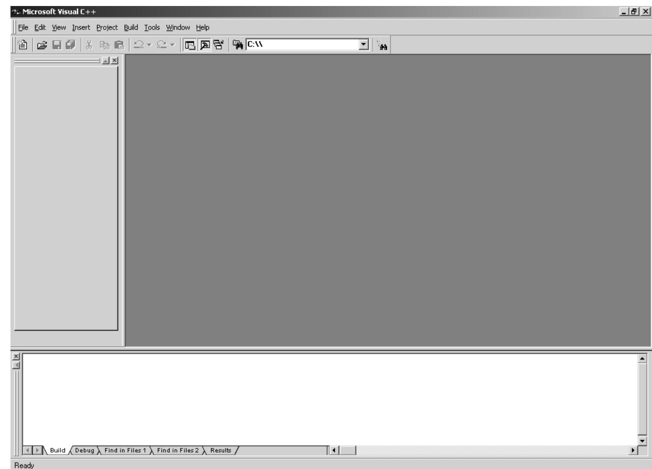

Figure 1. The Visual C++ Development Environment.

Creating an IMA Project. Once Visual C++ has been started, a new workspace and project must be created. A workspace simply contains multiple projects. To create a new IMA project, select *File→New...* to display the *New Project Dialog* (see Figure 2). Selecting the *Projects* tab will display a list of project types. For IMA 1.0 projects, the *ATL COM AppWizard* item should be selected. Entering a directory such as the desktop or your personal network drive for the *Location* field is recommended. A project name must also be specified. The project name will be both the filename for your DLL, and the *projectname* portion of the *projectname.classname.1* PID descriptor. Project names in IMA 1.0 should be short and reflect the types of components provided. *ControlMechs* or *CameraReps* are good examples.

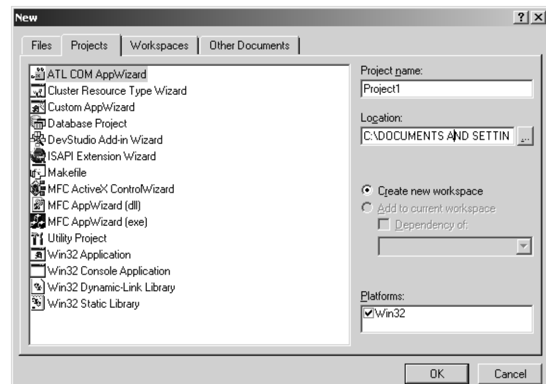

Figure 2. The New Project Dialog.

After the project name has been specified, pushing the *OK* button will lead to the *ATL COM AppWizard Dialog*. (see Figure 3). The first question asked by the wizard is which COM server to generate. IMA 1.0 components are contained in *Dynamic Link Libraries (DLLs)*, which should be selected. None of the other options on this page should be checked[1]. Pushing *Finish* will then generate a confirmation screen. Pushing *OK* at the confirmation screen will return you to the main screen, with a new node added to the *Workspace View (WV)* on the left hand side of the screen.

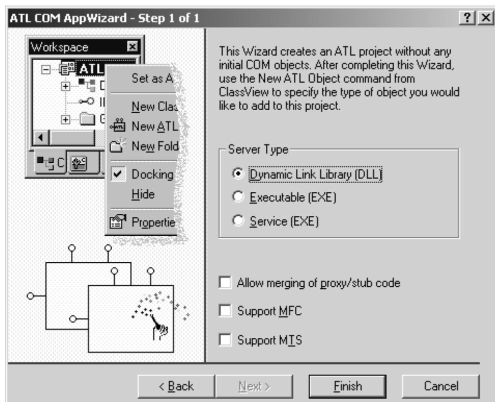

Figure 3. Selecting a DLL project.

Referencing BasicIMA_i.c. Before we can add components to or compile the project, certain compiler options must be configured. The first of these involves adding the *BasicIMA_i.c* file to the project's build list. Without this file included in the project, Visual C++ won't recognize the IIDs and CLSIDs for IMA 1.0 components. To add this file, first select the *Fileview* tab from the *Workspace* panel (see Figure 4). Next, right-click the *Header Files* item and select *Add Files To Folder* to bring up the *Add File Dialog*. Selecting the *I:\Include\BasicIMA_i.c* file and pushing *OK* will then add it to the project.

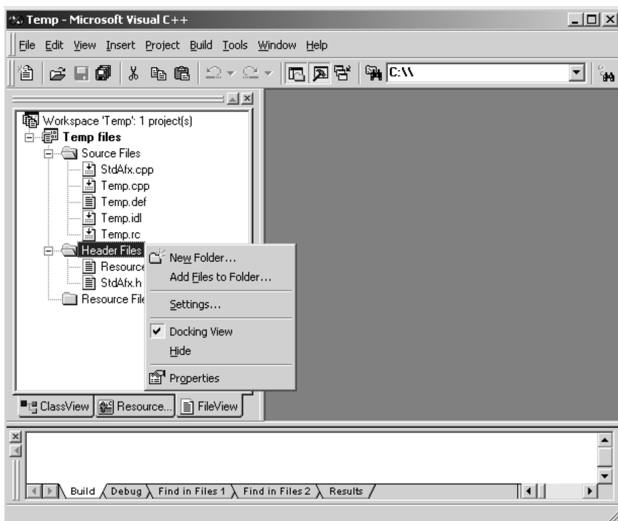

Figure 4. Referencing the BasicIMA_i.c file.

[1] If a manager was being created, the *Support MFC* box would need to be checked.

Disabling Precompiled Headers. Once the *BasicIMA_i.c* file has been added to the project, its precompiled header must be disabled for the project to compile. This is accomplished by selecting *Project→Settings* from the application menu to display the *Project Settings Dialog* (see Figure 5). Selecting the *BasicIMA_i.c* item, the *C/C++* tab, and the *Precompiled Headers* item in the *Category* field will then allow the *Not using precompiled headers* option to be selected.

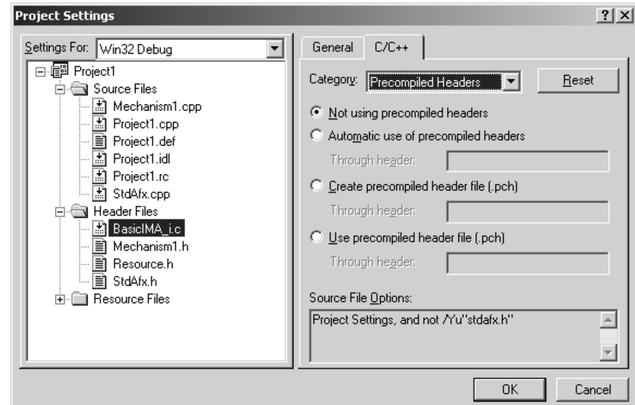

Figure 5. Disabling precompiled headers.

Enabling Exception Handling. Enabling exception handling will reduce the number of compiler warnings generated for your project. This can be done in the same dialog by selecting the *C++ Language* item under the *Category* listbox on the *C/C++* tab (see Figure 6). Checking the *Enable Exception Handling* checkbox will turn on the feature.

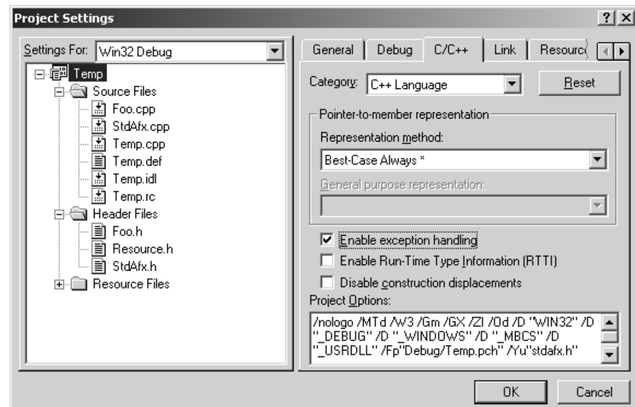

Figure 6. Enabling Exception Handling.

Referencing AgentBuilder. After disabling precompiled headers and enabling exception handling, AgentBuilder must be configured as the host process for debugging components. Selecting the *Debug* tab and the *General* item in the *Category* listbox allows the AgentBuilder path to be specified in the *Executable for debug session field* (see Figure 7). After this had been done, executing the project (pushing F5) will run the component's DLL within AgentBuilder, allowing breakpoints to be used.

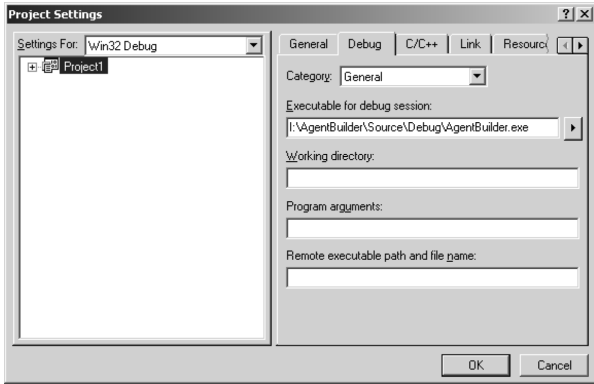

Figure 7. Specifying AgentBuilder as the host process.

At this point, pushing the *OK* button will commit the changes to your project settings, allowing the IMA 1.0 project to compile correctly.

4. INSERTING A NEW COMPONENT.

Creating the ATL Object. After having created and configured the IMA project, you will want to add components to it. We now provide a walkthrough of adding a Mechanism component to a project. First, select the *Classview* tab on the *Workspace* panel and right-click on the project's name. Next, select *New ATL Object…* from the menu to display the *ATL Object Wizard Dialog* (see Figure 8).

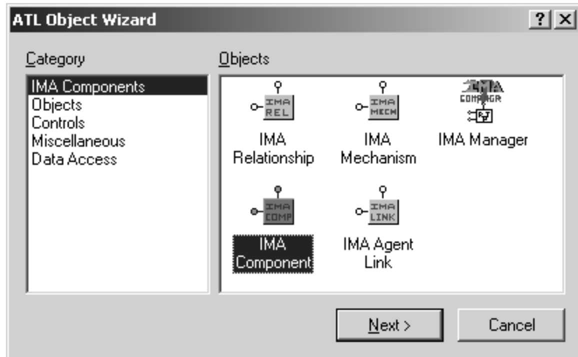

Figure 8. Specifying which type of IMA component.

Selecting the *IMA Components* item in the *Category* listbox should display the templates for various IMA components. If this category does not appear, or if the icons in Figure 8 are not present, please consult the *Configuring Component Templates* document on the *I:* drive. For walkthrough purposes, we will select the *IMA Mechanism* object and push the *Next* button.

Specifying Class Settings. Next, the ATL Object Wizard will ask for C++ and COM names for the class (see Figure 9). Typing a descriptive name such as *CameraMech* or *FuzzyControllerMech* into the *ShortName* field is recommended, as VC++ will generally fill out the other fields for you. It is not recommended that the other fields be filled in manually. Pushing *OK* will then create the class for you. The class in Figure 9 will have a COM class name of *Mechanism1*, a default interface called *IMechanism1*, and a *PID* of *Project1.Mechanism1*. Because the class was generated with the IMA template, it will also implement the IMA's *IComponent* and *IMechanism* interfaces for communication with the system.

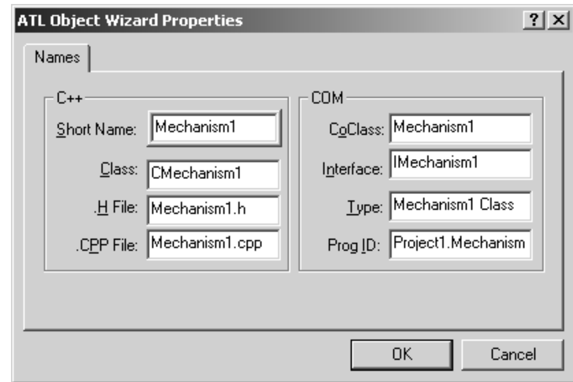

Figure 9. Specifying class settings.

Checking For Compiler Errors. At this point, the class has been generated and it is a good idea to check for compile errors before adding any new code to the project. Selecting *Build→Build* (or pushing F7) will attempt to compile, link, and register the project. It is acceptable for a series of *unwind semantics* warnings to be generated when compiling the IMA components, but no *errors* should be present. If errors are present, the previous steps should be checked for correctness.

Component Files. When a new IMA component is created, VC++ adds the source file (.CPP) and the header file (.H) for your component into the project. The source code file contains the actual code for your normal and COM/IMA methods and properties. The header file contains function prototypes for both these functions, as well as the inheritance of the IMA interfaces and the mapping of your default interface's methods and properties to functions on your object (see Figure 10). In addition to these two files, two sections of code are added to the *projectname.IDL* file that describe your object and its default interface in COM's *Interface Description Language (IDL)*.

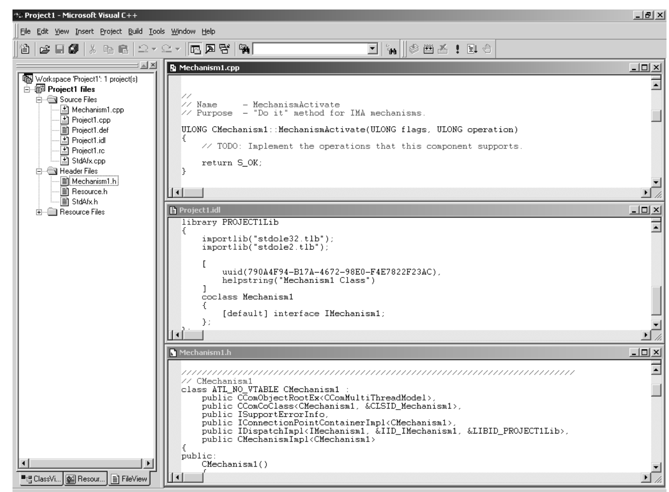
Figure 10. Component Files.

Due to the nature of COM programming, new IMA developers should not directly modify the header or IDL files themselves. Making a modification in either the

source code, header, or the IDL file usually involves making parallel changes in the other two files. If not done correctly, this can lead to mysterious compiler or runtime errors in the project. More information on how to program with ATL and IDL in COM can be found in Don Box's "*Essential COM*".

5. ADDING PROPERTIES AND METHODS.

Standard Properties & Methods. In IMA projects, there are two types of methods and properties: standard ones that are not exported by COM and those that are exported by COM. Standard properties, methods, and member variables can be added to the component as in normal C++ coding, or by right-clicking on the class's icon in the Class View and selecting *Add Member Function* or *Add Member Variable*. Properties and methods added this way are not visible by other IMA components and are typically for the component's internal use only.

COM Properties & Methods. In order to expose functionality to other components on the IMA network, COM properties and methods must be added to your component. These are the ones visible in AgentBuilder. Adding these by manually changing the .CPP, .H, and .IDL files is tricky and could irreversibly harm your project. It is suggested that the process be handled through VC++ by right-clicking on the I*ComponentName* icon (it looks like a metal link) nested under your class in the Class View (see Figure 11) and choosing the *Add Method* or *Add Property* options.

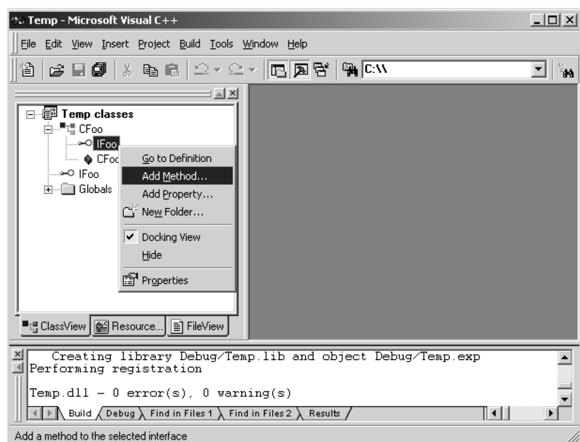

Figure 11. Adding COM methods and properties.

Adding COM Methods. To add a method to your component's default interface, right-click on the default interface in the *Classview*, then select *Add Method* (see Figure 12). The method's name and parameters should then be specified in the appropriate fields. Figure 12 shows a method that accepts an integer and a string argument, then returns an integer. As you type this information into the dialog box, it will generate and display the resulting IDL for you. Pushing *OK* will then add the function IDL, header, and implementation skeleton to your component files. Methods with return values should specify their last parameter as a pointer with an *[out, retval]* IDL prefix (see below).

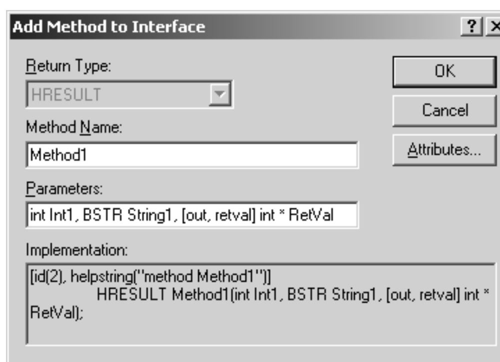

Figure 12. Adding a COM method.

Adding COM Properties. To add a property to your component's default interface, right-click on the default interface in the *Classview*, then select *Add Property* (see Figure 13). A pre-specified property type should be selected from the *Property Type* listbox and a property name specified. Indexed properties can be made by adding property parameters, such as integer index. Normal properties should have both a *get* and *put* function, but read-only properties can be made by unchecking the *Put Function* checkbox. At this point, pushing *OK* will add the specified property to your component files.

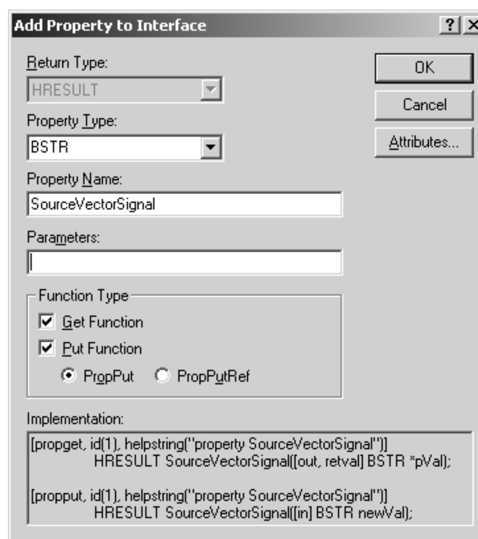

Figure 12. Adding a COM property.

In the figure above, we are adding a property called *SourceVectorSignal* that is of type *BSTR* (a COM string type). This is the typical format for a property that defines a binding target. The wizard will then generate a *GetSourceVectorSignal* function and a *PutSourceVectorSignal* function that will be called when the property is read or changed, respectively.

Properties, Methods, and HRESULTS. The distinction between properties and methods is often not very clear. In COM, both are examples of functions, as properties are implemented with *GetPropertyname* and *PutProp-*

ertyname functions. As a rule of thumb, COM properties should encapsulate member variables that are read or changed regularly and that do not impose significant processing penalties. Properties should also reflect configurable aspects of components, since they can be changed from AgentBuilder at runtime. Methods should reflect functionality that is meant to be called from programs only, or functionality that may invoke processing delays or generate errors. Because of this, calling methods from AgentBuilder is generally discouraged.

In COM, both properties and methods always return *HRESULT*s. This is an integer data type that reflects the status of the distributed function call. Typically, we assign this return value *S_OK* to indicate that all went well. Indicating a value other than *S_OK* will signal that an exception should be generated upon returning. Since HRESULTs are always returned in COM, a function's return value must be added to the parameter list as a pointer which COM can read and marshal (transmit via TCP/IP to the calling object). This pointer, as mentioned earlier, must be flagged with an *[out, retval]* prefix in the IDL description[2] to be recognized as a return value. The calling object must provide the memory space the pointer points to and read the value at that location upon return.

6. ADDING MECHANISM OPERATIONS.

IMA mechanisms support an additional type of method definition via the *MechanismActivate* function (see Figure 14). When called, this function tells the mechanism to execute one of its internal functions as specified by a numeric parameter. When using the *StateMachineEngine*, for example, the mechanism's path, desired operation, and flags are specified for each state. Then, when the *StateMachineEngine* is activated, the *MechanismActivate* function on the component is called with the appropriate *operation* and *flags* parameters.

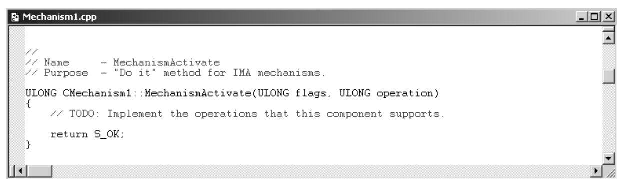

Figure 14. Template-generated MechanismActivate function.

Mechanisms can support multiple operations by providing a switch statement in their *MechanismActivate* function. Because the operations supported by a mechanism are not explicitly published in IMA 1.0, it is very important to comment the header of the *MechanismActivate* function with a list of operations and what they do.

7. IMPLEMENTING BINDINGS

Declaring Pointers. The first step in supporting bindings is declaring an *IUnknown* pointer member variable for each binding target. These pointer declarations are added into your member variables section of the component's .H file as follows:

```
IUnknown*     m_pIUnkVectorIn;
IUnknown*     m_pIUnkVectorOut;
```

These are declared as *IUnknown* pointers due to the way COM and IMA work with them internally. The prefix on the variables is Hungarian notation for m*ember variable pointer to an IUnknown interface.*

Setting Up Component Links. After your pointers have been declared, your component must be configured to recognize them as bindings. This can be accomplished by adding the following code to your class constructor:

```
SetupComponentLink(&IID_IUnknown, &m_pIUnkVectorIn);
SetupComponentLink(&IID_IUnknown, &m_pIUnkVectorOut);
```

This code activates macros that initialize and handle the binding process for you.

Retrieving & Changing Target Paths. After component links have been set up, you must add code to link changes in a string method to the binding system. The following code should be placed in the *get* and *put* functions for one of your component's string properties:

```
STDMETHODIMP CMech::get_InputVector(BSTR * pVal)
{
        GetComponentLinkName(&m_pIUnkVectorIn, pVal);
        return S_OK;
}

STDMETHODIMP CMech::put_InputVector(BSTR newVal)
{
        UpdateComponentLinkName(&m_pIUnkVectorIn, newVal);
        return S_OK;
}
```

The code provided automatically updates the binding when the path is changed in AgentBuilder.

Obtaining Target Pointers. Now that the component is configured to handle bindings, your code will want to be able to resolve those bindings into a pointer to the target objects:

```
long VectorLen = 0;
IVectorSignal* pIVSInput = NULL;

CComQIPtr <IVectorSignal, &IID_IVectorSignal > pIVSInput(m_pIUnkVectorIn);

if(pIVSInput == NULL) return S_FAIL;
pIVSInput->get_VectorLength(&VectorLen);
pIVSInput->Release();
return S_OK;
```

This code snippet declares a pointer to the *VectorSignal* interface on our binding target, then attempts to obtain that interface from the target. If obtaining the interface fails, the function exits with a failure code. Otherwise, the vector length is retrieved from the *VectorSignal*. The pointer

[2] In the IDL description only, not in the .H or .CPP files.

to the target component is then released and the function returns.

8. ICOMPONENT FUNCTIONS

When implementing any IMA component, certain functions on the IComponent interface are overridable in the source file for your convenience:

FinalComponentInitialization. Called when the initialize button is pushed in AgentBuilder. This function allows your component to perform any final setup before being bound or activated. This may be called more than once after a component is inserted.

SaveComponentToStream. Called by AgentBuilder when an components are saved to an agent file. This function should save the state of your component to the provided binary stream.

InitComponentFromStream. Called by AgentBuilder when components are loaded from an agent file. This function should load the state of your component from the provided binary stream.

GetMaxStreamSize. Called by AgentBuilder to estimate the size of your component's data. This function should return the size of the stream necessary to save the component.

Additionally, two more overridable functions are provided in the header file for your component:

FinalConstruct. Called by AgentBuilder after the component has been successfully constructed. This function should allocate hardware resources and setup component links.

FinalRelease. Called when the component is being removed from the agent. This function should release hardware resources and any other COM pointers.

9. IMECHANISM FUNCTIONS

When a mechanism is generated using the IMA 1.0 templates, two functions are generated in the source file:

MechanismActivate. This function is called by engines, AgentBuilder, or other components to access a mechanism's functionality at runtime. The *operation* and *flags* parameters allow the client to specify which functionality your component should execute.

MechanismReset. This function is called by engines, AgentBuilder, and other components to reset the internal state of your mechanism.

10. INTERFACING IMA 1.0 WITH QNX

Invoking agents written with TAO on the QNX operating system is performed through a TAO/DCOM bridge. The IMA bridge is implemented in an IMA component which handles access to agents hosted in QNX. In QNX, there are arm and hand control agents with the following methods:

void get_joint_angleR();
void set_joint-angleR();
void set_joint_pressureR();
void get_joint_pressureR();
void set_XYZ_angleR();
void get_XYZ_angleR();
void close_handR();
void open_handR();

These methods are for the right arm, but the same methods exist for the left arm when followed by a capital *L*. When the method is executed on the bridge component, it automatically calls QNX agents in Brutus for the left arm and Popeye for the right arm. Note that Brutus and Popeye should be booted in QNX before using the TAO/DCOM bridge. For more information, please see the *IMA 1.0 TAO/DCOM Bridge Manual*.