# Interfacing Ada 95 to Microsoft COM and Distributed COM Technologies

David Botton

Interactive Intelligence, Inc.
600 West Hillsboro Blvd., Suite 325
Deerfield Beach, Florida 33441
(954) 698-0030, ex 187

David@Botton.com

## *Abstract*

COM, the Component Object Model, and DCOM, Distributed COM, were introduced as the underlying technologies of Microsoft's second version of OLE, but have since proven their utility as powerful model for distributed systems. This paper intends to introduce COM and DCOM to the Ada community and demonstrate how to create and use objects that conform to these models.

## *Introduction*

Microsoft introduced Object Linking and Embedding 2 (OLE 2), a collection of objects providing various services for linking and embedding windows application objects. This new version of OLE was built conforming to the Component Object Model (COM). COM is a specification to provide language independent and location independent objects that expose specific interfaces. As support and tools for creating custom, non OLE COM objects became available, COM began to surface as a very flexible means of implementing distributed component based programming on Microsoft Windows platforms. Support for COM and Distributed COM (DCOM) has also been made available on a number of non-Microsoft platforms by a product called EntireX from Software AG (http://www.softwareag.com). Today, almost every new technologies released by Microsoft has been based on COM and DCOM.

Using COM objects requires access by the language or tool to the Win32 API and the ability to interface to C. Additionally, the ability to create interfaces conforming to the COM binary standard is needed. Ada 95 provides the ability to interface to C and many Ada 95 compilers that generate code for Windows NT provide the capability of creating binaries that comply with the COM standard.

This paper demonstrates the fundamentals of interfacing Ada95 to Microsoft's COM and Distributed COM technologies. Three applications are constructed, a COM client, a single apartment in-process COM object, and an application to register a COM object on a remote machine for Distributed use. Creating more robust COM objects that build on these applications will require, as a base, knowledge of Microsoft IDL and how to use the extended COM types VARIANT and BSTR from the Win32 API.

Full source code for this paper and additional information on this subject may be found on-line at the Ada Source Code Treasury (http://www.botton.com/ada).

## *COM Technical Overview*

A brief overview of the technical aspects of COM is given here, further elaboration and detail is found in the remainder of the article.

COM specifies the creation of binary objects, for language independence, that expose interfaces in the form of C++ style VTBLS, i.e. a pointer to a table of function pointers. The first three functions in the VTBL are always AddRef, Release, and QueryInterface. The AddRef and Release functions provide reference counting for the interface and COM object allowing the COM object to remove itself from memory when needed or release resources on a per interface basis. The QueryInterface function provides access to other interfaces exposed by the COM object. Additional functions for the interface are added to the end of the VTBL. When one interface is said to be a superset of another it is said to have *inherited* from the latter. The first three functions compose the IUknown interface and therefore all interfaces are said to inherit from IUknown. A globally unique identifier to facilitate versioning and interoperability identifies each interface.

The COM object is created by another object called a class factory. The class factory itself conforms to the COM specification, but exposes a well-known interface and is created and exposed by the module (DLL or executable) containing the COM objects by calls to the operating system (OS) API or through an OS specified DLL entry point. Access to the class factory object's interfaces by clients of the COM objects is through OS API calls. It is the OS's responsibility to locate and start if needed the module containing the class factory and COM objects.

Since all access to the COM object and class factory is through interfaces comprising strictly functions, the OS is free to insert proxy objects when needed to create location independence. These proxy objects are either created by the OS or provided by the COM objects. It is by way of these proxies that COM objects can exist in different processes then the client or even on different machines in a transparent manner to the client.

## *Creating a COM Client*

Creating an Ada95 application that uses a COM or DCOM object requires five steps. The full example referred to here and compiling instructions for tested compilers are available on-line at http://www.botton.com/ada/os/com-simple.html.

### 1. Identify the class identifier of the COM object and interface identifiers to be used from the COM object

One or more COM objects are housed inside of both executables and dynamic link libraries (shared libraries). Operating systems supporting COM provide a facility, transparent to the client, called the Service Control Manager (SCM) that will find the object's location, locally or on a remote machine, and provide the services necessary to make creating instances of this object possible. In order to maintain uniqueness of objects a Globally Unique Identifier (GUID) is used. For classes this is called the Class ID (CLSID) and for interfaces, the Interface ID (IID). Finding the CLSIDs and IIDs will require documentation provided by the creator of the COM object. This may be found in the form of C/C++ header files or IDL files, searching through the system registry with a tool like regedit, or using an application like the OLE/COM Object Viewer from Microsoft.

GUIDs are usually written in their canonical form {XXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}. IIDs and CLSIDs are constructed in the example by filling in the GUID record subtypes in this way:

```
-- IID for IBeep
-- {0FE0EE21-8AA2-11d2-81AA-444553540001}
IID_IBeep         : aliased IID := (16#0FE0EE22#,16#8AA2#,16#11d2#,
                       (Char'Val(16#81#),Char'Val(16#AA#),Char'Val(16#44#),
                        Char'Val(16#45#),Char'Val(16#53#),Char'Val(16#54#),
                        Char'Val(16#00#),Char'Val(16#01#)) );

-- Class ID of an object to create that has an interface to IBeep
-- {0FE0EE22-8AA2-11d2-81AA-444553540001}
CLSID_BeepClass   : aliased CLSID := (16#0FE0EE21#,16#8AA2#,16#11d2#,
                       (Char'Val(16#81#),Char'Val(16#AA#),Char'Val(16#44#),
                        Char'Val(16#45#),Char'Val(16#53#),Char'Val(16#54#),
                        Char'Val(16#00#),Char'Val(16#01#)) );
```

## 2. Construct an Ada 95 representation of the COM object's interfaces

The COM object's binary format is identical to Microsoft's implementation of C++ classes with virtual functions. This structure (the interface) is a record with the first 32 bits containing a pointer to another record that contains pointers to functions, this is known as a Virtual Table (VTBL). The Ada95 function prototypes used in creating the VTBL are created in the same fashion as Ada 95 function prototypes for C, but the first parameter is always a pointer to the interface.

The Ada 95 representation of the binary structure in the example were built to represent the following IDL code:

```
interface IBeep : IUnknown
{
   [
      helpstring("Make a beep")
   ]
   HRESULT Beep();
}
```

COM interfaces derive from an interface called IUnknown or a derivative of IUnknown. IUnknown provides reference counting for the interface and type coercion to other interfaces that may be present in the COM object. When creating Ada95 representations of the interface it is necessary to prototype all of the functions in the VTBL contained in the parent interfaces starting with the top-level parent (only single inheritance exists in interfaces).

An additional form of COM interface called a dispinterface is derived from IDispatch. This form of interface is used through the OLE automation services and is not discussed here. (See "Writing an OLE Automation Controller in Ada 95", http://www.aonix.com/Support/Ada/olepap1.html)

The example creates the binary structure using the following code:

```
-- IBeep just contains a pointer to its VTBL
type IBeep is
   record
      lpVtbl: Pointer_To_IBeepVtbl;
   end record;

-- IBeepVtbl contains pointers to all the IBeep methods
type IBeepVtbl is
   record
      QueryInterface: af_IBeep_QueryInterface;
      AddRef        : af_IBeep_AddRef;
      Release       : af_IBeep_Release;
      Beep          : af_IBeep_Beep;
```

```
   end record;
```

The prototype of the Beep method in the above IDL appears as:

```
type af_IBeep_Beep is access function (
   This: access IBeep)
   return HRESULT;
pragma Convention(Stdcall, af_IBeep_Beep);
```

Almost all function prototype will return an HRESULT type. This is an unsigned 32 bit integer that encodes a severity code, a facility code, and an information code. Important and common values for HRESULTs related to using COM are:

| | |
|---|---|
| S_OK | Successful operation |
| S_FALSE | Successful operation, but logically false |
| E_FAIL | Operation failed |
| E_NOTIMPL | Method not implemented |
| E_UNEXPECTED | Method called at an incorrect time |

## 3. Create an instance of the COM object

The COM libraries are initialized with a call to CoInitialize(System.null_address) (The null address is an artifact left from the original 16 bit version of this API function) in every thread of execution that will be creating or using COM objects. Pointers to COM Interfaces created in one thread can not be passed to another thread with out marshaling. There are alternate means of initializing the COM libraries using calls to CoInitializeEx, that make it possible to freely pass interface pointers between threads and that can potentially make operation of a multi-threaded application more efficient.

The example uses the API function CoCreateInstance to create the object and return a pointer to its IBeep interface. There is an alternate way of creating COM objects using a factory object and is the method used internally by CoCreateInstance. An example of how to do this can be found at http://www.botton.com/ada/os/com-factory.html.

The SCM will locate the object on the local or foreign machine using the COM object's registry settings. It is possible to tell the SCM to create an instance on specifically on another machine by using the CoCreateInstanceEx Win32 API function instead of CoCreateInstance.

## 4. Use the COM object

Calling functions in the interface requires passing the interface pointer itself, known as the this pointer, as the first argument to the function call, for example:

```
hr := BeepInterface.lpvtbl.Beep(BeepInterface);
if hr /= S_OK then
   raise com_error;
end if;
```

## 5. Clean up

After using an interface, the interface must be released using the IUnkown function Release. Internally during the call to CoCreateInstance, the IUnkown method calls the IUnkown function AddRef. Any copies

made of the interface pointer require a call to AddRef at that time and Release when the copy is no longer being used. The Release function is like any other member of the IBeep interface, except that it returns a reference count instead of an HRESULT:

```
refcount := BeepInterface.lpvtbl.Release(BeepInterface);
```

Before your application closes down, it should also call the Win32 API procedure CoUnitialize to release any resources being used by the application for COM and DCOM.

## *Creating a COM Object*

It is possible to create different types of COM objects that work with any type of client. The key difference between these types is in their threading model and container type. The example creates a single apartment model in-process COM object. This creates an easy to code and very flexible form of COM object, but sacrifices on efficiency during multi-threaded access. The single apartment model designates that the object was potentially not written in a thread-safe manor. Windows provides the thread safety when needed behind the scenes using a hidden GUI window. The COM object is designed for use in the same process space as the application and is therefor contained in a DLL. This doesn't mean that the object is restricted to only running in process or even on the same machine. The full example referred to here and compiling instructions are available on-line at http://www.botton.com/ada/os/make-com.html.

### 1. Create an IDL file that describes your COM object (beep.idl)

There are two parts to the IDL file in the example, the interface definitions and the type library definition. The interface created in this example constrains itself to using OLE Automation types:

```
boolean, unsigned char, double, float, int, long, short, BSTR, CURRENCY, DATE, SCODE,
enum, IDispatch*, IUknown*, SAFEARRAY of an OLE automation type, and a pointer to any
of the above types.
```

This allows specification of the oleautomation tag in the interface definition telling Windows that if it needs to marshal the interfaces it can use the OLE marshaling code. This effectively allows the object to be distributed on a network or run it out of process with out writing any marshaling code or creating proxy and stub implementations. Part of the type library definition is the CoClass. This designates an object that will implement the listed interfaces.

### 2. Compile the IDL file in to a TypeLibrary (beep.tlb)

The IDL file is compiled using Microsoft's MIDL compiler. MIDL produces a number of different C/C++ files that are not needed for the example in addition to the type library with the extension .tlb. A precompiled version of the type library is included with the example.

### 3. Create a resource file and embed the TypeLibrary in to the DLL (beep.rc)

There are two COM specific resources that are incorporated in the resource file, a self-registration key and the type library. The self-registration key is contained in a standard version resource as a value that takes the form of:

```
VALUE "OLESelfRegister", ""
```

This indicates to the RegSvr32.exe program or any other application that registers COM objects that this DLL contains code to handle COM registry settings.

The TypeLibrary is embedded as a resource using the following:

```
1 TypeLib "beep.tlb"
```

It is possible for an object to have more then one TypeLibrary resource and access to each is possible through various methods in the Win32 API. A precompiled version of the resource is included with the example.

## 4. Create the DLL entry points for COM (BeepDll.adb & BeepDll.ads)

There are two COM specific DLL entry points needed to package COM objects in a DLL, DllGetClassObject and DllCanUnloadNow. Additionally, there are two entry points needed for self-registration, DllRegisterServer and DllUnregisterServer.

The OS COM support libraries call DllGetClassObject in order to request a Class Factory object that supports the interface IClassFactory for a specified CLSID. In order to support licensing a request can also be made to DllGetClassObject for a Class Factory object that supports an alternate interface, IClassFactory2 that uses license keys for class creation. (see http://www.botton.com/ada/os/com-lic.html for more information on using COM objects with licenses).

DllCanUnloadNow is called periodically by the OS COM support in order to identify if there is a need to maintain the DLL in memory. DllCanUnloadNow checks to see if any components created from the DLL are still in use, or if the DLL has been locked in memory by calls to the IClassFactory member function LockServer.

In the example, this is done through the use of two global counters:

```
if (g_cServer=0) and (g_cComponents=0) then
   return S_OK;
else
   return S_FALSE;
end if;
```

Returning S_OK indicates to the OS that it is safe to unload the DLL from memory.
Regsvr32 or any other application that handles DLL registration calls DllRegisterServer. It registers the type library by loading it with the API function LoadTypeLib and then registers it with RegisterTypeLib. Following that it adds the COM registry entries making it available for the SCM and other COM and OLE facilities. COM keys are registered under HKEY_CLASSES_ROOT. There are three groups of registry keys that are used.

The first group registers the COM object on the system and is the only required set of entries to make available the COM object. These keys are registered under HKEY_CLASSES_ROOT\CLSID. Each CLSID receives one new key that takes the form of its CLSID in canonical form, a GUID with braces. In this key two string values are added, a default value naming the CLSID and an AppID. The AppID is a GUID in canonical form that represents an entry under HKEY_CLASSES_ROOT\AppID, which will be described shortly. The AppID can be the same as the CLSID and often is, but when registering many COM objects that will share the same AppID properties an additional GUID can be created for this purpose.

Under the CLSID key is three more keys, a key representing the server type (in this case InProcServer32, but alternatively LocalServer32 or InProcHandler32). This key contains a default value of the full server path and a value called ThreadingModel that describes the object's default threading model. The second

key is ProgID and its default value is the ProgID of the COM object which will be described shortly. The last key is the VersionIndependantProgID and its default value is the COM object's version independent ProgID.

The second group of registry entries describes the COM object in a user-friendly fashion allowing lookups of the COM object's CLSID. The lookups are performed using the Win32 API function CLSIDFromProgID or when using applications like Visual Basic. By convention, these keys are formed using the library name and CoClass name used in the IDL file, in the example BeepLibrary and BeepClass. The ProgID would be BeepLibrary.BeepClass.1. The one represents the version of the COM object. The VersionIndependantProgID is the same with out the one. These keys are placed directly under HKEY_CLASSES_ROOT. Their default value is the COM object's name, and they contain a sub key called CLSID with the default value of the COM object's CLSID. The VersionIndependantProgID key has an additional sub key called CurVer with the default value of ProgID.

The last group of registry keys is used to configure the COM object for DCOM. The key is placed under HKEY_CLASSES_ROOT\AppID with the default value of the Object's it will handle or other user recognizable form. A utility called dcomcnfg that configures the security for DCOM objects uses this default value for user interaction. Since the example is an InProcServer, we also need to define a DLL surrogate that will handle loading the DLL for DCOM use. The value DllSurrogate is used to indicate the application to server this purpose, or blank to use the OS dllhost application.

## 5. Implement the Class Factory Object (factory.ads & factory.adb)

The object returned by DllGetClassObject is not the COM object to be used by the client, but an object used to create instances of the COM object. This object is in itself a COM object derived from IClassFactory that is derived from IUnknown.

Creating the binary structure for creating COM objects is near identical to the structure used when using COM objects. The function prototypes and the VTBLs are constructed in the same fashion. The difference lies in the creation of the final record type, where a pointer to an instance of the VTBL is needed.

In the example, the VTBL is constructed for the IClassFactory interface and an instance is placed in the Class_Factory object:

```
type Class_Factory_Vtbl is
   record
      QueryInterface: af_QueryInterface := Factory.QueryInterface'Access;
      AddRef        : af_AddRef := Factory.AddRef'Access;
      Release       : af_Release := Factory.Release'Access;
      CreateInstance: af_CreateInstance := Factory.CreateInstance'Access;
      LockServer    : af_LockServer := Factory.LockServer'Access;
   end record;
pragma Convention(C, Class_Factory_Vtbl);

type Class_Factory_Vtbl_Pointer is access all Class_Factory_Vtbl;

Vtbl : aliased Class_Factory_Vtbl;

type Class_Factory is
   record
      lpVtbl : Class_Factory_Vtbl_Pointer := Vtbl'Access;
      m_cRef : Win32.Long := 1;
   end record;
pragma Convention(C, Class_Factory);
```

The m_cRef member of the Class_Factory contains a reference count for the object. Since the object will be created and returned to the OS in DllGetClassObject , it is created with one reference, relieving DllGetClassObject from first calling AddRef from one of the Class_Factory supported interfaces.

Since the class factory implements IClassFactory, it also needs to implement its parent IUnknown. IUnknown is implemented in the class factory in the same manor as other COM objects. AddRef adds a reference count and Release removes a reference count. If the reference count reaches zero, the object frees itself from memory. The QueryInterface method is passed an IID that represents the interface requested. This is implemented a as a simple set of if then statements:

```
if riid.all = Beep_Global.IID_IUnknown then
   ppvObject.all := This.all'Address;
elsif riid.all = Beep_Global.IID_IClassFactory then
   ppvObject.all := This.all'Address;
else
   ppvObject.all := System.Null_Address;
   return E_NOINTERFACE;
end if;

This.m_cRef := This.m_cRef+1;

return S_OK;
```

Since a reference is being passed out of the function, the implementation also adds a reference count to the object. If an interface is requested that is unsupported an error E_NOINTERFACE is returned to the requesting client. If the object requests a pointer IUnknown, it is returned the VTBL for any interface derived from IUnknown since the first three entries in its VTBL will be the same as IUnkown.

IClassFactory contains two methods, CreateInstance and LockServer. LockServer adds or subtracts a count to a global variable that when not equal to zero indicates to DllCanUnloadNow that the client has requested the Dll to remain in memory. CreateInstance creates an instance of the requested COM object and adds a count to the global component count indicating to DllCanUnloadNow that the Dll there are live components and the Dll should not be removed from memory. When calling the CreateInstance method a IID indicating the requested interface for the COM object to return is also passed. The example uses the newly created COM objects QueryInterface method to return the appropriate interface matching that IID.

### 6. Implement the COM object (BeepCom.ads & BeepCom.adb)

The implementation for the actual COM object is the same pattern as that followed by the class factory. In this case the COM object will support IBeep which is derived from IUknown. The only difference in the implementation is in the IUknown interface method Release. In addition to subtracting one from the objects reference count, when the reference count reaches zero, the global component variable indicating the number of live components is also reduced by one.

## *Creating an application to register a COM Object on a remote machine*

The location of a COM object is transparent to a COM client and can be located on remote machines. It is possible to request remote connections, but alternately an OS supporting COM can be configured to look for the object at a remote location. The complete source code for this example is located at http://www.botton.com/ada/os/com-remote.html.

It is possible to configure the remote machine by hand, but this example automates the process by creating all the appropriate registry settings with the exception of permissions. The settings are identical to that used by the server less the registry keys for apartment model and server type. An additional value, RemoteServerName, is added to the AppID key with the location of the host machine in place of the DllSurrogate value.

In addition to running this example, additional configuration for DCOM needs to be performed on the host machine using dcomcnfg. In dcomcnfg the Beep Class application is selected then the properties

button is clicked. Under the security tab, permissions need to be added for users that will be using the COM object for both the access and launch settings. Finally, under the identity tab, the interactive user option is selected. Normally it would not be necessary to modify the identity, but since the example COM object pops up a dialog box on the host machine, it needs to run as the interactive user.

Once the example has been executed and the permissions set, the COM client will access the COM object on the host machine with out changing a single line of code.

## *Conclusion*

Ada 95 can directly utilizes the powerful capabilities of COM and DCOM. The major complexities of COM programming are in the initial construction. These examples open the technology and terms to the Ada 95 community to appreciate and incorporate in their projects. Further research and development in to the creation of tools to generate bindings and skeleton code for COM objects is underway, information is available at http://www.botton.com/ada/com.