Encapsulate Instrument Driver DLLs WITH COM

Here's how to use an out-of-process COM server to encapsulate instrument driver DLLs to avoid conflicts when designing test systems.

Terry Leeper

Sometimes in a test system it is desirable for more than one computer program or process to make use of an instrument at the same time. This can present difficulties.

For example, the operating system in this article (I assume a Windows OS) will be time-slicing between the processes. It is therefore possible that one process could be in the "middle" of something and be interrupted by another. This problem is relatively easy to solve by locking the instrument using an OS synchronization object, such as a critical sections or mutexes.

Another problem arises because many instrument drivers are DLLs (dynamic link libraries). It is very easy when writing instrument drivers to get the state partially held in DLL memory. Since separate processes do not share memory, it is possible that one process will have a different state from another. This difficulty is not so easy to solve because test system designers do not control the instrument driver's contents.

Here's how you can deal with this problem using an out-of-process COM (component object model) server.

THE PROBLEM DESCRIBED

If instrument information is kept in DLL memory, it is possible for each of the two processes to have different states for the instrument (*see Figure 1, right*). This can lead to unexpected test system failure. "Wrapping" the driver DLL in an out-of-process or local COM yields what is shown in *Figure 2 (right)*. In this case, only one instance of the DLL exists, so all processes

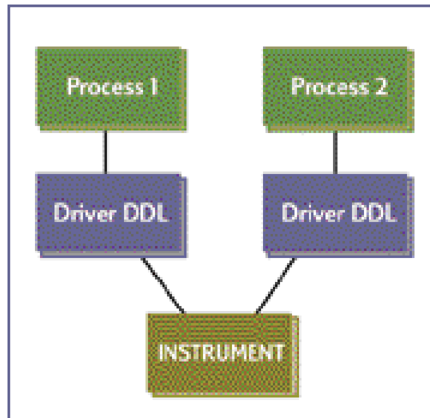

Figure 1: Two processes using the same instrument with a DLL driver.

that use the instrument access it through the same instance.

Since all accesses to the server cross process boundaries, it is not surprising that the primary issue for a local COM server is performance. While this article will not explore this issue broadly, here is one example to give you a flavor. On

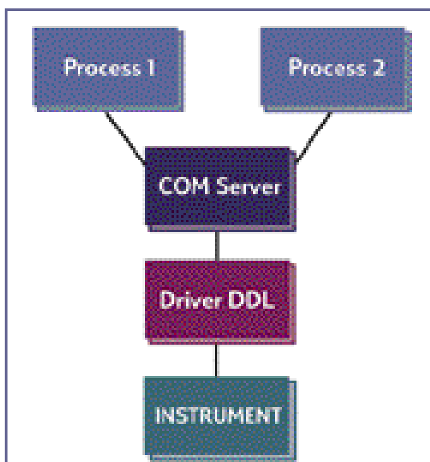

Figure 2: Two processes using the same instrument through a local COM server.

NT 4 using a 200MHz Pentium Pro processor it took about 250 microseconds to pass 20 bytes of data to the server. If this performance degradation is acceptable for your application, then the COM server is an excellent way to proceed.

Another difficulty facing system designers is that a driver may have many functions. Consequently, wrapping the entire driver can be a formidable task. However, a specific application may use only a small percentage of the driver's API. Thus, only these APIs need to be mapped into the wrapper.

COM IN A NUTSHELL

COM is often presented in a series of complex pieces. This can be simplified for the case of an instrument server, where your goal is to cut through the complexity to give a how-to approach.

COM is based on the notion of an object. The object makes itself known to the world through interfaces. Thus, an object exposes one or more interface.

Interfaces in turn are made up of methods. Methods are functions. So, the way C++ uses a COM object is that a pointer is obtained to one of its interfaces. It is through this pointer that methods (i.e., functions) are called.

In NT a COM object is registered in the registry. Each object is given a unique identifier called a Universally Unique Identifier (UUID) or Globally Unique Identifier (GUID). This is how the COM system finds an object. Fortunately, the details of this are performed by the development environment.

A COM interface may contain any number of methods. There are certain low-level methods required by COM that must occur in any interface. You need

Listing 1: A typical method entry in IDL.

```
[id(1), helpstring("method SetPortDirection")] HRESULT SetPortDirection([in]
short portnum,[in] short direction);
```

Listing 2: A short array of type input and another of output.

```
[id(1),helpstring("method method")] HRESULT method([in] short size1,[in,size_is(size1)]
short * array1, [in,out] short * size2,[out,size_is(*size2)]short *array2);
```

Listing 3: API's exported by the card's drivers.

```
SetPortDirection(short portnum, short
    direction)
portnum = 0 or 1
direction 0=Input, 1=Output
GetPort(short portnum, short * value)
portnum (0 or 1)
value: value of port

SetPort(short portnum, short value)
portnum (0 or 1)
value: value to set port to.
```

not be concerned with these required methods because the development environment automatically handles them. Thus, from the server designer's point of view, all that you need to do is create the necessary methods to expose the desired driver functions.

INTERFACE DESCRIPTION LANGUAGE

Before delving into the server details, there are some COM server issues that need to be covered.

The server is a separate process. A program that uses a server is called a client or container. Since separate processes do not share memory space, you need some methodology to take function parameters across process boundaries. In COM this methodology is called marshaling. Marshaling is really nothing more than copying parameters from one process into or from another process. This is accomplished by a DLL called a proxy, which is automatically generated by the development environment.

COM learns about an object through its type library. The type library is created from the Interface Description Language or IDL, a description of a COM object that the C++ compiler and other applications, such as Visual Basic, use. The proxy is built using the information in the IDL file. IDL is a text language. It's compiled into a type library using the MIDL compiler.

Your development environment handles most IDL details; however, server designers need to know enough IDL to enter the parameters to server methods. *Listing 1* at the top of this page shows a typical method entry in IDL.

A series of comma-separated items inside brackets precede a method's parameters. A variety of attributes tell COM about the parameters. IN, for example, instructs COM to copy the data in from the client process into the server process. OUT is the reverse process; i.e., OUT copies the data out from the server process into the client process. IN,OUT has data going both directions.

The Retval parameter will look like the return value of the function. When it is used in C++ or Visual Basic, Retval will appear as if the parameter does not even exist in the function, but is the return value of the function. Only [OUT] parameters can be tagged as Retval.

An input or output array parameter is declared as a pointer (such as short). You communicate the array size to COM by having size parameters in the method and then using a special attribute, size_is, to tell COM which parameter holds the size.

Suppose you have a short array of type input and another of output, such as in *Listing 2* above. Here, size1 tells COM how big array1 is, and size2 tells COM how big array2 is. COM can use these to copy the arrays from one process to the other. Note that size_is requires a parameter that is either [in] or [in,out] and that for the output array, array2, the client should allocate the array and pass its size in *size2.

It is important to remember that *size2 must be set to the number of items put into array2 before returning from the method.

COM supports the standard types needed by an instrument server. In particular, it supports shorts, longs, doubles, and pointers to these types. However, strings, such as char *, are more complex. Visual Basic has its own string type called a BSTR, and COM pretty much adopted it.

BSTRs are formatted as two-byte characters (i.e., wide character), with the length encoded inside the string at its beginning. A pointer to a BSTR will point just past the length. I would recommend that you handle all string data as BSTRs in COM.

Many C++ classes have been developed to deal with BSTRs. In my opinion, none of these are any easier than the standard Windows API . (***For a list of relevant APIs, see "Windows APIs," at the end of this article.***) One advantage of BSTR arguments is that size parameters are not needed.

THE BASIC SERVER

You build your server with Microsoft Visual C++ using the ActiveX Template Library (ATL). This article assumes version 6.0 of the C++ compiler and 3.0 of ATL.

To create a server project from the Visual C++ development environment, select New from the File menu and then select the Projects tab. Choose a project type of ATL COM AppWizard. In the right pane, select a path where you want the project to go then put in a name for your project (atlserver is used in this example). Press OK. Select the server type as executable, then press Finish (and OK from the info dialog) and the project will be created.

To continue with the project, you need an I/O card or instrument. For example purposes, we'll make up a simple digital I/O card with two 8-bit ports. Suppose the API in *Listing 3* (*above*) were exported by the card's driver. These APIs are what you need to expose from the COM server as methods.

In the Visual C++ development environment, select the tab marked ClassView from the left pane. Right-click upon the atlserver line, and you should see a menu that includes New ATL Object. Selecting New ATL Object should bring up a dialog box with Simple Object highlighted in its right pane. Now, press the Next button. This brings up a tabbed dialog with two tabs, Names and

Listing 4: An example of error handling.

```
STDMETHODIMP CSimpDIGIO::SetPortDirection(short portnum, short direction)
{
        if( !SetPortDirection(portnum,direction) )
{
                        Error("The driver entry point SetPortDirection failed");
                        return E_FAIL;
}

        return S_OK;
}
```

Listing 5: An example of using smart pointers.

```
#include "stdafx.h"

#import "..\atlserver\atlserver.tlb"//the path should be to YOUR server's subdirectory
using namespace ATLSERVERLib;

int main(int argc, char* argv[])
{

CoInitialize(NULL);

        {
                try()
                {
                        //here is where your program would use the server
                        ISimpDIGIOPtr pI(__uuidof(SimpDIGIO));

                        pI->SetPortDirection(0,1); //use the pointer
                }
                catch( _com_error cerr)
                {
                        wprintf(L"Got error %d\n",cerr.Description() );
                }

        }
CoUninitialize();


}
```

Attributes. In the Names tab at the Short Name edit box fill in the name for your digital I/O card object. SimpDIGIO is the name I'll use.

Now select the Attributes tab. The Threading Model should be Apartment, the interface dual, and No should be selected for aggregation. These should be the default values.

Select Support IsupportErrorInfo for richer error propagation. Press OK to make the object.

At this point you have a complete server. It has an object named SimpDIGIO with an interface named ISimpDIGIO. The next step is to add the methods for the specific digital I/O functions.

To do this, return to the ClassView pane and click upon the "+" box next to the atlserver name. This will open the server object. Right-click over the "spoon" beside the ISimpDIGIO interface and select AddMethod from the menu it activates. This brings up a dialog box.

Enter the first method SetPortDirection for the methods name. Now for IDL. In the parameters box type:

```
[in] short portnum,[in] short direction
```

This sets up the method SetPortDirection to have two input parameters. Entered SetPort in similarly.

The GetPort takes an out parameter, so enter its parameters:

```
[in] short portnum,[out,retval] short * value
```

Note the use of Retval. When you look at this object through a browser such as in Visual Basic, this method will appear to have only the portnum parameter. Value will be the return value of the function.

Open the file SimpDIGIO.cpp. At the bottom should be the three methods just created. Adding code to them will finish the server. For example, add SetPortDirection in the hypothetical card's driver:

```
STDMETHODIMP CSimpDIGIO::SetPort
Direction(short portnum, short direction)
  {
  SetPortDirection(portnum,direction); //our
hypothetical driver entry point

        return S_OK;
}
```

That's all there is to it.

BUT WHAT ABOUT ERRORS?

Error handling is illustrated in ***Listing 4 (above, left)***. The ATL system provides the Error routine. E_FAIL is a general failure code for a method.

Do the same for the other two entry points and your server is done. To experiment further, comment out the calls to your hypothetical I/O card. Now the server should compile and link.

As previously mentioned, the proxy is a DLL program created to allow COM to do its magic. The proxy contains all of your server's entry points. A call into the proxy entry point will be shunted to COM, which will marshal the parameters and call the server through a remote procedure call mechanism.

MIDL compiler creates the proxy when you build the project, but it needs to be installed. To do this, go to your Projects subdirectory. Run the following command from a DOS prompt: nmake–atlserverps.mk. This will both make and register the server in the NT registry. The proxy can be registered directly (if it has already been made) by the command: regsvr32 atlserverps.dll. Run the make command every time you add a method or change parameters.

The server is ready to use. From Visual Basic go to the project->references menu. Select the atlserver type library from the choices.

Here's a fragment of Visual Basic code that would use the server:

```
Dim x as new ISimpDIGIO
s.SetPortDirection 1
```

To see how Visual Basic will handle

server errors go into the SetPortDirection in your server C++ project and make SetPortDirection look like:

```
STDMETHODIMP CSimpDIGIO::SetPortDirection(short portnum, short direction)
{
        Error("The driver entry point SetPortDirection failed");
        return E_FAIL;
}
```

Rebuild the server and run it in Visual Basic to see how errors are handled.

To use the object in Visual C++ create a new "hello world" console app using the Win32 Application project type from the Visual C++ project tab.

To use COM objects in C++ you want to use smart pointers; i.e., pointers that know about your object (*see Listing 5, previous page*).

The Import command is given the path to the type library, which will be in the directory where you built the server. This is how C++ gets at the information about the server object. "Using namespace" results from the fact that #import will create a namespace that is the name of the type library (e.g., look in the server's .idl file for the library statement).

CoInitialize is part of the COM system and must be called before your program can reference anything in the COM system. The smart pointer is called pI. The purpose of the redundant looking block in this program is to set the scope for pI. It goes out of scope—causing its destructor to execute—before CoUninitialize is called (nothing in COM can be accessed after that call).

Take note of the name given to the pointer type by the #import directive: ISimpDIGIOPtr. The #import simply takes the name of the interfaces it finds in the type library and appends Ptr. The __uuidof(SimpDIGIO) tells COM which object to use. For example, SimpDIGIO is the name given to the object and ISimpDIGIO is the interface it exposes. One or more object can expose an interface, which is why the object's name needs to be specified here.

Notice that Visual C++ will know about your object when you use pI.

The try recover block shows you how to get at errors generated by the server.

CONCLUSION

In short, that's what you need to know to create a COM local server to surround an instrument driver DLL. This allows for a single access to an instrument with a driver that is a DLL. If any instrument state is kept in the DLL, the COM server will keep the state correct for all processes that access the instrument.

Terry Leeper *holds B.S. and M.A. degrees inmathematics and has worked for Hewlett-Packard for 18 years. You can contact him through e-mail c/o de-editors@helmers.com.*

Windows APIs

Many C++ classes have been developed to deal with BSTRs. In my opinion none of these are any easier than the Standard Windows API. Listed below are the relevant API. *—TL*

BSTR SysAllocString(wchar * string)—Create a BSTR from a null terminated wide character string. Use: BSTR s = SysAllocString(L"hello"); //
Note: The L modifier makes "hello" a wide character string.

Note—This functions attaches a null byte to the string

BSTR SysAllocStringByteLen(LPCSTR str, unsigned int len)—str is a pointer to a null terminated string

(char *) or NULL—Leaves the string uninitialized. Allocates a BSTR of length bytes.
Use: BSTR s= SysAllocStringByteLen(NULL,100); //
which creates a BSTR 100 bytes long (but only 50 wide chars long).

void SysFreeString(BSTR s)—Free a string allocated with one of the above.

size_t wcstombs(char *mbstr, const wchar_t *wcstr, size_t count)—A C runtime library routine to convert null terminated wide (2-byte) char strings to char * strings. Note that this API handles NULL bytes similar to strncpy.

size_t mbstowcs(wchar_t *wcstr, const char *mbstr, size_t count);—The reverse of the above.

For example to convert a char buffer of data , which may contain null bytes, 1000 items long into a BSTR do the following:

```
BSTR S;

S = SysAllocStringByteLen(NULL,2000);//byte len = 2* # of wide chars
for(int I=0; I<1000; I++) S[I]=data[I];
```

To convert a C style string to a BSTR do:

```
BSTR S;
WCHAR * buff = new WCHAR[strlen(data)];

mbstowcs(buff,data,strlen(data));
S=SysAllocString( buff );

delete [] buff;
```

Copyright, *Desktop Engineering* 1999. This article appeared in the Oct. '99 issue of *Desktop Engineering* magazine, a Helmers Publication. It may not be copied or stored in any form without the express permission of the publisher.