Integrating C++ and MFC Code with Graphical Programming

Monday Aug 24, 11:30-12:30

Thursday Aug 27, 10:15-11:15 and 11:30-12:30

Red River (4B)

Audrey F. Harvey

Senior Architect

NI WEEK '98

Overview

- G is a powerful programming language
- Can extend G with C++ and MFC
  - User interface
  - Operating system services
  - Managing complex data sets
- Easy to do
  - If you know C++ and MFC

NI WEEK '98

G is a powerful programming language. Sometimes you need functionality that is not available in the G language. You may need special user interface pieces or direct access to operating system services that are not built into the G language. C++ is also useful for managing complex collections of data. Fortunately it is very straightforward to integrate Microsoft Foundation Classes (MFC) and C++- based dynamic link libraries (DLLs) into your G application thus extending your programming possibilities. There is almost no limitation to what you can do.

This presentation assumes that you are already familiar with the Visual C++ development environment and basic MFC programming principles. This presentation focuses on using those skills with G applications.

MFC – Microsoft Foundation Classes

Visual C++, a C++ development environment provided by Microsoft.

Presentation Outline

- Creating an MFC-based DLL Application
- Interfacing C++ to DLL entry points
- Integrating MFC-based dialogs
- Managing C++ objects from G
- Handling Windows callback messages
- Event-driven notification (occurrences)
- Handling multiple threads in the DLL

NI WEEK '98

In this presentation, we cover several aspects of integrating MFC-based C++ DLLs into your G application. We will cover user interface possibilities as well as using objects from your G application and various options for interacting with G occurrences from your DLL.

– Creating an MFC-based DLL Application.

– Interfacing C++ to DLL entry points.

– Integrating MFC-based dialogs into a G application.

– Managing C++ objects from a G application.

– Handling system callbacks from from the Windows Message system.

– Event-driven notification to G (occurrences) from your DLL.

– Handling multiple threads in your DLL.

Creating an MFC-based DLL App

- Use the Visual C++ AppWizard to create the DLL template
- MFC-based DLLs inherit from CWinApp
- Add code to:
  - InitInstance - initialization
  - ExitInstance - clean up

NI WEEK '98

In your Visual C++ development environment you can use the AppWizard to create an MFC-based DLL by selecting the MFC AppWizard(dll) project when you create a new workspace. You can then choose to statically or dynamically link to the MFC DLLs. When you create your project, you also need to specify whether or not to include Automation or Windows Sockets features in your DLL.

The AppWizard automatically creates a DLL project with a single class declared as your application class. This class inherits from the CWinApp class. You can also override the CWinApp virtual functions "InitInstance" and "ExitInstance" for DLL-specific initialization and termination code. These functions are called as threads attach and detach from your DLL.

Demo - Using AppWizard

NIWEEK'98

Sample “DLL Application” Class Code:

```
class CDummyDLLApp : public CWinApp
{
public:
        CDummyDLLApp();
        int MyFunction(int iVal);
// Overrides
        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CDummyDLLApp)
        public:
        virtual BOOL InitInstance();
        virtual int ExitInstance();
        //}}AFX_VIRTUAL
        //{{AFX_MSG(CDummyDLLApp)
// NOTE - the ClassWizard will add and remove member functions here.
// DO NOT EDIT what you see in these blocks of generated code !
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};
/////////////////////////////////////////////////////////////////////////////
// The one and only CDummyDLLApp object
CDummyDLLApp theApp;
```

Interfacing C++ to DLL Entry Points

- Add member functions to the CWinApp derived class for each DLL function
- Create and export C functions that call these class methods

NI WEEK '98

This CWinApp derived class is useful for adding member functions that represent the entry points to the DLL. You can thus use this class as the interface or wrapper for all the other objects in your DLL. This is a design pattern known as a "facade".

To add DLL entry points, you just need to add global functions that are declared as exported:

```
extern "C" __declspec(dllexport) int MyExportedFunction(int iVal )

{
        AFX_MANAGE_STATE(AfxGetStaticModuleState());
        return theApp.MyFunction(iVal);
}
```

It is also necessary to call the AFX_MANAGE_STATE function when entering each DLL entry point.

This DLL entry point is a C function, but it then invokes a method on a global C++ object. You cannot pass classes or references to classes over this interface.

Calling the DLL Entry Point

- Pasting the DLL entry point declaration in your calling Virtual Instrument (VI) helps

NI WEEK'98

Once you build your DLL, you can call the DLL entry point from a VI using the Call Library Function node. I find it a useful convention to paste the DLL entry point declaration in my VI diagram below the Call Library Function. Seeing the function declaration helps in configuring the DLL node to have all the right parameters in the right order. It also helps any readers of the G code! This example also shows using the error I/O clusters within the VI calling the DLL node even though in this particular case there is no error returned from the DLL. Using error I/O cluster in your DLL-calling VIs is usually a good idea. It makes it easier to manage errors in your higher level G application.

Integrating MFC Dialogs with G

- Easy to do
- Useful for special user interface of system functions
- ActiveX controls in your G-based GUI may be a better alternative

NIWEEK'98

It can be very useful to incorporate MFC-based dialogs in your G application, and it is easy to make them appear no different from your G-based GUI.

Some applications are – using interface elements that are not available in G such as the tree control; or interacting directly with some system functions such as browsing the network.

Note that ActiveX controls are a very nice option for advanced user interface elements, so it may not be worth building a DLL just for a couple of special dialogs.

Creating the Dialog

- Insert new dialog into project resources
- Create a CDialog derived class

NI WEEK '98

MFC-based dialogs inherit from the class CDialog.

To create a dialog, you simply insert a new dialog into your project resources and then pop-up on the dialog and invoke the Class Wizard to create a class for the dialog. This class will inherit from CDialog. As you add controls and indicators you use the Class Wizard to create the various member functions that handle the GUI interaction. To learn how to build and write code for dialogs, you really need to study one of the many Visual C++ and MFC books available.

Invoking the Dialog

- Add the CDialog derived class as member variable to the CWinApp class
- Call DoModal() member function to invoke dialog

NI WEEK'98

Once you have the dialog functionality in place, create a member variable in your CWinApp derived class for each dialog. Then you simply need to create a member function for your CWinApp derived class that invokes the dialog by calling the DoModal() member function of the CDialog derived class.

```
extern "C" __declspec(dllexport) HRESULT BV_InvokeDialog(char*
                pszMessage,BOOL* bResult)
{
                AFX_MANAGE_STATE(AfxGetStaticModuleState());
                return theApp.InvokeDialog(pszMessage, bResult);
}
```

Calling the "DoModal()" function on the CDialog derived class invokes the dialog and returns the result (whether OK or Cancel was pressed).

```
HRESULT CDialogTestApp::InvokeDialog(char* pszMessage, BOOL* bResult)
{
                // TODO: Add your command handler code here
                m_myDialog.m_message = pszMessage;
                int nRetVal = m_myDialog.DoModal();
                *bResult = (nRetVal == IDOK);
                return NOERROR;
}
```

Dialog Cosmetics

- To make the MFC dialog “look” like a G Window:
  - Add the LabVIEW or BridgeVIEW icon to the dialog
  - Set the 3D-look and Client Edge dialog properties

NI WEEK 98

You can make an MFC-based dialog appear cosmetically just like a G Window. You need to add the BridgeVIEW or LabVIEW icon to the dialog.

Adding the dialog to the icon is done during dialog initialization. You need to add the icon to your project resources.

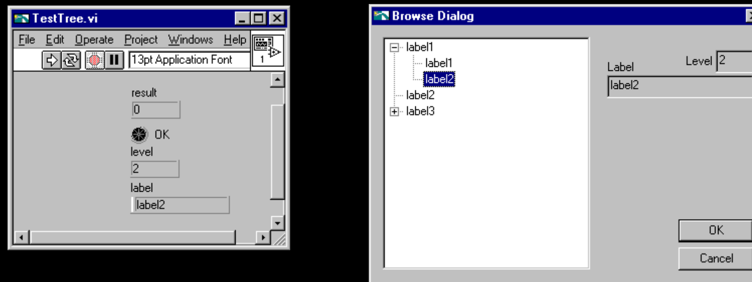
```
BOOL TestDialog::OnInitDialog()
{
        CDialog::OnInitDialog();
        //Set Icon to Match BridgeVIEW
        HICON hIcon = theApp.LoadIcon(IDI_BVICON);
        SetIcon(hIcon, FALSE );

        return TRUE;  // return TRUE unless you set the focus to a control
                // EXCEPTION: OCX Property Pages should return FALSE
}
```

Also, in the properties page for the dialog, set the 3D-look (More Styles tab) and Client edge (Extended Styles tab) properties. The dialog will now blend in with the rest of your G-based GUI.

Demo - Dialog Examples

- Simple dialog
- Dialog with tree control
- Passing data to and from the dialog



NIWEEK'98

Demo – The DialogTest project shows a both a basic dialog and a dialog with a tree control. This example also illustrates how data is passed to the dialogs from the G application and returned to the G application.

Managing Objects from G

- May need to manipulate many object instances from the G application
- Use a unique refnum to identify an object instance in G
- CRef<> template creates objects that generate unique refnums automatically

NI WEEK '98

In any complex C++ DLL, you inevitably end up managing diverse collections of objects. Care must be taken when manipulating this objects from your G application. In particular, you do not want to pass the object pointer directly to the G application, because if the object no longer exists, a crash will likely result. This, of course, is also true inside your C++ application. Instead, you should use a mechanism, such as a "safe pointer", that can be verified whenever it is passed into the DLL from the G application.

Many of the objects I access from my G applications are derived from a base class template called CRef. This template automatically assigns a reference number to each object during the object construction, and stores the object reference and pointer in a global CMap. When the object is destroyed, it removes itself from the Map. The object's refnum is always used in the G application. In this way, you can detect stale or invalid refnums before an object is used. Using a template provides a type-safe way of managing different objects - each class derives from a typedef that is created from the CRef template.

The CRef Template

NIWEEK'98

See the CRef.h file in the GObjects Project for the source code to the CRef template.

```
const DWORD NULL_REFNUM = 0;

template <class T> class CRef
{
        static DWORD refSource;
        DWORD refnum;
protected:
        static CMap<DWORD, DWORD, T*, T*> RefMap;
public:
        CRef(DWORD dwRefSource = refSource++);
        ~CRef();
        DWORD GetRefnum() { return refnum; };
        static T* Lookup(DWORD dwRefnum);
};
```

Object Type Management

- Create a unique G type for each object class
  - Use the Data Log File Refnum with an Enum
  - Make it a Typedef
  - Prevents miswiring of different object types

NI WEEK '98

Also, in the G application, I create a specific "type" for each object class. This just helps in terms of type enforcement at the G diagram level, so that different object class refnums cannot be wired to each other.

A tip from LabVIEW developer Stepan Riha to do this easily: drop down a data log file refnum; create an enum control and give the first (and only) item the name of your object; then place the enum inside the data log file refnum.

By using a different named enum for each object type, you can have a typed refnum for each object, and you will not accidentally wire different object refnums together (you will get broken wires if you do).

Make this control a strict typedef that is then used by all VIs that interact with that particular object class. Then if you need to change it, all the VIs are automatically updated.

You also have to pass the object reference into your Call Library Node by type. The refnum is a 32-bit integer.

Demo - GObjects Project

NIWEEK'98

Demo – The GObjects Project illustrates the basic principles of object manipulation from G Applications.

It illustrates how the CRef template class handles unique refnums and objects are created, used, and deleted. It also illustrates how to use the Data Log File Refnum for strict typing of objects in the G application.

Handling Windows Messages

- Some MFC class methods rely on Windows Message system
- LabVIEW and BridgeVIEW relinquish control to Windows regularly
- Easy to use callbacks in a DLL

NIWEEK'98

Working with MFC classes that rely on Windows Messages.

Several MFC classes use the Windows Message Queue to "callback" some of the class methods on events. As part of normal operation, LabVIEW and BridgeVIEW regularly relinquish control to the Windows Message Queue. Thus, if your DLL uses objects that count on callbacks from the windows message queue, you receive those callbacks in your DLL without doing anything special. You do not have to create separate threads in order to be called back. You can develop a simple, single-threaded DLL, but still operate in an asynchronous manner.

Example – Sockets

- Example – the CAsyncSocket class
- Using this class directly in a DLL makes it possible to wait on multiple TCP/IP clients without polling

NI WEEK'98

This technique is illustrated by using the MFC-based abstraction of Windows Sockets instead of using the G TCP/IP primitives to poll several clients for messages. You can configure the CAsyncSocket class to have its methods called whenever TCP/IP messages are received, when connections are made and closed. With this set of functions, a single TCP/IP Server can monitor and wait on messages from multiple clients without having to poll each client in turn.

The BVSockets Project example derives the classes CListenSocket from CSocket and CClientSocket from CAsyncSocket. The CListenSocket::OnAccept method is called whenever a new client tries to connect. This causes a new CClientSocket to be created. The CClientSocket::OnReceive method for a on the CClientSocket object for specific client connection instance is called whenever a new TCP/IP message is received from a client connection. Similarly the OnClose method is called whenever the client disconnects.

Each asynchronous callback signals the G application through an occurrence whenever something of interest occurs.

```
class CListenSocket : public CSocket

void CListenSocket::OnAccept(int nErrorCode)
{
        CSocket::OnAccept(nErrorCode);
        theApp.ProcessPendingAccept();
}

class CClientSocket : public CAsyncSocket

void CClientSocket::OnReceive(int nErrorCode)
{
        CAsyncSocket::OnReceive(nErrorCode);
        theApp.ProcessPendingRead(this);
}

void CClientSocket::OnClose(int nErrorCode)
{
        // TODO: Add your specialized code here and/or call the base class
        CAsyncSocket::OnClose(nErrorCode);
        theApp.ProcessPendingClose(this);
}
```

Event-Driven Notification

- Event-driven notification (occurrences)
- Use the Post Message function to set occurrences
- Works across multiple processes

NIWEEK'98

Event-based signaling to the G application.

When your DLL receives a callback, it needs to signal your G application. There are several ways to synchronize your G application with events in your DLL. A fairly straightforward way to do this is with occurrences.

Two methods are available. One method is to use the Windows PostMessage function. In order to use PostMessage you must obtain the G environment Window handle and the code for the occurrence message. We use a simple CIN that returns that information. The specific occurrence refnum for the event is also one of the arguments in the PostMessage function. This occurrence is created on the G diagram and then passed into the DLL for later use.

```
BOOL bDone = PostMessage(m_hGWnd, m_hGWndMsg, 0, m_occRefnum);
```

Using PostMessage is simple and works as well from another application as it does from a DLL. However, it goes through the Windows Message Queue which adds latency to the event notification.

Using Occur Directly

- Use the LabVIEW/BridgeVIEW Occur function to set occurrences directly
- Only works from a DLL loaded into the LabVIEW/BridgeVIEW process

NI WEEK '98

Another option is to call the LabVIEW/BridgeVIEW Occur function directly, to set the specified occurrence. The Occur function is thread safe. It also bypasses the Windows Message Queue and therefore has less latency.

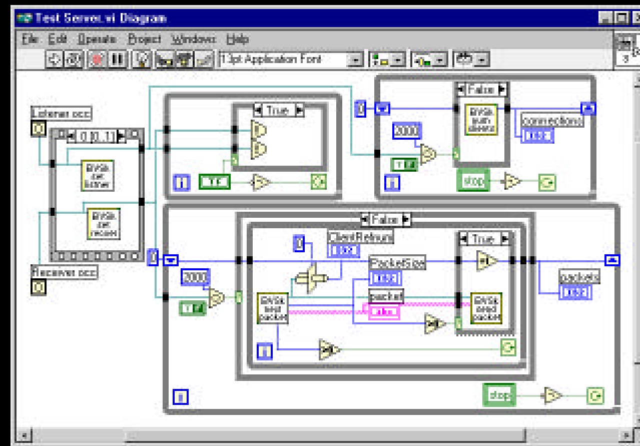
The prototypes for using the Occur function are not included in extcode.h, but are exported in LabVIEW.lib.

```
typedef int Occurrence;
extern "C" MgErr Occur(Occurrence o);

                    MgErr err = Occur((Occurrence )m_occRefnum);
```

When using the Occur function in LabVIEW/BridgeVIEW, you must link your project with LabVIEW.lib, and use extcode.h as a header file. Both are available in the CINTOOLS directories. Some functions exported from LabVIEW.lib may conflict with the MFC library msvcrt.lib. You can configure your Visual C++ project settings to explicitly exclude this library from being used in the link.

Demo – BridgeVIEWSockets Project



NIWEEK'98

Demo – the BridgeVIEWSockets Project.

The Test Server VI creates a listener socket and then passes two occurrences into the BVSockets DLL; one for notification of a change in the number of clients, and one for whenever a client message is received. One loop then waits on each event and processes accordingly. A third loop signals the other two loops when the Front Panel stop button is pressed.

This example just handles the server side of the TCP/IP connection. You can use standard G TCP/IP primitives for all the client side code and just use the DLL functions for the server side.

This example uses builds TCP/IP packets that contain a 4 byte length field followed by a byte stream. This length plus data format is typical of G TCP/IP programs. In case you were wondering, the DLL internally takes care of the big endian to little endian conversions between the server and G client VIs. You must take this into account when you mix G with C++.

Multiple Threads in the DLL

- Some applications require more than one thread
- You can spawn threads in your DLL that run in parallel with your G application
- Use MFC Synchronization and Event Classes to handle multiple threads
  - CCriticalSection
  - CEvent

NI WEEK'98

Using multiple threads in your MFC DLL

Some type of functionality requires additional threads to be spawned in the DLL. You can use extra threads in your DLL that run in parallel with the G application. You can use the same occurrence signaling mechanisms that were used in the previous examples.

In using multiple threads, you must use synchronization classes to manage access to shared data structures. You may also need to use event classes for signaling between threads. Some common MFC classes are CCriticalSection and CEvent.

Using Multiple Threads

- Declare a thread
  - UINT MyThread(LPVOID pParam) {}
- Spawn the thread when needed
  - AfxBeginThread(MyThread, pParam);
- Terminate the thread when appropriate
  - Signal the thread to terminate
  - Wait for the thread to complete
    - GetExitCodeThread(...);

NI WEEK '98

Threads are created within your DLL by declaring a thread:

```
UINT TimerThread(LPVOID pParam)
{
		CTimerInfo* pTimerInfo = (CTimerInfo*)pParam;
		while (TRUE)
		{
				.....			// Thread code - some kind of long running
loop until signalled to exit
		}
		// Terminate this thread by exiting the proc.
		return 0;
}
```

and spawning this thread when needed:

```
m_pTimerThread = AfxBeginThread(TimerThread, &m_TimerInfo);
```

You also must ensure that the thread terminates when your application is finished.

Demo – Timer Test Project

NIWEEK'98

Demo – The Timer Test Project illustrates the basic principles of using a multi-threaded DLL with your G application.

It illustrates both spawning and terminating threads, signaling and synchronization between threads, and event notification back to your G application from a thread.

Integrating C++ and MFC Code
Graphical Programming

The End

NI WEEK '98

Four projects are included that demonstrate some key aspects of using C++ and MFC with G applications.

DialogTest – using MFC-based dialogs from G

GObjects – managing objects with G

BVSockets – handling windows messages and G event notification

TimerTest – handling multiple threads in your DLL

You have to know your C++ programming and MFC quite well in order to take advantage of the examples provided here. However, with the short learning curve of G, you can make some tradeoffs between G and C++ programming to get the best of both worlds.