OPC Custom Interface Tutorial

1. Introduction

1.1 The Purpose of this document

This tutorial will show the steps necessary to set up a Microsoft Visual C++ MFC dialog-based project for communication with an OPC Server.

It will show the implementation of the asynchronous read and write calls only, as synchronous reads and writes are trivial to implement by comparison.

The tutorial and source is intended to concentrate on demonstrating OPC connectivity, for that reason it makes no attempt to create a robust software application, error handling is not discussed and the GUI is kept very simple.

The tutorial should be looked at in conjunction with the provided Visual C++ project, as not all code in the example project is shown in this document.

1.2 Who should read this document?

This tutorial should be read by software and application engineers and others interested in using the OPC custom interface to access Omron hardware or software.

The target audience for this guide is application developers knowledgeable in industrial automation technology and Visual C++ (including MFC). No attempt is made to teach automation technology and terminology or any programming language.

2. Step By Step Tutorial

2.1 Creating the GUI

The first step is to create the Graphical User Interface (GUI). This can be achieved by the following steps:

Create a dialog based MFC application in Visual C++.
Create a menu with the following items:
    Server, Server | Connect
    Server, Server | Disconnect
Add method handlers for the menu items added.
Add the menu to the dialog.
Add a list box (IDC_LIST_ITEMS) to the left hand side of the dialog and assign it to the member variable m_ListItems.
Add three buttons next to the listbox, labelled Initialise, Async Read and Async Write.
Add an edit box next to each of these buttons (IDC_EDIT_READ and IDC_EDIT_WRITE). Assign to them the member variables m_ceItemRead and m_ceItemWrite, respectively.
Add the following OPC source files (these are installed by CX-Server OPC, and placed into an OPC source code subdirectory) to the workspace and make sure that they do *not* use precompiled headers.
    opccomn_i.c
    opcda_i.c
    opcenum_clsid.c

2.2 OPC Server Connection/Disconnection

To achieve OPC server connection and disconnection add the following OPC source code header files:
    opccomn.h
    opcda.h
    afxpriv.h (for string conversion macros)

Define the following constants in the dialogs header.

```
EXTERN_C const CLSID CLSID_OPCServerList;
EXTERN_C const IID IID_IOPCServerList;
EXTERN_C const IID IID_IOPCServer;
```

Add a variable of type IOPCServer* in the dialog'sheader (m_pOPCServer). This is defined in the opcda.h header file.

Add the following code to the Connect menu handler. This initialises the COM library, sets the CLSID and creates an instance of Omron's OPC Server. The

server ID has been hard-coded to “Omron.DataServer.1” in this example, but the ID for any server can be obtained through the registry.

```
HRESULT hr;
hr = CoInitialize(NULL);
if (FAILED(hr))
{
      // Do something
}
USES_CONVERSION;
CLSID clsid;
LPCOLESTR oleProgID = T2OLE("Omron.DataServer.1");
CLSIDFromProgID(oleProgID, &clsid);

CoCreateInstance(
      clsid,
      NULL,
      CLSCTX_LOCAL_SERVER,
      IID_IOPCServer,
      (void**)&m_pOPCServer);
```

For the USES_CONVERSION macro in the code above, you must ensure that the atlconv.h header has been included.

Place the disconnect code in the OnClose() event handler.

```
if (m_pOPCServer)
{
      m_pOPCServer->Release();
      m_pOPCServer = NULL;
}
```

2.3 Adding/Removing a Group

Add the following member variables in the dialogs header file:

```
IOPCServer* m_pOPCServer;
IUnknown*   m_pUnknown;
HANDLE      m_hAsyncCompleted;
OPCHANDLE   m_opcServerHandle;
DWORD       m_dwCPCookie;
OPCHANDLE   m_hGroup;
OPCHANDLE   m_hClientGroup;
```

Add code to add a group in the “Initialise” button’s event handler.

```
DWORD dwRevisedUpdateRate = 0;
HRESULT hr = E_FAIL;

if (m_pOPCServer)
{
      hr = m_pOPCServer->AddGroup(
            (LPCWSTR)"myGroup",
            TRUE,
            1000,
            m_hClientGroup,
            0,
```

```
            0,
            1033,
            &m_hGroup,
            &dwRevisedUpdateRate,
            __uuidof(IUnknown),
            (IUnknown**)&m_pUnknown);

      if (SUCCEEDED(hr))
      {
            // Assuming success for the time being.
      }
}
```

Add the remove group code to the OnClose() event handler.

```
hResult = m_pOPCServer->RemoveGroup(m_hGroup, TRUE);
```

2.4 Browsing Items

At the end of the AddGroup method call BrowseItems. This will populate the item list. The code for browse items is below.

```
USES_CONVERSION;

m_ListItems.ResetContent();
IOPCBrowseServerAddressSpace* pBAS = NULL;
IEnumString* pEnum = NULL;
LPOLESTR pStr, pStrItem;
ULONG actual;
long i = 0;

m_pOPCServer->QueryInterface(
      __uuidof(IOPCBrowseServerAddressSpace), (void**) &pBAS);
pBAS->BrowseOPCItemIDs(
            OPC_FLAT, T2W("*"), VT_UI4, // Unsigned longs only.
            OPC_READABLE|OPC_WRITEABLE, &pEnum);

while (pEnum->Next(1, &pStr, &actual) == S_OK)
{
      pBAS->GetItemID(pStr, &pStrItem);

      m_ListItems.AddString(CString(pStrItem));
      m_ListItems.SetItemData(i, (DWORD)VT_UI4);

      i++;

      CoTaskMemFree(pStr);
      CoTaskMemFree(pStrItem);
}

pEnum->Release();
pBAS->Release();
```

As you can see the code will only browse for items of type VT_UI4 (i.e. unsigned long). If you have no items of this type in your server you can either add some yourself or change the data type in this code and in other areas.

2.5 Adding/Removing an Item

Under the OnSelChange event we will add the functionality for adding an item. It is possible to add more than one item at a time but a more sophisticated mechanism for keeping track of these items would be required (MFC and STL provide neat classes for doing this).

```
void CCustomInterfaceTutorialDlg::OnSelchangeListItems()
{
	CString csSel;
	CString csPath;
	int nCurSel = m_ListItems.GetCurSel();
	m_ListItems.GetText(nCurSel, csSel);
	VARTYPE vtType = VT_UI4;

	HRESULT *pErrors = NULL;
	OPCITEMRESULT *pOIR = NULL;
	OPCITEMDEF oid;

	oid.szItemID            = csSel.AllocSysString();
	oid.vtRequestedDataType = vtType;
	oid.szAccessPath        = csPath.AllocSysString();
	oid.hClient             = (OPCHANDLE)nCurSel;
	oid.bActive             = TRUE;
	oid.dwBlobSize          = 0;

	IOPCItemMgt* pIIM;
	m_pUnknown->QueryInterface(
		__uuidof(IOPCItemMgt),
		(void**)&pIIM);
	pIIM->AddItems(1, &oid, &pOIR, &pErrors);

	m_opcServerHandle = pOIR->hServer;

	SysFreeString(oid.szItemID);
	SysFreeString(oid.szAccessPath);

	if (pOIR && pOIR->pBlob)
		CoTaskMemFree(pOIR->pBlob);

	if (pOIR)
		CoTaskMemFree(pOIR);

	if (pErrors)
		CoTaskMemFree(pErrors);
}
```

Add the following code to the OnClose() event handler prior to disconnecting from the server:

```
	HRESULT *pErrors = NULL;
	HRESULT hResult = NULL;

	IOPCItemMgt* pIIM;
	m_pUnknown->QueryInterface(
		__uuidof(IOPCItemMgt),
```

```
            (void**)&pIIM);
      hResult = pIIM->RemoveItems(
            1,
            &m_opcServerHandle,
            &pErrors);

      if (pErrors)
            CoTaskMemFree(pErrors);

      if (pIIM)
            pIIM->Release();
```

2.6 Using Advise and Unadvise

To be able to respond to events fired by the server (for asynchronous calls, for example) it is first necessary to "advise" the server (to let the server know about the client). The code shown below is straightforward code copied from the ATL source provided with Visual C++, with minor modifications to make sure that it works well in MFC. When the callback is no longer required it is necessary to use "unadvise".

Note: For this code to work the standard EnableAutomation() method must be called prior to it being executed. It is a matter of preference where this goes, but in this case it can be put in the OnInitDialog() event handler.

```
void CCustomInterfaceTutorialDlg::Advise(void)
{
      IConnectionPointContainer* pCPC;
      IConnectionPoint* pCP;
      HRESULT hResult;

      LPUNKNOWN pUnknown = GetIDispatch(FALSE);

      hResult = m_pUnknown->QueryInterface(
            __uuidof(IConnectionPointContainer),
            (void**)&pCPC);
      if (SUCCEEDED(hResult))
            hResult = pCPC->FindConnectionPoint(
                  __uuidof(IOPCDataCallback),
                  &pCP);
      if (SUCCEEDED(hResult))
            hResult = pCP->Advise(pUnknown, &m_dwCPCookie);
}

void CCustomInterfaceTutorialDlg::UnAdvise(void)
{
      IConnectionPointContainer* pCPC;
      IConnectionPoint* pCP;
      HRESULT hResult;

      hResult = m_pUnknown->QueryInterface(
            __uuidof(IConnectionPointContainer),
            (void**)&pCPC);
      if (SUCCEEDED(hResult))
            hResult = pCPC->FindConnectionPoint(
                  __uuidof(IOPCDataCallback),
                  &pCP);
```

```
	if (SUCCEEDED(hResult))
		hResult = pCP->Unadvise(m_dwCPCookie);
}
```

2.7 Callbacks

The following code may not be familiar to those software engineers who have not worked with MFC and COM, but it is the standard way of dealing with COM callback events. See MSDN's "TN038: MFC/OLE IUnknown Implementation" for further detailed information on this quite complicated area.

In the header insert the following code:

```
protected:

BEGIN_CONNECTION_PART(OPCGroup, OPCDataCallback)
CONNECTION_IID(IID_IOPCDataCallback)
END_CONNECTION_PART(OPCDataCallback)

DECLARE_CONNECTION_MAP()

DECLARE_INTERFACE_MAP()

	BEGIN_INTERFACE_PART(OnDataCallback, IOPCDataCallback)
	STDMETHOD(OnDataChange)(
		/*[in]*/ DWORD dwTransid,
		/*[in]*/ OPCHANDLE hGroup,
		/*[in]*/ HRESULT hrMasterquality,
		/*[in]*/ HRESULT hrMastererror,
		/*[in]*/ DWORD dwCount,
		/*[in, sizeis(dwCount)]*/ OPCHANDLE* phClientItems,
		/*[in, sizeis(dwCount)]*/ VARIANT* pvValues,
		/*[in, sizeis(dwCount)]*/ WORD* pwQualities,
		/*[in, sizeis(dwCount)]*/ FILETIME* pftTimeStamps,
		/*[in, sizeis(dwCount)]*/ HRESULT* pErrors);
	STDMETHOD(OnReadComplete)(
		/*[in]*/ DWORD dwTransid,
		/*[in]*/ OPCHANDLE hGroup,
		/*[in]*/ HRESULT hrMasterquality,
		/*[in]*/ HRESULT hrMastererror,
		/*[in]*/ DWORD dwCount,
		/*[in, sizeis(dwCount)]*/ OPCHANDLE* phClientItems,
		/*[in, sizeis(dwCount)]*/ VARIANT* pvValues,
		/*[in, sizeis(dwCount)]*/ WORD* pwQualities,
		/*[in, sizeis(dwCount)]*/ FILETIME* pftTimeStamps,
		/*[in, sizeis(dwCount)]*/ HRESULT* pErrors);
	STDMETHOD(OnWriteComplete)(
		/*[in]*/ DWORD dwTransid,
		/*[in]*/ OPCHANDLE hGroup,
		/*[in]*/ HRESULT hrMasterError,
		/*[in]*/ DWORD dwCount,
		/*[in, sizeis(dwCount)]*/ OPCHANDLE* phClientItems,
		/*[in, sizeis(dwCount)]*/ HRESULT* pError);
	STDMETHOD(OnCancelComplete)(
		/*[in]*/ DWORD dwTransid,
		/*[in]*/ OPCHANDLE hGroup);
	END_INTERFACE_PART(OnDataCallback)
```

In the source file insert:

```
BEGIN_CONNECTION_MAP(CCustomInterfaceTutorialDlg, CCmdTarget)
    CONNECTION_PART(CCustomInterfaceTutorialDlg, IID_IOPCDataCallback,
OPCDataCallback)
END_CONNECTION_MAP()

BEGIN_INTERFACE_MAP(CCustomInterfaceTutorialDlg, CCmdTarget)
    INTERFACE_PART(CCustomInterfaceTutorialDlg, IID_IOPCDataCallback,
OnDataCallback)
END_INTERFACE_MAP()

long FAR EXPORT
CCustomInterfaceTutorialDlg::XOnDataCallback::OnCancelComplete(DWORD
dwTransID, OPCHANDLE hGroup)
{
      METHOD_PROLOGUE(CCustomInterfaceTutorialDlg, OnDataCallback)
      return 1;
}

HRESULT FAR EXPORT
CCustomInterfaceTutorialDlg::XOnDataCallback::OnWriteComplete(DWORD
dwTransID, OPCHANDLE hGroup, HRESULT hrMasterError, DWORD dwCount,
OPCHANDLE* phClientItems, HRESULT* pError)
{
      METHOD_PROLOGUE(CCustomInterfaceTutorialDlg, OnDataCallback)
      return S_OK;
}

long FAR EXPORT
CCustomInterfaceTutorialDlg::XOnDataCallback::OnReadComplete(DWORD
dwTransid, OPCHANDLE hGroup, HRESULT hrMasterquality, HRESULT
hrMastererror, DWORD dwCount, OPCHANDLE* phClientItems, VARIANT*
pvValues, WORD* pwQualities, FILETIME* pftTimeStamps, HRESULT*
pErrors)
{
      METHOD_PROLOGUE(CCustomInterfaceTutorialDlg, OnDataCallback)
      ASSERT_VALID(pThis);
      return S_OK;
}

long FAR EXPORT
CCustomInterfaceTutorialDlg::XOnDataCallback::OnDataChange(DWORD
dwTransid, OPCHANDLE hGroup, HRESULT hrMasterquality, HRESULT
hrMastererror, DWORD dwCount, OPCHANDLE* phClientItems, VARIANT*
pvValues, WORD* pwQualities, FILETIME* pftTimeStamps, HRESULT*
pErrors)
{
      METHOD_PROLOGUE(CCustomInterfaceTutorialDlg, OnDataCallback)
      return S_OK;
}

ULONG FAR EXPORT
CCustomInterfaceTutorialDlg::XOnDataCallback::Release(void)
{
      METHOD_PROLOGUE(CCustomInterfaceTutorialDlg, OnDataCallback)
            return pThis->ExternalRelease();
      }
```

```
ULONG                           FAR                             EXPORT
CCustomInterfaceTutorialDlg::XOnDataCallback::AddRef(void)
{
      METHOD_PROLOGUE(CCustomInterfaceTutorialDlg,
OnDataCallback)
      return pThis->ExternalAddRef();
}

long                            FAR                             EXPORT
CCustomInterfaceTutorialDlg::XOnDataCallback::QueryInterface(RE
FIID iid, void** ppvObj)
{
      METHOD_PROLOGUE(CCustomInterfaceTutorialDlg,
OnDataCallback)
      return pThis->ExternalQueryInterface(&iid, ppvObj);
}
```

2.8 WaitForMessage and MessagePumps

The WaitForMessage and MessagePumps are used to process messages and allow the callbacks to take place. The message pump is found in Stdafx.cpp (this is standard code):

```
bool MessagePump()
{
      MSG msg;
      do
      {
            if (!::GetMessage (&msg,NULL,NULL,NULL))
            {
                  ::PostMessage(
                        msg.hwnd,
                        msg.message,
                        msg.wParam,
                        msg.lParam);

                        return false;
          }
          else
          {
                  ::TranslateMessage(&msg);
                  ::DispatchMessage(&msg);
          }
      }
      while (::PeekMessage(&msg,NULL,NULL,NULL,PM_NOREMOVE));

      return true;
}
```

Make sure it is declared as below, in stdafx.h:

```
extern bool MessagePump();
```

In the dialog's code add the function WaitForEvent shown below. You can see that the code extract is waiting for a particular event to be fired. This event is defined by the AsyncRead code (shown later in this tutorial).

```
bool CCustomInterfaceTutorialDlg::WaitForEvent(HANDLE hEvent)
{
    MSG msg;

      for (;;)
      {
            DWORD dwResult = ::MsgWaitForMultipleObjects(
                                    1,
                                    &hEvent,
                                    FALSE,
                                    300000,
                                    QS_ALLINPUT);
            switch (dwResult)
            {
                  case WAIT_OBJECT_0:
                  case WAIT_ABANDONED_0:
                        return true;
                  case WAIT_TIMEOUT:
                        return false;
                  case (WAIT_OBJECT_0 + 1):
                        while (::PeekMessage(
                                    &msg,NULL,NULL,NULL,PM_NOREMOVE))
                        {
                              if (!::MessagePump ())
                              {
                                    return false;
                              }
                        }
                        break;
                  default:
                        return true;
            }
      }
}
```

2.9 Asynchronous Read

The event, m_hAsyncCompleted, is set in the callback function OnReadComplete. At this point no more callbacks will be received. If you wanted to continue to process callbacks then the event variable should be set in a separate function, e.g. in a “stop” button handler.

```
void CCustomInterfaceTutorialDlg::OnButtonRead()
{
      Advise();

      HRESULT hResult = NULL;
      HRESULT* phErrors = NULL;
      DWORD dwCancelID;
      DWORD dwTransID = 1;
      IOPCAsyncIO2* pAsyncIO2 = NULL;

      ResetEvent(m_hAsyncCompleted);

      m_pUnknown->QueryInterface(
            __uuidof(IOPCAsyncIO2),
            (void**)&pAsyncIO2);
      hResult = pAsyncIO2->Read(
```

```
            1, &m_opcServerHandle,
            dwTransID, &dwCancelID, &phErrors);

      if (hResult != S_OK)  // Purposely checking against 0.
      {
      }
      else
      {
            WaitForEvent(m_hAsyncCompleted);
      }

      CoTaskMemFree(phErrors);
}
```

2.10 OnReadComplete

Notice the use of the SetEvent function and the note in “Async Read”.

```
      CString csValue;
      if (pvValues)
      {
            csValue.Format("%ld", pvValues[0].ulVal);
      }

      pThis->m_ceRead.SetWindowText(csValue.GetBuffer(0));
      SetEvent(pThis->m_hAsyncCompleted);
```

2.11 Asynchronous Write

The OnWriteComplete() callback was not used in the example code, but a real-world system would probably perform some error checking.

The asynchronous writing code is:

```
      Advise();

      HRESULT hResult = NULL;
      HRESULT* phErrors = NULL;
      DWORD dwCancelID;
      DWORD dwTransID = 1;
      IOPCAsyncIO2* pAsyncIO2 = NULL;

      ResetEvent(m_hAsyncCompleted);

      m_pUnknown->QueryInterface(
            __uuidof(IOPCAsyncIO2), (void**)&pAsyncIO2);

      CString csWriteContents;
      m_ceWrite.GetWindowText(csWriteContents);
      unsigned long nVal = (unsigned long)
            atoi(csWriteContents.GetBuffer(0));

      VARIANT variant;
      VariantInit(&variant);
      variant.ulVal = nVal;
      variant.vt = VT_UI4;
```

```
hResult = pAsyncIO2->Write(
      1, &m_opcServerHandle, &variant,
      dwTransID, &dwCancelID, &phErrors);

if (hResult != S_OK)  // Purposely checking against 0.
{
}
else
{
      WaitForEvent(m_hAsyncCompleted);
}

CoTaskMemFree(phErrors);
```

3. Conclusion

Hopefully you can see from the tutorial and associated Visual C++ project that Visual C++ programmers should find that using the custom interface for the OPC Server is not overly complicated once some code has been written. Most of the complex code shown is in fact standard functionality (e.g. the message pump and COM event callback handling) rather than OPC-specific.

Although some corners were cut in order to keep the code and explanation simple you should find that building a more complex and robust system (with multiple points and groups and sophisticated error checking) will not require large quantities of horrendously complex code.

The overall conclusion is that, for experienced programmers familiar with Visual C++, the Custom Interface provides a practical and sensible way of achieving medium to large OPC applications