OLE Embedded Components and Containers part 1

Program examples compiled using Visual C++ 6.0 compiler on Windows XP Pro machine with Service Pack 2. The Excel version is Excel 2003/Office 11. Topics and sub topics for this tutorial are listed below. Don't forget to read Tenouk's small disclaimer. The supplementary notes for this tutorial are IOleObject and OLE.

Index:

Intro

In this module, you'll get familiar with the core of **Object Linking and Embedding** (OLE). You'll learn how an **embedded component** talks to its **container**. This is the needed knowledge in order to use **ActiveX controls**, **in-place activation** (Visual Editing), and **linking**, all of which are described in Adam Denning's ActiveX Controls Inside Out (Microsoft Press, 1997), Kraig Brockschmidt's Inside OLE, 2d ed. (Microsoft Press, 1995), and other books.
You'll get started with a Microsoft Foundation Class mini-server, an out-of-process OLE component program that supports in-place activation but can't run as a stand-alone program. Running this component will give you a good idea of what OLE looks like to the user, in case you don't know already. You'll also see the extensive MFC support for this kind of application. If you work at only the top MFC level, however, you won't appreciate or understand the underlying OLE mechanisms. For that, you'll have to dig deeper. Shepherd and Wingo's MFC Internals (Addison-Wesley, 1996) provides extensive coverage of the internal workings of MFC's OLE Document support.
Next you'll build a container program that uses the familiar parts of the MFC library but supports embedded OLE objects that can be edited in their own windows. This container can, of course, run your MFC mini-server, but you'll really start to learn OLE when you build a mini-server from scratch and watch the interactions between it and the container.

Embedding vs. In-Place Activation (Visual Editing)

Visual Editing is Microsoft's name for **in-place activation**. A component that supports in-place activation also supports **embedding**. Both in-place activation and embedding store their data in a **container's document**, and the container can activate both. An in-place-capable component can run inside the container application's main window, taking over the container's menu and toolbar, and it can run in its own top-level window if necessary. An embedded component can run only in its own window, and that window has a special menu that does not include file commands. Figure 1 shows a Microsoft Excel spreadsheet **in-place activated** inside a Microsoft Word document. Notice the Excel menus and toolbars.
Some container applications support only embedded components; others support both in-place and embedded components. Usually, an in-place container program allows the user to activate in-place components either in place or in their own windows. You should be getting the idea that **embedding is a subset of in-place activation**. This is true not only at the user level but also at the OLE implementation level. Embedding relies on two key interfaces, `IOleObject` and `IOleClientSite`, which are used for in-place activation as well.

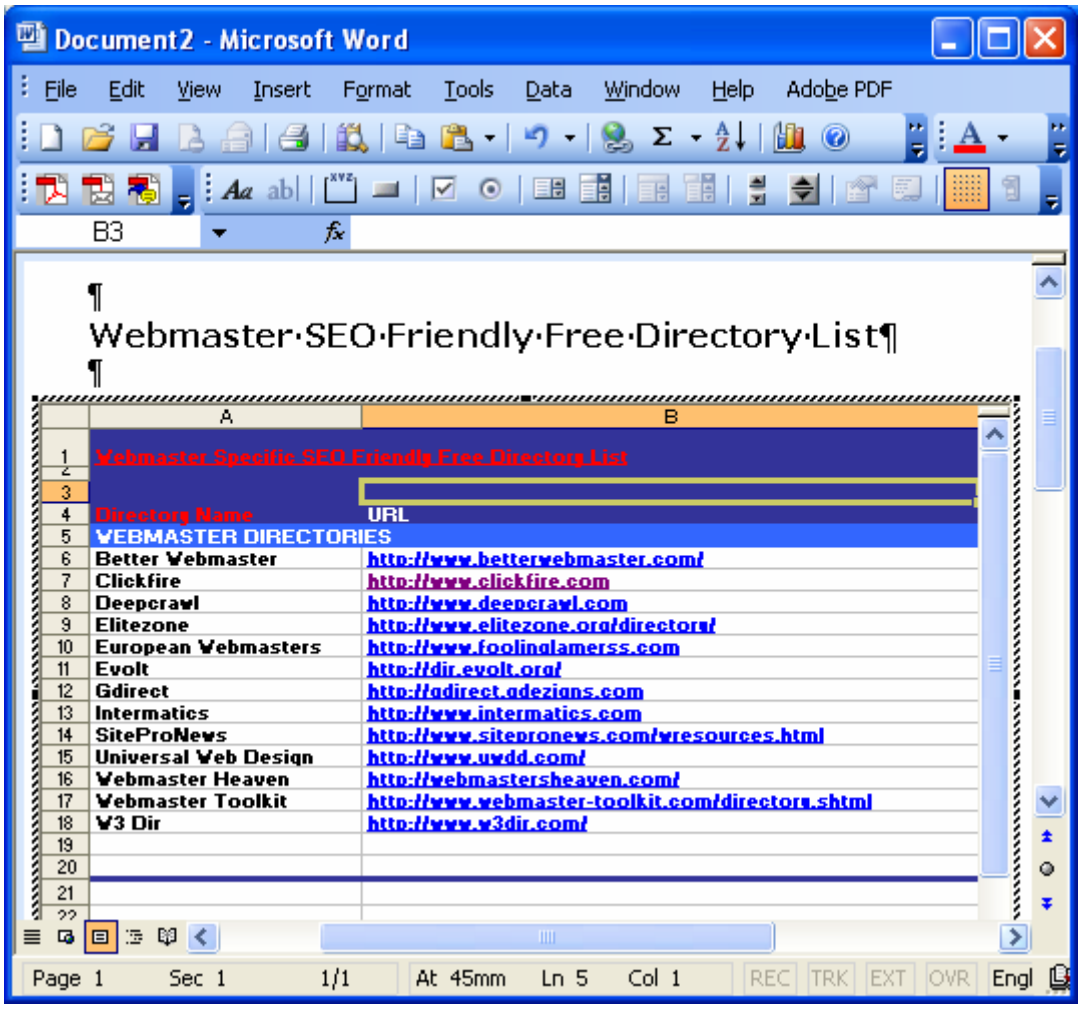

Figure 1: An Excel spreadsheet activated inside a Word document.

Mini-Servers vs. Full Servers (Components): Linking

A **mini-server** can't be run as a stand-alone program; it depends on a container application to launch it. It can't do its own file I/O but depends on the container's files. A **full server**, on the other hand, can be run both as a stand-alone program and from a container. When it's running as a stand-alone program, it can read and write its own files, which mean that it supports OLE linking. With **embedding**, the container document contains all the data that the component needs; with **linking**, the container contains only the name of a file that the component must open.

The Dark Side of Visual Editing

We're really enthusiastic about the COM architecture, and we truly believe that ActiveX Controls will take over the programming world. We're not so sure about Visual Editing, though, and we aren't alone. From our cumulative experience meeting developers around the world, we've learned that few developers are writing applications that fit the "objects embedded in a document" model. From our programming experiences, we've learned that it is tricky for containers and components to coordinate the size and scale of embedded objects. From our "user" experience, we've learned that in-place activation can be slow and awkward, although the situation is improving with faster computers. If you don't believe us, try embedding an Excel worksheet in a Word document, as shown in Figure 28-1. Resize the worksheet in both the active mode and the non-active mode. Notice that the two sizes don't track and that processing is slow.

Consider the need for drawing graphics. Older versions of Microsoft PowerPoint used an in-place component named Microsoft Draw. The idea was that other applications could use this component for all their graphics needs. Well, it didn't work out that way, and PowerPoint now has its own built-in drawing code. If you have old PowerPoint files with Microsoft Draw objects, you'll have a hard time converting them.

Now consider printing. Let's say you receive a Word document over the Internet from Singapore, and that document contains the metafiles for some embedded objects. You don't have the objects' component programs, however. You

print the document on your trusty 1200-dpi color laser printer, and the metafiles print with it. Embedded object metafiles can be rendered for a specific printer, but it's doubtful that the person in Singapore used your printer driver when creating the document. The result is less-than-optimal output with incorrect line breaks.
We do believe, however, that the OLE embedding technology has a lot of potential. Playing sounds and movies is cool, and storing objects in a database is interesting. What you learn in this module will help you think of new uses for this technology.

Windows Metafiles and Embedded Objects

You're going to need a little more Windows theory before you can understand how **in-place** and **embedded** components draw in their clients' windows. We've avoided metafiles up to this point because we haven't needed them, but they've always been an integral part of Windows. Think of a metafile as a cassette tape for GDI instructions. To use a cassette, you need a player/recorder, and that's what the metafile device context (DC) is. If you specify a filename when you create the metafile DC, your metafile will be **saved on disk**; otherwise, it's **saved in memory** and you get a handle.
In the world of OLE embedding, **components create metafiles and containers play them**. Here's some component code that creates a metafile containing some text and a rectangle:

```
CMetaFileDC dcm; // MFC class for metafile DC
VERIFY(dcm.Create());
dcm.SetMapMode(MM_ANISOTROPIC);
dcm.SetWindowOrg(0, 0);
dcm.SetWindowExt(5000, -5000);
// drawing code
dcm.Rectangle(CRect(500, -1000, 1500, -2000));
dcm.TextOut(0, 0, m_strText);
HMETAFILE hMF = dcm.Close();
ASSERT(hMF != NULL);
```

It's possible to create a metafile that uses a **fixed mapping mode** such as `MM_LOENGLISH`, but with OLE we'll always use the `MM_ANISOTROPIC` mode, which is **not fixed**. The metafile contains a `SetWindowExt()` call to set the x and y extents of the window, and the program that plays the metafile calls `SetViewportExt()` to set the extents of the viewport. Here's some code that you might put inside your container view's `OnDraw()` function:

```
pDC->SetMapMode(MM_HIMETRIC);
pDC->SetViewportExt(5000, 5000);
pDC->PlayMetafile(hMF);
```

What's supposed to show up on the screen is a rectangle 1-by-1-cm square because the component assumes the `MM_HIMETRIC` mapping mode. It will be 1-by-1 cm as long as the viewport extent matches the window extent. If the container sets the viewport extent to (5000, 10000) instead, the rectangle will be stretched vertically but the text will be the same size because it's drawn with the non-scalable system font. If the container decided to use a mapping mode other than `MM_HIMETRIC`, it could adjust the viewport extent to retain the 1-by-1-cm size.
To reiterate, the component sets the window extent to the assumed size of the viewable area and draws inside that box. If the component uses a negative y extent, the drawing code works just as it does in `MM_HIMETRIC` mapping mode. The container somehow gets the component's extent size and attempts to draw the metafile in an area with those `HIMETRIC` dimensions.
Why are we bothering with metafiles? Because the container needs to draw something in the component's rectangle, even if the component program isn't running. The component creates the metafile and hands it off in a data object to the in-process OLE handler module on the container side of the **Remote Procedure Call** (RPC) link. The handler then caches the metafile and plays it on demand and also transfers it to and from the container's storage. When a component is in-place active, however, its view code is drawing directly in a window that's managed by the container.

The MFC OLE Architecture for Component Programs

We're not going into too many details here, just enough to allow you to understand the new files in the next example. You need to know about three new MFC base classes: `COleIPFrameWnd`, `COleServerDoc`, and `COleServerItem`. When you use AppWizard to generate an OLE component, AppWizard generates a class derived from each of the base classes, in addition to an application class, a main frame class, and a view class. The

COleIPFrameWnd class is rather like CFrameWnd. It's your application's main frame window, which contains the view. It has a menu associated with it, IDR_SRVR_INPLACE, which will be merged into the container program's menu. When your component program is running in place, it's using the in-place frame, and when it's running stand-alone or embedded, it's using the regular frame, which is an object of a class derived from CFrameWnd. The embedded menu is IDR_SRVR_EMBEDDED, and the stand-alone menu is IDR_MAINFRAME. The COleServerDoc class is a replacement for CDocument. It contains added features that support OLE connections to the container. The COleServerItem class works with the COleServerDoc class. If components never supported OLE linking, the functionality of the two classes could be combined into one class. Because stand-alone component programs do support linking, the MFC architecture dictates that both classes be present in all components. You'll see in the EX28C example that we can make our own simple mini-server without this division.
Together, the COleServerItem class and the COleServerDoc class implement a whole series of OLE interfaces, including IOleObject, IDataObject, IPersistStorage, and IOleInPlaceActiveObject. These classes make calls to the container, using interface pointers that the container passes to them. The important things to know, however, are that your derived CView class draws in the component's in-place-active window and that the derived COleServerItem class draws in the metafile on command from the container.

The EX32A Example: An MFC In-Place-Activated Mini-Server

You don't need much OLE theory to build an MFC mini-server. This example is a good place to start, though, because you'll get an idea of how containers and components interact. This component isn't too sophisticated. It simply draws some text and graphics in a window. The text is stored in the document, and there's a dialog for updating it. Here are the steps for creating the program from scratch:

Run AppWizard to create the EX32A project in the ex32a directory or wherever you have designated your project directory.

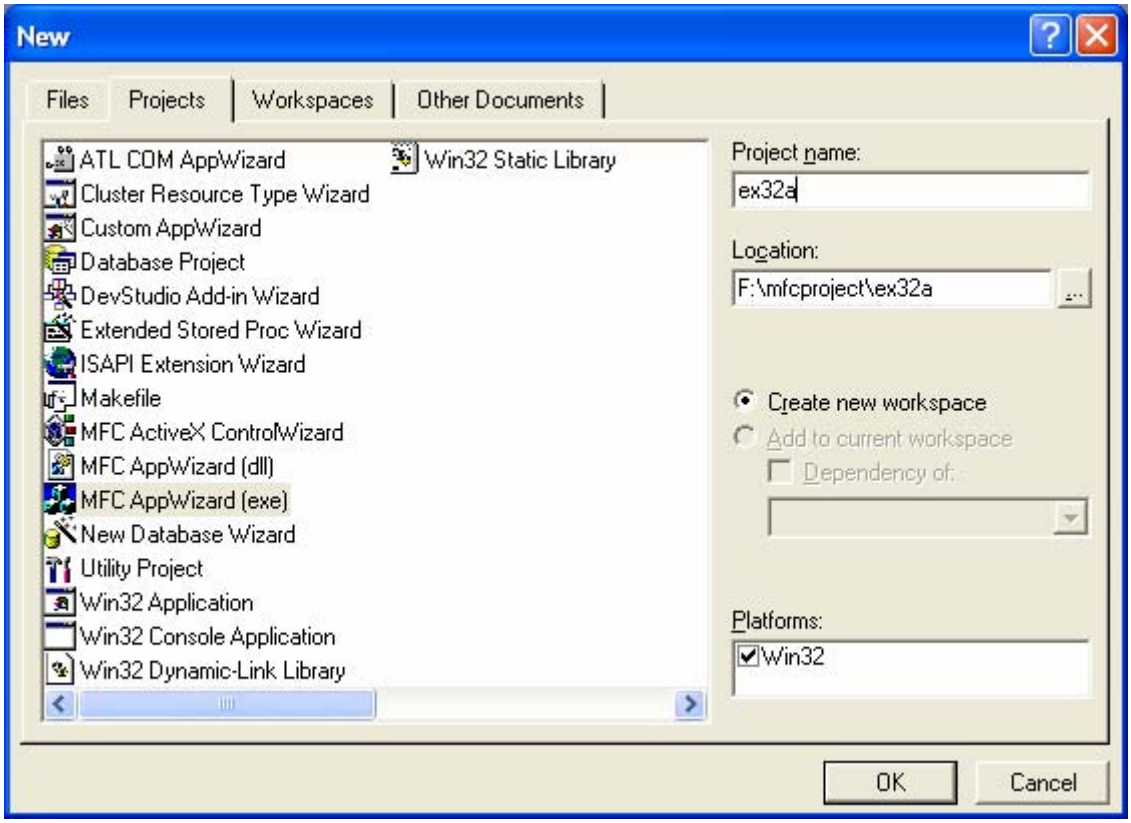

Figure 2: EX32A – Visual C++ new project dialog.

Select **Single document** interface.

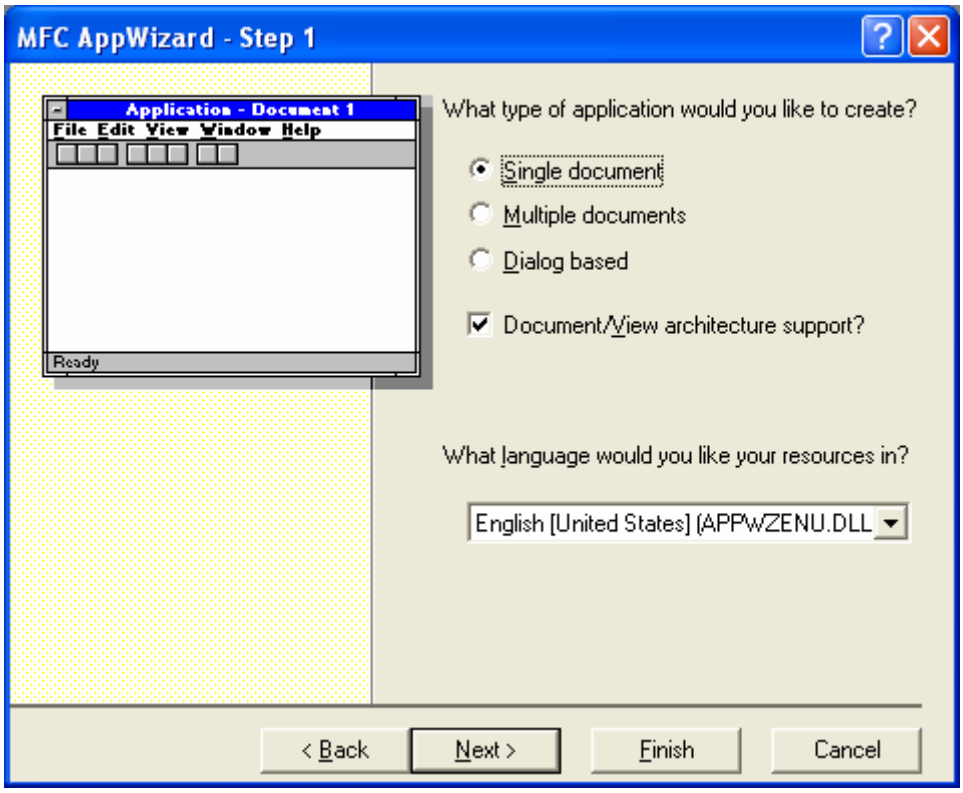

Figure 3: EX32A – AppWizard step 1 of 6.

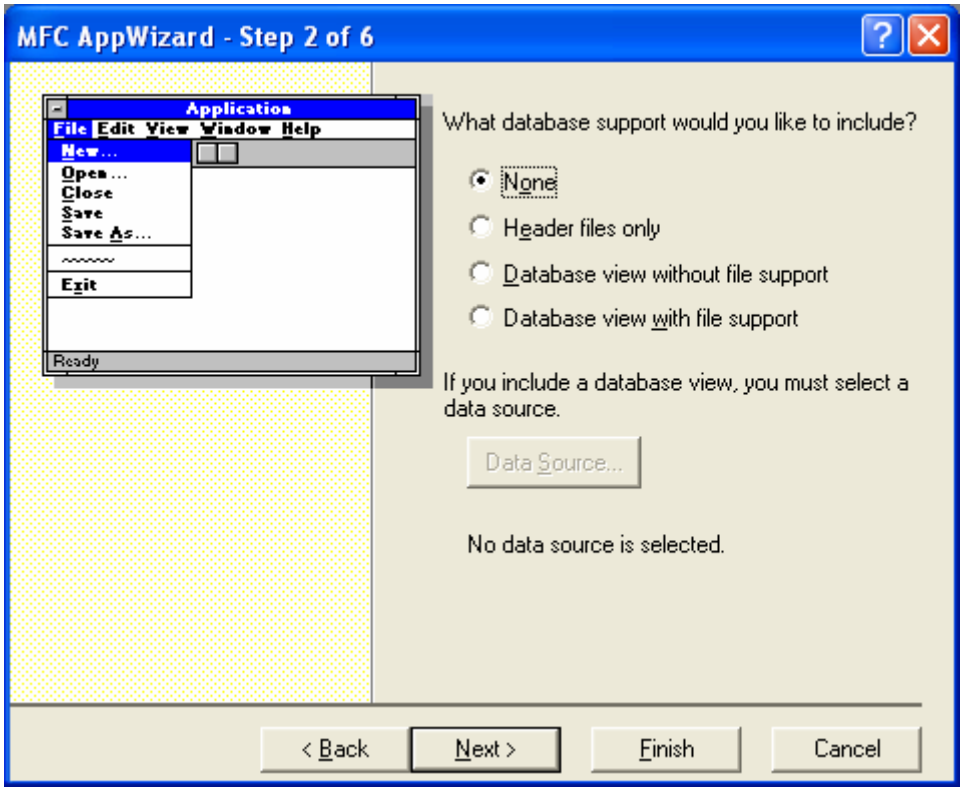

Figure 4: EX32A – AppWizard step 2 of 6.

Click the **Mini-Server** option in the AppWizard Step 3 dialog shown here.

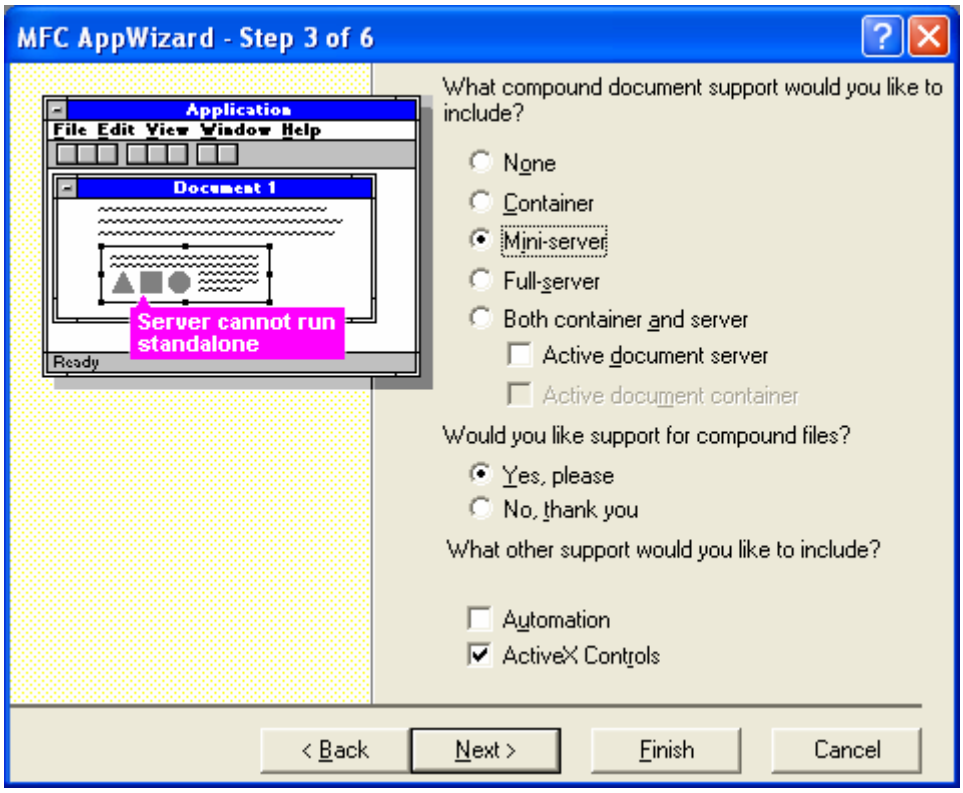

Figure 5: EX32A – AppWizard step 3 of 6.

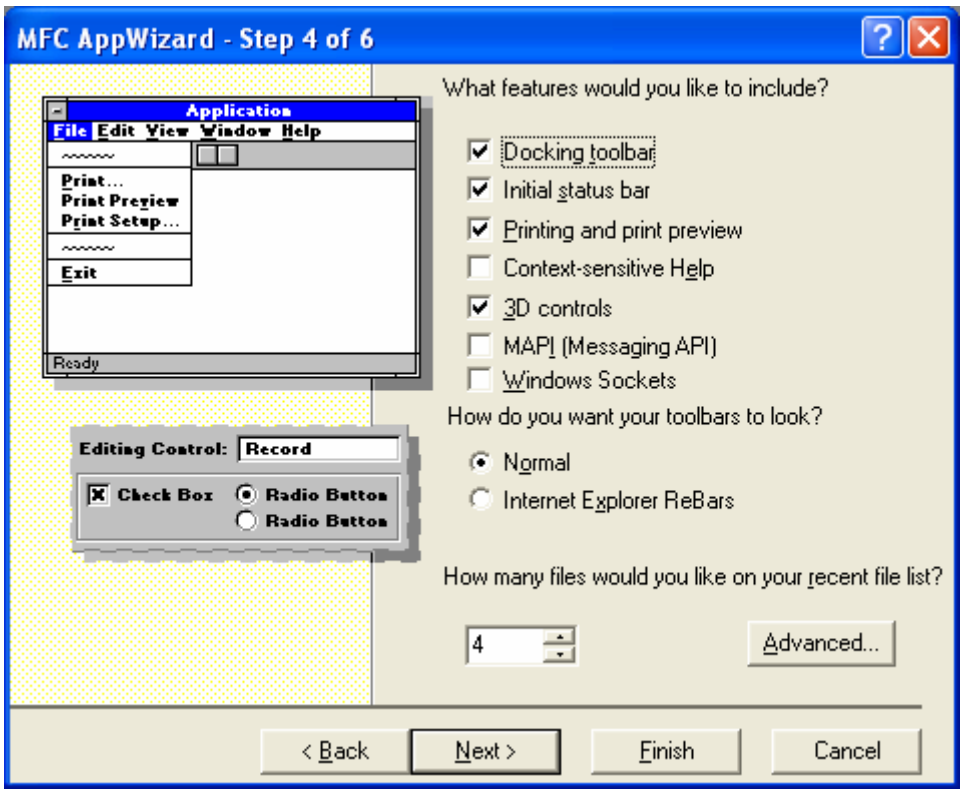

Figure 6: EX32A – AppWizard step 4 of 6.

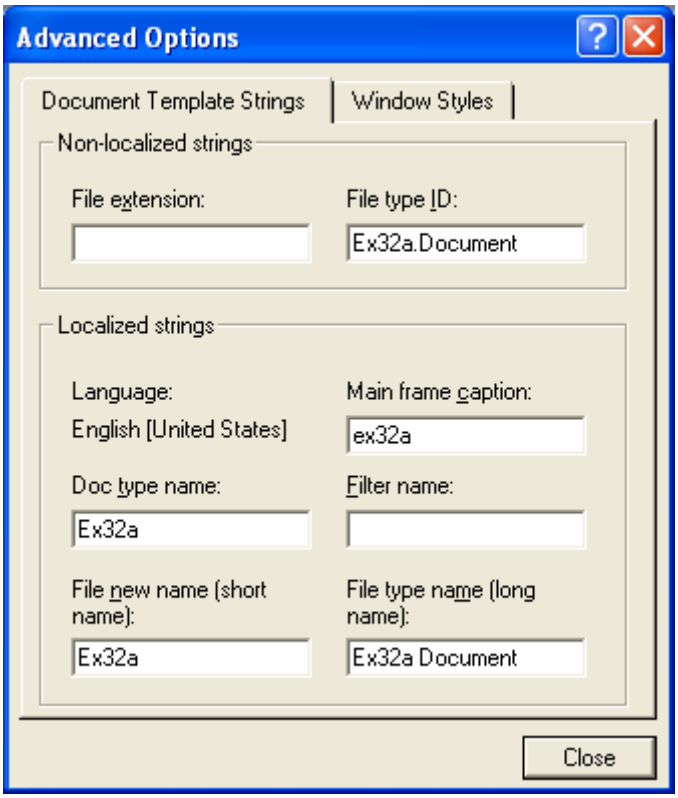

Figure 7: EX32A – the default localized strings.

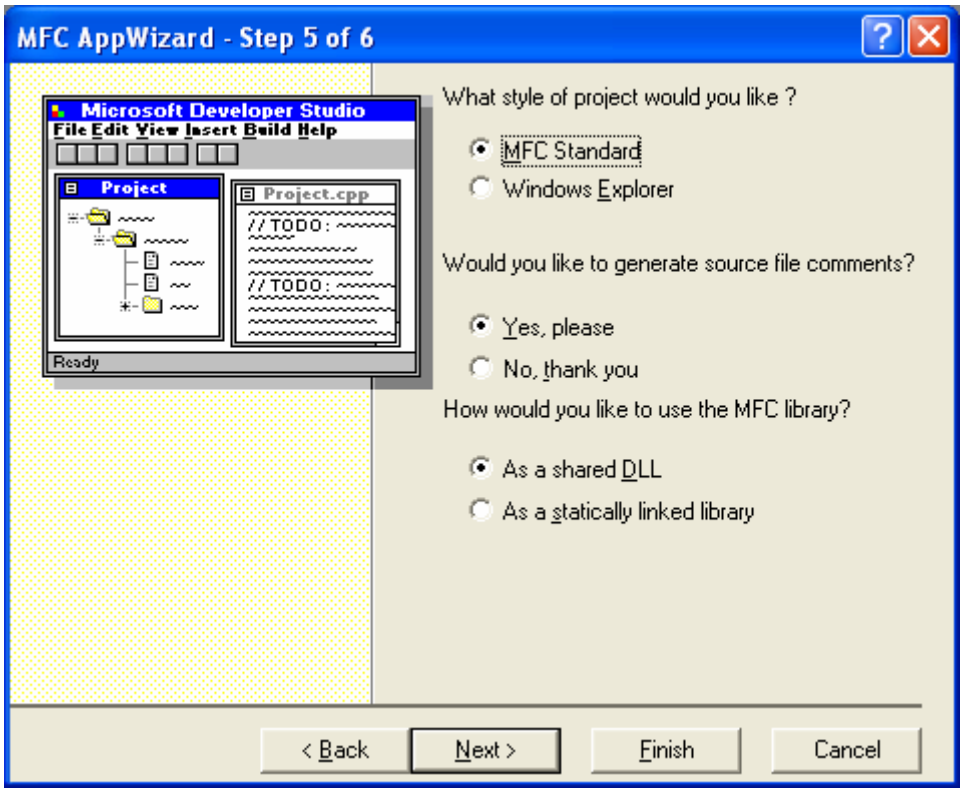

Figure 8: EX32A – AppWizard step 5 of 6.

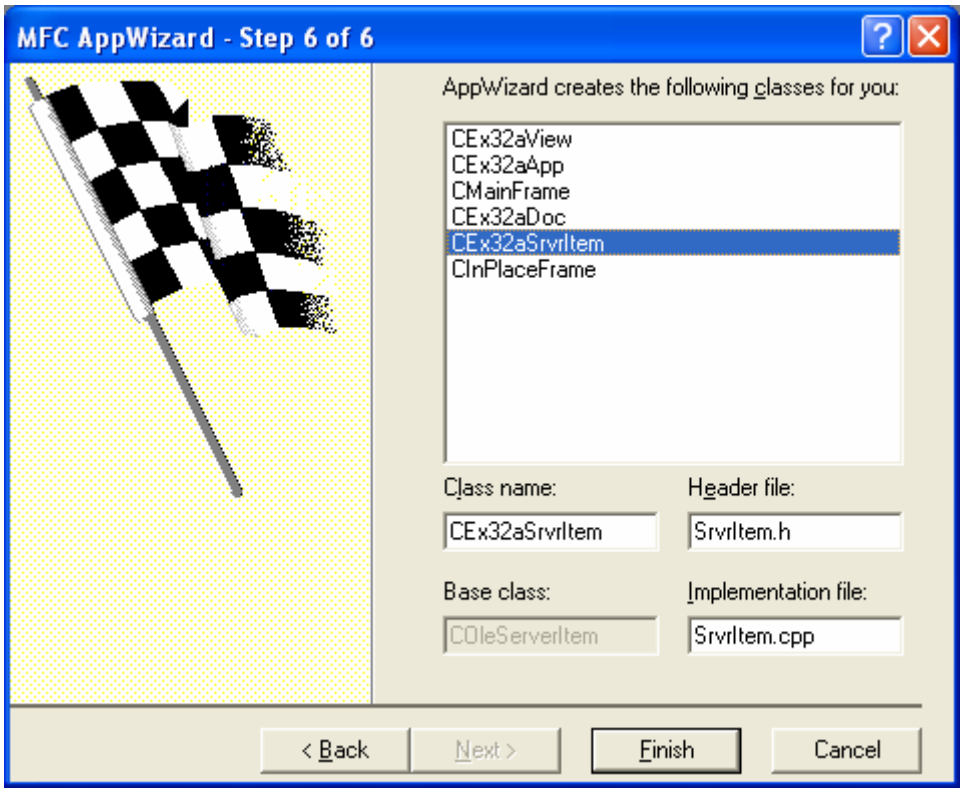

Figure 9: EX32A – AppWizard step 6 of 6.

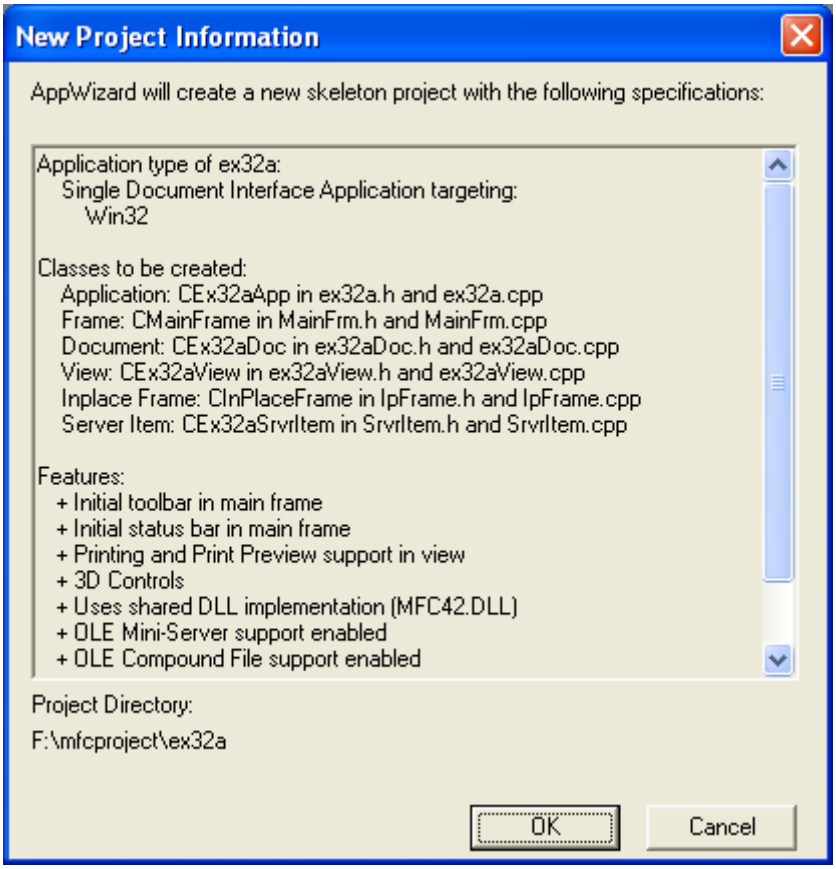

Figure 10: EX32A project summary.

Examine the generated files. You've got the familiar application, document, main frame, and view files, but you've got two new files too as shown in the following Table and Figures.

Header	Implementation	Class	MFC Base Class
SrvrItem.h	SrvrItem.cpp	`CEx32aSrvrItem`	`COleServerItem`
IpFrame.h	IpFrame.cpp	`CInPlaceFrame`	`COleIPFrameWnd`

Table 1.

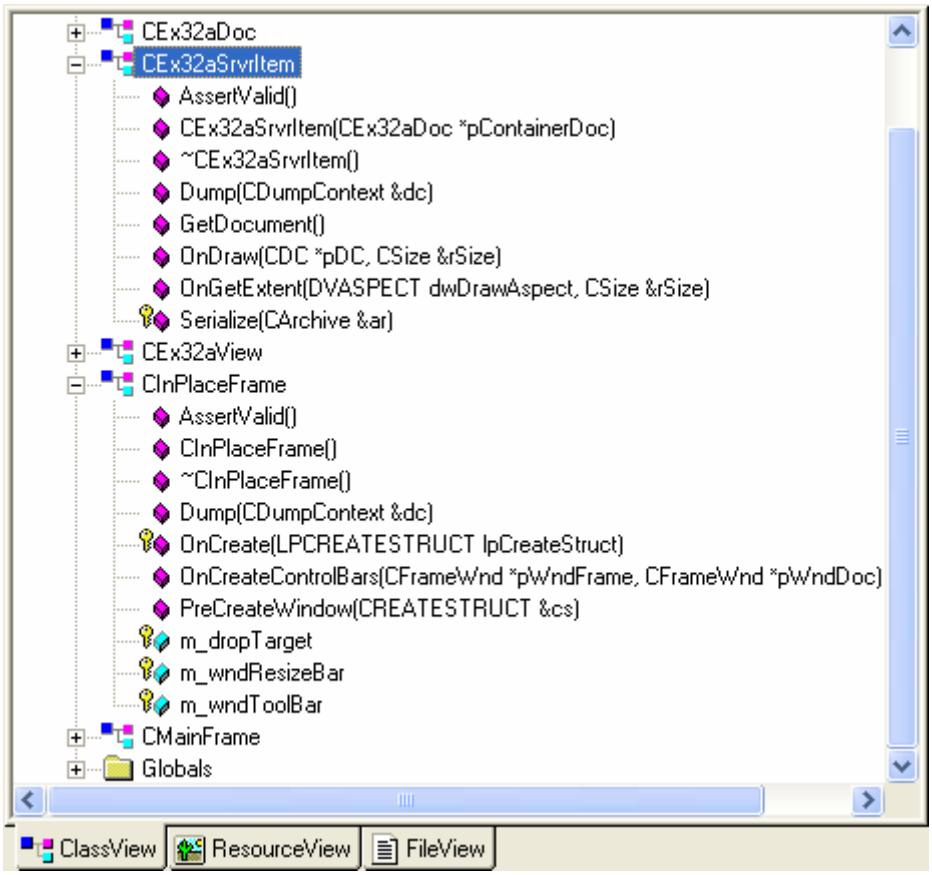

Figure 11: Generated classes for EX32A viewed through ClassView.

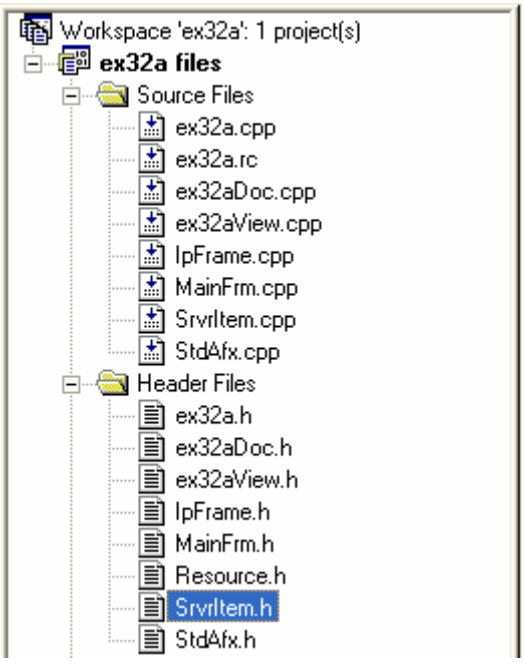

Figure 12: Generated files for EX32A seen through FileView.

Add a text member to the document class. Add the following public data member in the class declaration in **ex32aDoc.h**:

```
CString m_strText;
```

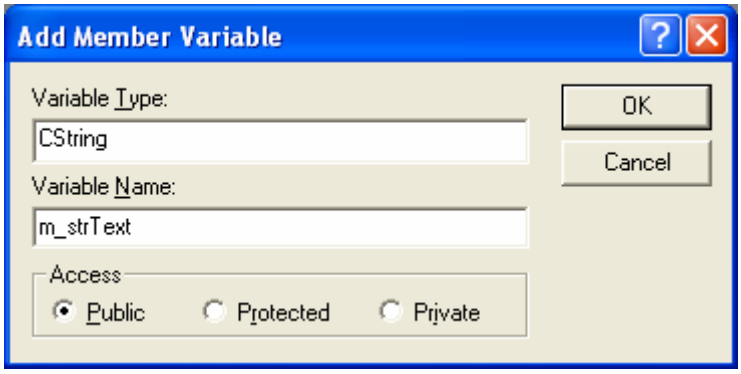

Figure 13:Adding `m_strText` member variable.

```
// Implementation
public:
    CString m_strText;
    virtual ~CEx32aDoc();
#ifdef _DEBUG
```

Listing 1.

Set the string's initial value to "`Initial default text`" in the document's `OnNewDocument()` member function.

```
m_strText = "Initial default text";
```

```
BOOL CEx32aDoc::OnNewDocument()
{
    m_strText = "Initial default text";

    if (!COleServerDoc::OnNewDocument())
        return FALSE;
```

Listing 2.

Add a dialog to modify the text. Insert a new dialog template with an edit control, and then use ClassWizard to generate a `CTextDialog` class derived from `CDialog`. Just use the default IDs.

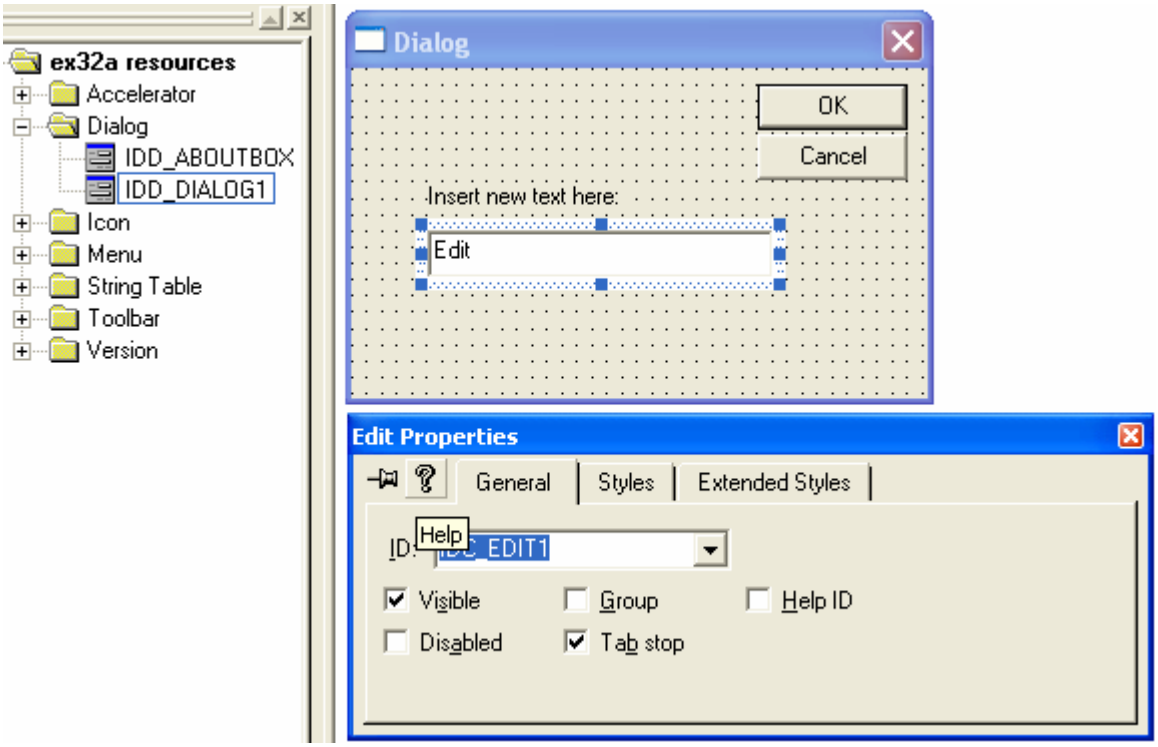

Figure 14: Adding dialog and Edit control.

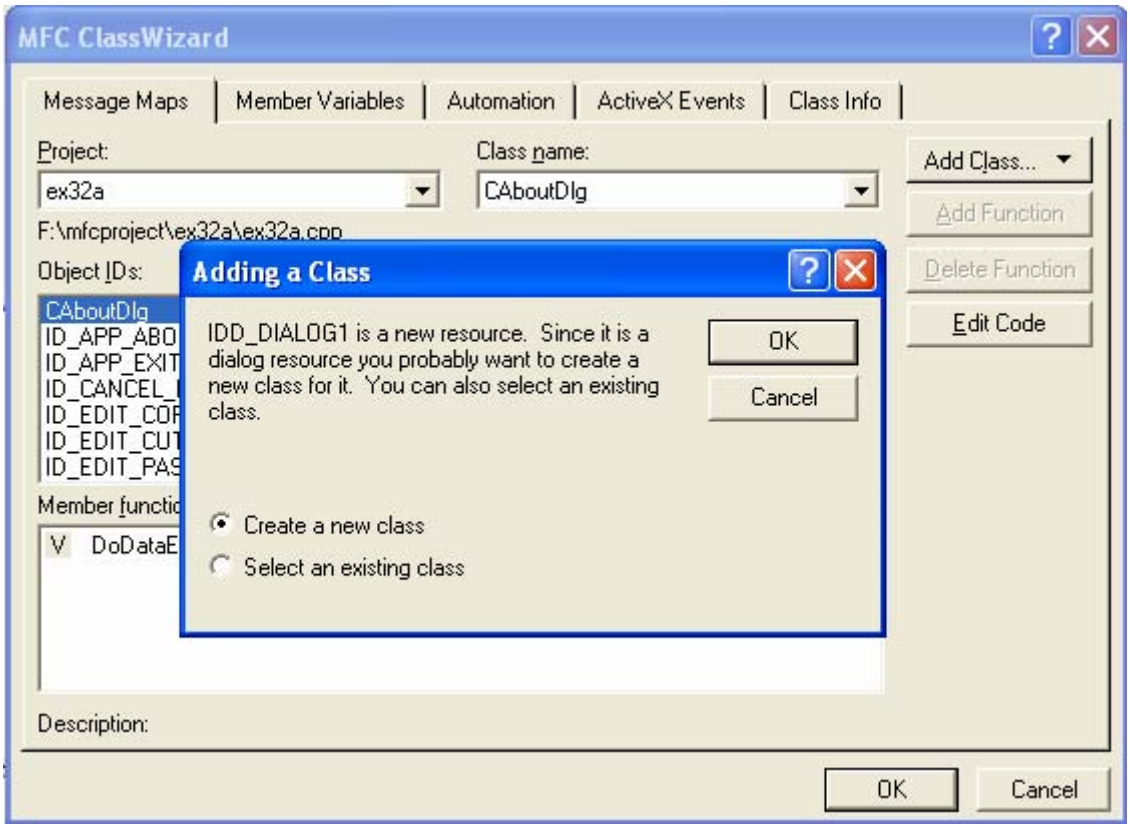

Figure 15: Adding new class dialog prompt.

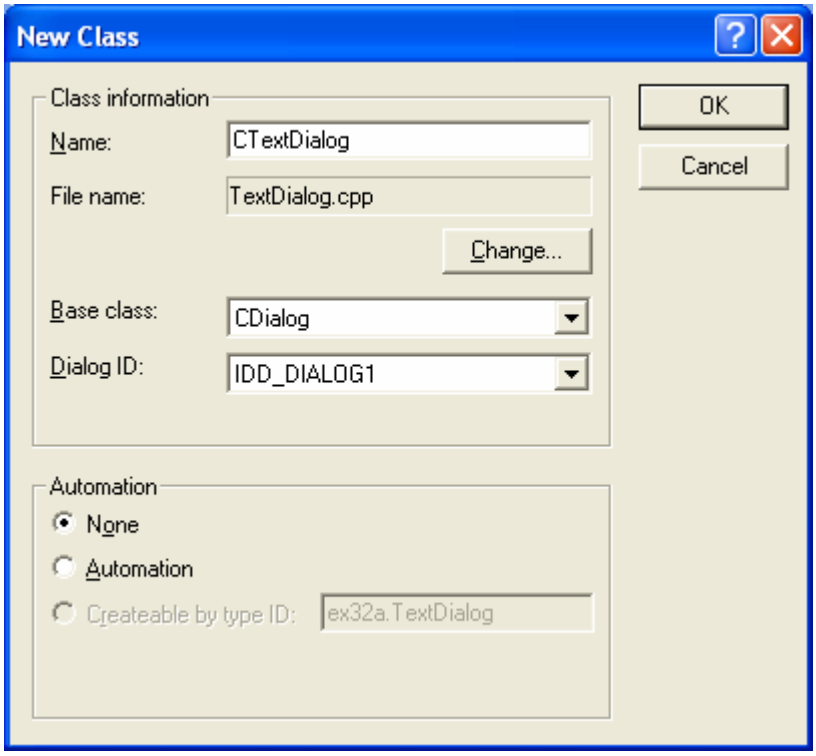

Figure 16: Adding `CTextDialog` class and its information.

Don't forget to include the dialog class header in **ex32aDoc.cpp**.

```
#include "stdafx.h"
#include "ex32a.h"

#include "ex32aDoc.h"
#include "SrvrItem.h"
#include "TextDialog.h"

#ifdef _DEBUG
```

Listing 3.

Also, use ClassWizard to add a `CString` member variable named `m_strText` for the edit control.

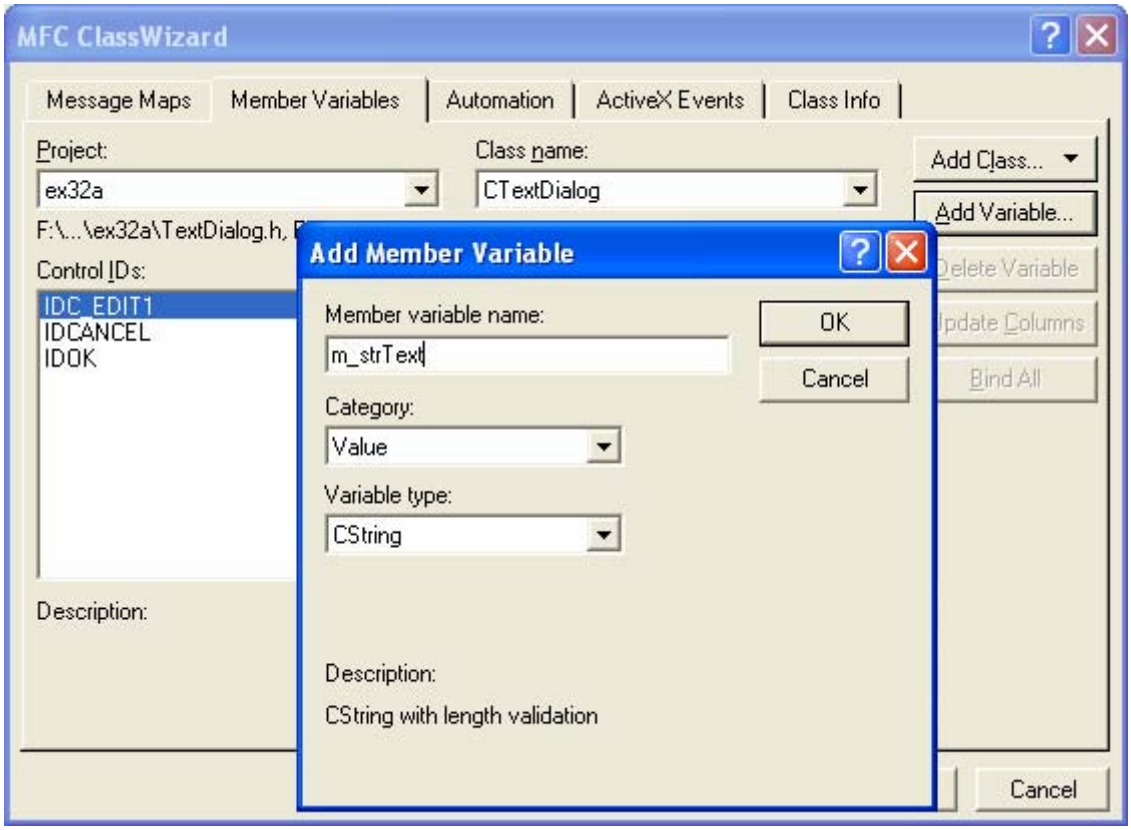

Figure 17: Adding `m_strText` member variable to `IDC_EDIT1` of `CTextDialog` class.

Add a new menu command in both the **embedded** and **in-place** menus. Add a **Modify** menu command in both the `IDR_SRVR_EMBEDDED` and `IDR_SRVR_INPLACE` menus. To insert this menu command on the `IDR_SRVR_EMBEDDED` menu, use the resource editor to add an `EX32A-EMBED` menu item on the top level, and then add a **Modify** option on the submenu for this item. Next add an `EX32A-INPLACE` menu item on the top level of the `IDR_SRVR_INPLACE` menu and add a **Modify** option on the `EX32A-INPLACE` submenu.
To associate both **Modify** options with one `OnModify()` function; use `ID_MODIFY` as the `ID` for the **Modify** option of both the `IDR_SRVR_EMBEDDED` and `IDR_SRVR_INPLACE` menus. Then use ClassWizard to map both **Modify** options to the `OnModify()` function in the **document** class.

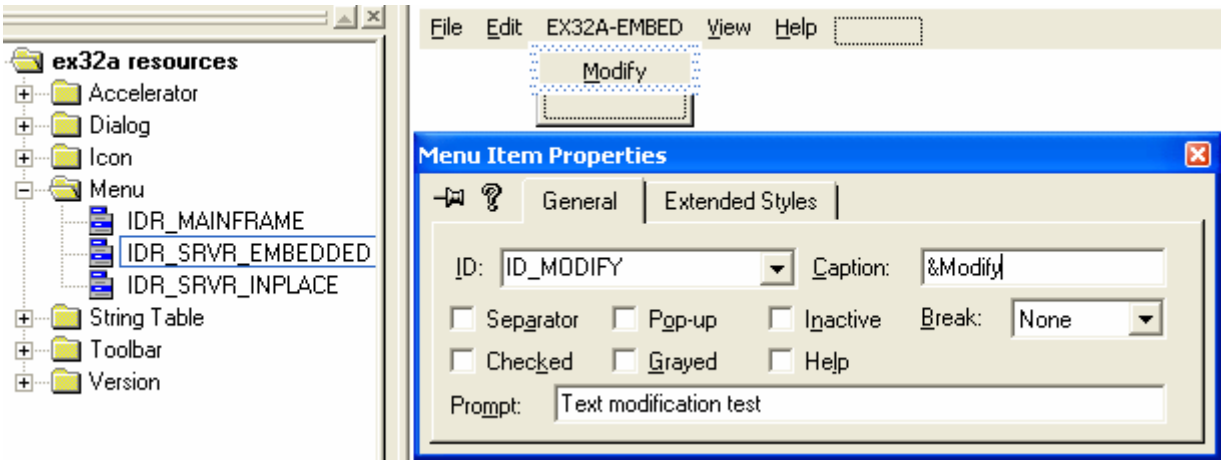

Figure 18: Adding new menu and its item to `IDR_SRVR_EMBEDDED`.

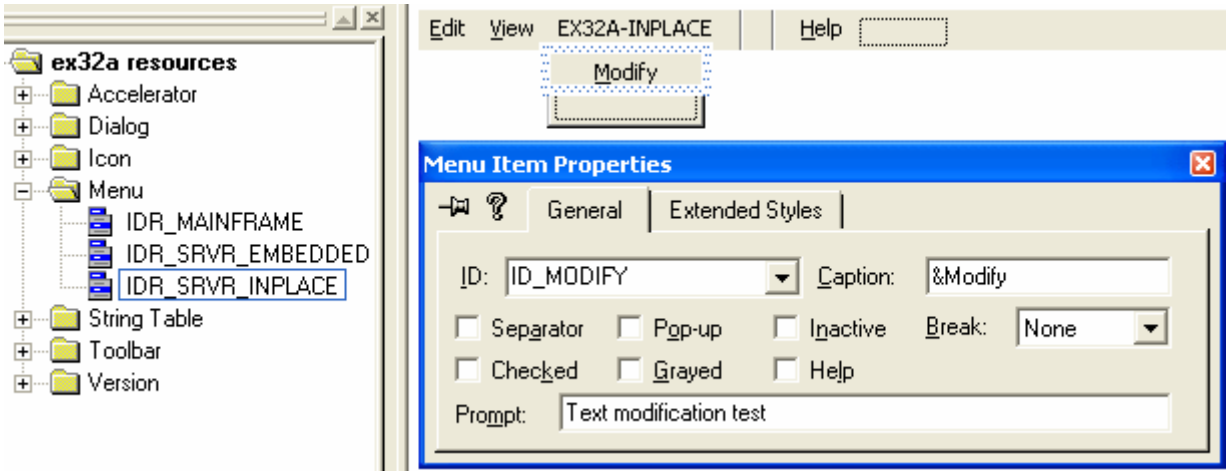

Figure 19: Adding new menu and its item to `IDR_SRVR_INPLACE`.

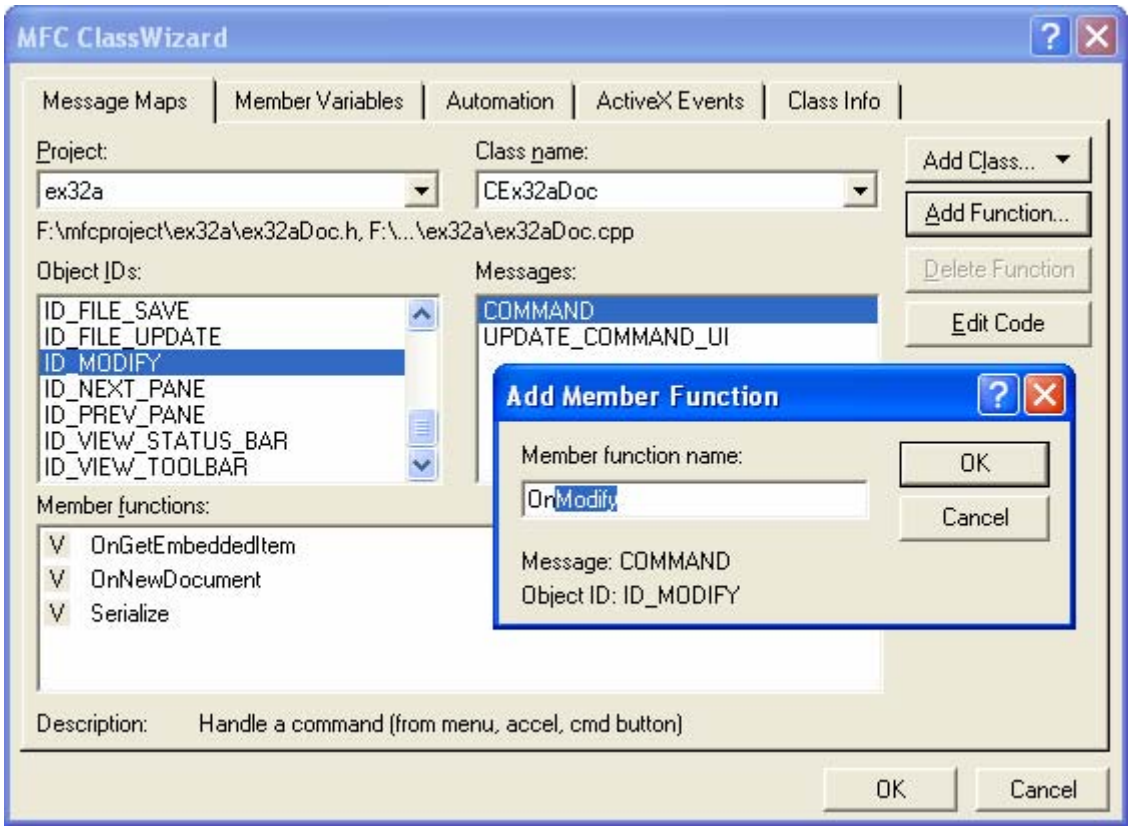

Figure 20: Adding command handler to `ID_MODIFY` of `CEx32aDoc` class.

Code the `ID_MODIFY` command handler as shown here:

```
void CEx32aDoc::OnModify()
{
    // TODO: Add your command handler code here
    CTextDialog dlg;
    dlg.m_strText = m_strText;
    if (dlg.DoModal() == IDOK)
    {
        m_strText = dlg.m_strText;
        // Trigger CEx32aView::OnDraw
        UpdateAllViews(NULL);
        // Trigger CEx32aSrvrItem::OnDraw
        UpdateAllItems(NULL);
        SetModifiedFlag();
    }
}
```

```
// CEx32aDoc commands

void CEx32aDoc::OnModify()
{
    // TODO: Add your command handler code here
    CTextDialog dlg;
    dlg.m_strText = m_strText;
    if (dlg.DoModal() == IDOK) {
        m_strText = dlg.m_strText;
        UpdateAllViews(NULL); // Trigger CEx32aView::OnDraw
        UpdateAllItems(NULL); // Trigger CEx32aSrvrItem::OnDraw
        SetModifiedFlag();
    }
}
```

Listing 4.

Override the view's `OnPrepareDC()` function. Use ClassWizard to generate the function.

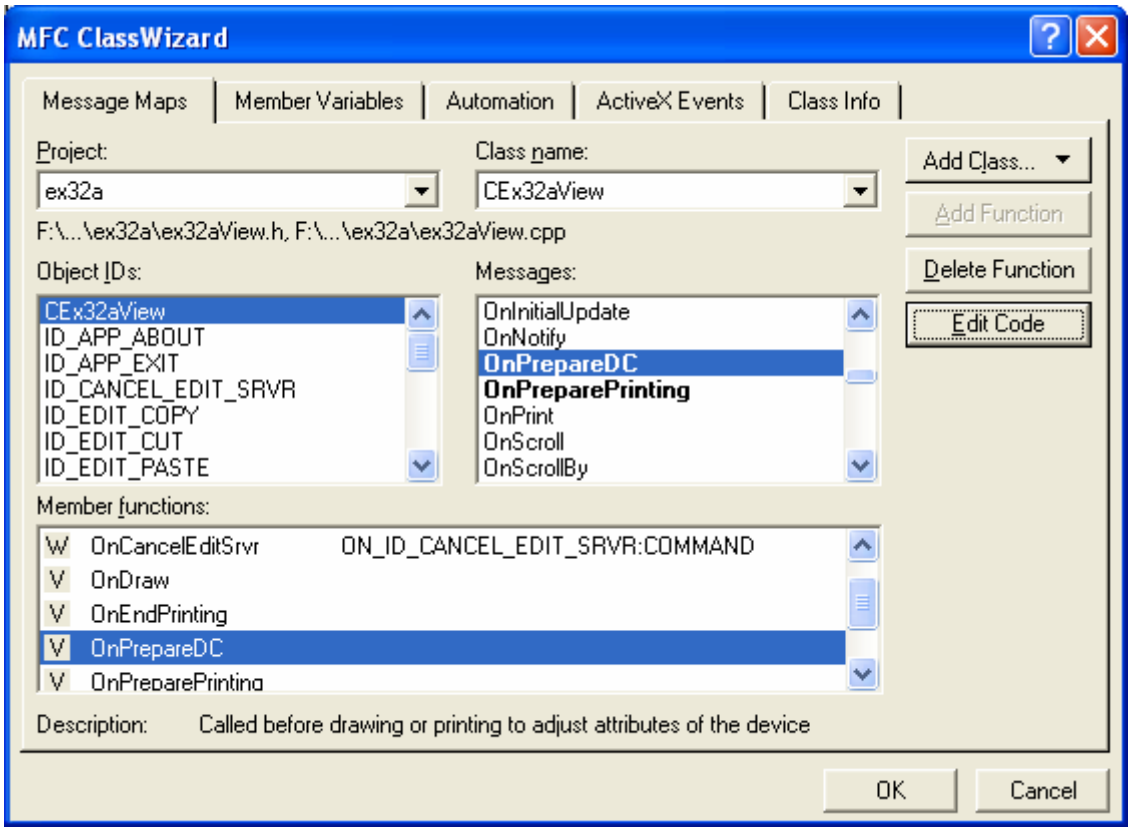

Figure 21: Overriding the view's `OnPrepareDC()` function.

Then replace any existing code with the following line.

```
pDC->SetMapMode(MM_HIMETRIC);
```

```
// CEx32aView message handlers

void CEx32aView::OnPrepareDC(CDC* pDC, CPrintInfo* pInfo)
{
    // TODO: Add your specialized code here and/or call the base class
    pDC->SetMapMode(MM_HIMETRIC);
}
```

Listing 5.

Edit the **view**'s `OnDraw()` function. The following code in **ex32aView.cpp** draws a 2-cm circle centered in the client rectangle, with the text wordwrapped in the window.

```
void CEx32aView::OnDraw(CDC* pDC)
{
    CEx32aDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    CFont font;
    font.CreateFont(-500, 0, 0, 0, 400, FALSE, FALSE, 0,
                    ANSI_CHARSET, OUT_DEFAULT_PRECIS,
                    CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
                    DEFAULT_PITCH | FF_SWISS, "Arial");
```

```
    CFont* pFont = pDC->SelectObject(&font);
    CRect rectClient;
    GetClientRect(rectClient);
    CSize sizeClient = rectClient.Size();
    pDC->DPtoHIMETRIC(&sizeClient);
    CRect rectEllipse(sizeClient.cx / 2 - 1000,
                     -sizeClient.cy / 2 + 1000,
                      sizeClient.cx / 2 + 1000,
                     -sizeClient.cy / 2 - 1000);
    pDC->Ellipse(rectEllipse);
    pDC->TextOut(0, 0, pDoc->m_strText);
    pDC->SelectObject(pFont);
}
```

```
// CEx32aView drawing

void CEx32aView::OnDraw(CDC* pDC)
{
    CEx32aDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    CFont font;
    font.CreateFont(-500, 0, 0, 0, 400, FALSE, FALSE, 0,
                    ANSI_CHARSET, OUT_DEFAULT_PRECIS,
                    CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
                    DEFAULT_PITCH | FF_SWISS, "Arial");
    CFont* pFont = pDC->SelectObject(&font);
    CRect rectClient;
    GetClientRect(rectClient);
    CSize sizeClient = rectClient.Size();
    pDC->DPtoHIMETRIC(&sizeClient);
    CRect rectEllipse(sizeClient.cx / 2 - 1000,
                     -sizeClient.cy / 2 + 1000,
                      sizeClient.cx / 2 + 1000,
                     -sizeClient.cy / 2 - 1000);
    pDC->Ellipse(rectEllipse);
    pDC->TextOut(0, 0, pDoc->m_strText);
    pDC->SelectObject(pFont);
}
```

Listing 6.

Edit the server **item**'s `OnDraw()` function. The following code in the **SrvrItem.cpp** file tries to draw the same circle drawn in the view's `OnDraw()` function. You'll learn what a server item is shortly.

```
BOOL CEx32aSrvrItem::OnDraw(CDC* pDC, CSize& rSize)
{
    // Remove this if you use rSize
    UNREFERENCED_PARAMETER(rSize);

    CEx32aDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    // TODO: set mapping mode and extent
    //  (The extent is usually the same as the size returned from
    //  OnGetExtent)
    pDC->SetMapMode(MM_ANISOTROPIC);
    pDC->SetWindowOrg(0, 0);
    pDC->SetWindowExt(3000, -3000);

    CFont font;
    font.CreateFont(-500, 0, 0, 0, 400, FALSE, FALSE, 0,
                    ANSI_CHARSET, OUT_DEFAULT_PRECIS,
                    CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
                    DEFAULT_PITCH | FF_SWISS, "Arial");
```

```
    CFont* pFont = pDC->SelectObject(&font);
    CRect rectEllipse(CRect(500, -500, 2500, -2500));
    pDC->Ellipse(rectEllipse);
    pDC->TextOut(0, 0, pDoc->m_strText);
    pDC->SelectObject(pFont);

    return TRUE;
}
```

```
BOOL CEx32aSrvrItem::OnDraw(CDC* pDC, CSize& rSize)
{
    // Remove this if you use rSize
    UNREFERENCED_PARAMETER(rSize);

    CEx32aDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    // TODO: set mapping mode and extent
    //  (The extent is usually the same as the size returned from
    //  OnGetExtent)
    pDC->SetMapMode(MM_ANISOTROPIC);
    pDC->SetWindowOrg(0, 0);
    pDC->SetWindowExt(3000, -3000);

    CFont font;
    font.CreateFont(-500, 0, 0, 0, 400, FALSE, FALSE, 0,
                    ANSI_CHARSET, OUT_DEFAULT_PRECIS,
                    CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
                    DEFAULT_PITCH | FF_SWISS, "Arial");
    CFont* pFont = pDC->SelectObject(&font);
    CRect rectEllipse(CRect(500, -500, 2500, -2500));
    pDC->Ellipse(rectEllipse);
    pDC->TextOut(0, 0, pDoc->m_strText);
    pDC->SelectObject(pFont);

    return TRUE;
}
```

Listing 7.

Edit the **document**'s `Serialize()` function. The framework takes care of loading and saving the document's data from and to an OLE stream named `Contents` that is attached to the object's main storage. You simply write normal serialization code, as shown here:

```
void CEx32aDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        ar << m_strText;
    }
    else
    {
        ar >> m_strText;
    }
}
```

```
// CEx32aDoc serialization

void CEx32aDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        ar << m_strText;
    }
    else
    {
        ar >> m_strText;
    }
}
```

Listing 8.

There is also a `CEx32aSrvrItem::Serialize` function that delegates to the document `Serialize()` function.

Build and register the EX32A application. If the program is run standalone, the following prompt will be displayed.

Figure 22: EX32A program output.

You must run the application directly once to update the Registry.
Test the EX32A application. You need a container program that supports in-place activation. Use Microsoft Excel or a later version. Choose the container's **Insert Object** menu item. If this option does not appear on the **Insert** menu, it might appear on the **Edit** menu instead. The steps are shown below.

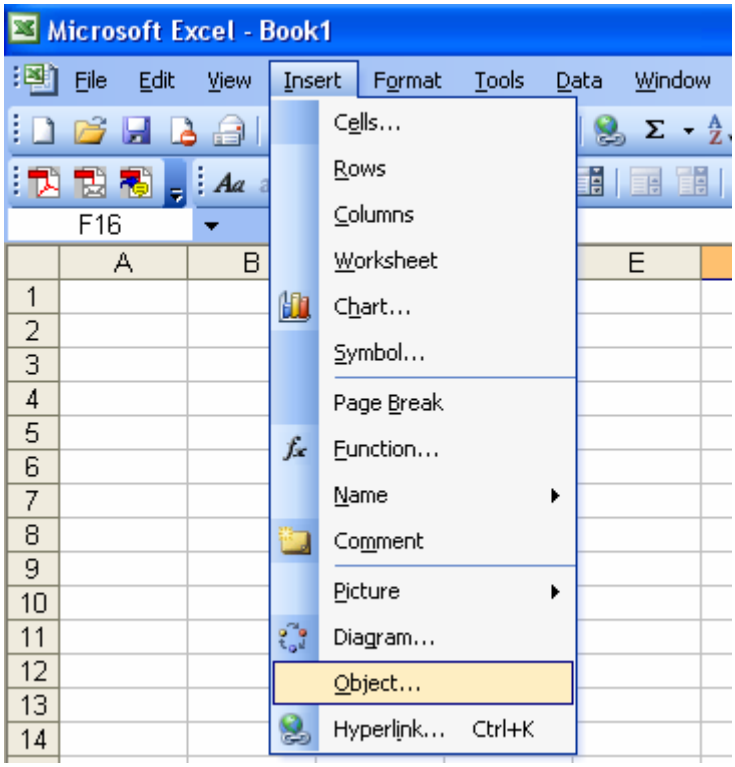

Figure 23: Inserting EX32A object into Excel worksheet.

Then select **Ex32a Document** from the list. Click the **OK** button.

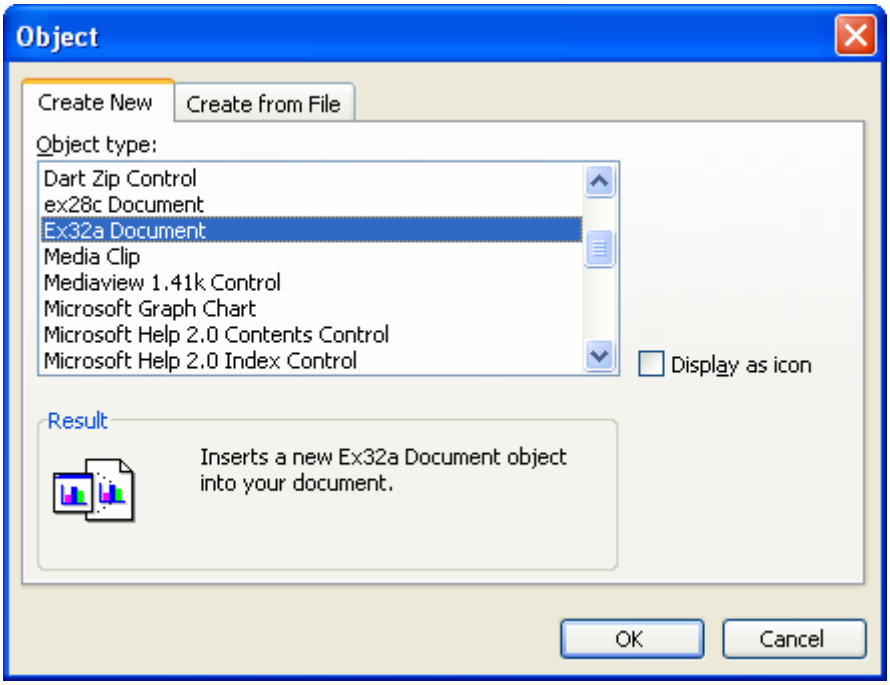

Figure 24: Selecting **Ex32a Document** object.

You debug an embedded component the same way you debug an Automation EXE component. See the sidebar, "Debugging an EXE Component Program", for more information.
When you first insert the EX32A object, you'll see a **hatched border**, which indicates that the object is **in-place** active. The bounding rectangle is 3-by-3-cm square, with a 2-cm circle in the center, as illustrated here. Notice that the component's `IDR_SRVR_INPLACE` menu is visible.

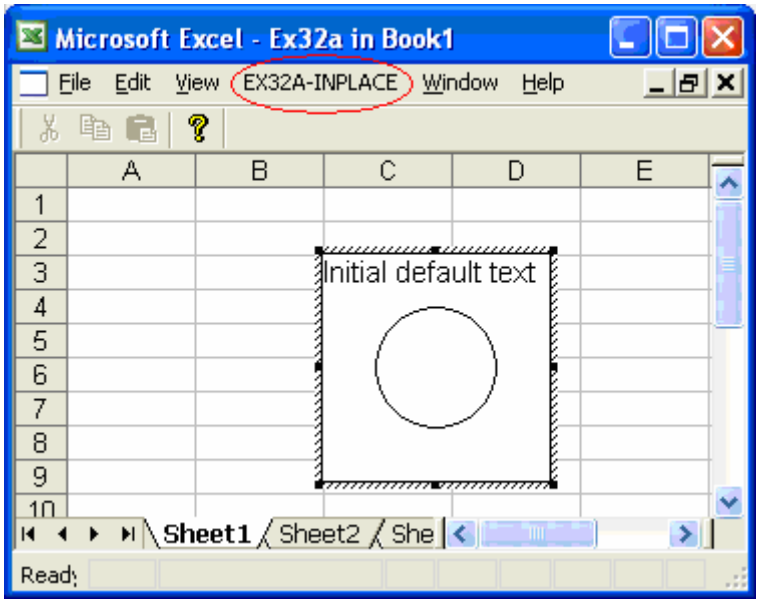

Figure 25: EX32A object in Excel worksheet.

If you click elsewhere in the container's window (Excel worksheet), the object becomes inactive, and it's shown like this.

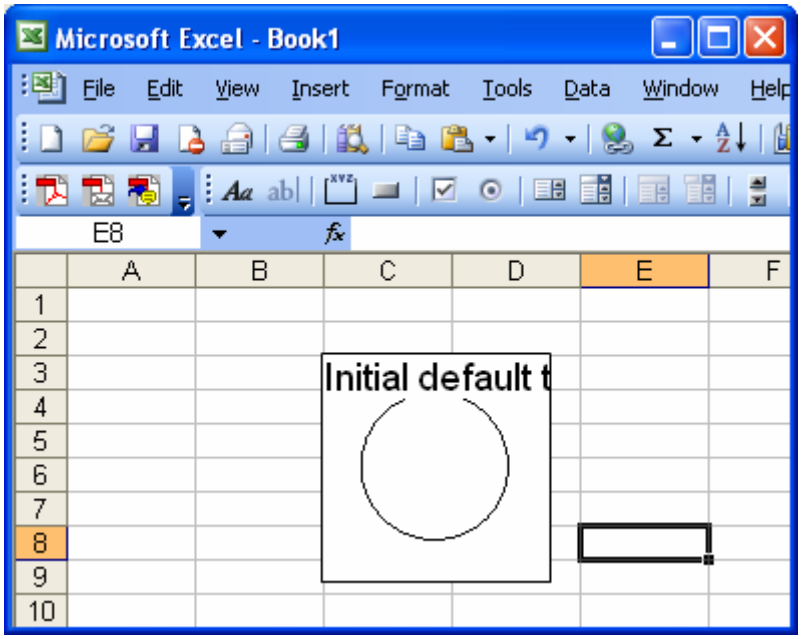

Figure 26: The EX32A object in inactive mode.

In the first case, you saw the output of the view's `OnDraw()` function; in the second case, you saw the output of the server item's `OnDraw()` function. The circles are the same, but the text is formatted differently because the server (component) item code is drawing on a metafile device context.
If you use the resize handles to extend the height of the object (click once on the object to see the resize handles; don't double-click), you'll stretch the circle and the font will get bigger, as shown below in the first figure. If you reactivate the object by double-clicking on it, it's reformatted as shown in the second figure that follows.

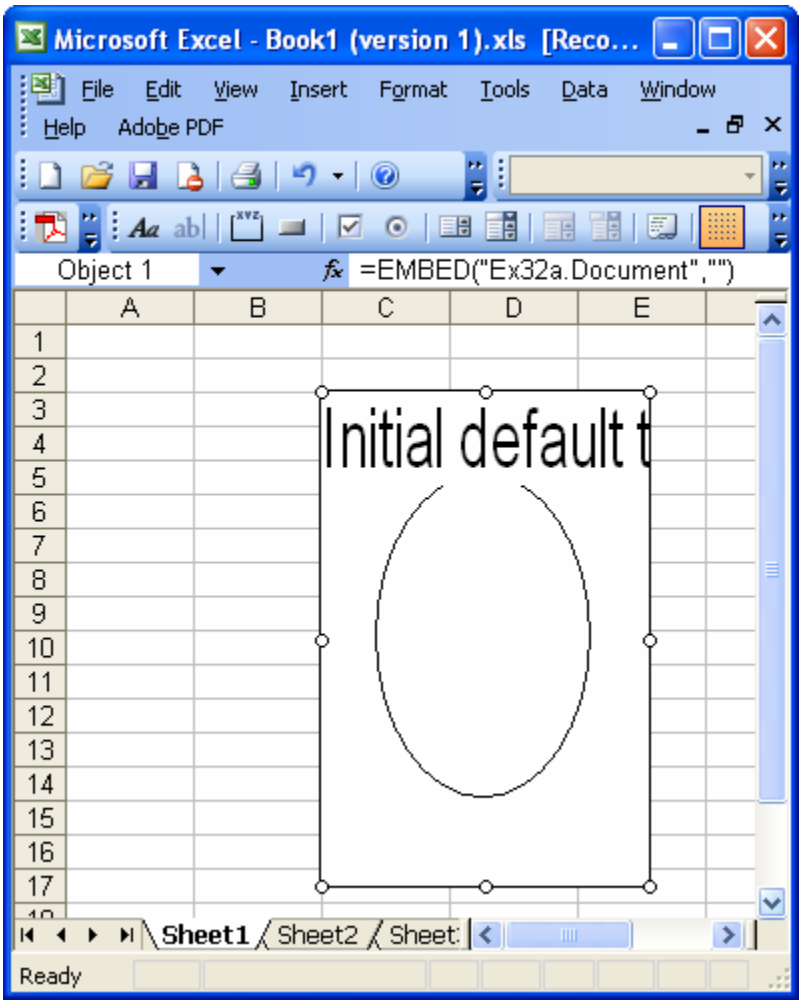

Figure 27: Resizing the object.

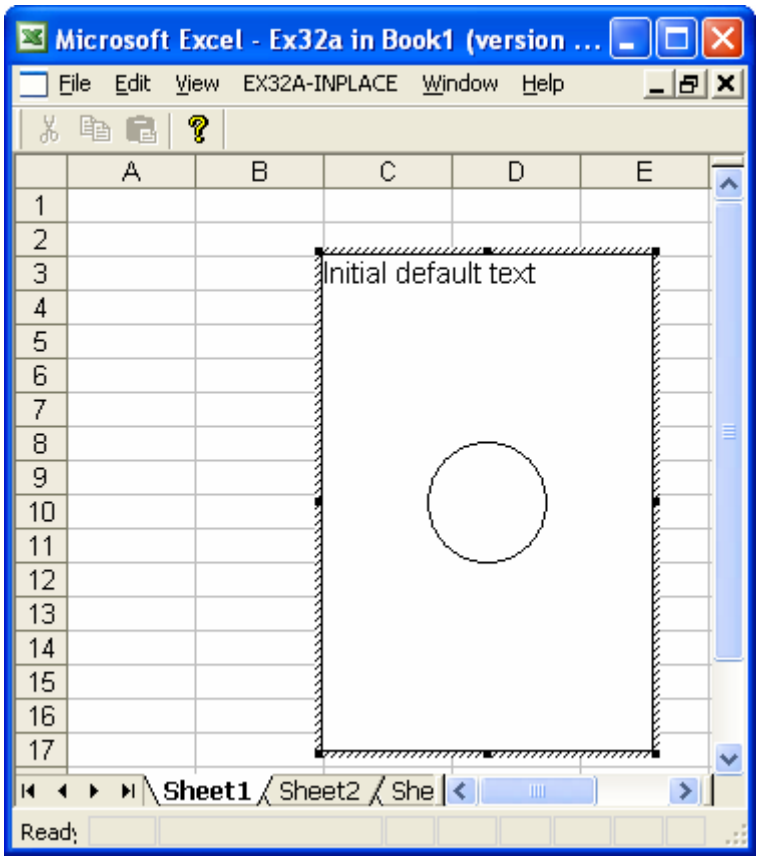

Figure 28: Reactivate the object after resizing it.

Click elsewhere in the container's window, single-right-click on the object, and then choose **Ex32a Object** from the menu. Choose **Open** from the submenu. This starts the component program in **embedded mode** rather than in **in-place** mode, as shown here.

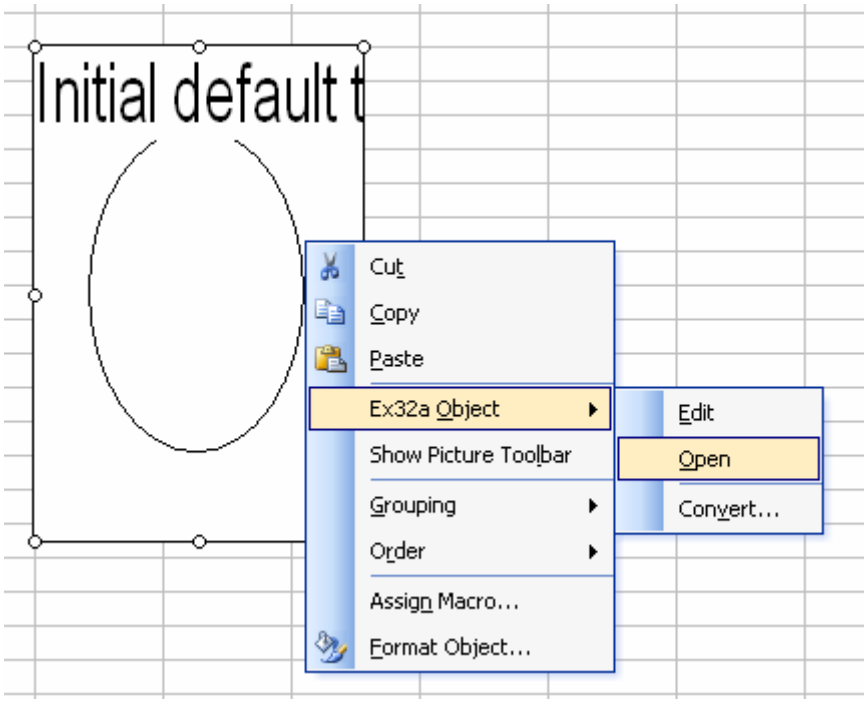

Figure 29: Opening the object in an embedded mode.

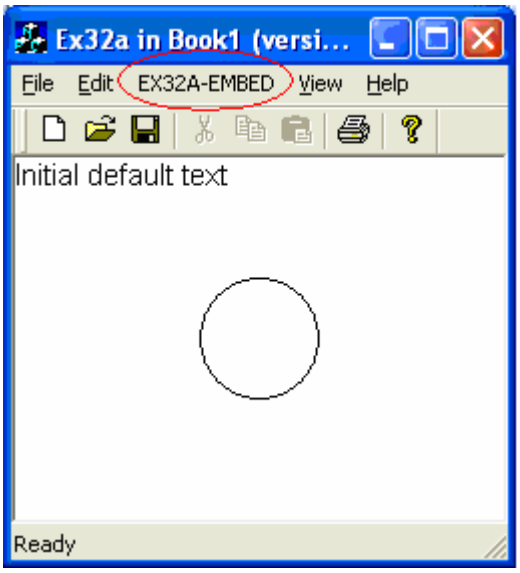

Figure 30: Object in embedded mode. Notice the top menu.

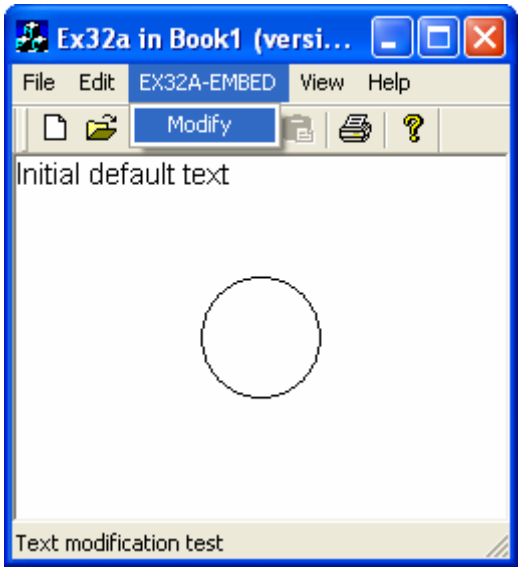

Figure 31: Using **Modify** menu in embedded mode.

Figure 32: Modifying the text.

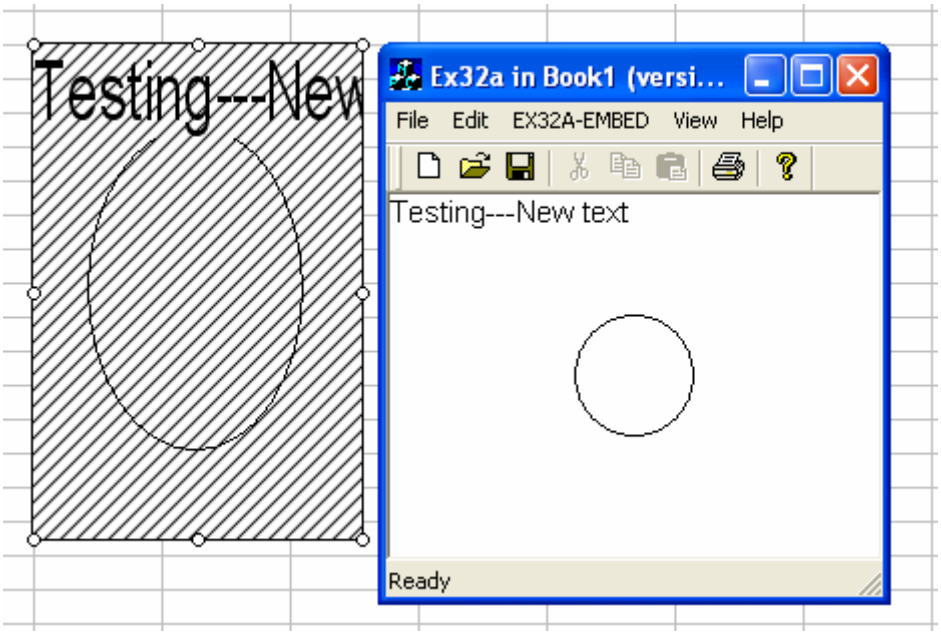

Figure 33: New text in action.

Notice that the component's `IDR_SRVR_EMBEDDED` menu is visible.

Continue on next module...part 2

----------End part 1----------

Further reading and digging:

1. MSDN MFC 6.0 class library online documentation - used throughout this Tutorial.
2. MSDN MFC 7.0 class library online documentation - used in .Net framework and also backward compatible with 6.0 class library
3. MSDN Library
4. DCOM at MSDN.
5. COM+ at MSDN.
6. COM at MSDN.
7. Windows data type.
8. Win32 programming Tutorial.
9. The best of C/C++, MFC, Windows and other related books.
10. Unicode and Multibyte character set: Story and program examples.