



Microsoft

# ADSI 2.5

*Active Directory Service Interfaces Beta*

## ADSI Extension

---

### **Abstract**

ADSI Extension allows ISVs and corporate developers to extend interfaces and methods in directory objects.

For more information about the capabilities of ADSI Extension, see the ADSI Extension White Paper located at <http://www.microsoft.com/adsi>.

© 1998 Microsoft Corporation. All rights reserved.

*THIS IS PRELIMINARY DOCUMENTATION. The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.*

*This BETA document is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS DOCUMENT.*

*The BackOffice logo, Microsoft, Windows, and Windows NT are registered trademarks of Microsoft Corporation.*

*Other product or company names mentioned herein may be the trademarks of their respective owners.*

*Microsoft Corporation • One Microsoft Way • Redmond, WA 98052-6399 • USA  
0X98*

---

**CONTENTS**

**INTRODUCTION.....1**

**EARLY BINDING SUPPORT.....2**

**LATE BINDING SUPPORT.....11**

**FOR MORE INFORMATION.....17**

---

## EARLY BINDING SUPPORT

The ADSI Extension model allows developers to extend an ADSI object and associate the extension with a specific class in the directory. The extension will be part of the ADSI programming model. The ADSI extension benefits both script and extension writers. For script writers or extension consumers, it eliminates the necessity to learn a new programming paradigm. For extension writers, it allows them to extend the set of methods for interfaces that are already a part of the ADSI infrastructure, which saves time during the development process by leveraging the functionality that already exists in ADSI.

To understand how ADSI extensions can simplify a development project, imagine a scenario where a corporate developer or ISV needs to develop a backup program that allows an administrator to back up all of the computers in the organizational unit that he or she manages. After an extension is added to the ADSI model, it will be possible for the administrator to use a simple script to perform the backup, as shown in the following example:

```
Dim ou as IADsContainer
Set ou = GetObject("LDAP://OU=Sales, DC=ArcadiaBay, DC=COM")
ou.Filter = Array("computer")
For each comp in ou
    Debug.Print comp.Get("dnsHostName")
    Debug.Print comp.LastBackUp
    comp.BackUpNow
Next
```

As you can see in this example, the extension writer only needs to provide **LastBackUp** and **BackUpNow** for the property and the method, in order to implement the backup process.

ADSI supports two levels of extension:

- Early Binding Support.
- Late Binding Support.

We will discuss both techniques in this walkthrough.

**Note:** The tutorial is compiled and linked under the Visual C++ 6.0 environment and uses the ADSI 2.5 Beta libs. ADSI extensions can also be created using previous versions of Visual C++.

You can download the ADSI 2.5 Beta from <http://www.microsoft.com/adsi>.

Early binding support allows Visual Basic and Visual C++ developers to access your extension. Each extension needs to be associated with an object class in the directory. You may associate one extension to many different object classes in the

---

directory.

Early binding support is the easiest form of extension to add under the ADSI extension model. In fact, you will need to add support for early binding in order to make your existing COM servers support ADSI extensions. There are two steps to adding early binding support to existing COM servers. First, you will need to provide support for aggregation. Next, you will need to set up registration for the new extension.

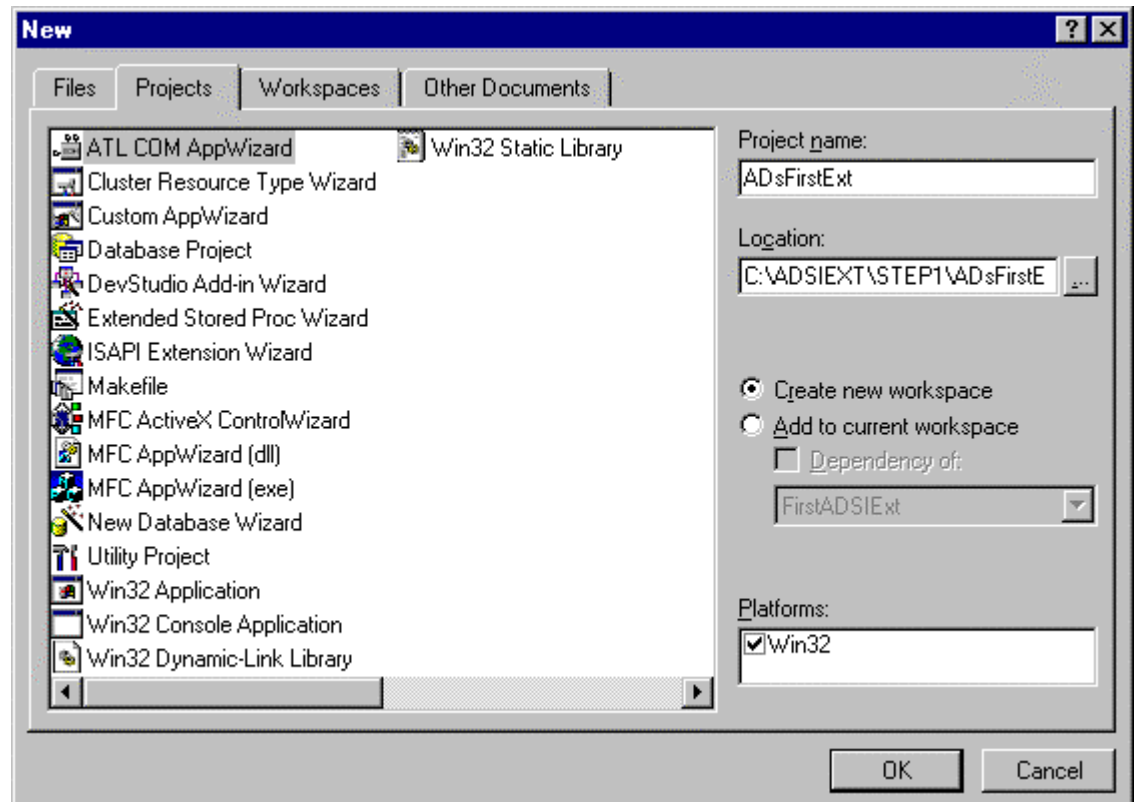
### **Adding Support for Aggregation**

To support early binding in your extension, you must first add support for aggregation. To do this, you will need to do the following:

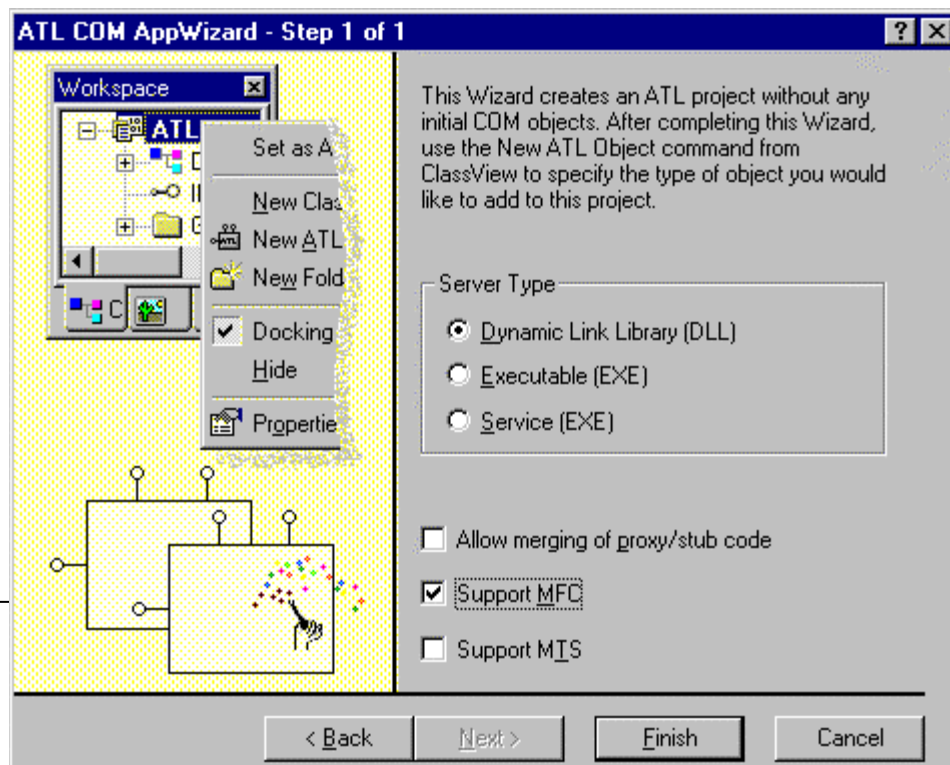
- Create a new ATL project.
- Create a new class and interface.
- Set up the project environment.
- Create a new method.
- Add method implementation information.

#### **To Create a New ATL Project**

1. Open Visual C++, select **New** from the **Project** menu. The following dialog window will be displayed.



2. Select **ATL Com AppWizard**.
3. Type *ADsFirstExt* as the project name, select your desired location.
4. Select **OK**. The following dialog box will be displayed.
5. For Server Type, select Dynamic Link Library.
6. Support MFC is optional.
7. Click Finish.



8.

### To Create a New ATL Class and Interface

1. Select **New Class** from the **Insert** menu.
2. Select **ATL Class** as the **Class Type**.
3. Enter **HelloWorld** as the **Class Information Name**.
4. Select **Dual** as the interface type.
5. Select **1** for **Number of interfaces**.
6. Select the **Aggregatable** check box.
7. Click **OK**.

**New Class**

Class type: ATL Class

Class information

Name: HelloWorld

File name: HelloWorld1.cpp

Change...

Interface type: ☒ Dual ☐ Custom

Interfaces

Number of interfaces: 1

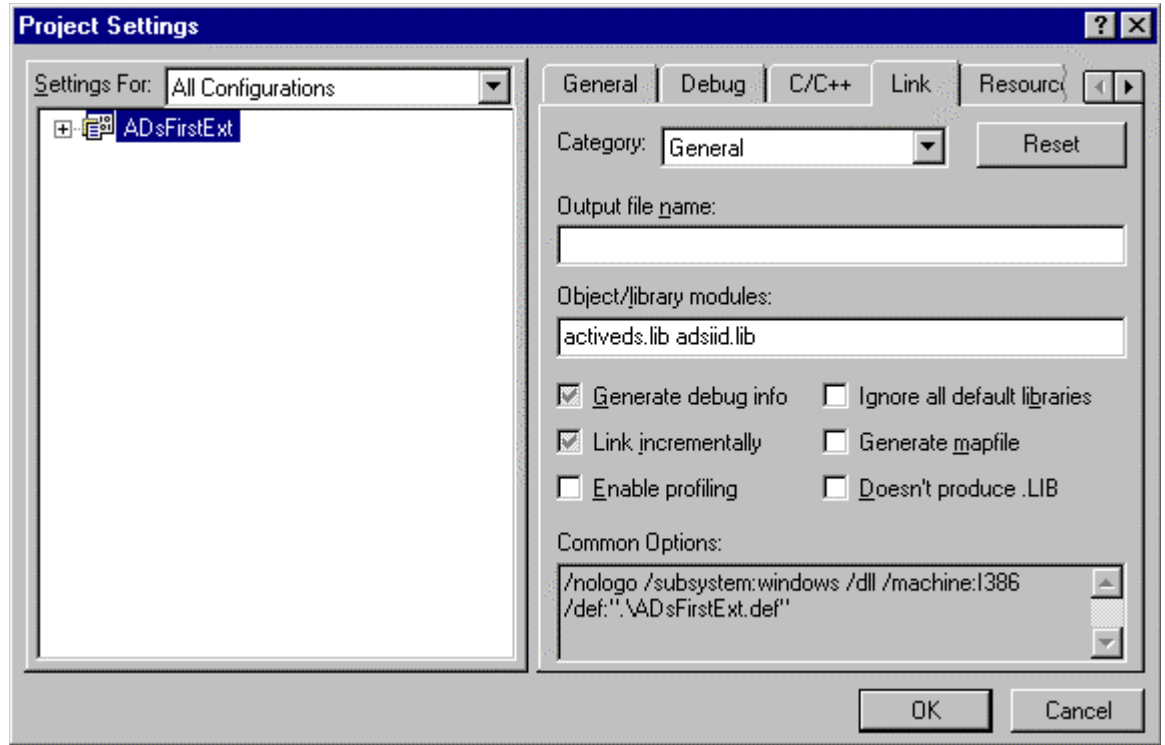
Names: IHelloWorld

Edit...

☒ Aggregatable

### To Set Up the Project Environment

1. Select **Settings...** from the **Project** menu. The **Project Settings** dialog window will be displayed.

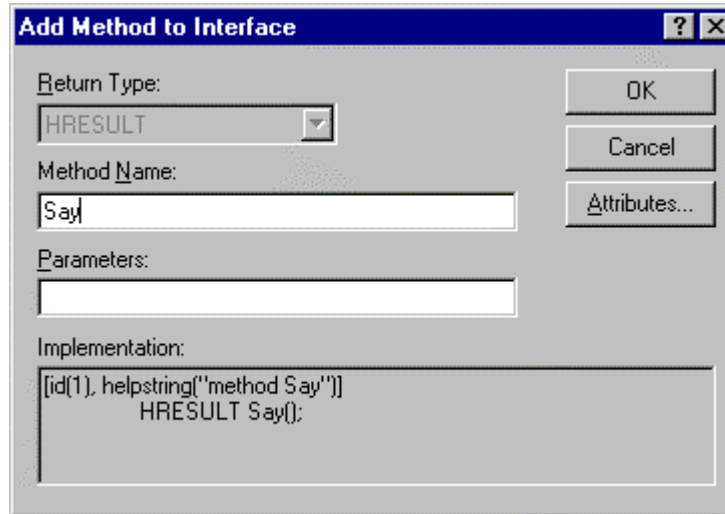


2. In the **Settings For** field, select **All Configurations**.
3. Select the **Link** tab.
4. In the **Object/library modules** field, enter *activeds.lib* and *adsiid.lib*.
5. Click **OK**.
6. Open *stdafx.h* either by selecting **Open** from the **File** menu, or using the workspace window.
7. After `#include <atlcom.h>` add the line `#include <activeds.h>`.



### To Create a New Method

1. Right-click `IHelloWorld` in the workspace window. The **Add Method to Interface** dialog window will be displayed.



2. Type **Say** in the **Method Name** field.
3. Click **OK**.

### To Add Method Implementation Information

1. Open the *HelloWorld.CPP* file.
2. Enter the following code in the **Implementation** field:

```
STDMETHODIMP HelloWorld::Say()
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState())
    IADs *pADs;
    HRESULT hr;
    BSTR bstr;
    TCHAR szText[128];

    hr = OuterQueryInterface(IID_IADs, (void **) &pADs );
    if ( !SUCCEEDED(hr) )
    {
        return hr;
    }
    hr = pADs->get_Name(&bstr);
    if ( !SUCCEEDED(hr) )
    {
        pADs->Release();
        return hr;
    }
    wsprintf(szText, _T("Hello %S"), bstr );
    SysFreeString(bstr);
    ::MessageBox(NULL, szText, _T("From ADSI Extension"), MB_OK);
    pADs->Release(); return S_OK;
}
```

---

## Adding Registration Code for the Extension

Now you will need to add information to the registrar for the extension.

### To Add Registration Information

1. Write the **RegisterADsExt** function using the following code as an example:

```
STDAPAPI RegisterADsExt(void)
{
    HRESULT hr;
    HKEY hKey;
    DWORD dwDisposition;

    ///////////////////////////////////
    // Register the class
    ///////////////////////////////////
    hr = RegCreateKeyEx( HKEY_LOCAL_MACHINE,
                        _T("SOFTWARE\
\Microsoft\ADs\Providers\LDAP\Extensions\User\{E1E3EDF8-48D1-11D2-
B22B-0000F87A6B50}"),
        0,
        NULL,
        REG_OPTION_NON_VOLATILE,
        KEY_WRITE,
        NULL,
        &hKey,
        &dwDisposition );
    ///////////////////////////////////
    // Register the Interface
    ///////////////////////////////////
    const wchar_t szIf[] = L"{E1E3EDF7-48D1-11D2-B22B-0000F87A6B50}\\0\\0";

    hr = RegSetValueEx( hKey, _T("Interfaces"), 0, REG_BINARY, (const BYTE *)
szIf, sizeof(szIf));

    RegCloseKey(hKey);
    return S_OK;
}

STDAPAPI RegisterADsExt(void);
STDAPAPI DllRegisterServer(void)
{
    // registers object, typelib and all interfaces in typelib
    if ( SUCCEEDED(_Module.RegisterServer(TRUE) ) )
    {
        return RegisterADsExt();
    }
    return E_FAIL;
}
```

**Note:** ADSI extension registration requires the REG\_MULTI\_SZ. ATL type. The registrar does not support this type in Visual C++ 6.0. However, Visual C++ 6.1 will support this registry type. We encourage you to use the ATL registrar when this support is available. For now, we will use the Registry APIs.

2. Replace the highlighted strings with the following information:

- **LDAP** You can extend other providers. In this beta, only WinNT and LDAP is extensible.

- 
- **User** You can replace this with any other objectClass, for example Computer, PrintQueue, etc.
  - **CLSID { xxxxx }** You must replace this parameter with your own clsid. See your .RGS file to find the clsid.
  - **Interface ID {xxxxx}** You must replace this with your own interface ID. See the .IDL file for this ID number.
3. Call the **RegisterADsExt** function from the **DllRegisterServer** function.
  4. Optionally, you can create an implementation of **UnRegisterADsExt**, and add it to the **DllUnregisterServer** function.

### Compiling Your Program

Now that you have created all the essential elements, you can compile your program using the following step:

On the **Build** menu, select **Rebuild All**.

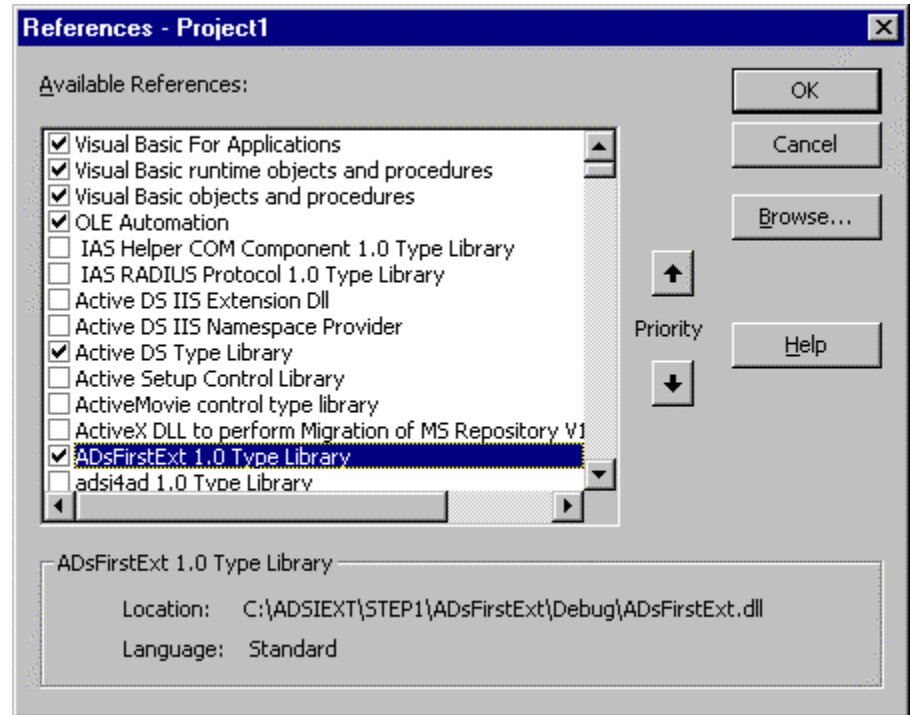
Your extension is now ready for testing.

### Testing your Extension

The easiest way to test your extension is to write a simple VB program:

1. Launch Visual Basic.
2. Select **Standard Project (.EXE)**.
3. On the **References - Project** dialog window, select the following **Available References** check boxes:
  - **Active DS Type Library**
  - **AdsFirstExt 1.0 Type Library**

## LATE BINDING SUPPORT



4. Double-click the form and add the following code:

```
Private Sub Form_Load()  
Dim ext As HelloWorld  
Dim usr As IADsUser  
  
Set usr = GetObject("LDAP://CN=Guest,CN=Users,DC=YourDomain,,DC=com")  
Debug.Print Usr.Name  
Set ext = usr  
ext.Say  
  
End Sub
```

4. The following dialog window should appear.



Next, you will need to add support for late binding. If your extension supports late binding, it can be accessed using late binding automation clients, such as Windows Script Host™, VBS, Active Server Pages, Visual Basic and Visual C++.

The following example shows late binding code for accessing your extension:

```
Set usr = GetObject("LDAP://CN=Guest,CN=Users,DC=YourDomain,,DC=com")  
Debug.Print Usr.Name  
Usr.Say
```

To support late binding in your extension, you must also support the

---

**IADsExtension** interface, in addition to all of the requirements needed for early binding. Step 2 folder in your ADSI Extension sample tutorial supports late binding.

### To Reroute the Internal IDispatch

The extension needs to reroute its **IDispatch** implementation to the aggregator (that is, the ADSI objectmanager ), and let ADSI make an invocation decision. ADSI may invoke itself, another extension, or your extension, depending on the algorithm.

1. Comment out **IDispatchImpl**, as shown in the following example:

```
//public IDispatchImpl<IHelloWorld, &IID_IHelloWorld,  
//&LIBID_ADSFIRSTEXTLib>
```

2. Add the following string:

```
public IHelloWorld,
```

3. To implement the **IDispatch** portion of **IHelloWorld**, you will need to declare them as shown in the following example:

```
// IDispatch  
STDMETHOD(GetTypeInfoCount)(UINT* pctinfo);  
STDMETHOD(GetTypeInfo)(UINT itinfo, LCID lcid, ITypeInfo** pptinfo);  
STDMETHOD(GetIDsOfNames)(REFIID riid, LPOLESTR* rgpszNames, UINT cNames,  
LCID lcid, DISPID* rgdispid);  
STDMETHOD(Invoke)(DISPID dispidMember, REFIID riid,  
LCID lcid, WORD wFlags, DISPPARAMS* pdispparams, VARIANT* pvarResult,  
EXCEPINFO* pexcepinfo, UINT* puArgErr);
```

4. Do the following to make the implementation:

```

////////////////////////////////////
// Delegating IDispatch Methods to the aggregator
////////////////////////////////////
STDMETHODIMP HelloWorld::GetTypeInfoCount(UINT* pctinfo)
{
    IDispatch *pDisp;
    HRESULT hr;
    hr = OuterQueryInterface( IID_IDispatch, (void**) &pDisp );
    if ( SUCCEEDED(hr) )
    {
        hr = pDisp->GetTypeInfoCount( pctinfo );
        pDisp->Release();
    }
    return hr;
}

STDMETHODIMP HelloWorld::GetTypeInfo(UINT itinfo, LCID lcid, ITypeInfo**
pptinfo)
{
    IDispatch *pDisp;
    HRESULT hr;
    hr = OuterQueryInterface( IID_IDispatch, (void**) &pDisp );
    if ( SUCCEEDED(hr) )
    {
        hr = pDisp->GetTypeInfo( itinfo, lcid, pptinfo );
        pDisp->Release();
    }

    return hr;
}

STDMETHODIMP HelloWorld::GetIDsOfNames(REFIID riid, LPOLESTR* rgpszNames,
UINT cNames, LCID lcid, DISPID* rgdispid)
{
    IDispatch *pDisp;
    HRESULT hr;
    hr = OuterQueryInterface( IID_IDispatch, (void**) &pDisp );
    if ( SUCCEEDED(hr) )
    {
        hr = pDisp->GetIDsOfNames( riid, rgpszNames, cNames, lcid,
rgdispid);
        pDisp->Release();
    }

    return hr;
}

STDMETHODIMP HelloWorld::Invoke(DISPID dispidMember, REFIID riid,
LCID lcid, WORD wFlags, DISPPARAMS* pdispparams, VARIANT*
pvarResult, EXCEPINFO* pexcepinfo, UINT* puArgErr)
{
    IDispatch *pDisp;
    HRESULT hr;
    hr = OuterQueryInterface( IID_IDispatch, (void**) &pDisp );
    if ( SUCCEEDED(hr) )
    {
        hr = pDisp->Invoke( dispidMember, riid, lcid, wFlags,
pdispparams, pvarResult, pexcepinfo, puArgErr);
        pDisp->Release();
    }

    return hr;
}

////////////////////////////////////
// End delegating IDispatch Methods
////////////////////////////////////

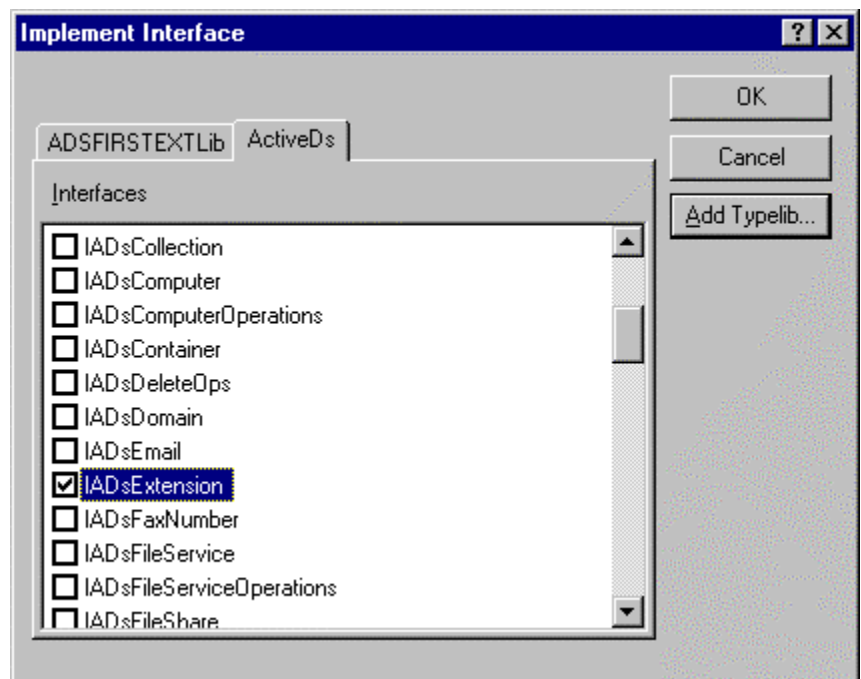
```

## Adding an IADsExtension Interface Implementation

If you use Visual C++ 6.0 or higher, you can use this procedure to add the **IADsExtension** interface implementation. Otherwise, you will need to manually type the **IADsExtension** declaration and implementation, or you can copy and paste the code from the tutorial.

## To Add IADsExtension to the Type Library

1. Right-click the **Class** object, and select **Implement Interface...** The **Implement Interface** dialog box will be displayed.
2. Click the **Add Type Library...** button.
3. Select the **Active Ds** tab.
4. Select the **IADsExtension** check box.



Th

is will comment out the import .tlb in *HelloWorld.h*, since we have included the *activeds.h* file.

```
// #import "c:\winnt\system32\activeds.tlb"
```

For developers who do not use Visual C++ 6.0, you must also do the following steps:

1. Add public **IADsExtension** as the class declaration.
2. Add COM\_INTERFACE\_ENTRY(IADsExtension) in the mapping for COM\_MAP.

### To Add the Type Info Member Variable

The type info member variable is used to hold the type information for the **>HelloWorld** interface.

1. Add the following line in the class definition:

```
protected:
    ITypeInfo    *m_pTypeInfo;
```

2. Add the constructor and destructor definitions:

```
HelloWorld();
~HelloWorld();
```

3. Implement the constructor and destructor definitions:

```
HelloWorld::HelloWorld()
{
    HRESULT hr;
    ITypeLib *pITypeLib;
    m_pTypeInfo = NULL;
    hr = LoadRegTypeLib( LIBID_ADSFIRSTTEXTLib,
                        1,
                        0,
                        PRIMARYLANGID(GetSystemDefaultLCID()),
                        &pITypeLib );

    if ( SUCCEEDED(hr) )
    {
        hr = pITypeLib->GetTypeInfoOfGuid( IID_IHelloWorld, &m_pTypeInfo );
    }

    pITypeLib->Release();
}
HelloWorld::~HelloWorld()
{
    if ( m_pTypeInfo )
    {
        m_pTypeInfo->Release();
    }
}
```

### To Add the IADsExtension Interface Implementation

1. Go to **IADsExtension** interface implementations in the *HelloWorld.h* file.
2. Add the following code:



---

```

STDMETHOD(Operate)(ULONG dwCode, VARIANT varData1, VARIANT varData2,
VARIANT varData3)
{
    HRESULT hr = S_OK;
    switch (dwCode)
    {
        case ADS_EXT_INITCREDENTIALS:
            // you can prompt for a credential if you like
            // MessageBox(NULL, "INITCRED", "ADsExt", MB_OK);
            break;
        default:
            hr = E_FAIL;
            break;
    }
    return hr;
}

STDMETHOD(PrivateGetIDsOfNames)(REFIID riid, OLECHAR ** rgpszNames,
unsigned int cNames, LCID lcid, DISPID * rgdispid)
{
    if (rgdispid == NULL)
    {
        return E_POINTER;
    }
    return DispGetIDsOfNames(m_pTypeInfo, rgpszNames, cNames,
rgdispid);
}

STDMETHOD(PrivateInvoke)(DISPID dispidMember, REFIID riid, LCID lcid, WORD
wFlags, DISPPARAMS * pdispparams, VARIANT * pvarResult, EXCEPINFO *
pexcepthinfo, UINT * puArgErr)
{
    return DispInvoke( (IHelloWorld*)this,
        m_pTypeInfo,
        dispidMember,
        wFlags,
        pdispparams,
        pvarResult,
        pexcepthinfo,
        puArgErr );
}

```

3. Recompile the file.

---

## FOR MORE INFORMATION

### Testing the Late Binding

To make things interesting, try testing the component using VB Script (although you could also test it using a Visual Basic sample). Use the following *test.vbs* script:

```
Set usr = GetObject("LDAP://CN=Guest,CN=Users,DC=actived,dc=nttest,  
dc=microsoft,dc=com")  
wscript.echo Usr.Name  
Usr.Say  
wscript.echo User.Get "samAccountName"
```

To run the test, type **test** at the command prompt. The following dialog window should appear (the same as it did in the early binding test).



For the latest information on ADSI, check out our World Wide Web site at:

<http://www.microsoft.com/adsi>.

For ADSI feedback, send mail to

<mailto:adsi@microsoft.com>.