



Microsoft

# ADSI 2.5

*Active Directory Service Interfaces*

## ADSI Extension

---

### **Abstract**

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

For more information on ADSI Extension, please see walkthrough and tutorial in ADSI SDK 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*

<b>INTRODUCTION.....</b>	<b>2</b>
<b>ARCHITECTURE.....</b>	<b>5</b>
<b>EARLY BINDING SUPPORT.....</b>	<b>7</b>
Step 1: Make your component aggregate-able	7
Step 2: Registering Extension	7
Getting ADSI Interfaces from your extension	10
Type Library	10
What's happening under the hood	10
<b>LATE BINDING SUPPORT.....</b>	<b>12</b>
IADsExtension	12
IADsExtension::Operate	13
IADSEExtension::PrivateGetIDsOfNames	13
IADSEExtension::PrivateInvoke	14
IADsExtension Usage	15
Supporting Dual or Dispatch Interfaces	16
What's happening under the hood	18
<b>REVISITING COM AGGREGATION RULES WITH ADSI EXTENSION</b>	<b>19</b>
<b>WHAT DOES A CLIENT SEE?.....</b>	<b>20</b>
<b>RESOLUTION OF MULTIPLE AGGREGATION COMPONENTS</b>	
<b>SUPPORTING THE SAME INTERFACE.....</b>	<b>21</b>
<b>LATE BINDING VS V-TABLE ACCESS IN EXTENSION MODEL</b>	<b>22</b>
<b>RESOLUTION OF FUNCTION/PROPERTY NAME CONFLICTS IN</b>	
<b>AUTOMATION IN EXTENSION.....</b>	<b>23</b>
<b>MORE ON RESOLUTION OF AUTOMATION CONFLICTS: SAME</b>	
<b>FUNCTION NAME BUT DIFFERENT PARAMATERS.....</b>	<b>26</b>
<b>FOR MORE INFORMATION.....</b>	<b>27</b>

An ADSI Extension is a special COM object that allows developers to extend an ADSI object. Each extension is associated with a specified class in the directory. With this extension model, developers can give more dynamic meaning to an object by adding methods. In a traditional directory-programming model, object is accessed via setting and getting object's attributes. Using ADSI, you will be able to add semantic to an object in the directory.

---

How the extension methods are implemented is up to the extension writer. An extension writer may implement a method that is totally outside the scope of the directory. For example, an ISV for backup and restore software plans to extend a computer object. It creates two methods call *BackUp* and *Restore*. These methods will operate remotely on the physical computer pointed by the computer object in the directory. By participating as an extension, the component automatically gains access to ADSI infrastructure and is viewed by ADSI clients as an integral part of ADSI programming model.

Of course, not all components are appropriate as an ADSI Extension. The following guideline should help you decide if creating ADSI extension is

- *Create an extension to integrate your component with the directory object.* Since a user object is in directory, an HR developer may create an ADSI Extension that will populate other information in the SQL Server database every time a user is created.
- *Create an extension if your component requires a directory lookup.* Your component may require a directory as a starting point for directory look up. For example, you are creating a new application. Your application object, let's say *FooApp*, can be published in the directory. Note: in many directory services, such Active Directory™, you are allowed to extend the schema. Your application may support status notifications to a collection of mail servers. You decide to make this application to support an ADSI extension.

Now, a user may search for all instances of *FooApp* in the directory. For each object returned, the user may issue *NotifyNow()*. Your application/extension will be able to get more information about the current object in the directory (i.e getting the server names) and notify each server asynchronously.

- *Create an extension as a junction point between name spaces and programming models.* An ISV invents a new object model for print services. The *printQueue* object already defines in the directory. The ISV can create an ADSI extension associate its extension with *printQueue* object. The ADSI users will be able to bind to a *printQueue* object and starts using ISV's object model. From the ADSI client's perspective, this junction point is transparent.
- *Create an extension to simplify tasks.* Many tasks in the directory can be accomplished by searching and setting multiple attributes in an object or multiple objects. By creating a simple function, it makes life much easier for application and script writers.

For ADSI clients, extension enriches the ADSI programming environment in many

---

ways:

- ADSI clients do not have to learn a new programming model. The extensions are part of ADSI. They would use the same paradigm for searching, data manipulation, securing objects.
- Administrator may manage other related directory enabled application via extension.
- The extension consumer will view ADSI object and extension as one integrated object.
- Existing components may be appropriate to be integrated with ADSI. Hence, it leverages existing investments, and creates powerful synergy between components.

ADSI Extension was designed with the following goals:

- *Easy to implement.* With the current existing Microsoft technology, Visual C++™, Active Template Library™, one can develop extensions in few hours.
- *Clients view one IDispatch.* From the script and automation writers perspective the extension methods and properties are transparently blend into one ADSI object.
- *Independent.* Extension writers can independently extend the ADSI without prior knowledge of all existing extensions.

Let's look at a scenario. Imagine a corporate developer or an ISV is to develop a back up program. This back up program allows an administrator to back up all computers in the organizational unit he or she manages. With the ADSI extension, the following script is possible:

```
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
```

The *LastBackUp* and *BackUpNow* are property and method provided by the extension writer.

The code illustrates the benefits for both extension consumers and providers. The extension writer does not have to reinvent a new way of Filtering, Searching, and

## ARCHITECTURE

Accessing the directory. The extension consumer, on the other hand, does not have to re-learn a new programming paradigm. New methods and properties that are provided by the extension writer are viewed as if they are part of ADSI.

ADSI Extension is based on the COM Aggregation model with several enhancements described later. Extensions still must satisfy all COM Rules. Please consult COM specification for more information.

For our convenience, let's review the COM Aggregation model quickly

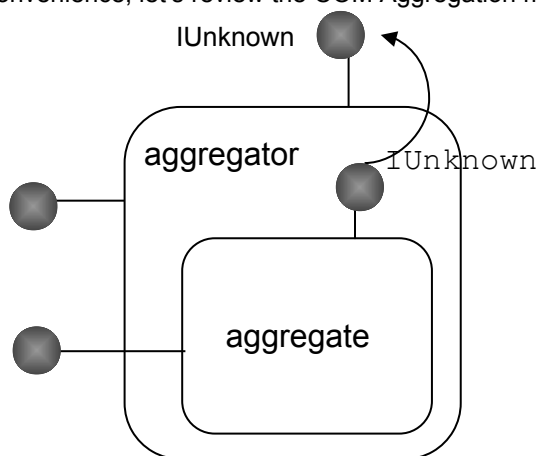


Figure 1. COM Aggregation Model

An *aggregate*, also known as inner object, is an aggregate-able object, which will be created by an aggregator. Your extension object is an aggregate.

An *aggregator*, also known as outer object, is an object that creates an aggregate. ADSI is an aggregator<sup>1</sup>.

The inner object delegates its IUnknown to aggregator's IUnknown

ADSI Extensions adds the following enhancements to COM Aggregation to satisfy its requirements:

- Allows each extension writers to extend ADSI object independently. An extension writer may register its extension to ADSI without worrying the existence of other extensions. In the COM aggregation model, the aggregator must know the aggregate's CLSID. ADSI relaxes this requirement by making ADSI acts as the aggregator for all its extensions. Hence, instead of forming a layer of nested components, the extensions are sibling to each others.
- *One object, one IDispatch* . From day one, automation support is one of highest requirements in ADSI. Automation is achieved by supporting *IDispatch*. Extensions are also encouraged to support the *IDispatch* or dual interface. However, there should be one *IDispatch* on a given object. ADSI integrates and collects many *IDispatches* from extensions, presents them as one consistent

<sup>1</sup> To be exact, the aggregator is an ADSI Provider

---

## EARLY BINDING SUPPORT

IDispatch to the automation controller. Each extension, when it's aggregated, must re-route its IDispatch calls to ADSI's IDispatch.

All these solutions are possible because of services provided by the ADSI Object Manager, which resides on each ADSI provider.

Figure below shows the ADSI Extension Model architecture:

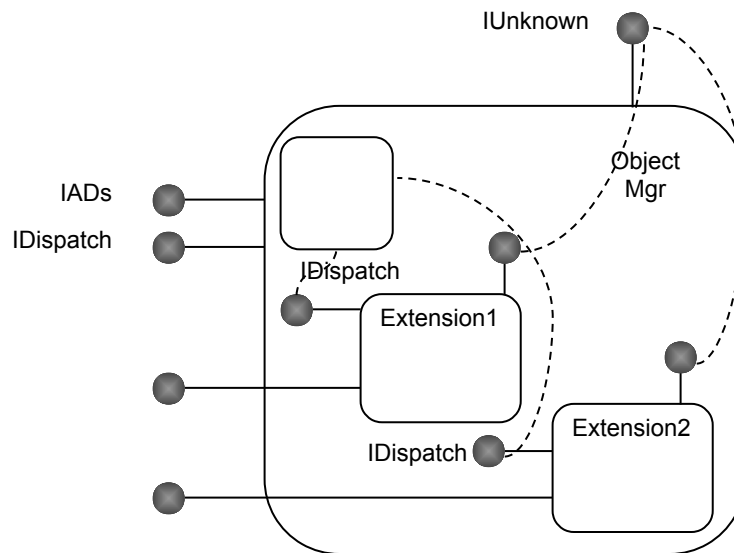


Figure 2. ADSI Extension Component Model

ADSI supports two level of extension:

- **Early Binding Support.** This is the first level of extension. Extension must support extension's registration and implements the new interfaces. The extension consumers must use tools or scripting hosts that support early binding, for example Visual C++ , Visual Basic.
- **Late Binding Support.** It satisfies all early binding requirements, and implementing an additional interface, *IADsExtension*. The extension consumers can use any tools that are able to act as an automation controller, such as Windows Scripting Host™, Active Service Page™, HTML with VB Script.

Let's begin by looking at a scenario when the early binding support in place.

---

Consider the following code snippet:

```
Dim x as IADsUser
Dim y as IADsExt1
Dim z as IADsExt2

Set x = GetObject("LDAP://CN=JSmith, OU=Sales,
                  DC=Microsoft,DC=COM")
x.SetPassword("newPassword")

Set v = x
v.MyNewMethod( "\\srv\public")
y.MyProperty = "Hello World"

Set z = v
z.OtherMethod()
z.OtherProperty = 4362

Debug.Print x.LastName

Set z = GetObject("LDAP://CN=Alice,OU=Engr,
                  DC=Microsoft,DC=COM")
z.OtherProperty = 5323
```

Two extension components extend a *user* object. Each extension publishes its own interface(s). Each extension is not aware other extension's interface(s), only ADSI is aware of the existence of both extensions. Each extension will delegate its IUnknown to ADSI. ADSI will act as an aggregator for both extensions, and any other future extensions. Querying an interface from any extension, or ADSI will yield the same consistent result.

Let's look at the steps to make an extension.

### Step 1: Make your component aggregate-able

Follow the COM specification for making an aggregate-able component. In summary, you must honor the pUnknown passes to your component during CreateInstance, and delegate the pUnknown to the aggregator's unknown if the component is being aggregated.

### Step 2: Registering Extension

Now you must decide which directory classes you want to extend. This is not the same as ADSI interfaces (for example *IADsUser*, *IADsComputer*). Directory objects are persisted in the directory, while your extension and ADSI will be running on the client's machine. Directory object examples are *user*, *computer*, *printQueue*, *serviceConnectionPoint*, *nTDSservice*. You can add a new class in Active Directory and create a new extension for this new class as well.

The registry keys are used to associate a directory class name with the ADSI



extension components. The figure below represents the existing registry layout as well as new keys.

- A new key – “*Extension*” contains a list of keys where each key represents a class in the directory. For each class, for example *User*, *PrintQueue*, *Computer*, maintains a list of sub-keys.
- Each of sub-keys contains the CLSID of the ADSI extension component.
- Each CLSID sub key contains a multi-valued string entry. You should *only* list the interfaces that participate in the aggregation.

Note: The extension objects are still required to register standard COM keys.

```
HKLM
|_Software
|   |_Microsoft
|       |_ADS
|           |_Providers
|               |_LDAP
```

<continue from above>

```
LDAP
|_Extensions
|   |_<ClassNameA>
|       |_<CLSID of ExtensionA1>
|           Interfaces(REG_MUTI_SZ) {List of Interfaces}
|
|       |_<CLSID of ExtensionA2>
|           Interfaces(REG_MUTI_SZ) {List #1 of Interfaces}
|
|   |_<ClassNameB>
|       |_<CLSID of ExtensionB1>
|           Interfaces(REG_MUTI_SZ) {List of Interfaces}
|
|       |_<CLSID of ExtensionB2> {List of Interfaces}
|           Interfaces(REG_MUTI_SZ) {List #1 of Interfaces}
```

**Example:**

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Ads\Providers\LDAP\Extensions\PrintQueue\{9f37f39c-6f49-11d1-8c18-00c04fd8d503}]
"Interfaces"={466841B0-E531-11d1-8718-00C04FD44407} {466841B1-E531-11d1-8718-00C04FD44407}
```

The list of interfaces from Extension1 can be different from that of Extension2. The object, when it's alive, will support the interfaces of the aggregator (ADSI) and all the interfaces provided by the aggregate's Extension1, Extension2. Resolution of conflict of interfaces (same interface supported by both the aggregator and the aggregates or by multiple aggregates) is determined by priorities<sup>2</sup> of the extensions.

---

<sup>2</sup> Priority keys will be defined after ADSI 2.5 Beta 1.

---

You may also associate your CLSID extension to multiple object class names, for example your extension can extend both user and contact objects.

Note: The extension behavior is added per object class, not per object instance
---

It is best to put the extension registration when you are registering COM components, i.e., *DllRegisterSvr*. You should also provide a facility to unregister your extension in *DllUnregisterServer*.

Here is an example on how to register an extension:

```
//////////
// 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 TCHAR szIf[] = _T("{E1E3EDF7-48D1-11D2-B22B-0000F87A6B50}");

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

RegCloseKey(hKey);

return S_OK;

}
```

## Getting ADSI Interfaces from your extension

---

Often time an extension needs to get the information from the directory object it binds to. For example, an extension for computer may want to get the “*dnsHostName*” of the current object from the directory. This can easy achieve by issuing a query interface to the aggregator’s unknown.

Example:

```
HRESULT hr;
IADs *pADs; ` ADSI Interface to get /set attributes

hr = m_pOuterUnk->QueryInterface(IID_IADs, (void**) &pADs);
if ( SUCCEEDED(hr) )
{
    VARIANT var;
    VariantInit(&var);
    hr = pADs ->Get(L"dnsHostName", &var);
    printf("%S\n", V_BSTR(&var));

    VariantClear(&var);
    pADs->Release();
}
```

## Type Library

The type library from all extensions will not be merged. The ADSI client must specifically include the type libraries for each extension. The type library is needed for Visual Basic to enable early bindings. C/C++ can directly QI the extension interfaces defined in the extension’s header files.

## What’s happening under the hood

Now, you’re done. This is how ADSI interacts with the extensions.

- Someone issues an ADSI Binding to a directory object. For example, *LDAP://CN=Jsmith, OU=Sales, DC=ArcadiaBay, DC=COM*
- ADSI finds out that the object has an object class of ‘user’.
- ADSI looks up in the registry and finds extensions’ CLSIDs for ‘user’. Note, ADSI actually caches this information.
- Someone QIs for IID\_ImyExtension. ADSI searches the interfaces, associated with the ‘user’ object, both its own interfaces first and other extension interfaces. The Interface conflict and resolution will be discussed later.
- If found, ADSI creates the component that support IID\_ImyExtension, and QI for the extension. The interface will be given to the consumer.

---

## LATE BINDING SUPPORT

- Now, with this interface, one can call the interface's methods.
- Next, one will QI for IID\_IYourExtension which is in a different component. This component will delegate this Query Interface to the aggregator's unknown, which happens to be ADSI itself.
- Again, ADSI searches the interfaces and create the component.

Let's revisit our scenario. When a late binding support is in place, every function call must go through to one IDispatch, *ADSI's IDispatch*, before it re-routes the appropriate extension.

Consider the following code snippet:

```
Set x = GetObject("LDAP://CN=JSmith, OU=Sales,
                  DC=Microsoft,DC=COM")
x.SetPassword("newPassword")
```

```
x.MyNewMethod( "\\srv\public")
x.MyProperty = "Hello World"
```

```
x.OtherMethod()
x.OtherProperty = 4362
```

```
Debug.Print x.LastName
```

Notice in the example above, there aren't any explicit Query Interfaces to get to your extensions. The extensions must re-route their IDispatch calls to ADSI's IDispatch. ADSI makes the decision and resolves any conflict if it occurs, then it re-routes back to the appropriate extension via an interface called, *IADsExtension*. Hence, any extension wants to support late binding must implement IADsExtension.

## IADsExtension

IADsExtension defines as:

```
IADsExtension : public IUnknown
{
public:
    virtual HRESULT STDMETHODCALLTYPE Operate(
        /* [in] */ DWORD dwCode,
        /* [in] */ VARIANT varData1,
        /* [in] */ VARIANT varData2,
        /* [in] */ VARIANT varData3) = 0;

    virtual HRESULT STDMETHODCALLTYPE PrivateGetIDsOfNames(
        /* [in] */ REFIID riid,
        /* [in] */ OLECHAR __RPC_FAR * __RPC_FAR *rgszNames,
        /* [in] */ unsigned int cNames,
        /* [in] */ LCID lcid,
        /* [out] */ DISPID __RPC_FAR *rgDispId) = 0;

    virtual HRESULT STDMETHODCALLTYPE PrivateInvoke(
        /* [in] */ DISPID dispidMember,
```

```

/* [in] */ REFIID riid,
/* [in] */ LCID lcid,
/* [in] */ WORD wFlags,
/* [in] */ DISPPARAMS __RPC_FAR *pdispparams,
/* [out] */ VARIANT __RPC_FAR *pvarResult,
/* [out] */ EXCEPINFO __RPC_FAR *pexcepinfo,
/* [out] */ unsigned int __RPC_FAR *puArgErr) = 0;

};

```

## IADsExtension::Operate

This function is called by the aggregator (ADSI). The extension should interpret each code and parameters according to the provider's documentation.

```

HRESULT Operate(

    DWORD dwCode,

    VARIANT varData1,

    VARIANT varData2,

    VARIANT varData3)

```

### Parameters

*dwCode* -[in] indicates the type of data.

*VarData1 to 3* – [in] Custom parameters supplied by the provider. The value depends on the *dwCode*.

At present, the only information defined by the ADSI is ADS\_EXT\_INIT.

DwCode	VarData1	VarData2	VarData3
ADS_EXT_INIT	Not define	Not define	Not define

### HRESULT

The expected HRESULT returned are S\_OK, E\_FAIL or E\_NOTIMPL.

If the extension return E\_FAIL, E\_NOTIMPL, ADSI simply ignores this return value.

---

## **IADSExtension::PrivateGetIDsOfNames**

This function is called by the aggregator (ADSI). The function is called after ADSI determines the extension to service the dispatch. The extension could use the type info for getting the DISPID, i.e via `DispGetIDsOfNames`.

**HRESULT PrivateGetIDsOfNames(REFIID riid, OLECHAR FAR\* FAR\* rgszNames, unsigned int cNames, LCID lcid, DISPID FAR \* rgDispId)**

### **Parameters**

All parameters have the same meaning as the parameters in the standard `IDispatch::GetIDsOfNames()`

### **Remark**

The extension component must return a unique id (`rgDispId`) for each method or property defined in *all* the dual interfaces supported. The restriction on uniqueness is *only within* this extension. The ADSI provider will resolve and ensure uniqueness higher up - among dispids of *all* extension objects and the aggregator(ADSI) itself.

`rgDispId` returned by `PrivateGetIDsOfNames` *must* be between 1 and  $2^{24}-1$ , or -1 (`DISPID_UNKNOWN`)<sup>3</sup>.

## **IADSExtension::PrivateInvoke**

The `PrivateInvoke` is normally called by ADSI after calling *PrivateGetIDsOfNames*. The extension should call the method that actually implements the method. Alternatively, the extension can use type info and call *DispInvoke*.

**PrivateInvoke(DISPID dispIdMember, REFIID riid, LCID lcid, WORD wFlags, DISPPARAMS FAR \* pDispParams, VARIANT FAR \* pVarResult, EXCEPINFO FAR \* pExcepInfo, unsigned int FAR \* puArgErr)**

---

<sup>3</sup> All non-positive dispids ( $\leq 0$ ), except for `DISPID_UNKNWON` (-1), from ADSI clients are handled by the *aggregator* in the ADSI provider, and will *not* be passed into the extension object through `IADsExtension::PrivateInvoke()`. At present, ADSI provider only supports `DISPID_VALUE` (0) and `DISPID_NEWENUM` (-4).

---

### **Parameters**

All parameters have the same meaning as the parameters in standard `IDispatch::Invoke()`.

The `dispIdMember` is the `rgDispId` returned by `PrivateGetIDsOfName()`.

---

## IADsExtension Usage

IADsExtension is an optional interface. The extension writer should implement this interface when at least one of the following conditions is met:

1. The extension component needs an initialization notification as defined by ADSI\_EXT\_ dwCode in *Operate()* method.
2. The extension component supports any dual or dispatch interface.

If an extension component support IADsExtension for reason number 1 only, *IADsExtension::PrivateGetIDsOfNames()* and *IADsExtension::PrivateInvoke()* can simply return E\_NOTIMPL. On the other hand, if an extension component supports this interface for reason number 2 only, *Operate()* can simple ignore the data and return HRESULT as E\_NOTIMPL.

### *Example of IADsExtension implemented by an extension.*

```
STDMETHOD(Operate)(ULONG dwCode, VARIANT varData1, VARIANT varData2,
VARIANT varData3)

{
    HRESULT hr = S_OK;

    switch (dwCode)
    {
        case ADS_EXT_INIT:

            // 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 *
pexcepinf, UINT * puArgErr)

{
    return DispInvoke( (IHelloWorld*)this,

        m_pTypeInfo,

        dispidMember,

        wFlags,

        pdispparams,

        pvarResult,

        pexcepinf,

        puArgErr );
}

```

## Supporting Dual or Dispatch Interfaces

Like dispatch interface, all dual interfaces *must* inherit from an *IDispatch* which delegate all of its IDispatch functions (GetIDsOfNames, Invoke, GetTypeInfo, GetTypeInfoCount) back to the IDispatch of the *aggregator (ADSI)*. To execute the delegation, extension object should query for the *aggregator's* IDispatch (IID\_IDispatch, or IID\_IADs) from m\_pOuterUnknown and release the pointer after use.

If your extension can be a standalone component, you may want to check if you are being aggregated. If it's being aggregated, you should re-route the dispatch functions to the aggregator's IDispatch, otherwise (a stand-alone version) you can call your internal implementation of IDispatch, or you can just call your IADsExtension's implementation.

**Example re-routing IDispatch call to aggregator's IDispatch :**

```
////////////////////////////////////
// Delegating IDispatch Methods to the aggregator
////////////////////////////////////
STDMETHODIMP MyExtension::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 MyExtension::TypeInfo(UINT itinfo, LCID lcid, ITypeInfo**
pptinfo)
{
    IDispatch *pDisp;
    HRESULT hr;
    hr = OuterQueryInterface( IID_IDispatch, (void**) &pDisp );
    if ( SUCCEEDED(hr) )
    {
        hr = pDisp->TypeInfo( itinfo, lcid, pptinfo );
        pDisp->Release();
    }

    return hr;
}

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

    return hr;
}

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

```

---

## REVISITING COM AGGREGATION RULES WITH ADSI EXTENSION

```
        hr = pDisp->Invoke( dispidMember, riid, lcid, wFlags,  
                           pdispparams, pvarResult, pexcepinfo, puArgErr);  
        pDisp->Release();  
    }  
  
    return hr;  
}
```

ADSI strongly encourages extension writers to support dual interfaces instead of dispatch interfaces in their extension objects. A dual interface allows a client to have faster access, as long as vtable access is enabled in the client (please see "*Late Binding VS v-table Access*"). Based on our current model, implementing dual interfaces should not be more difficult than implementing dispatch interfaces.

### What's happening under the hood

That's it, you're done! Here is what happening in the late binding case:

- Someone issues an ADSI Binding to a directory object. For example, *LDAP://CN=Jsmith, OU=Sales, DC=ArcadiaBay, DC=COM* via COM late binding. This QIs ADSI for IDispatch.
- ADSI finds out that the object has an object class of 'user', create an object that support appropriate ADSI interfaces, ie *IADs*, *IADsUser*.
- ADSI looks up in the registry and finds extensions' CLSIDs for 'user'. Note, ADSI actually caches this information.
- Someone issues "MyNewMethod". ADSI looks up its DISPID, and other ADSI Extensions' dispIDs. ADSI, then, decides the extension that serves this call, by calling the extension's *IADsExtension*.
- The extension executes the function.
- Now, the script writer invokes "YouNewMethod" via the current extension's IDispatch. The extension's IDispatch's implementation delegates back ADSI IDispatch.
- ADSI IDispatch, again, searches for the appropriate extension (or itself), then it calls the appropriate extension via the extension's *IADsExtension*.

Note: The way ADSI choosing the appropriate is deterministic, as describe later.

For a quick reference, we include the COM aggregation rules along with ADSI extension rules.

---

## WHAT DOES A CLIENT SEE?

- CreateInstance() returns a pointer to an IUnknown which does *not* delegate any function calls to the *aggregator*.
  - IUnknown::QueryInterface() returns pointers to the interfaces it supports and error for interfaces it doesn't.
  - IUnknown::AddRef() increments the reference count on the extension object (*aggregatee*) itself.
  - IUnknown::Release() decrements the reference count on the extension object (*aggregatee*) itself and self-destroy when the count is 0.
- The extension object should store the aggregator's IUnknown pointer (e.g. m\_pOuterUnknown) during CreateInstance().
- All interfaces the extension object supports, including IADsExtension should inherit from the IUnknown which *do* delegate all function calls back to the *aggregator*.
  - IUnknown::QueryInterface() call m\_pOuterUnknown->QueryInterface().
  - IUnknown::AddRef() calls m\_pOuterUnknown->AddRef().
  - IUnknown::Release() calls m\_pOuterUnknown->Release().

The extension object writers can choose any internal implementation they prefer but the object writers must obey standard COM aggregation rules. Please note that extension object does *not* have to work standalone. Extensions are designed to work as an aggregate. However, an extension can be written to work both standalone and as an aggregate.

In addition to standard COM aggregation support, extension object may support IADsExtension for more advanced features. If late binding support is supported the extension should

- Delegate IDispatch's functions back to the *aggregator*.
- Implement the Dispatch functionality in IADsExtension

- An ADSI client sees the ADSI and all of its extension objects as one object.

---

## RESOLUTION OF V-TABLE ACCESS IN EXTENSION MODEL COMPONENTS SUPPORTING THE SAME INTERFACE

- An ADSI client sees **one IDispatch** which handles **all** the **dual** and **dispatch** interfaces in the object, whether the dual/dispatch interface is implemented by the aggregator in the provider or by an extension. (please see "*Resolution of Function/Property Name Conflicts in Automation*") )
- ADSI does not expose any typeinfo through IDispatch::GetTypeInfo or IDispatch::GetTypeInfoCount. ADSI provides typeinfo through type library<sup>4</sup>.

It's highly unlikely that two extensions expose the same interface to ADSI. If this happens, however, the following rules apply:

- If an interface, e.g IFOO, is supported by both the *aggregator (ADSI)* and any extension objects, QueryInterface() will **always** return the *ADSI's* IFOO.
- If an interface, e.g. IFOO, is not supported by the *aggregator(ADSI)* but is supported by more than one extension objects, QueryInterface() will **always** return the IFOO of the first extension object listed in the registry which supports it.

Please note that the ordering of components in the registry will also affect the resolution of name conflicts in automation (please see "*Resolution of Function/Property Name Conflicts in ADSI Extension Model*")

A dual interface allows direct v-table access to all its functions while a dispatch interface does not. A C/C++ client can query for a dual interface pointer and invoke its functions by direct v-table access. This provides a faster access than invoking the function via IDispatch::GetIDsOfNames() and IDispatch::Invoke(). This is especially true in our extension model since all dual interfaces in a extension object must delegate their GetIDsOfNames() and Invoke() back to the *aggregator (ADSI)* first. The *aggregator* then needs to perform extra internal works to sort out which extension object, or maybe itself, provides support for the function being called and redirects the call to the appropriate object.

Visual Basic also invokes a dual-interface function by a direct v-table access if it has a pointer to the interface and access to typeinfo from the type library. ADSI clients written in VB can specify a pointer to a dual interface, e.g *IADsDualInf*, explicitly and thus enable v-table access to functions, e.g. Func1, in the interface.

Example:

---

<sup>4</sup> ADSI ships the type library (activeds.tlb) to provide type info. Extension object writers should provide their own typelib for any typeinfo they would like to expose.

---

## RESOLUTION OF FUNCTION/PROPERTY NAME CONFLICTS IN AUTOMATION IN EXTENSION

```
Dim inf as IADsDualInf
Set inf = GetObject(...) 'an object supporting IADsDualInf
inf.Func1 'IADsDualInf::Func1() will be invoked through direct v-table access
```

Since a dispatch interface does not support v-table access, the above does not apply. In other words, a dispatch function is always invoked through `IDispatch::GetIDsOfNames()` and `IDispatch::Invoke()` only.

Current releases of VBS and JavaScript also do not support v-table access. Hence, a dual interface in a VBScript and JavaScript environment behaves like a dispatch interface.

In this section, an *object* means the object as a whole seen by an ADSI client. In other words, *ADSI and all of its extensions*.

If two or more *dual/dispatch* interfaces in an object support a function (property) of the same name *Func1*, the *Func1* invocation will be determined by:

- 1) if the client has a pointer to one of *dual (only)* interfaces which *support a function called Func1 AND*
- 2) if the automation environment supports v-table access.

In "Late Binding VS v-table Access in ADSI Extension Model", we have briefly explained how a client can get a pointer to a dual interface and what types of environment support v-table access. Further details can be obtained from references on COM/OLE, VB/VBA/VBS and JavaScript.

If both conditions 1) and 2) are true, i.e. a client has a pointer to *dual (only)* interface *IDualInf1* which *supports a function called Func1* and the automation environment supports v-table access, *IDualInf1::Func1* will be invoked -- directly through ADSI v-table access.

If condition 1) or/and 2) fails, `IDispatch::GetIDsOfNames()` and `IDispatch::Invoke()` will be called to invoke a *Func1*. Since all extension objects redirect the `IDispatch` functions back to the *aggregator*, the *aggregator* controls of which *Func1* will be invoked. The rules are:

If any interface, and there will only be one if any<sup>5</sup>, in the *aggregator (ADSI)* supports a function called Func1, the aggregator will invoke its own Func1.

Otherwise, the *aggregator* will go through each of its extensions, in the order listed in the registry, and find the first extension which implements a function called Func1. It is possible, but highly unlikely, that multiple *dual /Dispatch* interfaces in this first extension have a function called Func1. It is up to the extension to determine which Func1 should **always** be invoked in automation<sup>1</sup>.

Example in VB:

Consider the following assumptions:

- *IADs0* does **not** support *Func0*.
- *IADs1* which supports *Func1* and *Func0*.
- *IADs2* which supports *Func2* and *Func0*.
- All three interfaces are dual interfaces.

IADs0 : IDispatch	IADs1 : IDispatch	IADs2 : IDispatch
{ OtherFunc(); }	{ Func0() Func1(); }	{ Func0() Func2(); }

```
Dim myInf1 as IADs1  
  
myInf1.Func1 ' IADs1::Func1 is invoked via direct v-table access  
  
myInf1.Func2 ' IADs2::Func2 is invoked via GetIDsOfNames/Invoke
```

Note that even though IADs1 does **not** support Func2, an ADSI client see **one IDispatch** which supports all the **dual** and **dispatch interfaces** in our model. Thus, the ADSI client can directly call Func2 via *myInf1.Func2* **without** figuring out of which dual interface supports Func2.

<sup>5</sup> Uniqueness on Dispid is restricted to *within each* extension object only. The *aggregator* in ADSI provider will resolve and ensure uniqueness higher up - among *all* extension objects and the *aggregator* itself - as long as dispids returned by IADsExtension::PrivateGetIDsOfNames() are always between -1 and 2<sup>24</sup>-1.

---

```
myInf1.Func0
```

Note that both IADs1 and IADs2 have a function called Func0, but since the client

- 1) has a pointer to *dual* interface IADs1 which *has a function called Func0* AND
- 2) VB supports direct v-table access assuming type info is available through type library,

Hence, IADs1::Func0 is invoked via direct v-table access

```
Dim myInf2 as IADs2
Set myInf2 = myInf1 ' Querying for pointer to IADs2
myInf2.Func0
```

This time, since the client has a dual interface pointer to IADs2 instead of IADs1, IADs2::Func0 is invoked via direct v-table access

```
Dim myInfNone as IADs0

Set myInfNone = myInf1 'or directly to the aggregated object
which supports both IADs1 & 2

myInfNone.Func0
```

Again, both IADs1 and IADs2 have a function called Func0, but this time, the client has a pointer to *dual* interface IADs0 which does *not* have a function called Func0. Therefore, there is no v-table direct access can be performed. Instead, IDispatch::GetIDsOfNames()/Invoke() are called to invoke Func0.'

case a) IADs1 and IADs2 are implemented by two COM components Ext1 and Ext2 respectively

i) Ext1 is an extension which comes before Ext2 in the registry  
IADs1::Func0 is invoked.

ii) extension Ext2 comes first in the registry before extension Ext1, the  
IADsInterface2::Func0 is invoked.

case b) if IADs1 and IADs2 are implemented by the same extension object Ext. The Func0 is always invoked by the Ext's IADsExtension::PrivateGetIDsOfNames() and PrivateInvoke().

It's completely up to the extension writers to determine how to resolve conflicts of functions (or properties) of different dual/dispatch interfaces having the *same name* in an extension. The implementation of IADsExtension::PrivateGetIDsOfNames()



---

## MORE ON RESOLUTION OF AUTOMATION CONFLICTS: SAME FUNCTION NAME BUT DIFFERENT PARAMETERS

and `PrivateInvoke()` should resolve this conflict. For example, `IFOO::Func1` and `IMoreFOO::Func1` where `IFOO` and `IMoreFOO` are dual/dispatch interfaces supported by the same extension object. `PrivateGetIDsOfNames()` and `PrivateInvoke()` must determine which `Func1` should **always** be called.

The same applies to a conflicting dispid in different dual or dispatch interfaces. For example, the dispid of `IFOO::Y` is 2 in `ifoo.odl` (or `ifoo.idl`). The dispid of `IMoreFOO::X` is 2 also in `imorefoo.odl`. `IADsExtension::PrivateGetIDsOfNames()` must return a unique (within the extension itself) dispid for each instead of returning the same dispid for both.

As a reference, ADSI resolves the first problem by not supporting multiple interfaces with conflicting function (or property) names. It resolves the later problem by adding a unique (within the same extension object) "interface number" to the unused bits of the dispid.

We have discussed the resolution of function name conflicts – same function name and same parameter list (number, type and order) - in automation. But what if two functions have the same name but different parameters (number, type or order)? Unfortunately, if an ADSI *client* invoke the function with `IDispatch::GetIDsOfNames()` without specifying the parameters by using multiple names, ADSI Extension model cannot distinguish the functions. Based on the resolution scheme discussed in the above paragraph, the first extension in the registry which supports this function through one of its interfaces will have its version of this function invoked. And the call may fail or yield incorrect results.

For example,

Extn1 (first in the registry under class CA – higher priority) supports `IInterface1`.

Extn2 (third in the registry under class CA – lower priority) supports `IInterface2`.

`IInterface1` supports `Foo1(int param1, int param2)`.

`IInterface2` supports `Foo1(int param1)`

An ADSI client has a dispatch interface pointer to an object of class CA: It wants to invoke `IInterface2::Foo1()`. If the client calls `pDispatch->GetIDsOfNames(IID_NULL, rgpszNames, 1, MY_LCID, rgDispId)` by just storing the function name "Foo1" in `rgpszNames[0]`, then `IInterface1::Foo1()`, **instead of** the desired `IInterface2::Foo1()`, will be invoked and the function will fail since the number of parameters are different.

Extension writers should minimize this problem by prefixing their function names with their own specific identifiers and avoiding interface design with functions of the

---

## FOR MORE INFORMATION

same name but different parameters.

In case this situation of name conflict does happen, ADSI clients can avoid this problem by direct v-table access if the interface is a dual interface. (See discussion above.) If direct v-table access is not possible, ADSI clients should call `IDispatch::GetIDsOfNames()` with multiple names - specifying the function names as well as the parameters in the array `rgszNames` described above.

VB 5.0 does **not** call `IDispatch::GetIDsOfNames()` with multiple names. That is, it only passes the function name to `GetIDsOfNames()` but not arguments also. However, VB5 allows users to invoke a function by direct v-table access (see discussion above) instead of invoking `IDispatch::GetIDsOfNames()` if the interface is a dual interface. User should use direct v-table access if possible.

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>