

# Implementing Side-by-Side Component Sharing in Applications (Expanded)

David D'Souza, BJ Whalen, and Peter Wilson  
Microsoft Corporation

## Summary

Explores the implementation of side-by-side shared components in Microsoft® Windows® 2000 and Windows 98 Second Edition, as discussed in the Windows Certification specification. Covers the creation of new side-by-side components and the use of DLL/COM redirection to handle incompatibility among different versions of the same component. Includes guidelines for writing and installing side-by-side components and for repackaging and testing applications.

## Introduction

Modern operating systems and applications are built from many components. A component is a self-contained software entity, offering a set of functions that can be used broadly by a variety of applications. Since individual components are used by more than one application, component sharing is essential.

Successful global component sharing requires that any shared component function exactly like previous versions of that component. In practice, however, 100 percent backward compatibility is difficult if not impossible to achieve because of the difficulty in testing all configurations in which the shared component may be used. Both newer and older applications end up using the same component, and over time, fixing and improving the component becomes increasingly difficult.

As well, the practical functionality of a component is not easily defined. Applications may become dependent on unintended side effects that are not considered part of the core function of the component. For example, an application may become dependent on a bug in the component, and when the component developer chooses to fix that bug, the application fails, in what has come to be called "DLL Hell." The sheer volume of applications that use each component only deepens the problem.

This lack of backwards compatibility can result in the inability to deploy a new application without breaking applications already deployed or compromising the functionality of the new application. Any new application requires a version of a shared component that is different from the version already deployed. To provide for successful sharing while enhancing application stability, Microsoft has introduced side-by-side sharing in Windows 2000 and Windows 98 Second Edition, creating a way to share components through selective isolation.

## A Little Background

Before we get into the details of side-by-side sharing, let's look at some of the background issues and the problems of DLL Hell.

## Component Sharing

Windows has embraced the concept of sharing since its inception. All operating systems balance the need to provide a robust, full set of services against the resource constraints of the hardware for which the operating system was designed. Until recently, CPU usage and disk space were very tight resources on the PC platform. An obvious way to fit operating system and application code into a small space has been to share code as much as possible. Among many other benefits, code sharing improves the leverage of hardware resources and minimizes the exposed front that quality assurance must test. Code sharing is part of what has made Windows successful.

Windows does not restrict sharing to just code. Application and component state can be found throughout the operating system in the form of Registry state, application-specific data storage in the file system, and Windows APIs that expose global namespaces. Such sharing provides for a high

# Implementing Side-by-Side Component Sharing in Applications (Expanded)

David D'Souza, BJ Whalen, and Peter Wilson  
Microsoft Corporation

level of interoperability between applications produced by multiple software vendors, bringing cost reduction and heightened software efficiency.

However, there are costs to sharing. Sharing means that applications become interdependent upon one other, introducing an element of fragility. Changes to one component may produce unintended effects in other components. Typically, an application may become dependent on a particular version of a shared component. Another application may be installed with an upgraded (or downgraded) version of that shared component, and the first application may suffer from that change. In the extreme, applications that once worked well mysteriously start to function oddly, or even fail. This condition is often referred to as "DLL Hell."

## Isolation

The opposite of sharing in a system is isolation. Applications can be isolated by statically binding all resources and code into the application. However, complete isolation is not feasible today for applications that rely on COM or any other globally stored system resources.

One solution for reducing application fragility is to selectively isolate applications and components. Under this scenario, applications may all have access to the same component, but multiple versions of that component now become available. Component producers have the freedom to produce new versions of old components, making improvements and fixing bugs. Customers, on the other hand, can choose the version that fits with a particular application. It's like going to an auto parts store and picking up a fuel pump for your 1984 Chevy. You find the pump in stock, alongside pumps that fit other models, later years. With components, the key is to provide the version that is appropriate to each application and isolate the different versions from each other. Further, with redirection, applications can be configured to use the component version that fits with that particular application, regardless of what other versions are currently deployed or will be deployed in the future.

## Side-by-Side Sharing

To encourage such isolation, Microsoft Windows 2000 and Windows 98 Second Edition feature a new form of component sharing called side-by-side sharing, which uses selective isolation to minimize DLL Hell. Side-by-side sharing allows multiple versions of the same component (COM or Win32®) to run at the same time in different processes. Applications can then use the specific version of a component for which they were designed and tested, even if another application requires a different version of the same component. This arrangement allows developers to build and deploy more reliable applications because developers are able to specify the version of the component they will use for their application, independent of the other applications on the system.

## New Component Sharing Strategies

Side-by-side sharing in Windows 2000 and Windows 98 Second Edition follows two strategies:

- **Creating side-by-side components.** Developers build new components that support the simultaneous execution of multiple versions of that component. Consumers of these new components are now able to use the version for which they were built and installed, regardless of what other versions are installed on the computer.
- **DLL/COM redirection.** Developers and administrators repackage existing applications and components so that the required versions of shared components are privatized to the application that needs them, each functioning side by side with other versions.

# Implementing Side-by-Side Component Sharing in Applications (Expanded)

David D'Souza, BJ Whalen, and Peter Wilson  
Microsoft Corporation

Side-by-side components, whether newly created or part of an existing, reconfigured application, are not supported in all scenarios.

- The primary scenario for creating side-by-side components occurs when components are hosted, in process, within another container application. For example, if your controls are to be used by a desktop Windows application (written Microsoft Visual Basic® or Visual C++®), the controls are a good candidate for side-by-side design. Side-by-side components are not recommended for use in an IIS Server context, or in an MTS Server.
- The primary scenario for DLL/COM redirection occurs when new client applications are being deployed to computers that already support several other applications, or where the new client application needs to be made more resilient to changes in shared components caused by the installation of other applications. DLL/COM redirection should not be used in the context of an IIS Server (Web application) or in an MTS/COM+1.0 Server. Microsoft is working on other solutions to address isolation for these scenarios in the future. Nonredirected components are supported as usual in these environments.

**Note:** DLL/COM redirection focuses on addressing application conflicts when existing applications and components are being deployed. When developing a new application or new components the best strategy is to develop side-by-side components that are inherently isolated.

## Comparing the Two Strategies

The following table compares the two approaches, DLL/COM redirection and creating side-by-side (SxS) components, and gives guidance in selecting the right approach for your scenario.

**Table 1. Side-by-Side Strategies Compared**

Attributes	SxS Components	DLL/COM Redirection
What is the primary focus?	Building robust <b>new components</b> that in future versions will be immune from changes that render the component "backwards incompatible."	Insulating <b>existing applications</b> from problems caused by other applications installing incompatible shared DLLs.
Are specific versions of isolated to the applications that use them?	Yes. SxS components must always be deployed so that they are isolated to the applications that use them.	Yes. Applications that employ DLL/COM redirection use the versions of any shared components that are installed in the application directory, regardless of what versions are installed elsewhere on the system.
Is new code or changes to existing code required?	Yes. Building SxS components (or making existing components SxS) requires at a minimum that the COM registration code be changed to allow access by relative path. Additional coding changes may be required to ensure that global state is handled correctly between SxS running versions.	No. DLL/COM redirection allows applications to be reconfigured to install and run SxS without any code changes or recompilation. This allows administrators without access to the source code of an application to reconfigure it to address DLL Hell problems.
Does the strategy work in all scenarios?	Yes. SxS components are designed and coded to be installed and run	No. DLL/COM redirection does not require code changes. Existing

# Implementing Side-by-Side Component Sharing in Applications (Expanded)

David D'Souza, BJ Whalen, and Peter Wilson  
Microsoft Corporation

	SxS. Therefore, well-designed, developed, and tested SxS components (and the applications that use them) will be free of the problems associated with DLL Hell.	applications and components may not have been designed with the requirement that more than one version can run at the same time. Experience has shown that in most cases existing applications and components can run SxS, but testing is required to confirm this for a specific scenario. (See <a href="#">Selecting Components to Isolate.</a> )
--	---	---

As a general rule of thumb:

- If DLL Hell problems are preventing the deployment of an existing application, use DLL/COM redirection to isolate the conflicting components. (To further understand this option, see the scenarios below).
- If you are building a new application and want to design and develop defensively against DLL Hell problems, develop side-by-side components.

## Creating New Side-by-Side Components

Side-by-side sharing requires that when creating new applications, developers write side-by-side components. These are ordinary COM or Win32 components, except that they are installed into the application directory (or a subdirectory thereof) rather than the system directory. They are isolated to a specific application and are not globally shared across all applications.

Thus, a component can be installed safely side by side, with a different version of the same component that is installed elsewhere, in another application directory. If another application on the system requires (and thus installs) a different version, your application is unaffected. Both applications will run with their respective versions of the component.

In the event that another application installs a new version of a component on the system, your version of that component will remain untouched, since you installed it into your application directory. Your application continues to use the same version of the component that you shipped with your application, while the other application uses its version. The operating system can load both versions at once.

Similarly, because the side-by-side component is "private" to the application that installed it, the application uninstaller can always safely remove the side-by-side component, as by definition no other application can depend upon a private component.

**Note:** Side-by-side components must be registered properly with the operating system (described later in this paper) so that each version of the component won't conflict with other versions of the component that may exist.

**Note:** Windows 2000 and Windows 98 Second Edition both support side-by-side sharing. It is not supported for previous Windows operating systems; however, on these systems a side-by-side DLL (a DLL written according to the guidelines in the next section) can be installed into the system directory.

# Implementing Side-by-Side Component Sharing in Applications (Expanded)

David D'Souza, BJ Whalen, and Peter Wilson  
Microsoft Corporation

Thus, the DLL functions in a global-sharing (backward compatible) fashion. Applications must dynamically check the operating system version to determine which sharing technique to use.

## Guidelines for Writing Side-by-Side Components

The following guidelines outline the issues involved in producing side-by-side components (COM or Win32). When you write such components and place them in the application directory, your code is privatized to the application's context. When you root your data in the registry based on the application name, that data is privatized to the application.

Customers of your side-by-side component are insulated from changes that may be needed by other customers of your component. You are also able to update your components without fear of breaking existing applications. Applications are able to install your component without a reboot, even if another application is using a different version of your component.

**Note:** These guidelines are a more robust form of the current Windows Certification guidelines that tell you to put Win32 DLLs into your application directory.

### General Issues

- Do not attempt to replace any of the files protected by System File Protection that ship with Windows 2000, including most .sys, .dll, .exe, and .ocx files.
- You must test all components to ensure side-by-side validity, especially in areas where sharing can occur, since there is no side-by-side enforcement by current operating systems.
- Gather all version-specific names together into #defines to enable easy source code migration from one version to another. Doing so allows you to change the version in one place; all registry keys then change automatically. For example:

```
#define MyRegistryKey "MyAppv1.0.0.0"
```

**When you release a new version of your component, you just need to change the version in one place.**

- Beware of the loss of the ability to apply quick fixes to components, since they are now in arbitrarily located application directories. As a component producer, you may not know all of the places where you need to fix your component. Application vendors need to distribute updates to customers.
- Beware of inadvertent cross-process sharing. For example, shared memory sections can cause problems because the section is not shared across different versions of your component.
- You need to design all shared data structures to be side-by-side (different for each version of the component), including memory mapped files, mutexes, named pipes, and hardware drivers.
- You must store any nonpersistent data in the TEMP directory.
- Do not put user data in global locations. There should be a clear separation of application data and user data.

# Implementing Side-by-Side Component Sharing in Applications (Expanded)

David D'Souza, BJ Whalen, and Peter Wilson  
Microsoft Corporation

## Strengthening Your Components

- You must properly reference count your GUIDs in the global registry across installs and uninstalls of your component by different vendors. Maintaining a reference count under the GUID key is the most reliable way to do that.

```
HKEY_CLASSES_ROOT\CLSID\{GUID}\InprocServer32
Default=foo.dll
ThreadingModel=Apartment
RefCount=1
```

**Note:** You must reference count the DLL independently from the GUID.

- Do not register full path names for COM registrations; you can depend on directory search of the application directory to find your component and its dependencies.
- If you change metadata (such as the threading model) under the COM GUID, you need to rename and apply a new GUID for your component, since it is effectively a new version of the component. You need to do this because the data in the registry would not be valid in another application's context, with that application's private version of your component).
- You must contain the type library within your DLL and not in a separate file.
- Do not use the LoadRegTypeLib function to load type libraries through the registry. The registration of the type library in the registry is global state and runs counter to the rules of privatization. Instead, use LoadType Lib.
- The ability of OLEAUT to create a new version of external type libraries is fragile and prone to failure. It is advised that you do not use external type libraries for side-by-side components.
- When modifying an existing component to make it side-by-side, you are changing the way it is activated to use a relative path and isolate global state. It's important that you give it a new CLSID, ProgId, and rename the file, and then use this CLSID, ProgId, and new file name for future side-by-side versions. Doing so avoids conflicts that would otherwise occur if a non-side-by-side version of a component were registered over a side-by-side version. Side-by-side components are not backward compatible with previous versions that are themselves not side-by-side.

## State Storage

- Regarding state (settings) stored in the registry, you need to privatize state to the application context that runs. You can use the GetModuleFileName() function to set up a virtual root. This should be done for HKLM and HKCU branches.
- Registry settings must be done on a per-version basis to obtain registry isolation. Registry keys are a common way in which components save their state. Since different versions of your component may exist on your machine, it is important that it be as easy as possible to version your keys as you recompile. Getting a good set of header files and helper APIs makes this easy.
- Store your registry state in keys with the following naming convention:

```
HKCU\MyCompany\MyComponent\VersionXXXX\
```

# Implementing Side-by-Side Component Sharing in Applications (Expanded)

David D'Souza, BJ Whalen, and Peter Wilson  
Microsoft Corporation

For example, assume a configuration setting called **EnableSuperCoolFeature** with the value of true or false. Traditionally, you would store this information in the registry as follows:

```
HKEY_CurrentUser\Software\MyCompany\MyComponent\  
    EnableSuperCoolFeature = TRUE
```

With side-by-side sharing, you would need to store it as follows:

```
HKEY_CurrentUser\Software\MyCompany\MyComponent\Version01.01  
    EnableSuperCoolFeature = TRUE
```

- Alternatively, if you determine that you need isolation per application, you may use

```
HKCU\MyCompany\MyComponent\VersionXXYY\SomeApplication\
```

where "SomeApplication" is the return value of **GetModuleFileName**. Doing so enables a component to isolate its settings to only the application under which it is currently running.

- Ideally, you should support a persistence model so that the application takes responsibility for persisting your state and doesn't alter the registry. An application should not need to directly touch the component's registry entries. Instead, the component should offer APIs that save or restore settings that are side-by-side compatible.
- For interaction with a global state, settings stored in other places besides the registry should be stored in a side-by-side manner. Such stores include:
  - A protected store (pstore)
  - A WinInet cache
  - A Microsoft SQL Server™ or Microsoft Jet database

## Installing Side-by-Side Components Before You Install

Before you install side-by-side components, you must determine whether or not they are supported on your operating system. The following code detects whether or not side-by-side sharing is available. If it is not, components must be installed in the system directory.

```
BOOL bPlatformSupportsSideBySide(void)  
{  
    OSVERSIONINFOEX osvix ;  
  
    osvix.dwOSVersionInfoSize = sizeof(OSVERSIONINFOEX);  
  
    // If platform does not support OSVERSIONINFOEX structure, it does not  
    // have side by side support  
    // In the kernel we have made these modifications hand-in-hand...  
    //  
    if (!GetVersionEx((OSVERSIONINFO *)&osvix))  
    {  
        return FALSE ; // no DLL redirection  
    }  
  
    // For the other platform Ids, assume has side-by-side support  
    return TRUE ;  
}
```

# Implementing Side-by-Side Component Sharing in Applications (Expanded)

David D'Souza, BJ Whalen, and Peter Wilson  
Microsoft Corporation

}

## Installing and Uninstalling

It is critical that you install and uninstall your component correctly. Ideally, there should be no install and uninstall procedure for your component other than copying it into the application directory or deleting it from the application directory. However, if you need to do COM registration or any other initial setup of your component, then you must do it in a way that is side-by-side compatible.

Windows 2000 includes Windows Installer version 1.1, which will support install and uninstall of side-by-side components. (Windows Installer will also be available online after the release of Windows 2000.) When registering your side-by-side COM component, you need to make sure the Attributes column in the Class table has the `msidbClassAttributesRelativePath` bit set. This will register the component with a relative path name, allowing multiple copies of the same component to exist.

It is important to remember that while your component is private in the application directory, some other application may have installed a different version on this machine. While installing or uninstalling this component, you do not want to do anything that will be disruptive to other applications. Thus, the application using your component will depend on you to install it properly via the self-registration entry points `DLLRegisterServer` or `DLLUnregisterServer` (for COM components) or `DllInstall` (for Win32 or COM components). For more information on these functions, see "Register Server" in the Platform SDK.

To properly install your component in the application directory, install as you would a regular component, with the following additional steps:

1. When you register GUIDs, make sure they have relative path names.
2. Make sure to have a reference count on your GUID. This will help you track how many times you have installed or uninstalled this GUID.
3. If the GUID exists, increment your reference count.  
Or  
If it doesn't exist, you need to add the GUID and put in a reference count of one. For example:

```
{00000109-0000-0010-8000-00AA006D2EA4}  
\InprocServer32  
Default = "mycomponent.dll"  
ReferenceCount=1
```

Note: Type libraries should be contained within your DLL, and need not be registered in the system registry.

To properly uninstall your component in the application directory, install as you would a regular component, with the following additional step:

- Decrement your reference count. If it reaches 0, you know you can delete the GUID because there are no other users. If it is greater than 0, another application is installed on the system and is depending on the registry state.

# Implementing Side-by-Side Component Sharing in Applications (Expanded)

David D'Souza, BJ Whalen, and Peter Wilson  
Microsoft Corporation

## DLL/COM Redirection

DLL/COM redirection requires that when the application is deployed, the application executable and all isolated components be installed into the application directory rather than the system directory. Additionally, a ".local" file is deployed to the application directory to modify the Windows binding behavior, so that the application binds to the isolated components rather than the globally shared version.

Thus, the application will use a component that can run safely side by side with a different version of the same component that is installed elsewhere, in another application directory or in the system directory. If another application on the system requires a different version, your application is unaffected and both applications will run with their respective versions of the component.

In the event that another application installs a new version of a component on the system, the application's version of that component will remain untouched, since you installed it into your application directory. Your application continues to use the same version of the component that you shipped with your application, while the other application uses its version. The operating system can load both versions in memory at once.

**Note:** Isolated COM components must be registered properly with the operating system so that each version of the component won't conflict with other versions of the component that may exist. This registration requires that while the implementation of the component can change between versions, such registered COM meta-data as CLSID, ProgID, Type Library, and Threading Model cannot change between versions.

**Note:** Windows 2000 and Windows 98 Second Edition both support DLL/COM redirection. It is not supported for previous Windows operating systems.

## Using DLL/COM Redirection

DLL/COM redirection allows the developer or the administrator to selectively isolate existing components to the application they are building and deploying. This section discusses how to activate DLL/COM redirection, and how to select which components to isolate.

## Activating DLL/COM Redirection

DLL/COM redirection is activated on an application-by-application basis by the presence of a ".local" file. The ".local" file is an empty file in the same directory as the application's .exe file, with the same name as the application's .exe file with ".local" appended to the end of the name.

For example, to activate DLL/COM redirection for an application called "myapp.exe," create an empty file called "myapp.exe.local" in the same directory where myapp.exe is installed.

Once DLL/COM redirection is activated, whenever the application loads a DLL or an OCX, Windows looks first for the DLL or OCX in the directory where the application's .exe file is installed. If a version of the DLL or OCX is found in the directory where the application's .exe file is installed, the application uses it regardless of any directory path specified in the application or the registry. If a version of the DLL or OCX is not found in the directory where the application's .exe file is installed, the normal search path or server path is used.

# Implementing Side-by-Side Component Sharing in Applications (Expanded)

David D'Souza, BJ Whalen, and Peter Wilson  
Microsoft Corporation

## Selecting Components to Isolate

DLL/COM redirection allows the isolation of existing components where applications installed on a computer require different versions of the same component. No code changes are required to the component since, once activated, DLL/COM redirection changes the Windows binding behavior.

Up to now, however, executing different versions of components side by side has not typically been a design consideration. Whilst components can easily be installed side by side (installed in a shared location and isolated to one or more applications), they may not run side by side. This happens because some components use global state (such as settings stored in the registry), assuming that there will be only one version of the component on the computer at any time. Additionally, the component may make assumptions about the specific directory in which it is installed when locating other resources that it needs.

For this reason it is imperative to test an application that uses isolated components both installed on their own and installed in the context of the other applications from which the components are isolated. Microsoft's experience has indicated that in most scenarios commonly shared components can run side by side, but in some cases it may be necessary to close one application before running the next. The following guidelines should be followed when selecting components to isolate:

- Do not attempt to isolate any of the files protected by System File Protection that ship with Windows 2000, including most .sys, .dll, .exe, and .ocx files.
- You must test all applications to ensure side-by-side validity, especially in areas where sharing can occur, since there is no side-by-side enforcement by current operating systems.
- Beware of the loss of the ability to apply quick fixes to components, since they are now in arbitrarily located application directories. As an administrator, you need to know all of the places where you need to fix your component.

## Scenario I: Privatizing ActiveX Controls to an Application

In this scenario an administrator is unable to deploy a new application because the new application uses a version of an ActiveX control authored in Visual Basic that is different from the version required by currently deployed applications.

Over time, bug fixes and other modifications to the ActiveX control have introduced semantic differences such that applications that use the control are broken if the version with which they were tested is not used. The administrator needs to be able to run different versions of the ActiveX control with different applications that use it, avoiding having to fix and retest every application that would be affected by changing the ActiveX control.

Note In Visual Basic there is currently no easy way for developers to write inherently side-by-side ActiveX controls. This is because the registration of Visual Basic-authored ActiveX controls writes the fully qualified path to the OCX file into the registry when the control is registered.

The administrator is able to enforce the new application's use of the correct version of the ActiveX control, and make sure that the configuration of the existing applications is unchanged, by modifying the setup of the new application to:

- Install the new version of the ActiveX control in the directory where the application's .exe file is located.

# Implementing Side-by-Side Component Sharing in Applications (Expanded)

David D'Souza, BJ Whalen, and Peter Wilson  
Microsoft Corporation

- In the directory where the application's .exe file is located, install a .local file specifying that whenever this application is run, the ActiveX control should be loaded from the directory where the application's .exe file is located.

## Scenario II: Privatizing Win32 DLLs to an Application

In this scenario the administrator learns that an existing application stops working after a new application is deployed. The administrator is able to diagnose that the problem is caused by modifications to a shared component that result in the new version of the shared component not supporting backwards compatibility with the previous version.

The administrator is able to fix the application by performing the following steps:

- Installing the previous version of the shared DLL into the directory where the existing application's .exe file is located.
- Creating a .local file in the directory where the existing application's .exe file is located. The .local file specifies that when the application is run, DLLs found in the directory where the application's .exe file is located should be loaded from there.

## Considerations When Installing Isolated COM Servers

DLL/COM redirection is achieved by installing the DLL or OCX file in a new location, private to the application. However, other system state associated with the COM Servers will not be isolated, which has some specific implications for isolating COM Servers.

When installing an isolated COM Server, care should be taken to ensure that if any version of the component is already installed on the computer (by another application, for example), the InprocServer file location is not overwritten with the new location of the isolated component. For isolated COM Servers, the InprocServer file location is ignored at runtime. However, existing applications not using DLL/COM redirection require that the InprocServer file location continue to specify the location of the previously installed COM Server. The implication of this is as follows:

- When installing an isolated COM Server, if any version of the COM Server is already installed on the computer, do not register the isolated COM Server at install time.

Conversely, if a version of the isolated COM Server is not already installed on the computer, it must be registered. The problem scenario here occurs when a COM Server is isolated to an application, is installed in the application's .exe directory, and then later nonisolated applications are installed that require that component. In this case, it is unlikely that the uninstall for the isolated application will treat isolated COM components as shared files, and the uninstall will break other applications. The implication of this is as follows:

- When installing an isolated COM Server, if no version of the COM Server is already installed on the computer, copy the DLL or OCX file to both the application's .exe directory and the system directory (or some other shared location), and register the copy in the system directory (or the other shared location).

For existing components, where versions may be shared by some applications and other versions may be privatized to others, a reasonable rule of thumb is as follows: After installing applications that use isolated versions of potentially shared components, ensure that both a shared version and an

# Implementing Side-by-Side Component Sharing in Applications (Expanded)

David D'Souza, BJ Whalen, and Peter Wilson  
Microsoft Corporation

isolated version of the component are installed, and that the shared version is registered. Doing this allows uninstallers to remove isolated versions without fear of breaking other applications.