Best Practices for WOW64

June 17, 2010

Abstract

The PC industry is gradually transitioning from an installed base of primarily 32-bit systems to one of primarily 64-bit systems. In the interim, many application developers will continue to build 32-bit versions of their applications or will provide both 32-bit and 64-bit versions. To ease the effort that is involved in porting applications and to help encourage adoption of 64-bit computing, Microsoft provides Windows® 32-bit On Windows 64-bit (WOW64), an emulation layer that enables 32‑bit Windows-based applications to run seamlessly on 64-bit Windows. This paper describes best practices for building applications that run on WOW64.

This information applies to 64-bit editions of the following operating systems:  
 Windows 7  
 Windows Server® 2008 R2  
 Windows Server 2008  
 Windows Vista®  
 Windows Server 2003  
 Windows XP

References and resources discussed here are listed at the end of this paper.

The current version of this paper is maintained on the Web at:  
 <http://www.microsoft.com/whdc/system/platform/64bit/WoW64_bestprac.mspx>

**Disclaimer**: This document is provided “as-is”. Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes.

© 2010 Microsoft Corporation. All rights reserved.

Document History

| **Date** | **Change** |  |  |  |
| --- | --- | --- | --- | --- |
| June 17, 2010 | Corrected typos in sample code | | | |
| November 2, 2009 | Added information about Sysnative path forms, references to files and variables, DLL binding, and sample code. | | | |
| May 8, 2009 | Updated for Windows 7 and Windows Server 2008 R2. Registry reflection has been removed and redirected keys have changed. | | | |
| July 11, 2008 | Updated for Windows Vista | | | |
| May 17, 2006 | First publication | | | |

Contents

[Introduction 3](#_Toc264472143)

[Limitations of WOW64 4](#_Toc264472144)

[Registry Redirection and Registry Reflection 4](#_Toc264472145)

[Registry Redirection 4](#_Toc264472146)

[Registry Reflection 5](#_Toc264472147)

[File System Redirection 6](#_Toc264472148)

[Enabling and Disabling File System Redirection 6](#_Toc264472149)

[References to Files and Variables 7](#_Toc264472150)

[Binding to 32-bit and 64-bit DLLs 9](#_Toc264472151)

[Application Installation and Startup 10](#_Toc264472152)

[Installing Applications 10](#_Toc264472153)

[Starting Applications 11](#_Toc264472154)

[Sample Code 12](#_Toc264472155)

[Kernel-Mode Drivers 12](#_Toc264472156)

[Best Practices 13](#_Toc264472157)

[Resources 14](#_Toc264472158)

# Introduction

The PC industry is gradually transitioning from an installed base of primarily 32-bit systems to one of primarily 64-bit systems. In the interim, many application developers will continue to build 32-bit versions of their applications or will provide both 32-bit and 64-bit versions. To ease the effort that is involved in porting applications and to help encourage adoption of 64-bit computing, Windows® provides Windows 32-bit On Windows 64-bit (WOW64), an emulation layer that enables 32-bit Windows-based applications to run seamlessly on 64-bit Windows. This paper describes best practices for building applications that run on WOW64.

At a high level, WOW64 is a collection of user-mode DLLs that intercepts calls to and from 32-bit processes and translates them. Figure 1 shows how a 32-bit process runs on 64-bit Windows.

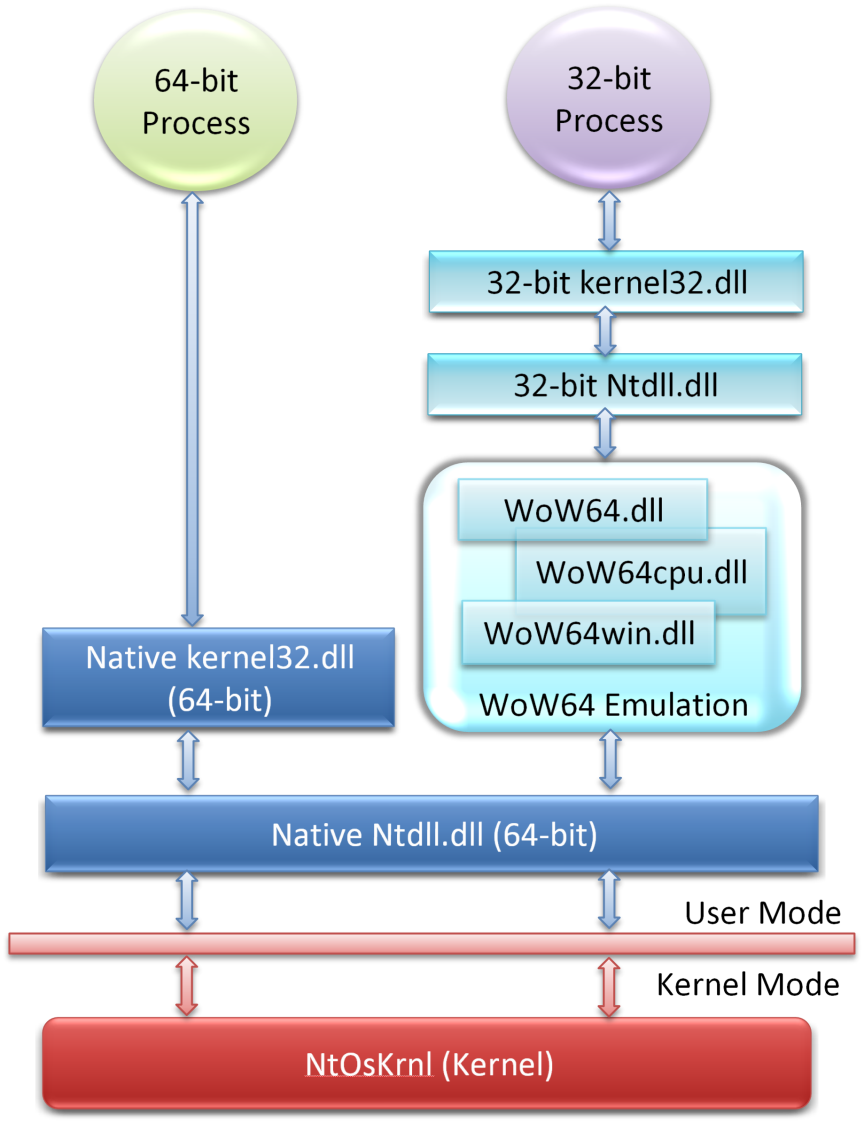


Figure 1. How a 32-bit process runs on 64-bit Windows

When a 32-bit application runs, the native library loader starts first. It recognizes that the executable image is a 32-bit process and handles it in a special manner. The native loader sets up a WOW64 emulation environment for 32-bit processes and transfers control to the 32-bit loader (in the 32-bit Ntdll.dll).

The WOW64 emulation layer runs between the 32-bit application and the native (64‑bit) Ntdll.dll and translates the application’s calls to the 32-bit Ntdll.dll to calls to the 64-bit Ntdll.dll. Any return calls are similarly translated and passed to the 32-bit application.

**Note** This paper uses the term bitness to refer to the distinction between 32-bit and 64-bit hardware and the potential related differences in applications, processes, and similar components.

# Limitations of WOW64

WOW64 has the following limitations:

* The address space is limited to 2 GB by default, and 4 GB if /LARGEADDRESSAWARE is used. For more information, see “Memory Limits for Windows Releases” on the MSDN® website.
* A 32-bit process cannot load a 64-bit DLL (except for certain system DLLs).
* Running 16-bit processes is not supported. For information on 16-bit installer programs, see “Running 32-bit Applications” on the MSDN website.
* The Virtual DOS Machine (VDM) API is disabled.
* Page-size dependent APIs such as Address Windowing Extension (AWE), scatter/gather I/O, and write tracking are not available on the Intel Itanium processor family (IPF). For more information, see “Running 32-bit Applications” on the MSDN website.
* The physical address extension (PAE) API is not available on IPF.
* Microsoft® DirectX® hardware acceleration APIs are not supported on IPF.

# Registry Redirection and Registry Reflection

Registry redirection and registry reflection enable 32-bit code to access the appropriate registry data on 64-bit machines. Redirection splits the registry into 32‑bit and 64-bit nodes. Versions of Windows earlier than Windows 7 and Windows Server® 2008 R2 support registry reflection for certain redirected registry keys. Reflection synchronizes the values of keys across the two registry views.

This section briefly describes these two features and suggests solutions for applications to avoid maintaining separate 32-bit and 64-bit registry data.

## Registry Redirection

Many applications store 32-bit specific or 64-bit specific configuration information in the registry. To allow separate configuration of native and WOW64 applications, the registry is split at important nodes into a 32-bit part and a 64-bit part. The registry redirection mechanism allows a 64-bit process to see the native registry view and a 32-bit process to see the 32-bit registry view. This separates the registry data of 32‑bit and 64-bit applications, so that they can run and safely interoperate without conflicts. For more information, see “Running 32-bit Applications” on the MSDN website.

By default, a 32-bit application that is running on WOW64 accesses the 32-bit registry view and a 64-bit application accesses the 64-bit registry view. The KEY\_WOW64\_32KEY flag enables a 64-bit application to access reflected keys in the 32-bit registry view, and KEY\_WOW64\_64KEY enables a 32-bit application to access reflected keys in the 64-bit registry view. Applications should not use **Wow6432Node** in a registry key path because this name is reserved for use by Windows and might change in the future.

In some situations, however, separate 32-bit and 64-bit registry data is not desirable. You can use one of the following approaches to ensure correct operation without maintaining separate registry data:

* Ensure that all readers and writers disable redirection before they access this data and use the KEY\_WOW64\_64KEY flag to target the 64-bit version of the data. You can safely use this flag on 32-bit versions of Windows, which ignore it. If you create a new registry key in a redirected location, this is the preferred approach. By using the flag, you can target the 64-bit data.
* Let writers target the bitness-specific location, but ensure that readers specify the flags, read both locations, and choose which one to use. Both 32-bit and 64‑bit implementations of DLLs must support the same behavior. To use this approach effectively, you must design the schema of the registry data to use keys to distinguish the data and to allow for merging.

If your application previously used the second approach, you can migrate to the first approach by following these guidelines:

* Migrate the existing data to the new location during application installation and setup.
* Ensure that no callers write data to the wrong location. The easiest way is to ensure that callers use an API to write to the registry instead of writing it directly.

Windows 7 and Windows Server 2008 R2 redirect a different group of keys than earlier versions of Windows. For a list of redirected keys, see “Registry Redirector” on the MSDN website.

## Registry Reflection

In versions of Windows earlier than Windows 7, some redirected registry keys were reflected as this section describes. In Windows 7 and Windows Server 2008 R2, no keys are reflected. Formerly reflected keys are now shared and are visible to both the 32-bit and 64-bit registry hives. For more information, see “Removal of Windows Registry Reflection” on the MSDN website.

The following information applies only to versions of Windows earlier than Windows 7.

To prevent registry key collisions, WOW64 presents 32-bit applications with an alternative view of the registry by splitting key portions of the registry into a 32-bit version and a 64-bit version. In spite of this separation of the registry, applications must be aware of the different views of the registry. Registry reflection is a mechanism to keep specific registry keys and values synchronized by copying them intelligently between the two registry views. For more information, see “Running 32‑bit Applications” on the MSDN website.

Some applications require both 32- and 64-bit versions to access the same registry data. To emulate the automatic bidirectional reflection of keys that are not reflected, an application can create a symbolic link between two registry keys. A symbolic link in the registry provides a shortcut to another key. All symbolic links address a single physical key and a change to any one symbolic link reflects instantaneously in all the linked keys.

An application can use this behavior to emulate only *full* *reflection* of a key. Imagine that an application ‘*HelloWorld’* uses a key K under **HKEY\_LOCAL\_MACHINE  
\Software\HelloWorld**. Because **HKLM\Software** is split into a 32-bit and 64-bit view, let us denote the key in the 32-bit view to be K32 and that in the 64-bit hive to be K64. If the key K contains the same value regardless of the application type (native or WOW64), then a symbolic link from K32 to K64 allows a single copy of the key to be accessed. This allows better interoperability and management of the application.

# File System Redirection

File system redirection enables 32-bit code that uses file names or paths that refer to 64-bit data or modules to instead be redirected to 32-bit data or modules.

A 32-bit process cannot load a 64-bit DLL and, similarly, a 64-bit process cannot load a 32-bit DLL. Because the Windows system folder contains both the installed applications and their DLLs, it must be separated into a native system folder for the 64-bit applications (%windir%\System32) and a WOW64 folder for the 32-bit applications (%windir%\Syswow64).

Developers often hard code the system folder path name in their applications. Therefore, to preserve application compatibility, the 64-bit system folder is still called System32. To enable 32-bit applications that have hard-coded paths to transparently access the WOW64 system directory, the WOW64 layer provides a file system redirector.

All accesses that a WOW64 process makes to the **%**windir**%**\System32 directory are redirected to the **%**windir%\Syswow64 directory. Therefore, with file system redirection enabled, a 32-bit application accesses the same contents for both the System32 and Syswow64 directories.

## Enabling and Disabling File System Redirection

File system redirection is enabled for all WOW64 applications by default.

To disable file system redirection on a per-thread basis, use the **Wow64DisableWow64FsRedirection** function. To revert to the default behavior, use the **Wow64RevertWow64FsRedirection** function.

Redirection applies only to the thread that called the function to enable or disable it. Therefore, APIs that execute their code on a different thread are not affected by disabling redirection. As a general rule, an application should disable redirection only around calls to **CreateFile** and should never disable it before calling any other APIs*.*

In Windows Vista® and later versions of Windows, the 32-bit application should start the application from %windir%\Sysnative instead of %windir%\System32 to avoid redirection. The Sysnative alias indicates to WOW64 that the file system should not redirect the access.

Applications such as virus scanners typically disable file system redirection to access the contents of the native system directory. We urge developers to exercise caution because if redirection is not reenabled in time, subsequent calls that depend on redirected paths fail. For example, with file system redirection disabled, an API that tries to load a Syswow64 (32-bit) DLL loads the 64-bit version from the System32 directory and fails. Furthermore, DLLs are not always loaded explicitly. One DLL can have dependencies on other DLLs, or Win32® or a third-party function can load a DLL on behalf of an application. In any of these situations, an application could depend on a redirected path and fail.

## References to Files and Variables

Applications must ensure that references to files and variables work independent of the bitness of the operating system. System variables, [environment variables](http://msdn.microsoft.com/en-us/library/aa384274(VS.85).aspx), KNOWNFOLDERID values (called CSIDL values on versions earlier than Windows Vista), and file system path forms that include the Sysnative alias can cause the following problems:

* Some variables work only on 64-bit versions of Windows.

For example, %ProgramW6432% and FOLDERID\_ProgramFilesX64 are not supported on 32-bit operating systems. Applications must test the bitness of the operating system before they use such variables.

* Some variables are a function of the bitness of the caller and therefore evaluate differently when called from 32-bit and 64-bit processes. This behavior supports separation of data by bitness.

For example, installation programs typically use %ProgramFiles% or FOLDERID\_ProgramFiles to determine where to install the application. Dual-bitness applications can use these variables appropriately to reference those programs, but they do not work for 32-bit-only or 64-bit-only applications.

* Some variables work only if the process is 64-bit.

For example, FOLDERID\_ProgramFilesX64 does not work for 32-bit callers. In versions of Windows earlier than Windows 7, %ProgramW6432% did not work in the context of 32-bit processes. An application must determine whether it is running in a 64-bit process before it uses these variables.

* Some variables work only in 32-bit processes.

For example, the %windir%\Sysnative path form enables 32-bit applications to target 64-bit files in the system directory.

Do not use these variables to create references to files, shortcuts (.lnk), or locations that must work independent of the bitness of the caller or must work on a 32-bit version of Windows.

References to registry data that uses environment variables are similarly problematic. Normally REG\_EXPAND\_SZ indicates that registry data contains an environment variable, but REG\_SZ values sometimes contain variables that clients manually expand by using **ExpandEnvironmentStrings** and similar system functions. When an application calls **RegGetValue**, Windows by default expands the values of environment variables according to the bitness of the caller, but the application can override this default in the call.

On 64-bit versions of Windows, use variables that evaluate the same regardless of the bitness of the caller. Avoid using the following variables to create a reference to an application that does not support both 32-bit and 64-bit:

* %ProgramFiles%, FOLDERID\_ProgramFiles, FOLDERID\_ProgramFilesCommon, and the older CSIDL\_PROGRAM\_FILES and CSIDL\_PROGRAM\_FILES\_COMMON.

Windows expands these variables according to the bitness of the caller, as described previously. Instead of %ProgramFiles%, use %ProgramW6432% or do not use an environment variable at all.

* FOLDERID\_ProgramFilesX64, FOLDERID\_ProgramFilesCommonX64.

These variables work only in 64-bit processes.

For a list of the environment variables that WOW64 defines, see “WOW64 Implementation Details” on the MSDN website.

Table 1 summarizes the issues that are involved in the references to files and variables, and Table 2 lists the default locations to which commonly used paths and variables point.

Table 1. Summary of Rules for Referencing Files and Variables

| **Application Bitness** | **32-bit Windows** | **64-bit Windows** |
| --- | --- | --- |
| 32-bit | Do not use variables that exist only on 64-bit platforms. | * Do not use variables that vary in evaluation based on caller’s bitness. * Do not use variables that work only in the context of a 64-bit caller. |
| 64-bit | Not supported. | * Do not use variables that vary in evaluation based on caller’s bitness. * Do not use paths that include Sysnative. |

Table 2. Summary of Defaults for Variables and Paths

|  |  |  |  |
| --- | --- | --- | --- |
| Type of reference | 32-bit caller on 32-bit hardware | 32-bit caller on  64-bit hardware | 64-bit caller on  64-bit hardware |
| %ProgramFiles% | C:\Program Files | C:\Program Files (x86) | C:\Program Files |
| %ProgramFiles(x86)% | Not supported | C:\Program Files (x86) | C:\Program Files (x86) |
| %ProgramW6432% | Not supported | C:\Program Files | C:\Program Files |
| %CommonProgramFiles% | C:\Program Files  \Common Files | C:\Program Files (x86)  \Common Files | C:\Program Files  \Common Files |
| %CommonProgramFiles(x86)% | Not supported | C:\Program Files (x86)  \Common Files | C:\Program Files (x86)  \Common Files |
| %CommonProgramW6432% | Not supported | C:\Program Files  \Common Files | C:\Program Files  \Common Files |
| FOLDERID\_ProgramFiles, CSIDL\_PROGRAM\_FILES | C:\Program Files | C:\Program Files (x86) | C:\Program Files |
| FOLDERID\_ProgramFilesX86, CSIDL\_PROGRAM\_FILESX86 | C:\Program Files | C:\Program Files (x86) | C:\Program Files (x86) |
| FOLDERID\_ProgramFilesX64 | Undefined | C:\Program Files | C:\Program Files |
| FOLDERID\_ProgramFilesCommon, CSIDL\_PROGRAM\_FILES\_COMMON | C:\Program Files  \Common Files | C:\Program Files (x86)  \Common Files | C:\Program Files  \Common Files |
| FOLDERID\_ProgramFilesCommonX86, CSIDL\_PROGRAM\_FILES\_COMMONX86 | C:\Program Files  \Common Files | C:\Program Files (x86)  \Common Files | C:\Program Files (x86)  \Common Files |
| FOLDERID\_ProgramFilesCommonX64 | Undefined | C:\Program Files  \Common Files | C:\Program Files  \Common Files |
| Sysnative | Not defined | C:\Windows  \System32 | Not defined |

## Binding to 32-bit and 64-bit DLLs

If you use DLLs to which both 32-bit and 64-bit clients must bind, you must manage the references to the DLLs to ensure that binding succeeds. The following are possible solutions to this problem:

* Use a binding technology such as COM that automatically binds to the correct bitness. This is the preferred solution.
* Deploy the DLLs in a folder (usually under \Program Files), and add that folder to the PATH environment variable. As a general rule, approaches that rely on environment variables for code binding are more likely to encounter problems than the use of COM or other solutions that do not. However, if you must use this approach, follow these guidelines for naming the DLLs:

Name the 32-bit and 64-bit DLLs differently. This enables the loader to target the right version of the DLL based on unique names.

Use the same DLL name for both the 32-bit and 64-bit DLLs. Store them in different folders, and add both folders to the PATH variable. The loader skips DLLs that have the wrong bitness and therefore binds to the correct library.

* Place the DLLs in the appropriate system directories: 64-bit DLLs in C:\Windows\System32\Dll.dll and 32-bit DLLs in C:\Windows\Syswow64\Dll.dll. The file system redirector can then choose the right DLL for clients of the different bitness. However, as a general rule, applications should not store DLLs in the system directories. Instead, applications should use the Program Files directory to keep their state information separate from that of the system.

# Application Installation and Startup

One key to ensuring that your applications operate properly is to install them in the appropriate location. The installation location, the bitness of the application, and the bitness of the process that starts the application together determine whether Windows applies file system redirection at startup.

## Installing Applications

Applications can be deployed on 64-bit versions of Windows in any of three ways:

* 32-bit application.

Install 32-bit applications in C:\Program Files (x86).

* 64-bit application.

Install 64-bit applications in C:\Program Files.

* Dual-bitness application.

Install 32-bit and 64-bit versions of the application in the appropriate locations as listed in the previous bullets.

Dual-bitness deployment supports applications that must be compatible with both 32-bit and 64-bit in-process extensions, such as Microsoft Internet Explorer, Windows Media Player and Windows Explorer. If Microsoft supplies the application with Windows, the 64-bit version of the application is stored in C:\windows\system32 and the 32-bit version is stored in C:\windows\syswow64.One such example of this is Notepad.exe.

Because Windows maintains separate folders for 32-bit and 64-bit applications only in Program Files, dual-bitness applications cannot rely on the system to put binaries of different bitness in different folders on a per-user basis. The application installation or deployment procedure must either create separate locations for the binaries or give the binaries different names.

Applications that have both 32-bit and 64-bit components can store their data in either the 32-bit or 64-bit location. Within this location, the application must store its different bitness components in separate subfolders. Visual C++ is an example of such an application.

Applications that have both 32-bit and 64-bit components side by side are difficult to install and uninstall cleanly. We recommend that both types of components be installed at the same time, if possible.

If a 64-bit application is being installed and a 32-bit version of the same application is already installed on the system, the newly installed 64-bit application is likely to overwrite existing configuration data. To avoid problems, the 64-bit installer (and uninstaller) must be able to detect the presence of an older version and store its own data in separate locations for compatibility.

## Starting Applications

Table 3 summarizes the behavior when an application starts a 32-bit application, a 64-bit application, or an interpreted file that is installed in the System32 or Syswow64 directory.

Table 3. Behavior of Applications and Interpreted Files at Process Startup

| **Bitness of running application** | **32-bit application** | **64-bit application** | **Interpreted file** |
| --- | --- | --- | --- |
| 32-bit application | Both processes run on WOW64. No applica­tion changes are required. | In Windows Vista and later versions of Windows, the 32-bit application should start the application from %windir%\Sysnative.  In earlier releases, the 32-bit application must disable redirection, start the 64-bit application, and reenable redirection. However, note that we do not recommend starting processes while redirection is disabled. | The 32-bit version of the interpreter is started. The interpreter and any processes that the script starts are run on WOW64. |
| 64-bit application | File system redirection does not apply. The appli­cation must be aware of the SysWow64 directory. | Both processes are native. No application changes are required. | The 64-bit version of the interpreter is started. |

As the table shows, Windows applies files system redirection when a 32-bit application accesses an application that is stored in the system directory (C:\Windows\System32). To ensure that applications in the system directory work independent of the bitness of the caller, consider designing such applications to support both bitnesses. The Microsoft-supplied Notepad.exe application is an example.

If you cannot provide a dual-bitness application, you can bypass redirection in either of the following ways:

* Move the application out of the system directory so that redirection does not occur. Only system components should be stored in the Windows\System32 directory.
* Use the Sysnative path form to bypass redirection when a 32-bit process that runs on 64-bit Windows references a specific program in the system directory. For example, Windows does not use redirection when a 32-bit process specifies a Sysnative path in a call to **CreateProcess** or **ShellExecute**. However, such applications must determine the bitness of the process at runtime, because Sysnative works only in 32-bit callers on 64-versions of Windows.

## Sample Code

This following code sample shows a helper function to determine the bitness of the process and the system.

\_\_inline bool Is32BitProcessRunningOnWow64()

{

#ifdef \_WIN64

// 64-bit code, obviously not running in a 32-bit process

return false;

#else

BOOL fWow64Process;

return IsWow64Process(GetCurrentProcess(), &fWow64Process) &&   
 IfWow64Process;

#endif

}

The next code sample shows how to use the Is32BitProcessRunningOnWow64 function (which is defined in the previous sample) with a Sysnative path to start the appropriate version of Notepad.exe.

// Start the platform native version of notepad in a way that

// works from 64-bit or 32-bit code.

// - on 64-bit OS 64-bit notepad

// - on 32-bit OS 32-bit notepad

wchar\_t szNotepadPath[MAX\_PATH];

ExpandEnvironmentStrings(Is32BitProcessRunningOnWow64() ?

L"%windir%\\sysnative\\notepad.exe" :

L"%windir%\\system32\\notepad.exe", szNotepadPath,  
 ARRAYSIZE(szNotepadPath));

ShellExecute(NULL, NULL, szNotepadPath, NULL, NULL,   
 SW\_SHOWNORMAL);

The following code sample shows how to get a string that identifies the native Program Files folder in either 32-bit or 64-bit code.

// Get the platform native "Program Files" in a way that works

// from 64-bit or 32-bit code

wchar\_t szProgramFiles[MAX\_PATH];

ExpandEnvironmentStrings(Is32BitProcessRunningOnWow64() ?

L"%ProgramW6432%" : L"%ProgramFiles%", szProgramFiles,  
 ARRAYSIZE(szProgramFiles));

Finally, this example shows how to get the native Program Files folder as a shell item from 64-bit code.

// Get the platform native version of Program Files as a shell item

// but account for the fact that this code is only supported from

// 64-bit callers on a 64-bit system.

#ifdef \_WIN64

IShellItem \*psiPrograms;

SHGetKnownFolderItem(FOLDERID\_ProgramFilesX64,

KF\_FLAG\_DEFAULT\_PATH, NULL, IID\_PPV\_ARGS(&psiPrograms));

#endif

# Kernel-Mode Drivers

A 32-bit driver cannot run on the 64-bit Windows kernel, so it must be ported. For more information, see “Porting Your Driver to 64-Bit Windows” in the Windows Driver Kit (WDK) and the Porting Issues Checklist in that reference.

You should take special care when you port I/O dispatch routines because 64-bit device drivers must support I/O requests from 32-bit applications. WOW64 does not translate the I/O control (IOCTL) path because it does not have enough information to interpret the data buffer. The driver must be able to handle both 32-bit and 64-bit data structures in its IOCTL path.

A kernel-mode driver can determine whether it is being invoked in the context of WOW64 or of a native process by using **IoIs32bitProcess**. In addition to this device driver interface (DDI), two user-mode functions in Kernel32.dll can detect WOW64 processes: **IsWow64Process** and **GetNativeSystemInfo**.

# Best Practices

The following are best practices:

* File system redirection is on a per-thread basis. Therefore, isolate operations that require disabling redirection in a separate thread.
* Reenable redirection as soon as possible after performing the task.
* Be aware of interoperability when you install a 64-bit process alongside its 32-bit version.
* When using interprocess communication methods such as sockets, pipes, remote procedure call (RPC), and COM, test for bit-awareness in the way that you handle data.
* Avoid accessing 64-bit processes from 32-bit processes.

# Resources

#### WHDC

64-bit System Design nodal page

<http://www.microsoft.com/whdc/system/platform/64bit/default.mspx>

#### Windows Driver Kit (WDK)

Porting Your Driver to 64-Bit Windows

<http://msdn.microsoft.com/en-us/library/ff559747.aspx>

#### MSDN

Accessing an Alternate Registry View

<http://msdn.microsoft.com/en-us/library/aa384129(VS.85).aspx>

File System Redirector

<http://msdn.microsoft.com/en-us/library/aa384187(VS.85).aspx>

KNOWNFOLDERID

<http://msdn.microsoft.com/en-us/library/bb762584(VS.85).aspx>

Memory Limits for Windows Releases

<http://msdn.microsoft.com/en-us/library/aa366778.aspx>

Programming Guide for 64-bit Windows

<http://msdn.microsoft.com/en-us/library/bb427430(VS.85).aspx>

Removal of Windows Registry Reflection

<http://msdn.microsoft.com/en-us/library/dd464643(VS.85).aspx>

Registry Redirector

<http://msdn.microsoft.com/en-us/library/aa384232(VS.85).aspx>

Running 32-bit Applications

[http://msdn.microsoft.com/en-us/library/aa384249.aspx](http://msdn.microsoft.com/en-us/library/aa384249.aspx%20)

WOW64 Implementation Details

<http://msdn.microsoft.com/en-us/library/aa384274(VS.85).aspx>