

# Introducing DirectX Foundation

Microsoft® DirectX® Foundation is a set of low-level application programming interfaces (APIs) for creating games and other high-performance multimedia applications. It includes support for two-dimensional (2-D) and three-dimensional (3-D) graphics, sound and music, input, force feedback, and network communication for applications such as multiplayer games.

This overview gives general introductory information about DirectX Foundation. Information is divided into the following sections:

- About DirectX Help
- What's New in DirectX Foundation?
- Components of DirectX Foundation
- DirectX Samples
- DirectX Tutorials
- DirectX Tools
- C and C++ Programming Topics
- Visual Basic Programming Topics
- Further Information

## About DirectX Help

This documentation includes information about developing Microsoft® DirectX® applications in C, C++, and Microsoft® Visual Basic®.

DirectX fully supports both C and C++. Although this document most often uses the C++ syntax, some of the examples and tutorials (as well as some of the sample applications) are in C. In addition, the following topics provide information on using C with DirectX:

- Using Macro Definitions
- Accessing COM Objects by Using C

For the sake of simplicity, this documentation generally uses "C++" to mean "C or C++" when presenting information for those languages, as opposed to Visual Basic. You can assume that discussions of C++ concepts are also valid for C, with the necessary changes in syntax.

To provide only the information that is relevant to your programming environment, this document implements language filtering. On the title bar of many topics (including this one) you will find a button for a pop-up language menu. This menu gives you the choice of seeing documentation tailored either to C++ (and C) or to Visual Basic. If you choose **Show All**, you will see the language-specific information for both languages. Regardless of your language selection, you will always see information (such as general concepts) that is relevant to all languages.

---

[\[C++\]](#)

This text appears only if you have selected **C++** or **Show All** from the language menu.

---

[\[Visual Basic\]](#)

This text appears only if you have selected **Visual Basic** or **Show All** from the language menu.

---

The buttons that can appear on the title bar of topics are described as follows:

| Button | Meaning  |
|--------|--|
| L      | Shows when language filtering is in the topic. Select your programming language from those available on the menu.  |
| +      | Denotes that additional information is available and provides links to those topics. This is called the See Also section.  |
| !      | Shows the Requirements (previously known as QuickInfo) for using reference material. The information may include operating system versions and header information. |

The following conventions are used in the syntax of methods, functions, and other API elements, as well as typographic conventions used in explanatory material and sample code:

| Convention         | Meaning  |
|--------------------|--|
| <i>Italic text</i> | Denotes a placeholder or variable. You must provide the actual value. For example, the statement SetCursorPos( <i>X</i> , <i>Y</i> ) requires you to substitute values for the <i>X</i> and <i>Y</i> parameters. |
| <b>Bold text</b>   | Denotes a function, procedure, structure, macro, interface, method, data type, or other keyword in the programming interface or language.  |
| []                 | Encloses optional parameters.  |
| ...                | Specifies that the preceding item may be repeated.   |

|                           |  |
|---------------------------|--|
| <b>FULL BOLD CAPITALS</b> | Used for most type and structure names.                |
| <b>FULL CAPITALS</b>      | Used for enumeration values, flags, and constants.     |
| <b>monospace</b>          | Sets off code examples and shows syntax spacing.       |
| <b>.</b>                  | Represents an omitted portion of a sample application. |
| <b>.</b>                  |  |
| <b>.</b>                  |  |

## What's New in DirectX Foundation?

The following are some of the new features in DirectX 7.0:

- A new Visual Basic type library makes key DirectX features available to applications developed in that environment.
- Microsoft® Direct3D® Immediate Mode API offers many new features, including: device-state blocks, geometry blending, cubic environment mapping, user-defined clipping planes, and more. For details, see Immediate Mode Changes for DirectX 7.0.
- A new Direct3DX Utility Library is available now as a helper layer that sits on top of Direct3D and Microsoft® DirectDraw® to simplify common tasks encountered by 3-D graphics developers.
- Microsoft® DirectMusic® supports the Downloadable Sounds level 2 standard.
- Microsoft® DirectInput® now supports up to eight mouse buttons, exclusive access to the keyboard, and delayed start for force-feedback effects. New methods allow reading and writing of force-feedback effect files, and a Force Editor application is provided for creating effects.
- Improved hardware voice management is provided in Microsoft® DirectSound®.
- New 3-D audio processing algorithms result in better CPU utilization in DirectSound.
- DirectDraw stereo support has been added for active devices.
- Ripple launching is provided for Microsoft® DirectPlay® lobbied applications.
- Improved documentation includes dynamic filtering for language-specific information.
- More sample applications are provided.

If you need to refer to information provided with a previous release of DirectX, the legacy documentation is available for download from <http://www.microsoft.com/directx>.

## Components of DirectX Foundation

DirectX Foundation is made up of the following components:

- Direct3D provides a high-level Retained Mode interface that allows applications to easily implement a complete 3-D graphical system, and a low-level Immediate Mode interface that lets applications take complete control over the rendering pipeline. For more information about Immediate Mode, see [About Direct3D Immediate Mode](#). For more information about Retained Mode, which is not part of DirectX Foundation, see [Direct3D Retained Mode](#).
- Direct3D now includes a Direct3DX Utility Library that is a helper layer to simplify common tasks encountered by 3-D graphics developers. See [About the Direct3DX Utility Library](#).
- DirectDraw accelerates hardware and software animation techniques by providing direct access to bitmaps in off-screen display memory, as well as extremely fast access to the blitting and buffer-flipping capabilities of the hardware. For more information, see [About DirectDraw](#).
- DirectInput provides support for every kind of input and force-feedback device. For more information, see [About DirectInput](#).
- DirectMusic works with message-based data and provides a complete system for playing music and DLS-based sound effects with run-time composition and variation. For more information about this component of DirectX, see [About DirectMusic](#).
- DirectPlay makes connecting games over a modem link or network easy. For more information, see [About DirectPlay](#).
- DirectSetup provides a one-call installation procedure for DirectX. For more information, see [About DirectSetup](#).
- DirectSound enables playback of wave sounds and includes support for hardware and software mixing, 3-D positioning, and sound capture. For more information, see [About DirectSound](#).

## DirectX Samples

Studying code from working samples is one of the best ways to understand DirectX. Sample applications are located in the `Dxf\Samples\Multimedia` folder of the DirectX SDK or in the DirectX code samples under the Platform SDK References section.

A brief overview of the samples is provided under each component:

- Direct3D Immediate Mode Tools and Samples
- Direct3DX Utility Library Samples
- DirectDraw Tools and Samples
- DirectInput Tools and Samples

- DirectMusic Samples
- DirectPlay Tools and Samples
- DirectSetup Samples
- DirectSound Tools and Samples

## DirectX Tutorials

The tutorials found throughout the DirectX components guide you step by step through the basics of implementing each component. Following the code path and explanations can help you understand how to use the APIs and avoid some of the pitfalls along the way.

The tutorials are listed under each component:

- Direct3D Immediate Mode Tutorials
- DirectDraw Tutorials
- DirectInput Tutorials
- DirectMusic Tutorials
- DirectPlay Tutorials
- DirectSound Tutorials

## DirectX Tools

The following tools can be useful in developing and troubleshooting DirectX applications:

- DirectX Caps Viewer
- DirectX Diagnostic Tool
- DirectX Property Sheet

### See Also

- Direct3D Immediate Mode Tools
- DirectDraw Tools
- DirectInput Tools
- DirectPlay Tools

## DirectX Caps Viewer

## Description

The DirectX Caps Viewer tool enumerates devices and capabilities for all the components of DirectX.

## Path

Executable: (*SDK root*)\Bin\DXUtils\DXCapsViewer.exe

## User's Guide

Select items in the tree view. Information appears in the pane on the right.

# DirectX Diagnostic Tool

## Description

DirectX Diagnostic Tool gathers information on the system and the DirectX components installed on it, as well as providing a number of tests to ensure that components are working properly. A special version of this tool installed with the DirectX SDK allows developers to report problems directly to the DirectX development team.

## Path

Executable: (*SDK root*)\Bin\DXUtils\Dxdiag.exe

## User's Guide

DirectX Diagnostic Tool has its own Help file, which can be accessed by clicking the **Help** button.

# DirectX Property Sheet

## Description

This component of Control Panel can be used to obtain information about the DirectX components, drivers, and hardware features available on the system. It can also be used to set the amount of debugging output generated by the debug versions of the DirectX direct-link libraries (DLLs) and to disable hardware-acceleration features for testing purposes.

## Path

Double-click the DirectX icon in Control Panel.

## C and C++ Programming Topics

---

### [\[Visual Basic\]](#)

This section pertains only to applications written in C++. For an introduction to programming for DirectX in Visual Basic, see Visual Basic Programming Topics.

---

### [\[C++\]](#)

This section introduces some general topics that pertain to programming with any of the DirectX Foundation APIs. The following topics are discussed:

- The Component Object Model
  - Using Callback Functions
  - Version Checking
  - Compiling DirectX Samples and Other DirectX Applications
  - Debugging DirectX Applications
- 

## The Component Object Model

---

### [\[Visual Basic\]](#)

This topic pertains only to applications written in C++. For an introduction to programming for DirectX in Visual Basic, see Visual Basic Programming Topics.

---

### [\[C++\]](#)

Most of the DirectX API is composed of objects and interfaces based on COM. COM is the foundation of an object-based system that focuses on reuse of interfaces. It is also an interface specification from which any number of interfaces can be built.

A DirectX application is built from instances of COM *objects*. You can consider an object to be a black box that represents hardware or data that you access through *interfaces*. Commands are sent to the object through *methods* of the COM interface. For example, the **IDirectDraw7::GetDisplayMode** method of the **IDirectDraw7** interface is called to get the current display mode of the display adapter from the DirectDraw object.

Objects can bind to other objects at run time, and they can use the implementation of interfaces provided by the other object. If you know that an object is a COM object and you know which interfaces that object supports, your application (or another object) can determine which services the first object can perform. One of the methods that all COM objects inherit, the **QueryInterface** method, lets you determine which interfaces an object supports and creates pointers to these interfaces. For more information, see the IUnknown Interface.

More information about COM programming for DirectX is found in the following topics:

- IUnknown Interface
  - C++ and the COM Interface
  - Retrieving Newer Interfaces
  - Interface Method Names and Syntax
  - Accessing COM Objects by Using C
  - Using Macro Definitions
- 

## IUnknown Interface

---

### [\[Visual Basic\]](#)

This section pertains only to applications written in C++. For an introduction to programming for DirectX in Visual Basic, see [Visual Basic Programming Topics](#).

---

### [\[C++\]](#)

All COM objects support an interface called **IUnknown**. This interface provides DirectX with control of the object's lifetime and the ability to retrieve other interfaces implemented by the object. **IUnknown** has three methods:

- **AddRef** increments the object's reference count by 1 when an interface or another application binds itself to the object.
- **QueryInterface** queries the object about the features that it supports by requesting pointers to a specific interface.
- **Release** decrements the object's reference count by 1. When the count reaches 0, the object is deallocated.

The **AddRef** and **Release** methods maintain an object's reference count. For example, if you create a `DirectDrawSurface` object, the object's reference count is set to 1. Every time a function returns a pointer to an interface for that object, the function then must call **AddRef** through that pointer to increment the reference count. Match each **AddRef** call with a call to **Release**. Before the pointer can be destroyed, you must call **Release** through that pointer. After an object's reference count reaches 0, the object is destroyed, and all interfaces to it become invalid.

The **QueryInterface** method determines whether an object supports a specific interface. If an object supports an interface, **QueryInterface** returns a pointer to that interface. You then can use the methods of that interface to communicate with the object. If **QueryInterface** successfully returns a pointer to an interface, it implicitly calls **AddRef** to increment the reference count, so your application must call **Release** to decrement the reference count before destroying the pointer to the interface.

---



## Requirements

**Windows NT/2000:** Requires Windows NT 3.1 or later.

**Windows 95/98:** Requires Windows 95 or later.

**Header:** Declared in Unknwn.h.

## IUnknown::AddRef

The **IUnknown::AddRef** method increases the reference count of the object by 1.

```
ULONG AddRef();
```

## Parameters

There are no parameters.

## Return Values

Returns the new reference count. This value is meant to be used for diagnostic and testing purposes only.

## Remarks

When the object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the **Release** method to decrease the object's reference count by 1.

## Requirements

**Windows NT/2000:** Requires Windows NT 3.1 or later.

**Windows 95/98:** Requires Windows 95 or later.

**Windows CE:** Unsupported.

**Header:** Declared in Unknwn.h.

## IUnknown::QueryInterface

The **IUnknown::QueryInterface** method determines whether the object supports a particular COM interface. If it does, the system increases the object's reference count, and the application can use that interface immediately.

```
HRESULT QueryInterface(  
    REFIID riid,  
    LPVOID* obp  
);
```

## Parameters

*riid*

Reference identifier of the interface being requested.

*obp*

Address of a pointer to be filled with the interface pointer if the query succeeds.

## Return Values

If the method succeeds, the return value is S\_OK.

If the method fails, the return value may be E\_NOINTERFACE or E\_POINTER. Some components also have their own definitions of these error values in their header files. In DirectInput, for example, DIERR\_NOINTERFACE is equivalent to E\_NOINTERFACE.

## Remarks

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **QueryInterface** method allows Microsoft and third parties to extend objects without interfering with existing or future functionality.

## Requirements

**Windows NT/2000:** Requires Windows NT 3.1 or later.

**Windows 95/98:** Requires Windows 95 or later.

**Windows CE:** Unsupported.

**Header:** Declared in Unknwn.h.

## IUnknown::Release

The **IUnknown::Release** method decreases the reference count of the object by 1.

**ULONG Release();**

## Parameters

There are no parameters.

## Return Values

Returns the new reference count. This value is meant to be used for diagnostic and testing purposes only.

## Remarks

The object deallocates itself when its reference count reaches 0. Use the **AddRef** method to increase the object's reference count by 1.

Applications must call this method to release only interfaces that it explicitly created in a previous call to **IUnknown::AddRef**, **IUnknown::QueryInterface**, or a creation function such as **DirectDrawCreate**.

## Requirements

**Windows NT/2000:** Requires Windows NT 3.1 or later.

**Windows 95/98:** Requires Windows 95 or later.

**Windows CE:** Unsupported.

**Header:** Declared in Unknwn.h.

## C++ and the COM Interface

---

### [\[Visual Basic\]](#)

This topic pertains only to applications written in C++. For an introduction to programming for DirectX in Visual Basic, see Visual Basic Programming Topics.

---

### [\[C++\]](#)

To C++ programmers, a COM interface is like an abstract base class. That is, it defines a set of signatures and semantics but not the implementation, and no state data is associated with the interface. In a C++ abstract base class, all methods are defined as *pure virtual*, which means that they have no code associated with them.

Pure virtual C++ functions and COM interfaces both use a device called a *vtable*. A *vtable* contains the addresses of all functions that implement the interface. If you want an application or object to use these functions, use the **QueryInterface** method to verify that the interface exists on an object and to obtain a pointer to that interface. After sending **QueryInterface**, the object sends your application or object a pointer to the *vtable*, through which this method can call the interface methods implemented by the object. This mechanism isolates any private data that the object uses and the calling client process from one another.

Another similarity between COM objects and C++ objects is that a method's first argument is the name of the interface or class, called the *this* argument in C++. Because COM objects and C++ objects are completely binary-compatible, the compiler treats COM interfaces like C++ abstract classes and assumes the same syntax. This results in less complex code. For example, the *this* argument in C++ is treated as an understood parameter and not coded, and the indirection through the *vtable* is handled implicitly in C++.

---

## Retrieving Newer Interfaces

---

### [Visual Basic]

This topic pertains only to applications written in C++. For an introduction to programming for DirectX in Visual Basic, see Visual Basic Programming Topics.

---

### [C++]

COM dictates that objects update their functionality, not by changing the methods within existing interfaces, but by extending new interfaces that encompass new features. By keeping existing interfaces static, an object built on COM can freely extend its services, while maintaining compatibility with older applications.

DirectX components follow this practice. For example, the DirectDraw component supports several interfaces to access a DirectDrawSurface object:

**IDirectDrawSurface**, **IDirectDrawSurface2**, **IDirectDrawSurface3**, **IDirectDrawSurface4**, and **IDirectDrawSurface7**. Each version of the interface supports the methods provided by its ancestor but adds new methods to support new features.

If your application does not need these new features, it does not need to retrieve newer interfaces. To take advantage of features provided by a new interface, you must call the object's **IUnknown::QueryInterface** method, specifying the globally unique identifier (GUID) of the interface that you want to retrieve. Interface GUIDs are declared in the corresponding header file.

The following example shows how to query for a new interface:

```
LPDIRECTDRAW  lpDD1;
LPDIRECTDRAW2 lpDD2;

ddrval = DirectDrawCreate( NULL, &lpDD1, NULL );
if( FAILED(ddrval))
    goto ERROROUT;

// Query for the IDirectDraw2 interface.
ddrval = lpDD1->QueryInterface(IID_IDirectDraw2, (void **)&lpDD2);
if( FAILED(ddrval))
    goto ERROROUT;

// Now that you have an IDirectDraw2, release the original interface.
lpDD1->Release();
```

In some rare cases, a new interface does not support some methods provided in a previous interface version. The **IDirect3DDevice2** interface is an example of this type of interface. If your application requires features provided by an earlier version of an interface, you can query for the earlier version as shown in the preceding example, using the GUID of the older interface to retrieve it.

## Interface Method Names and Syntax

---

### [\[Visual Basic\]](#)

This topic pertains only to applications written in C++. For an introduction to programming for DirectX in Visual Basic, see Visual Basic Programming Topics.

---

### [\[C++\]](#)

All COM interface methods described in this document are shown using C++ class names. This naming convention is used for consistency and to differentiate between methods used for different DirectX objects that use the same name, such as **QueryInterface**, **AddRef**, and **Release**. This convention does not mean that you can use these methods only with C++.

The syntax for the methods also uses C++ conventions. It does not include the *this* pointer to the interface. When programming in C, the pointer to the interface must be included in each method. The following example shows the C++ syntax for the **IDirectDraw7::GetCaps** method:

```
HRESULT GetCaps(  
    LPDDCAPS lpDDDriverCaps,  
    LPDDCAPS lpDDHELCaps  
);
```

The same example using C syntax would look like this:

```
HRESULT GetCaps(  
    LPDIRECTDRAW7 lpDD,  
    LPDDCAPS lpDDDriverCaps,  
    LPDDCAPS lpDDHELCaps  
);
```

The *lpDD* parameter is a pointer to the DirectDraw interface that represents the DirectDraw object.

---

## Accessing COM Objects by Using C

---

### [\[Visual Basic\]](#)

This topic pertains only to applications written in C. For an introduction to programming for DirectX in Visual Basic, see Visual Basic Programming Topics.

---

#### [C++]

Any COM interface method can be called from a C program. There are two things to remember when calling an interface method from C:

- The first parameter of the method always refers to the object that has been created and that invokes the method (the *this* argument).
- Each method in the interface is referenced through a pointer to the object's *vtable*.

The following example creates a surface associated with a DirectDraw object by calling the **IDirectDraw7::CreateSurface** method in C:

```
ret = lpDD->lpVtbl->CreateSurface (lpDD, &ddsd, &lpDDS,  
    NULL);
```

The *lpDD* parameter references the DirectDraw object associated with the new surface. The method call fills a surface-description structure (*&ddsd*) and returns a pointer to the new surface (*&lpDDS*).

To call the **IDirectDraw7::CreateSurface** method, first dereference the DirectDraw object's *vtable*, and then dereference the method from the *vtable*. The first parameter supplied in the method is a reference to the DirectDraw object that has been created and that invokes the method.

To illustrate the difference between calling a COM object method in C and C++, the same method is shown in C++ (C++ implicitly dereferences the *lpVtbl* parameter and passes the *this* pointer):

```
ret = lpDD->CreateSurface(&ddsd, &lpDDS, NULL)
```

---

## Using Macro Definitions

---

#### [Visual Basic]

This topic pertains only to applications written in C++. For an introduction to programming for DirectX in Visual Basic, see Visual Basic Programming Topics.

---

#### [C++]

Many of the header files for the DirectX interfaces include macro definitions for each method. These macros are included to simplify the use of the methods in your programming, and also have the advantage of expanding to appropriate calls in either C or C++ syntax, depending on whether or not `__cplusplus` is defined.

The following example uses the **IDirectDraw7\_CreateSurface** macro to call the **IDirectDraw7::CreateSurface** method. The first parameter is a reference to the DirectDraw object that has been created and invokes the method:

```
ret = IDirectDraw7_CreateSurface (lpDD, &ddsd, &lpDDS,  
    NULL);
```

To obtain a current list of the methods supported by macro definitions, see the appropriate header file for the DirectX component that you want to use.

---

## Using Callback Functions

---

### [\[Visual Basic\]](#)

This topic pertains only to applications written in C++. For an introduction to programming for DirectX in Visual Basic, see Visual Basic Programming Topics.

---

### [\[C++\]](#)

DirectX contains many methods that are used to enumerate hardware resources or other items available to the application. These methods all work in fundamentally the same way.

Prototype callback functions are documented for each DirectX component. The application declares its own functions with the same return values and parameters as these prototypes. Callback functions are declared as type **CALLBACK**, **WINAPI**, or **FAR PASCAL**, which are all equivalent.

When the application calls an enumeration method, it supplies a pointer to the appropriate callback function that it has implemented. The method calls the function once for each item that qualifies for enumeration and passes in information about the item. Within the function, the application can use this information to perform any sort of task, such as building a list or looking for particular device capabilities. The enumeration method returns when all items have been enumerated or when the callback function returns a value indicating that enumeration can stop (either **FALSE** or a particular **HRESULT**, depending on the return type of the callback).

For an example, see DirectInput Device Enumeration.

---

## Version Checking

In order to scale their functionality to the version of DirectX currently running on the user's system, applications may need to know what version of DirectX is currently installed.

The DirectX software development kit includes a sample application with source code at (*SDK root*) \Samples\Multimedia\DXMisc\Src\GetDXVer that determines whether DirectX is on the system and, if so, what version it is. Each new version of DirectX

supports all the functionality that was in older versions, so be sure to test for the desired version *or later*, so that your application will continue to run on future versions.

You can also test for particular functionality by using **QueryInterface**. For example, suppose your application is designed to run on DirectX 3.0 or later, but has optional force-feedback effects supported only by DirectX 5.0 and later. You can determine whether support for force feedback is available by obtaining the **IDirectInputDevice** interface and then querying for **IDirectInputDevice2**.

For a variety of reasons applications may also want to determine what version of the operating system is present. To provide for easier migration to future versions of components or versions of the Windows operating system, applications should check for the earliest compliant version, and allow for installation and execution on later versions. Knowing that Windows 2000 will contain DirectX 7.0 functionality or greater means developers should write code that detects the presence of Windows 2000 or greater and allows the application to run if it is found. Conversely, if Windows NT 4.0 is the version detected, DirectX 7.0 functionality is not available.

The following sample function returns TRUE if the operating system is the same as or later than the one indicated in the parameters, which indicate the major version, minor version, and service pack.

```
BOOL bIsWindowsVersionOK(DWORD dwMajor, DWORD dwMinor, WORD dwSPMajor)
{
    OSVERSIONINFO osv;

    // Initialize the OSVERSIONINFO structure.//
    ZeroMemory(&osv, sizeof(OSVERSIONINFO));

    osv.dwOSVersionInfoSize = sizeof(OSVERSIONINFO);
    GetVersionEx((OSVERSIONINFO*)&osv);

    // Check the major version.
    if (osv.dwMajorVersion > dwMajor)
        return TRUE;
    else if (osv.dwMajorVersion == dwMajor)
    {
        // Check the minor version.
        if (osv.dwMinorVersion > dwMinor)
            return TRUE;
        else if (osv.dwMinorVersion == dwMinor)
        {
            // Check the Service Pack.
            if (dwSPMajor && osv.dwPlatformId ==
                VER_PLATFORM_WIN32_NT)
            {
                HKEY hKey;
```



```
DWORD dwCSDVersion;
DWORD dwSize;
BOOL fMeetsSPRequirement = FALSE;

if (RegOpenKeyEx (HKEY_LOCAL_MACHINE,
    "System\\CurrentControlSet\\Control\\Windows",
    0,
    KEY_QUERY_VALUE,
    &hKey) == ERROR_SUCCESS)
{
    dwSize = sizeof(dwCSDVersion);
    if (RegQueryValueEx (hKey,
        "CSDVersion",
        NULL,
        NULL,
        (unsigned char*)&dwCSDVersion,
        &dwSize) == ERROR_SUCCESS)
    {
        fMeetsSPRequirement =
            (LOWORD(dwCSDVersion) >= dwSPMajor);
    }
    RegCloseKey(hKey);
}

return fMeetsSPRequirement;
}
return TRUE;
}
}
return FALSE;
}
```

---

## Compiling DirectX Samples and Other DirectX Applications

---

### [\[Visual Basic\]](#)

This section pertains only to applications written in C++. For an introduction to programming for DirectX in Visual Basic, see Visual Basic Programming Topics.

---

### [\[C++\]](#)

This section provides information about considerations specific to compiling DirectX applications. The following topics are discussed:

- Preparing for Compilation

- Component Version Constants
- 

## Preparing for Compilation

---

### [Visual Basic]

This topic pertains only to applications written in C++. For an introduction to programming for DirectX in Visual Basic, see Visual Basic Programming Topics.

---

### [C++]

The samples included in this SDK use Microsoft® Visual C++® project files (MDP files) that describe the appropriate source files, project resources, and linker settings for each sample. However, you might need to make additional preparations to ensure that the samples compile and link properly, or you might need to prepare settings for a new project of your own. (For the samples, developers that use other compilers can reference the generic makefiles included for information about the required libraries, then configure their environment appropriately.) The information provided here applies to the DirectX samples and the DirectX applications that you create.

After opening a project file in Visual C++, verify some settings before trying to compile the application. The following descriptions are valid for Microsoft Visual C++ 5.0 and 4.2. Other versions of Visual C++ and other compilers have equivalent settings. If you do not use these versions of Visual C++, refer to the documentation provided with your development environment for information on changing these settings.

### Note

The following discussion uses the default installation paths (C:\Mssdk\Include and C:\Mssdk\Lib) to describe file locations. Your installation paths might differ.

## Include search paths

Be sure that the search path for header files is correct and the directory for DirectX header files is the first path that the compiler searches. To check the include path, choose **Options** from the **Tools** menu; then select the **Directories** tab. The following dialog box will appear.

The topmost path indicates the folder that contains the latest DirectX header files. The default path is C:\Mssdk\Include. If the path is not present, add it to the list, and move it to the top of the search list by using the toolbar controls within the **Directories** tab.

## Linker search paths

Check the search paths and search order that the linker uses to search for link libraries. The link search paths are also listed on the **Directories** tab. Choose **Options** from the **Tools** menu; then select the **Directories** tab. When the dialog box appears, choose the **Library files** option in the **Show directories for:** drop-down list box. The topmost path should be the folder that contains the latest DirectX link libraries. The default path is C:\Mssdk\Lib. (Borland link libraries are provided in the Borland folder within the default folder.)

## Project link libraries

If you are using the provided sample project files, you do not need to verify these settings; they are specified with the provided project files. For new applications, choose **Settings** from the **Project** menu to make the following dialog box appear. (In Visual C++ 4.2, this is the **Settings** item on the **Build** menu.)

All DirectX applications link to the Dxguid.lib include library, which defines the globally unique identifiers (GUIDs) for all DirectX Foundation COM interfaces. (You can also define INITGUID prior to all other include and define statements in a single source module.) In addition, verify that the application is linked to the appropriate standard DirectX link libraries. The following table shows the various DirectX foundation components that might be used and the corresponding link libraries for those components.

| Component                 | Link library (*.lib)         |
|---------------------------|------------------------------|
| Direct3D Immediate Mode   | D3dim.lib, D3du.lib          |
| Direct3DX Utility Library | D3dx.lib, D3dx.d.lib (debug) |
| Direct3D Retained Mode    | D3drm.lib, D3dxof.lib        |
| DirectDraw                | Ddraw.lib                    |
| DirectInput               | Dinput.lib                   |
| DirectMusic               | None                         |
| DirectPlay                | Dplayx.lib                   |
| DirectSetup               | Dsetup.lib                   |
| DirectSound               | Dsound.lib                   |
| (All components)          | Dxguid.lib                   |

---

## Component Version Constants

---

[\[Visual Basic\]](#)

This topic pertains only to applications written in C++. For an introduction to programming for DirectX in Visual Basic, see Visual Basic Programming Topics.

---

#### [C++]

For backward compatibility with previous versions of DirectX, some DirectX components include variable API element definitions in their header files. Affected elements are typically capability structures or flag sets that are version-specific. Parts of some header files are surrounded by preprocessor conditionals that cause the preprocessor to filter out unneeded definitions. The value of the defined constant identifies a specific version of the component; if no value is defined, the headers set a value that identifies the DirectX version for which the header file was written. An example from the DirectDraw header file, Ddraw.h, is shown here:

```
#ifndef DIRECTDRAW_VERSION
#define DIRECTDRAW_VERSION 0x0700
#endif /* DIRECTDRAW_VERSION */
```

You can define other values for these constants to use newer versions of the header files with previous versions of the components. For example, to use the latest headers to compile against the DirectX 3.0 version of DirectDraw, define `DIRECTDRAW_VERSION` to be `0x0300`. You can set the value for the constant in your development environment, or you can change the value in the header file itself.

---

## Debugging DirectX Applications

This section covers the following topics pertaining to debugging C and C++ applications:

- Debug vs. Retail DLLs
- Symbol Information

### Debug vs. Retail DLLs

The DirectX SDK installation program provides the option of installing either debug or retail builds of the DirectX dynamic-link libraries (DLLs).

---

#### [Visual Basic]

If you are developing in Visual Basic, there is no advantage in installing the debug DLLs because your application would run more slowly.

---

#### [C++]

When developing software in C++, it is best to install the debug versions of the DLLs. The debug DLLs have additional code that validates internal data structures and output debug error messages (using the Microsoft® Win32® **OutputDebugString** function) while your program is executing. When an error occurs, the debug output gives you a more detailed description of the problem. The debug DLLs execute more slowly than the retail DLLs but are much more useful for debugging an application. Be sure to ship the retail version with your application.

To see the debug messages, configure your system so that debug output displays in a window or on a remote computer. A development environment such as Visual C++ allows you to do this. Consult the environment documentation for setup instructions.

## Symbol Information

### [Visual Basic]

This topic pertains only to applications written in C++. For an introduction to programming for DirectX in Visual Basic, see Visual Basic Programming Topics.

### [C++]

To ensure that the debugger can find all the relevant symbolic information when using debug builds, locate the symbol files as follows:

| OS                         | Debugger   | Location for .pdb file                   | Location for .db  |
|----------------------------|------------|--|-------------------|
| Windows 95<br>Windows 98   | Visual C++ | Same directory as binary                 | Same directory as |
| Windows NT<br>Windows 2000 | Visual C++ | Same directory as binary                 | %SystemRoot%\     |
| Windows NT<br>Windows 2000 | NTSD/KD    | %SystemRoot%\Symbols\<binary extension>\ | %SystemRoot%\     |

### Note

When debugging with the Visual C++ development system on Microsoft® Windows NT®/Windows® 2000, do not use the WIN32API Splitsym development tool. WIN32API Splitsym copies private symbolic information from the <binary>.dbg file into the symbol directory under the binary extension (for example, %SystemRoot%\Symbols\Dll\ ), then deletes the original file from the binary directory. Visual C++ relies on finding a private symbolic information (<binary>.dbg) file in the same directory as the binary itself. Therefore, for debug builds, you must only copy the private symbolic information, and not delete it. This is an issue only when using Visual C++ with Windows NT/Windows 2000. Consult Visual C++ documentation and Windows 2000 Driver Development Kit (DDK) documentation for further debugging information.

