Introducing DirectX 8.0

Microsoft® DirectX® 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 effects and music, input devices, and support for networked applications such as multiplayer games.

This document gives general introductory information about DirectX 8.0. Information is divided into the following sections.

- What's New in DirectX 8.0
- Using the DirectX 8.0 Documentation
- DirectX 8.0 Components
- DirectX Tools

[C++]

- Programming DirectX with C/C++

[Visual Basic]

- Programming DirectX with Visual Basic

- Further Information

What's New in DirectX 8.0

The following are some of the new features supported by Microsoft® DirectX® 8.0.

Complete integration of DirectDraw and Direct3D
Microsoft DirectDraw® and Microsoft Direct3D® are merged into a single DirectX Graphics component. The API has been extensively updated to make it even easier to use and to support the latest graphics hardware.

DirectMusic and DirectSound more integrated
Microsoft DirectMusic® and Microsoft DirectSound® are much more tightly integrated than with DirectX 7.0. Wave files or resources can now be loaded by the DirectMusic loader, and played through the DirectMusic performance, synchronized with MIDI notes.

DirectPlay updated
The Microsoft DirectPlay® component has been extensively updated to increase its capabilities and improve its ease-of-use. In particular, DirectPlay now supports voice communication between players.

DirectInput updated

Microsoft DirectInput® introduces one major new feature: action mapping. Action mapping enables you to establish a connection between input actions and input devices that does not depend on the existence of particular device objects. It simplifies the input loop and reduces the need for custom game drivers, custom device profilers, and custom configuration user interfaces in games.

[C++]

DirectShow included in DirectX

Microsoft DirectShow® is now part of DirectX and has been updated for this release.

Debug build available

You can use the DirectX Control Panel Application to switch between the debug and retail builds of DirectInput, Direct3D, and DirectMusic. To enable this feature, select the debug option when you install the SDK. This option installs both debug and retail DLLs on your system. The retail option installs only the retail DLLs.

Using the DirectX 8.0 Documentation

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**
Italic text	Denotes a placeholder or variable. You must provide the actual value. For example, the statement SetCursorPos(*X*, *Y*) requires you to substitute values for the *X* and *Y* parameters.
Bold text	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.
FULL BOLD CAPITALS	Used for most type and structure names.
FULL CAPITALS	Used for enumeration values, flags, and constants.
`monospace`	Used for code examples and syntax spacing.
. . .	Represents an omitted portion of a sample application.

DirectX 8.0 Components

Microsoft® DirectX® 8.0 is made up of the following components.

- DirectX Graphics combines the Microsoft DirectDraw® and Microsoft Direct3D® components of previous DirectX versions into a single API that you can use for all graphics programming. The component includes the Direct3DX utility library that simplifies many graphics programming tasks.
- DirectX Audio combines the Microsoft DirectSound® and Microsoft DirectMusic® components of previous DirectX versions into a single API that you can use for all audio programming.
- Microsoft DirectInput® provides support for a variety of input devices, including full support for force-feedback technology.
- Microsoft DirectPlay® provides support for multiplayer networked games.

[C++]

- Microsoft DirectShow® provides for high-quality capture and playback of multimedia streams.
- Microsoft DirectSetup is a simple API that provides one-call installation of the DirectX components.

DirectX Tools

The following tools can be useful in developing and troubleshooting Microsoft® DirectX® applications.

- DirectX Caps Viewer
- DirectX Control Panel Application
- DirectX Diagnostic Tool

DirectX Caps Viewer

Description

The Microsoft® 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 Control Panel Application

You can use the Microsoft® DirectX® Control Panel utility, which is installed with the SDK, to examine and modify the properties of DirectX components. This utility contains tabs for each DirectX component. Using this utility you can:

- Obtain version information for DirectX components.
- Change the debug output level and debug settings to aid in troubleshooting during the development process.
- View specific driver and hardware support information for DirectX components.
- Toggle between debug and retail versions of the run times for Microsoft Direct3D®, DirectMusic®, and DirectInput®.

DirectX Diagnostic Tool

Description

Microsoft® 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 enables developers to report problems directly to the DirectX development team.

Path

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

User's Guide

To display the DirectX Diagnostic Tool Help file, click the **Help** button.

Programming DirectX with C/C++

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
Most Microsoft® DirectX® application development with C/C++ involves conventional programming techniques. However, there are several aspects of DirectX programming that may be unfamiliar to some developers. This section provides a brief overview of several specialized topics, along with some guidelines for compiling and debugging DirectX applications.

- Using COM
- Using Callback Functions
- Version Checking
- Compiling DirectX Samples and Other DirectX Applications
- Debugging DirectX Applications

Using COM

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
The Component Object Model (COM) is an object-oriented programming model used by numerous applications. Because the bulk of the Microsoft® DirectX® run time is in the form of COM-compliant objects, all DirectX developers need to have at least a basic understanding of COM principles and programming techniques. However, although COM has a reputation for being difficult and complex, the COM programming required by most DirectX applications is straightforward.

There are two distinct types of COM programming:

- Using existing COM objects. This is not much more difficult than using C++ objects.
- Implementing your own COM objects. This can be a complicated and demanding task. Much of COM's reputation for complexity comes from this type of COM programming.

Most DirectX applications need to use only the COM objects provided by DirectX. They do not need to implement their own COM objects. In other words, most DirectX developers will need to concern themselves only with the first, and easiest, type of COM programming.

This section provides a brief introduction to using the COM objects provided by DirectX. It is primarily intended for novice COM programmers. For a more detailed discussion of how to use COM objects, or for information on how to implement your own COM objects, see Further Information.

What is a COM Object?

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
COM objects are basically black boxes that can be used by applications to perform one or more tasks. They are most commonly implemented as DLLs. Like a conventional DLL, COM objects expose methods that your application can call to perform any of the supported tasks. Applications interact with COM objects in somewhat the same way they do with C++ objects. However, there are some distinct differences.

- COM objects enforce stricter encapsulation than C++ objects. You cannot simply create the object and call any public method. A COM object's public methods are grouped into one or more *interfaces*. To use a method, you must create the object and obtain the appropriate interface from the object. An interface typically contains a related set of methods that provide access to a particular feature of the object. For example, the **IDirect3DCubeTexture8** interface contains methods that enable you to manipulate cube texture resources. Any methods that are not part of an interface are not accessible.
- COM objects are not created in the same way as C++ objects. There are several ways to create a COM object, but all involve COM-specific techniques. The Microsoft® DirectX® API includes a variety of helper functions and methods that simplify creating most of the DirectX objects.
- You must use COM-specific techniques to control the lifetime of the object.
- COM objects do not need to be explicitly loaded. COM objects are typically contained in a DLL. However, you do not need to explicitly load the DLL or link to a static library in order to use a COM object. Each COM object has a unique registered identifier that is used to create the object. COM automatically loads the correct DLL.

- COM is a binary specification. COM objects can be written in and accessed from a variety of languages. You don't need to know anything about the object's source code. For example, Microsoft Visual Basic® applications routinely use COM objects that were written in C++.

Objects and Interfaces

It is important to understand the distinction between objects and interfaces. In casual usage, an object may sometimes be referred to by the name of its principle interface. However, strictly speaking, the two terms are not interchangeable.

- An object may expose any number of interfaces. For example, while all objects must expose the **IUnknown** interface, they normally expose at least one additional interface, and they might expose many. In order to use a particular method, you must not only create the object, you must also obtain the correct interface.
- More than one object might expose the same interface. An interface is a group of methods that perform a specified set of operations. The interface definition specifies only the syntax of the methods and their general functionality. Any COM object that needs to support a particular set of operations can do so by exposing a suitable interface. Some interfaces are highly specialized and are exposed only by a single object. Others are useful in a variety of circumstances and are exposed by many objects. The extreme case is the **IUnknown** interface, which must be exposed by all COM objects.

Note

If an object exposes an interface, it must support every method in the interface definition. In other words, you can call any method and be confident that it exists. However, the details of how a particular method is implemented may vary from object to object. For example, different objects may use different algorithms to arrive at the final result. There is also no guarantee that a method will be supported in a non-trivial way. Sometimes an object exposes a commonly-used interface, but needs to support only a subset of the methods. You will still be able to call the remaining methods successfully, but they will return E_NOTIMPL. You should refer to the documentation to see how an interface is implemented by any particular object.

The COM standard requires that an interface definition must not change once it has been published. You cannot, for example, add a new method to an existing interface. You must instead create a new interface. While there are no restrictions on what methods must be in an interface, a common practice is to have the next-generation interface include all the of the old interface's methods, plus any new methods.

It is not unusual for an interface to have several generations. Typically, all generations perform essentially the same overall task, but they will be different in detail. Often, an object will expose every generation of interface. Doing so allows older applications to continue using the object's older interfaces, while newer applications can take advantage of the features of the newer interfaces. Typically, a family of interfaces will all have the same name, plus an integer indicating the generation. For example, if the original interface were named **IMyInterface**, the next two generations would be called **IMyInterface2** and **IMyInterface3**. Microsoft DirectX® typically labels successive generations of interfaces with the DirectX version number.

GUIDs

Globally Unique Identifiers (GUIDs) are a key part of the COM programming model. At its most basic, a GUID is a 128-bit structure. However, GUIDs are created in such as way as to guarantee that no two GUIDs are the same. COM uses GUIDS extensively for two primary purposes:

- To uniquely identify a particular COM object. A GUID that is assigned to a COM object is called a Class ID (CLSID). You use a CLSID when you want to create an instance of the associated COM object.
- To uniquely identify a particular COM interface. The GUID that is associated with a particular COM interface is called an Interface ID (IID). You use an IID when you request a particular interface from an object. An interface's IID will be the same, regardless of which object exposes the interface.

Note

For convenience, documentation normally refers to objects and interfaces by a descriptive name such as **IDirect3D8**. In the context of the documentation, there is rarely any danger of confusion. However, strictly speaking, there is no guarantee that another object or interface does not have the same descriptive name. The only unambiguous way to refer to a particular object or interface is by its GUID.

Although GUIDs are structures, they are often expressed as an equivalent string. The general format of the string form of a GUID is "{VVVVVVVV-WWWW-XXXX-YYYY-ZZZZZZZZZZZZ}", where each letter corresponds to a hexadecimal integer. For example, the string form of the IID for the **IDirect3D8** interface is:

```
{1DD9E8DA-1C77-4D40-B0CF-98FEFDFF9512}
```

Because the actual GUID is somewhat clumsy to use and easy to mistype, an equivalent name is usually provided as well. You can use this name instead of the actual structure when you call functions such as **CoCreateInstance**. The customary naming convention is to prepend either IID_ or CLSID_ to the descriptive name of the interface or object, respectively. For example, the name of the **IDirect3D8** interface's IID is IID_IDirect3D8.

HRESULT Values

All COM methods return a 32-bit integer called an HRESULT. With most methods, the HRESULT is essentially a structure that contains two separate pieces of information:

- Whether the method succeeded or failed.

- More detailed information about the outcome of the operation supported by the method.

Some methods return HRESULT values only from the standard set defined in Winerror.h. However, methods are free to return custom HRESULT values with more specialized information. These values are normally documented on the method's reference page.

Note

The list of HRESULT values that you find on a method's reference page is often only a subset of the possible values that may be returned. The list typically covers only those values that are specific to the method and those standard values that have some method-specific meaning. You should assume that a method may return a variety of standard HRESULT values, even if they are not explicitly documented.

While HRESULT values are often used to return error information, you should not think of them as error codes. The fact that the bit that indicates success or failure is stored separately from the bits that contain the detailed information allows HRESULT values to have any number of success and failure codes. By convention, success codes are given names with an S_ prefix, and failure codes with an E_ prefix. For example, the two most commonly used codes are S_OK and E_FAIL, which indicate simple success or failure, respectively.

The fact that COM methods may return a variety of success or failure codes means that you have to be careful how you test the HRESULT value. For example, consider a hypothetical method with documented return values of S_OK if successful and E_FAIL if not. However, remember that the method may also return other failure or success codes. The following code fragment illustrates the danger of using a simple test. The *hr* value is the HRESULT that was returned by the method.

```
if(hr == E_FAIL)
 {
   //Handle the failure
 }
else
 {
   //Handle the success
 }
```

As long as the method returns only E_FAIL to indicate failure, this test will work properly. However, the method might also return an error value such as E_NOTIMPL or E_INVALIDARG. That value would be interpreted as a success, perhaps causing your application to fail.

If you need detailed information on the outcome of the method call, you will need to test each relevant HRESULT value. However, you may be interested only in whether the method succeeded or failed. A robust way to test whether an HRESULT value indicates success or failure is to pass the value to the one of the following macros, defined in Winerror.h.

- The **SUCCEEDED** macro returns TRUE for a success code and FALSE for a failure code.
- The **FAILED** macro returns TRUE for a failure code and FALSE for a success code.

You can fix the preceding code fragment by using the **FAILED** macro.

```
if(FAILED(hr))
 {
  //Handle the failure
 }
else
 {
  //Handle the success
 }
```

This code fragment properly treats E_NOTIMPL and E_INVALIDARG as failures.

Although most COM methods return structured HRESULT values, a small number use the HRESULT to return a simple integer. Implicitly, these methods are always successful. If you pass an HRESULT of this sort to the **SUCCESS** macro, the macro will always return TRUE. A commonly used example is the **IUnknown::Release** method. This method decrements an object's *reference count* by one and returns the current reference count. See Managing the Object's Lifetime for a discussion of reference counting.

The Address of a Pointer

If you look at a few COM method reference pages, you will probably run across something like the following:

```
HRESULT CreateDevice(
  ...,
  IDirect3DDevice8** ppReturnedDeviceInterface
);
```

While a normal pointer is quite familiar to any C/C++ developer, COM often uses an additional level of indirection. This second level of indirection is indicated by a "**" following the type declaration, and the variable name typically has a "pp" prefix. For the example given above, *ppReturnedDeviceInterface* parameter is typically referred to as the *address of a pointer to* an **IDirect3DDevice8** interface.

Unlike C++, you do not access a COM object's methods directly. Instead, you must obtain a pointer to an interface that exposes the method. To invoke the method, you use essentially the same syntax that you would to invoke a pointer to a C++ method. For example, to invoke the **IMyInterface::DoSomething** method, you would use the following syntax.

```
IMyInterface *pMyIface;
...
pMyIface->DoSomething(...);
```

The need for a second level of indirection comes from the fact that you do not create interface pointers directly. You must call one of variety of methods, such as the **CreateDevice** method shown above. To use such a method to obtain an interface pointer, you declare a variable as a pointer to the desired interface, and pass the address of that variable to the method. In other words, you pass the method the address of a pointer. When the method returns, the variable will point to the requested interface, and you can use that pointer to call any of the interface's methods. See Obtaining and Using COM Interfaces for further discussion of how to use interface pointers.

Creating a COM Object

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
There are several ways to create COM objects. The two most common ones used in Microsoft® DirectX® programming are:

- Directly, by passing the object's CLSID to the **CoCreateInstance** function. The function will create an instance of the object, and it will return a pointer to an interface that you specify.
- Indirectly, by calling a DirectX method or function that creates the object for you. The method creates the object and returns an interface on the object. When you create an object this way, you usually cannot specify which interface should be returned.

Before you create any objects, COM must be initialized by calling the **CoInitialize** function. If you are creating objects indirectly, the object creation method will handle this task. If you need to create an object with **CoCreateInstance**, you must call **CoInitialize** explicitly. When you are finished, COM must be uninitialized by calling **CoUninitialize.** If you make a call to **CoInitialize** you must match it with a call to **CoUninitialize**. Typically, applications that need to explicitly initialize COM do so in their startup routine, and they uninitialize COM in their cleanup routine.

To create a new instance of a COM object with **CoCreateInstance**, you must have the object's CLSID. If this CLSID is publicly available, you will find it in the reference documentation or the appropriate header file. If the CLSID is not publicly available, you cannot create the object directly.

The **CoCreateInstance** function has five parameters. For the COM objects you will be using with DirectX, you can normally set the parameters as follows

- *rclsid*. Set this parameter to the CLSID of the object you want to create.
- *pUnkOuter*. Set this parameter to NULL. It is used only if you are aggregating objects. See Further Information for a discussion of aggregation.
- *dwClsContext*. Set this parameter to CLSCTX_INPROC_SERVER. This setting indicates that the object is implemented as a DLL and will run as part of your application's process.
- *riid*. Set this parameter to the IID of the interface that you would like to have returned. The function will create the object, and it will return the requested interface pointer in the *ppv* parameter.
- *ppv*. Set this parameter to the address of a pointer that will be set to the interface specified by *riid* when the function returns. This variable should be declared as a pointer to the requested interface, and the reference to the pointer in the parameter list should be cast as (LPVOID *).

For example, the following code fragment creates a new instance of the **DirectPlay8** object, and it returns a pointer to the **IDirectPlay8Peer** interface in the *g_pDP* variable. If an error occurs, a message box is displayed, and the application terminates.

```
IDirectPlay8Peer*  g_pDP = NULL;
...
CoInitialize( NULL );
...
hr = CoCreateInstance( CLSID_DirectPlay8, NULL, CLSCTX_INPROC_SERVER,
                    IID_IDirectPlay8Peer, (LPVOID*) &g_pDP );

if( FAILED( hr ) )
 {
   MessageBox( NULL, TEXT("Failed Creating IDirectPlay8Peer. "),
              TEXT("DirectPlay Sample"), MB_OK | MB_ICONERROR );
   return FALSE;
 }
```

Creating an object indirectly is usually much simpler. You pass the object creation method the address of an interface pointer. The method then creates the object and returns an interface pointer. When you create an object indirectly, you typically cannot choose which interface the method will return. However, you can often specify a variety of things about how the object should be created. For example, the following code fragment calls the **IDirect3D8::CreateDevice** method discussed earlier to create a device object to represent a display adapter. It returns a pointer to the object's **IDirect3DDevice8** interface. The first four parameters provide a variety of information needed to create the object, and the fifth parameter receives the interface pointer. See the reference documentation for details.

```
IDirect3DDevice8 *g_pd3dDevice = NULL;
```

```
...
if( FAILED( g_pD3D->CreateDevice(D3DADAPTER_DEFAULT,
                                 3DDEVTYPE_HAL,
                                 hWnd,
                                 D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                                 &d3dpp,
                                 &g_pd3dDevice )))
  return E_FAIL;
```

Using COM Interfaces

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
When the object is created, the creation method returns an interface pointer. You can then use that pointer to access any of the interface's methods. The syntax is identical to that used with a pointer to a C++ method. The following code fragment extends the example given in the previous section. After creating the **DirectPlay8** object, the example uses the **IDirectPlay8Peer** interface pointer returned by **CoCreateInstance** to initialize the object by calling the **IDirectPlay8Peer::Initialize** method. Error correction code is omitted for clarity.

```
IDirectPlay8Peer*  g_pDP = NULL;
...
CoInitialize( NULL );
...
hr = CoCreateInstance( CLSID_DirectPlay8, NULL,CLSCTX_INPROC_SERVER,
                       IID_IDirectPlay8Peer, (LPVOID*) &g_pDP );
hr = g_pDP->Initialize( NULL, DirectPlayMessageHandler, 0 );
```

Requesting Additional Interfaces

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
In many cases, the interface pointer that you receive from the creation method may be the only one that you need. In fact, it is not uncommon for an object to export only

one interface other than **IUnknown**. However, many objects export multiple interfaces, and you may need pointers to several of them. If you need more interfaces than the one returned by the creation method, there is no need to create a new object. Instead, you request another interface pointer by using the object's **IUnknown::QueryInterface** method.

If you create your object with **CoCreateInstance**, you can request an **IUnknown** interface pointer, and then call **IUnknown::QueryInterface** to request every interface you need. However, this approach is inconvenient if you need only a single interface, and it doesn't work at all if you use an object creation method that does not allow you to specify which interface pointer should be returned. In practice, you usually don't need to obtain an explicit **IUnknown** pointer because all COM interfaces inherit from or *extend* the **IUnknown** interface.

Extending an interface is similar to inheriting from a C++ class. The child interface exposes all of the parent interface's methods, plus one or more of its own. In fact, you will often see "inherits from" used instead of "extends". What you need to remember is that the inheritance is internal to the object. Your application cannot inherit from or extend an object's interface. However, you can use the child interface to call any of the child's or the parent's methods.

Because all interfaces are children of **IUnknown** you can use any of the interface pointers you already have for the object to call **QueryInterface**. When you do so, you must provide the IID of the interface you are requesting and the address of a pointer that will contain the interface pointer when the method returns. For example, the following code fragment calls **IDirectSound8::CreateSoundBuffer** to create a primary sound buffer object. This object exposes several interfaces. The **CreateSoundBuffer** method returns an **IDirectSoundBuffer8** interface. The subsequent code then uses the **IDirectSoundBuffer8** interface to call **QueryInterface** to request an **IDirectSound3DListener8** interface.

```
IDirectSoundBuffer8* pDSBPrimary = NULL;
IDirectSound3DListener8* pDSListener;
...
if(FAILED(hr = g_pDS->CreateSoundBuffer( &dsbd, &pDSBPrimary, NULL )))
  return hr;

if(FAILED(hr = pDSBPrimary->QueryInterface(IID_IDirectSound3DListener8,
                                (LPVOID *)&pDSListener)))
  return hr;
```

Managing a COM Object's Lifetime

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
When an object is created, the system allocates the necessary memory resources. When an object is no longer needed, it should be destroyed. The system can use that memory for other purposes. With C++ objects, you can control the object's lifetime directly with the *new* and *delete* operators. COM does not enable you to create or destroy objects directly. The reason for this practice is that the same object may be used by more than one application. If one of those applications were to destroy the object, the other applications would probably fail. Instead COM uses a system of *reference counting*, to control an object's lifetime.

An object's reference count is the number of times one of its interfaces has been requested. Each time an interface is requested, the reference count is incremented. An application *releases* an interface when that interface is no longer needed, decrementing the reference count. As long as the reference count is greater than zero, the object remains in memory. When the reference count reaches zero, the object destroys itself. You don't need to know anything about the reference count of an object. As long as you obtain and release an object's interfaces properly, the object will have the appropriate lifetime.

Note

Properly handling reference counting is a crucial part of COM programming. Failure to do so can easily create a memory leak. One of the most common mistakes that COM programmers make is failing to release an interface. When this happens, the reference count will never reach zero, and the object will remain in memory indefinitely.

Incrementing and Decrementing the Reference Count

Whenever you obtain a new interface pointer, the reference count must be incremented by a call to **IUnknown::AddRef**. However, your application does not usually need to call this method. If you obtain an interface pointer by calling an object creation method, or by calling **IUnknown::QueryInterface**, the object will automatically increment the reference count. However, if you create an interface pointer in some other way, such as copying an existing pointer, you must explicitly call **IUnknown::AddRef**. Otherwise, when you release the original interface pointer, the object may be destroyed even though you may still need to use the copy of the pointer.

You must release all interface pointers, regardless of whether you or the object incremented the reference count. When you no longer need an interface pointer, call **IUnknown::Release** to decrement the reference count. A common practice is to initialize all interface pointers to NULL, and set them back to NULL when they are released. That allows you to test all interface pointers in your cleanup code. Those that are non-NULL are still active and must be released before you terminate the application.

The following code fragment extends the sample discussed in Requesting Additional Interfaces to illustrate how to handle reference counting.

```
IDirectSoundBuffer8* pDSBPrimary = NULL;
IDirectSound3DListener8* pDSListener = NULL;
IDirectSound3DListener8* pDSListener2 = NULL;
...
//Create the object and obtain an additional interface.
//The object increments the reference count.
if(FAILED(hr = g_pDS->CreateSoundBuffer( &dsbd, &pDSBPrimary, NULL )))
  return hr;

if(FAILED(hr=pDSBPrimary->QueryInterface(IID_IDirectSound3DListener8,
                                           (LPVOID *)&pDSListener)))
  return hr;

//Make a copy of the IDirectSound3DListener8 interface pointer.
//Call AddRef to increment the reference count and to ensure that
//the object is not destroyed prematurely
pDSListener2 = pDSListener;
pDSListener2->AddRef();
...
//Cleanup code. Check to see if the pointers are still active.
//If they are, call Release to release the interface.
if(pDSBPrimary != NULL)
{
  pDSBPrimary->Release();
  pDSBPrimary = NULL;
}
if(pDSListener != NULL)
{
  pDSListener->Release();
  pDSListener = NULL;
}
if(pDSListener2 != NULL)
{
  pDSListener2->Release();
  pDSListener2 = NULL;
}
```

Using C to Access COM Objects

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
Although C++ is the most commonly used language for COM programming, you can also access COM objects using C. Doing so is relatively straightforward, but requires a somewhat more complex syntax.

- All methods have an additional parameter added to the beginning of the parameter list. This parameter must be set to the interface pointer.
- You must explicitly reference the object's *vtable*.

Every COM object has a vtable that contains a list of pointers to methods that the object exposes. An interface pointer points to the appropriate location in the vtable, which in turn contains a pointer to the particular method you are calling. The vtable is not mentioned elsewhere in this documentation because with C++, the vtable is essentially invisible. However, if you wish to call COM methods with C, you must include an additional level of indirection that explicitly references the vtable.

The following code fragment illustrates how to call the **IDirectPlay8Peer::Initialize** method with the C++ calling convention.

```
g_pDP->Initialize( NULL, DirectPlayMessageHandler, 0 );
```

To make the same method call from C, use the following syntax. The conventional name for the vtable pointer is lpVtbl.

```
g_pDP->lpVtbl->Initialize(g_pDP,NULL, DirectPlayMessageHandler, 0);
```

Some components have macros defined in their header files that resolve to the correct calling convention. See Using Macros to Call DirectX COM Methods for details.

Using Macros to Call DirectX COM Methods

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
Many of the Microsoft® DirectX® interfaces have macros defined for each method that make using the methods in your applications simpler. You can find definitions of these macros in the same header file as the interface declaration. The macros are designed to be used by both C and C++ applications. To use the C++ macros, you must define _cplusplus. Otherwise, the C macros will be used. The macro syntax is the same for both languages, but the header files include separate sets of macro definitions that expand to the appropriate calling convention.

For example, the following code fragment from the d3d.h header file shows the definitions of the C and C++ macros for the **IDirect3D8::GetAdapterIdentifier** method.

```
#if !defined(__cplusplus) || defined(CINTERFACE)
  ...
  #define IDirect3D8_GetAdapterIdentifier(p,a,b,c) (p)->lpVtbl->GetAdapterIdentifier(p,a,b,c)
  ...
  #else
  ...
  #define IDirect3D8_GetAdapterIdentifier(p,a,b,c) (p)->GetAdapterIdentifier(a,b,c)
  ...
#endif
```

To use one of these macros, you must first obtain a pointer to the associated interface. The first parameter of the macro must be set to that pointer. The remaining parameters map to the method's parameters. The macro's return value is the HRESULT value that is returned by the method. The following code fragment uses a macro to call the **IDirect3D8::GetAdapterIdentifier** method. *pD3D* is a pointer to an **IDirect3D8** interface.

```
  hr = IDirect3D8_GetAdapterIdentifier(pD3D,
                                       Adapter,
                                       dwFlags,
                                       pIdentifier);
```

DirectX COM Documentation Conventions

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
By convention, COM methods and interfaces are referred to in the documentation by their equivalent C++ class names. The **Initialize** method of the **IDirectPlay8Peer** interface is thus referred to as **IDirectPlay8Peer::Initialize**. The primary reason for this convention is that different interfaces may export a method with the same name but with entirely different functionality and syntax. For example, many interfaces have an **Init** or **Initialize** method, but the functionality and parameters may be quite different. Using the C++ class name is a convenient way to uniquely identify the method.

The text and sample code generally uses C++ conventions for calling COM methods. See Using C to Access COM Objects, for a discussion of how to convert a C++ method call to its C equivalent.

IUnknown Interface

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
All COM objects support an interface called **IUnknown**. This interface provides Microsoft® 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 Microsoft Direct3D® 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 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

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
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 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.
Header: Declared in Unknwn.h.

IUnknown::QueryInterface

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
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* ppvObj
);
```

Parameters

riid
: Reference identifier of the interface being requested.

ppvObj
: Address of a pointer to fill 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 Microsoft® 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 enables 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.
Header: Declared in Unknwn.h.

IUnknown::Release

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
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 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 the method explicitly created in a previous call to **IUnknown::AddRef**, **IUnknown::QueryInterface**, or a creation function such as **Direct3DCreate8**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Unknwn.h.

Using Callback Functions

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
A *callback function* is essentially an event handler that is implemented by an application and called by the system. Microsoft® Windows® applications typically implement multiple callback functions, each one designed for a particular set of events. When an event occurs, the system notifies the application by calling the appropriate callback function. The callback function also usually has a parameter list that the system can use to pass the application more detailed information about the event. The most common example of a callback function is the *window procedure*. This function is used by the system to pass Windows messages to the applications that owns the window.

Microsoft DirectX® uses callback functions for a variety of purposes. For example, your system supports multiple devices. Microsoft DirectInput® represents each device by a device object, that contains the details of the device's capabilities. Your application will typically need to enumerate the available devices and to examine the device objects in order to handle user input properly. To do this enumeration, you must implement a **DIEnumDeviceObjectsCallback** callback function.

You start the enumeration process by calling **IDirectInputDevice8::EnumObjects** and passing the method a pointer to your **DIEnumDeviceObjectsCallback** callback function. The system will call this function once for each device, and it will pass in a **DIDEVICEOBJECTINSTANCE** structure containing information about the device's capabilities. After your callback function has processed the information, it can return DIENUM_CONTINUE to request the next device object, or DIENUM_STOP to stop the enumeration.

For more information, see Implementing a Callback Function.

DirectX uses a number of other callback functions for a variety of purposes. For details, see the documentation for the particular DirectX component.

Implementing a Callback Function

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
Because callback functions have different purposes and usage, they are documented in the appropriate Reference section in much the same way as a regular function. However, a callback function reference is essentially a template that describes how to implement the function, not a normal API reference. The callback function reference provides the following information:

- How the function is used
- The function's syntax
- An explanation of the information that will be contained by each parameter
- An explanation of the possible return values

You should declare the function as a **CALLBACK** or **WINAPI** type. Either type is acceptable. You can use any function name that you wish. The name used in the documentation is simply a convenient label for a particular callback function.

Implement the function according to the description in the reference. The implementation details will depend on the particular function and the requirements of your application. See the sample code for some examples of how to implement various callback functions.

Pass a pointer to the function to the appropriate Microsoft® DirectX® component. The DirectX component can then use the function pointer to call the function. The way in which you pass this pointer varies from function to function, so you should see the particular function's reference for details.

The following code fragment is borrowed from the Microsoft DirectInput® Joystick sample. It sketches out the essential elements of a **DIEnumDeviceObjectsCallback** implementation that is used to enumerate the axes of a joystick.

```
//The function declaration
```

```
BOOL CALLBACK EnumAxesCallback( const DIDEVICEOBJECTINSTANCE* pdidoi,
                                VOID* pContext );
...
//Pass the function pointer to DirectInput by calling the
//IDirectInputDevice8::EnumObjects method
if ( FAILED( hr = g_pJoystick->EnumObjects( EnumAxesCallback,
                                            (VOID*)hDlg,
                                             DIDFT_AXIS )));
...
//The function implementation
BOOL CALLBACK EnumAxesCallback( const DIDEVICEOBJECTINSTANCE* pdidoi,
                                VOID* pContext )
{
//Process the information passed in through the two parameters
//Return DIENUM_CONTINUE to request the next device object
//Return DIENUM_STOP to stop the enumeration
}
```

Version Checking

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
Applications sometimes need to know which version of Microsoft® DirectX® is currently available on the system. For example, if an older version of DirectX is on the system, your application may need to scale itself to the capabilities of that version or install the most recent version.

There is no direct way to obtain the DirectX version number. However, each version has a characteristic set of objects and interfaces. Because any version of DirectX supports all previous versions, this set of interfaces and objects will be supported by the version in which they are introduced and all subsequent versions. Thus, the preferred way to determine whether your desired version is available is to test for its characteristic objects or interfaces. As long as those are present, your application will work normally even though you might be using a more recent version of DirectX.

For example, suppose you need version 6.1 support. The Microsoft DirectMusic® object (CLSID_DirectMusic) was introduced in DirectX version 6.1. You can test for the presence of the DirectMusic object by attempting to create it with **CoCreateInstance**. If you are successful, you have version 6.1 or later, and you will be able to use all the DirectX 6.1 capabilities.

Rather than provide a detailed list here of each version's characteristic interfaces and objects, you should refer to the DirectX Software Development kit's sample section. One of the samples is a function, **GetDXVersion**, that includes tests for all DirectX versions. **GetDXVersion** returns an integer that corresponds to the DirectX version that is present on the system. As long as this integer is greater than or equal to your desired version number, your application will run normally. You can find the sample code under your SDK root folder at \Samples\Multimedia\DXMisc\GetDXVer.

For more information, see Checking the Operating System Version.

Checking the Operating System Version

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
In addition to determining the Microsoft® DirectX® version, applications may also need to know what operating system they are running on, and perhaps which Service Pack has been installed. The preferred way to check the operating system version is to use the Microsoft Windows® function, **GetVersionEx.** This function returns an OSVERSIONINFO structure that contains a variety of information including:

- Whether the system is Microsoft Windows NT®-based or Windows 95/98/ME
- The major and minor version numbers
- The service pack number, for Windows NT-based systems

The general procedure is to determine the earliest version of the operating system that your application is compliant with. If that version or later is installed, you can safely install and run your application.

Note

The operating system information returned by **GetVersionEx** should not be used to test for the presence or version number of DirectX. In particular, the fact that a system is Windows NT-based does not necessarily mean that DirectX is absent. The Windows 2000 operating system, for example, includes DirectX 7.0 or later. Future versions of Windows NT-based operating systems can also be expected to include a version of DirectX. Use the procedures described in the previous section to determine whether DirectX is present and, if so, the version number.

The following sample function illustrates how to use **GetVersionEx** to test the operating system version. If the installed version is identical to or more recent than the version specified in the parameter list, the function returns TRUE. You can then safely install and execute your application. Otherwise the function returns FALSE, indicating that your application will not run properly, if at all.

```
#include <windows.h>
#include <stdio.h>
#include <tchar.h>

BOOL bIsWindowsVersionOK(   DWORD dwWin9xMajor, DWORD dwWin9xMinor,
                    DWORD dwWinNTMajor, DWORD dwWinNTMinor, WORD
wWinNTSPMajor )
{
   OSVERSIONINFO          osvi;

   // Initialize the OSVERSIONINFO structure.
   ZeroMemory( &osvi, sizeof( osvi ) );
   osvi.dwOSVersionInfoSize = sizeof( osvi );

   GetVersionEx( &osvi );  // Assume this function succeeds.

   // Split code paths for NT and Win9x
   if( osvi.dwPlatformId == VER_PLATFORM_WIN32_WINDOWS )
   {
      // Check the major version.
      if( osvi.dwMajorVersion > dwWin9xMajor )
         return TRUE;
      else if( osvi.dwMajorVersion == dwWin9xMajor )
      {
         // Check the minor version.
         if( osvi.dwMinorVersion >= dwWin9xMinor )
            return TRUE;
      }
   }
   else if( osvi.dwPlatformId == VER_PLATFORM_WIN32_NT )
   {
      // Check the major version.
      if( osvi.dwMajorVersion > dwWinNTMajor )
         return TRUE;
      else if( osvi.dwMajorVersion == dwWinNTMajor )
      {
         // Check the minor version.
         if( osvi.dwMinorVersion > dwWinNTMinor )
            return TRUE;
         else if( osvi.dwMinorVersion == dwWinNTMinor )
         {
            // Check the service pack.
            DWORD dwServicePack = 0;

            if( osvi.szCSDVersion )
            {
                _stscanf(   osvi.szCSDVersion,
                            _T("Service Pack %d"),
                            &dwServicePack );
            }
            return ( dwServicePack >= wWinNTSPMajor );
        }
    }
  }
  return FALSE;
}
```

Compiling DirectX Samples and Other DirectX Applications

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
This section provides information about considerations specific to compiling Microsoft® DirectX® applications. DirectX 8.0 supports only Microsoft Visual C++ 5.0 SP3, and later.

- Preparing for Compilation
- Component Version Constants

Preparing for Compilation

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
The samples included in this SDK use Microsoft® Visual C++® project files (DSP 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. The information provided here applies to the Microsoft DirectX® samples and the DirectX applications that you create.

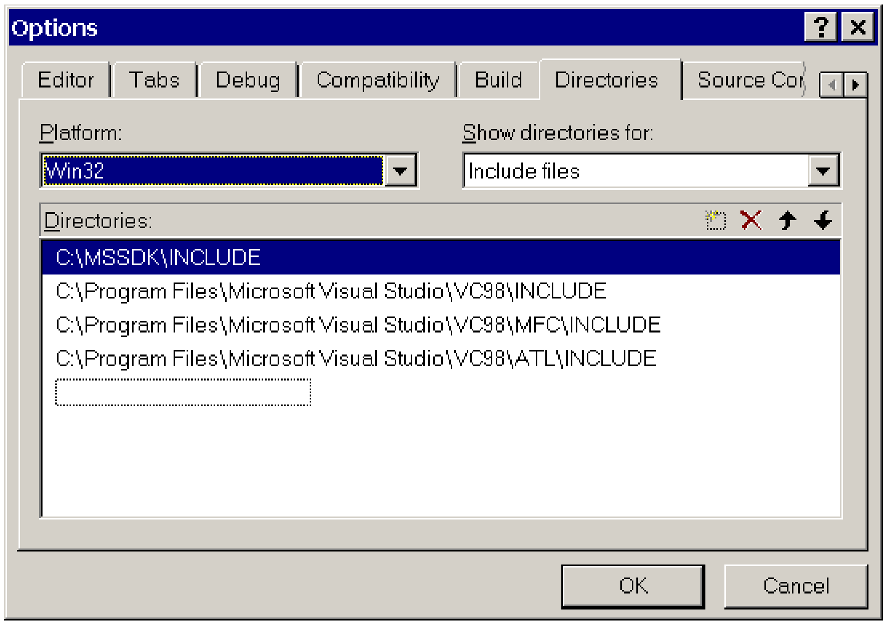
After opening a project file in Visual C++, verify some settings before compiling the application. The following descriptions are valid for Visual C++ 5 and 6.

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 and select the **Directories** tab. The following dialog box will appear.



Platform: Win32

Show directories for: Include files

Directories:

C:\MSSDK\INCLUDE
C:\Program Files\Microsoft Visual Studio\VC98\INCLUDE
C:\Program Files\Microsoft Visual Studio\VC98\MFC\INCLUDE
C:\Program Files\Microsoft Visual Studio\VC98\ATL\INCLUDE

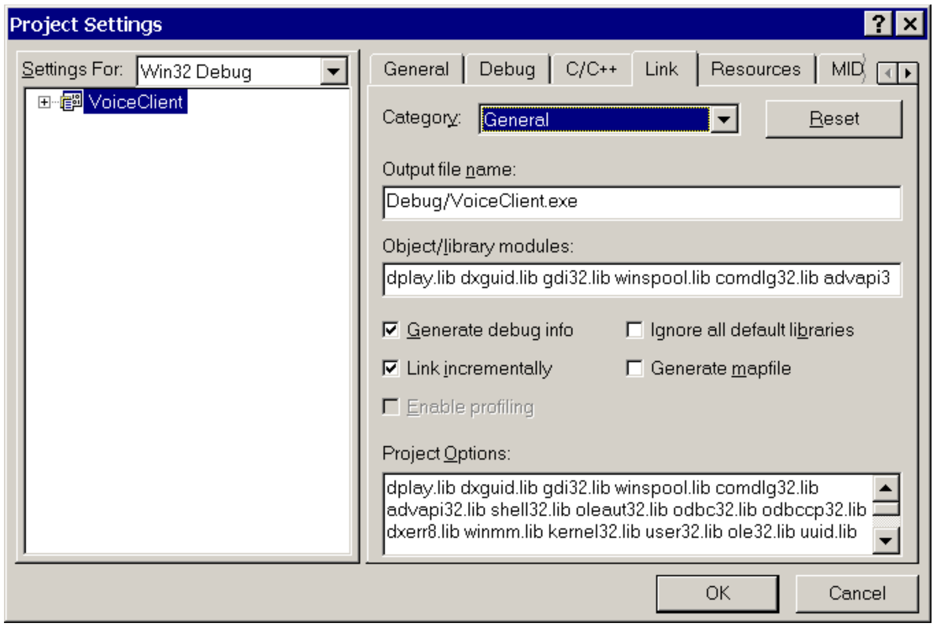
The topmost path indicates the folder that contains the latest DirectX header files. The default path is C:\Mssdk\Include. If the path is not present, add it to the list and move it to the top of the search list by using the toolbar controls within the **Directories** tab.

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 and select the **Directories** tab. When the dialog box appears, choose the **Library files** option in the **Show directories for** list box. The topmost path should be the folder that contains the latest DirectX link libraries. The default path is C:\Mssdk\Lib.

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 project files. For new applications, on the **Project** menu, click **Settings**. The following dialog box appears.



Settings For: Win32 Debug

VoiceClient

Category: General

Output file name:
Debug/VoiceClient.exe

Object/library modules:
dplay.lib dxguid.lib gdi32.lib winspool.lib comdlg32.lib advapi3

☑ Generate debug info
☐ Ignore all default libraries
☑ Link incrementally
☐ Generate mapfile
☐ Enable profiling

Project Options:
dplay.lib dxguid.lib gdi32.lib winspool.lib comdlg32.lib advapi32.lib shell32.lib oleaut32.lib odbc32.lib odbccp32.lib dxerr8.lib winmm.lib kernel32.lib user32.lib ole32.lib uuid.lib

You should verify that the application is linked to the appropriate standard DirectX link libraries.

Component Version Constants

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
For backward compatibility with earlier versions of Microsoft® 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 Microsoft Direct3D® header file, D3D8.h, is shown here.

```
#ifndef DIRECT3D_VERSION
#define DIRECT3D_VERSION        0x0800
#endif  //DIRECT3D_VERSION
```

You can define other values for these constants to use newer versions of the header files with older versions of the components. For example, to use the latest headers to compile against the DirectX 7.0 version of D3D, define DIRECT3D_VERSION to be 0x0700.

Debugging DirectX Applications

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
This section covers the following topics pertaining to debugging C and C++ applications.

- Debug vs. Retail DLLs
- The DirectX 8.0 Error Handling Utility Library

Debug vs. Retail DLLs

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
The Microsoft® DirectX® SDK installation program provides the option of installing either debug or retail builds of the DirectX dynamic-link libraries (DLLs).

When you develop software in C++, it is best to install the debug versions of the DLLs. This option installs both debug and retail DLLs on your system. The retail option installs only the retail 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.

You can use the DirectX Control Panel utility to switch between the debug and retail builds of Microsoft DirectInput®, Direct3D®, and DirectMusic®. To enable this feature, select the debug option when you install the SDK.

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

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

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

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\—and 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. Therefore, for debug builds, you must 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.

The DirectX 8.0 Error Handling Utility Library

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
The Microsoft® DirectX® 8.0 SDK includes a pair of utility functions and several macros designed to simplify the task of debugging your application. These functions and macros are contained in the Dxerr8.lib library. The declarations are in Dxerr8.h.

- DirectX 8.0 Error Handling Functions
- DirectX 8.0 Error Handling Macros

DirectX 8.0 Error Handling Functions

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
The following functions and macros are included in the DirectX error handling utility library.

- **DXGetErrorString8**
- **DXTrace**

DXGetErrorString8

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
Returns the name associated with a Microsoft® DirectX error® code.

```
TCHAR* DXGetErrorString8(
  HRESULT hr
);
```

Parameters

hr

[in] HRESULT value returned from a DirectX method. This handles only error codes.

Return Values

If successful, this function returns a pointer to a string that contains the name of the error code. If Unicode is set, **DXGetErrorString8** will return a Unicode string. Otherwise, it will return an ANSI string.

Remarks

This function is designed to retrieve the text equivalent of a DirectX error message from a Microsoft Direct3D®, D3DX, DirectPlay®, DirectInput®, DirectMusic®, or DirectSound® method. For example, if you set *hr* to 0x88768686, **DXGetErrorString8** will return D3DERR_DEVICELOST.

If an error code maps to more than one text string, **DXGetErrorString8** will return a generic string. For example, there are several DirectX error codes, such as DIERR_OUTOFMEMORY, that indicate that you are out of memory, and all map to the same value. If you set *hr* to any of these codes, **DXGetErrorString8** will return E_OUTOFMEMORY.

Requirements

Windows NT/2000: Requires Windows® 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Version: Requires DirectX 8.0.
Header: Declared in Dxerr8.h.
Import Library: Use Dxerr8.lib

DXTrace

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
Displays a message box, and passes the error code to **DXGetErrorString8**.

```
HRESULT DXTrace(
  char *strFile,
  DWORD dwLine,
  HRESULT hr,
  TCHAR *strMsg,
  BOOL bPopMsgBox
);
```

Parameters

strFile
: [in] Pointer to the name of the current file. You can set this parameter easily with the __FILE__ macro.

dwLine
: [in] Line number. You can set this parameter easily with the __LINE__ macro.

hr
: [in] HRESULT containing an error code. This value will be passed to **DXGetErrorString8** and converted to the equivalent name.

strMsg
: [in] Pointer to an optional message that will be displayed along with the file name, line number, and HRESULT.

bPopMsgBox
: [in] Specifies whether a message box should be displayed. If *bPopMsgBox* is set to TRUE and *hr* is non-zero, a message box will be displayed containing the values of the first four parameters. If *bPopMsgBox* is set to FALSE, the information is passed to the debugger.

Return Values

Returns zero if successful, or a non-zero value if not.

Requirements

Windows NT/2000: Requires Windows® 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Version: Requires DirectX® 8.0.
Header: Declared in Dxerr8.h.
Import Library: Use Dxerr8.lib

DirectX 8.0 Error Handling Macros

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
The following macros simplify using the **DXTrace** function.

- **DXTRACE_MSG**
- **DXTRACE_ERR**
- **DXTRACE_ERR_NOMSGBOX**

DXTRACE_MSG

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
Passes a string to the debugger.

```
HRESULT DXTRACE_MSG(
  char *str,
);
```

Parameters

str
    Pointer to a string that will be passed to the debugger.

Return Values

Returns zero if successful, or a non-zero value if not.

Remarks

The string is accompanied by the file name and line number. The macro declaration is:

```
DXTRACE_MSG(str) DXTrace( __FILE__, (DWORD)__LINE__, 0, str, FALSE )
```

Requirements

Windows NT/2000: Requires Windows® 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Version: Requires DirectX® 8.0.
Header: Declared in Dxerr8.h.

DXTRACE_ERR

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
Displays a message box with error information.

```
HRESULT DXTRACE_ERR(
  char *str,
  HRESULT hr
);
```

Parameters

str
: Pointer to a string to be displayed in the message box.

hr
: HRESULT containing an error code. This value will be passed to **DXGetErrorString8** and converted to the equivalent name.

Return Values

Returns zero if successful, or a non-zero value if not.

Remarks

The string and error name is accompanied by the file name and line number. The macro declaration is:

```
DXTRACE_ERR(str,hr) DXTrace(__FILE__,(DWORD)__LINE__,hr,str,TRUE )
```

Requirements

Windows NT/2000: Requires Windows® 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Version: Requires DirectX® 8.0.
Header: Declared in Dxerr8.h.

DXTRACE_ERR_NOMSGBOX

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
Passes error information to the debugger.

```
HRESULT DXTRACE_ERR_NOMSGBOX(
  char *str,
  HRESULT hr
);
```

Parameters

str

Pointer to a string to be passed to the debugger.

hr

HRESULT containing an error code. This value will be passed to **DXGetErrorString8** and converted to the equivalent name.

Return Values

Returns zero if successful, or a non-zero value if not.

Remarks

The string and error name is accompanied by the file name and line number. The macro declaration is:

```
DXTRACE_ERR_NOMSGBOX(str,hr) DXTrace(__FILE__,(DWORD)__LINE__,hr,str, FALSE)
```

Requirements

Windows NT/2000: Requires Windows® 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Version: Requires DirectX® 8.0.
Header: Declared in Dxerr8.h.

Programming DirectX with Visual Basic

This section is an introduction to developing applications with Microsoft® DirectX® for Microsoft Visual Basic®.

The following topics are covered:

- Referencing the Type Library
- Creating DirectX Objects

- Using GUIDs
- Passing Arrays to Methods
- Using Flags
- Using Bitmasks
- DirectX Enumerations
- The IUnknown Data Type
- Error Handling
- DirectX Class Reference

Referencing the Type Library

To use Microsoft® DirectX® for Microsoft Visual Basic® in a project, you must first ensure that the project has access to the type library.

To make sure that the project has access to the type library

1. In Microsoft Visual Studio®, on the **Project** menu, click **References**.
2. Select **DirectX 8 for Visual Basic Type Library**, and then click **OK**.

Creating DirectX Objects

Every Microsoft® DirectX® for Microsoft Visual Basic® application must have a **DirectX8** object. This is the parent object whose methods are used to create the principal objects for each DirectX component. The **DirectX8** object has many other methods, such as methods to perform calculations for Microsoft Direct3D®.

To create a **DirectX8** object, declare it as a new object. For example:

```
Public gObjDX As New DirectX8
```

Many of the primary DirectX component objects, such as **DirectSound8** are not created directly. Instead, you must create those objects by calling the appropriate **DirectX8** method. For example, to create a **DirectSound8** object, create a variable and assign it the return value of **DirectX8.DirectSoundCreate**.

```
Public gObjDSound As DirectSound8
Set gObjDSound = DirectX8.DirectSoundCreate(vbNullString)
```

The following objects must be created by calling the appropriate **DirectX8** method. See the DirectX Class Reference for details.

Direct3D8

DirectInput8

DirectMusicComposer8

DirectMusicLoader8

DirectMusicPerformance8

DirectPlay8Address

DirectPlay8Client

DirectPlay8LobbiedApplication

DirectPlay8LobbyClient

DirectPlay8Peer

DirectPlay8Server

DirectPlayVoiceClient8

DirectPlayVoiceServer8

DirectPlayVoiceTest8

DirectSoundCapture8

DirectSound8

DirectSoundEnum8

DirectXFile

Using GUIDs

Microsoft® DirectX® uses globally unique identifiers (GUIDs) extensively for purposes such as identifying objects. GUIDs are actually 128-bit integers, but they are commonly represented by their equivalent string, such as {AC330441-9B71-11D2-9AAB-0020781461AC}.

DirectX8 has a number of predefined GUIDs that are used to identify entities such as particular properties. If a method requires you to pass in a predefined GUID, it is handled as a **String** data type, and the string form of the GUID is expected. The most convenient approach is to use a string alias, if one exists. For example, the **DirectInput8.CreateDevice** method enables you to create a device for the standard keyboard by passing in **GUID_SysKeyboard** rather than the actual GUID string. You can also use the actual string, for example by copying and pasting it from a C++ header file.

To create a GUID

1. In the \Samples\Multimedia\VBSamples\DXMisc\Bin folder of the DirectX SDK, run Vbguidgen.exe.
2. Paste the GUID into your Microsoft Visual Basic® application, and place quotation marks around it.

You can also generate GUIDs at run time by calling the **DirectX8.CreateNewGuid** method.

Passing Arrays to Methods

Some methods, such as **DirectInputDevice8.SendDeviceData**, take a typed array as a parameter. For example, in the following code fragment, *formatArray* is an array of **DIDEVICEOBJECTDATA** types, *lCount* contains the number of elements in the array, and *lFlags* is a flag parameter that controls how the data is sent.

```
MyDevice.SendDeviceData(lCount, formatArray(), lFlags)
```

A few **DirectX8** methods use parameters with an **Any** type to return data. An example is the **DirectSoundCaptureBuffer8.ReadBuffer** method's *buffer* parameter. Typically you will want such methods to return their data as an array of bytes. However, you cannot simply pass the name of the array to the method because Microsoft® Visual Basic® arrays contain extra information ahead of the actual data. Instead, you must pass the first element of the array. For example, in the following code fragment, *dsCaptureBuffer* is a **DirectSoundCaptureBuffer8** object, *nBytes* is a **Long** value containing the number of bytes to be read, and *buffer* is a zero-based array of bytes:

```
ReDim buffer(nBytes)
Call dsCaptureBuffer.ReadBuffer(0, nBytes, buffer(0), 0)
```

Using Flags

Many object methods in Microsoft® DirectX® for Microsoft Visual Basic® have a parameter with the **Long** data type named *flags* or something similar. Types may also have one or more flag members. These *flag fields* are a compact way to select from a variety of options and are often used to specify the exact behavior of a method. For example, the *flags* parameter of **DirectInput8.GetDevicesBySemantics** enables you to define the scope of the enumeration by selecting from one of several options.

A flag is a single yes-no bit of information. In a flag field, each bit of the integer corresponds to a separate flag that represents an independent piece of information. If the bit is set, the flag is enabled. If the bit is zero, the flag is disabled. Flag fields typically contain a set of related flags that are represented by an enumeration. Normally, the numerical value of each member of the enumeration represents a different bit in the flag field. However, some members may represent combinations of two or more flags.

A common use of flag fields is to enable you to select one option from a variety of options. You normally do so by assigning the appropriate enumerated value to the flag parameter. However, because each flag corresponds to a different bit, flag fields can also be used to select two or more options from a list. To do so, set all the relevant bits by combining the enumerated values of the selected flags with the **Or** operator. For example, the **DirectInputDevice8.SetCooperativeLevel** method has a flag field that is used to specify the device's cooperative level. The following code fragment selects a combination of foreground and exclusive cooperative levels by combining the DISCL_FOREGROUND and DISCL_EXCLUSIVE flags.

```
' diDevice is a DirectInputDevice8 object, and hwnd is a window handle
diDevice.SetCooperativeLevel(hwnd, _
   DISCL_FOREGROUND Or DISCL_EXCLUSIVE)
```

Note

Examine the reference page carefully to determine how the flag field is to be used. Setting multiple flags in a flag field that is designed to have only one flag set may have unpredictable results.

You should not use the addition operator instead of the **Or** operator to combine flags. While the two operations are often equivalent, using the addition operator will sometimes lead to incorrect results. In particular, the two operators are usually not equivalent if one of the flags represents a combination of two or more flags. The following example illustrates this case. The CF_PURPLE flag represents a combination of CF_RED and CF_BLUE.

```
Enum COLORFLAGS
   CF_RED    = 1
   CF_BLUE   = 2
   CF_PURPLE = 3
   CF_GREEN  = 4
End Enum

Dim Color As Integer
Color = CF_RED Or CF_PURPLE
```

The value of *Color* is now 3. Setting CF_RED has no effect, as that bit was already set by CF_PURPLE.

The following operation, on the other hand, sets an incorrect value:

```
' This is wrong!
Color = CF_RED + CF_PURPLE
```

Adding the two flag values sets *Color* to 4, which is equivalent to CF_GREEN.

When you retrieve a value for flags, you can determine whether an individual flag is set by combining the flag value with the flag field using the **And** operator. If the bit in the flag field represented by a flag constant is set, an **And** operation will return a nonzero value indicating that the flag is "on". If the **And** operation returns zero, the flag is "off". For example the *caps* parameter of **DirectInputDevice8.GetCapabilities** method is used to return a **DIDEVCAPS** type that contains the device capabilities. The *lFlags* member of this type is a flag field that may have one or more of the flags specified by the **CONST_DIDEVCAPSFLAGS** enumeration set. If the DIDC_ATTACHED flag is set, the device is physically attached to the system. The following code fragment illustrates how to check whether the device is attached by examining the value of an **And** operation that combines the *lFlags* member with DIDC_ATTACHED.

```
Dim IsAttached As Boolean
IsAttached = diDevCaps.lFlags And DIDC_ATTACHED
```

Note that you should not simply test whether *lFlags* is equal to DIDC_ATTACHED. This flag field may have more than one flag set. If multiple flags are set, the following test will fail, even if DIDC_ATTACHED is set.

```
' This is wrong!
IsAttached = (diDevCaps.lFlags = DIDC_ATTACHED)
```

Using Bitmasks

Some Microsoft® DirectX® types, such as **D3DCAPS8**, have **Long** members that serve as bitmasks. Bitmasks are similar to flag fields, but they are normally used to mask off certain bits in a field. It is customary to use hexadecimal format to assign values to these bitmasks because it is easier to determine which bits will be affected. However, doing so may cause an error because Microsoft Visual Basic® normally attempts to convert hexadecimal constants to the shortest type. For example, &HFF would be converted to an **Integer** with a value of -1. If this value is then passed back to DirectX for Visual Basic, where a **Long** is expected, it is converted to &HFFFFFFFF.

To ensure that a hexadecimal bitmask is properly converted, place a second ampersand after the expression. For example:

```
DDSD.ddpfPixelFormat.lRBitMask = &HFF&
```

DirectX Enumerations

Most Microsoft® DirectX® applications must enumerate available resources, usually during initialization. For example, an application that uses DirectX Graphics might need to find out what display modes are available, or an application using Microsoft DirectInput® might need to enumerate the available buttons and axes on a joystick.

In DirectX for Microsoft Visual Basic®, applications handle enumeration by obtaining the appropriate enumeration object. When an enumeration object is created, it builds a collection that exists as long as the object exists. Your application can then access that collection by using enumeration object's methods.

DirectX supports many enumeration objects, each tailored to a specific task. You normally obtain an enumeration object by calling a method. For example, the **DirectX8** class contains two such methods. One of them, **DirectX8.GetDSEnum** returns a **DirectSoundEnum8** enumeration object. This object enables you to enumerate Microsoft DirectSound® devices to obtain a GUID for a suitable device. You can then use this GUID to create a **DirectSound8** object.

Other enumeration objects are obtained through various DirectX component objects. For example, you can call the **DirectInput8.GetDIDevices** method to obtain an enumeration object for input devices.

The following sample code illustrates how to enumerate all input devices attached to the system. The *di* variable is a **DirectInput8** object. The first step is to obtain a **DirectInputEnumDevices8** enumeration object.

```
Dim diEnum As DirectInputEnumDevices8
Set diEnum = di.GetDIDevices(0, DIEDFL_ATTACHEDONLY)
```

Next, use the **DirectInputEnumDevices8** methods to iterate through the attached devices. Examine each device to determine if it has the particular capabilities you need. In the following example, the names of the devices are put in a list box:

```
Dim diDevice As DirectInputDeviceInstance
Dim X As Integer

For X = 1 To diEnum.GetCount
    Set diDevice = diEnum.GetItem(X)
    Call List1.AddItem(diDevice.ProductName)
    'Examine device capabilities
    '...
Next X
```

Note
All collections created by DirectX enumerations are 1-based.

The IUnknown Data Type

The **IUnknown** interface is exposed by all COM objects. The interface is used by C++ developers to gain access to an object's functionality and to manage the object's lifetime. Microsoft® Visual Basic® developers rarely have to make use of **IUnknown**. However, an object's **IUnknown** interface is represented in Visual Basic by the **IUnknown** data type.

A handful of Microsoft DirectX® 8.0 Visual Basic methods have return values or parameters that are declared as **Unknown**. The type of these values is undefined, and will be either an **IUnknown** type, or another object type. DirectX 8.0 uses **Unknown** declarations for two cases.

When an object is not represented by a class. Such objects expose no methods or properties to your application. However, a reference may be passed to your application by one method so that you can then pass that reference to another method. For example, the return value of the **DirectMusicSegment8.GetAudioPathConfig** is declared as **Unknown**. The method returns an **IUnknown** value that is a reference to a configuration object that is not represented by a class. Your application then passes the object reference to another method as a parameter.

When a parameter needs to accept more than one object type. Declaring a parameter as **Unknown** allows it to accept any object type. For example, the first parameter of **DirectMusicBand8.Unload** is declared as **Unknown**. The parameter can accept either a **DirectMusicAudioPath8** or a **DirectMusicPerformance8** object reference.

> **Note** An **Unknown** declaration is similar to an **Any** declaration. When a parameter or return value is declared as **Any**, you can pass or set any data type. Similarly, when a parameter or return value is declared as **Unknown**, you can pass or set any object type. You must be careful to use the correct object type, however, or the results will be unpredictable.

An **IUnknown** variable does not provide direct access to an object's methods or to the **QueryInterface**, **Addref**, and **Release** methods that are used in C++. When you have an object reference in an **IUnknown** variable, you must set that variable to an appropriately declared class variable in order to use the object's methods.

The following code fragment illustrates how to use the **IUnknown** data type.

```
'dmSeg is a DirectMusicSegment8 object
'dmPerf is a DirectMusicPerformance8 object

Dim config as IUnknown
Dim audioPath As DirectMusicAudioPath

Set config = dmSeg.GetAudioPathConfig
Set audioPath = dmPerf.CreateAudioPath(config)
```

The value that is returned by **DirectMusicSegment8.GetAudioPathConfig** is a reference to a configuration object that is not represented by a class. All you need to do with this object reference is pass it to another method, in this case, DirectMusicPerformance8.CreateAudioPath.

The following use of the I**Unknown** data type is incorrect.

```
'objDiDev is a DirectInputDevice8 object

Dim unk As IUnknown
Set unk = objDiDev
NumItems = unk.GetDeviceData(diDeviceData,0) 'Incorrect!
```

The attempt to call the **DirectInputDevice8.GetDeviceData** method will raise a "method not found" error.

Error Handling

In Microsoft® Visual Basic® a method call either succeeds or it fails. If the call fails, an error is raised, and an error number is set in the global Visual Basic **Err** object. Your application must then branch to an error handler that examines the **Err** object and deals with the error, or execution terminates. However, underlying every Microsoft DirectX® Visual Basic method is a C++ method that handles errors somewhat differently. In some cases, you will need to know something about C++ return values to interpret Visual Basic errors.

In C and C++, every method call returns an **HRESULT** value. This value is an integer with a specific format that indicates the outcome of the operation. In some cases, the **HRESULT** that a method returns simply indicates whether the operation succeeded or failed. However, methods may sometimes use many **HRESULT** values to indicate a variety of success or failure modes. There may, in fact, be more than one return value that indicates success. Many of **HRESULT** values are system standards, and are defined in the Winerror.h header file. System standard values typically use an S_ prefix for success codes and an E_ prefix for failure. For example, S_OK and E_FAIL are commonly used codes for success and failure, respectively. When the C++ code that typically underlies a Visual Basic method returns a standard **HRESULT** value, the Visual Basic run time traps that value, and sets a standard Visual Basic error code.

Many DirectX methods also support one or more non-standard **HRESULT** values. These values are normally defined in an enumeration, and the meaning of each value is given in the method reference. Because they are not standard values, there is no corresponding Visual Basic error code. Instead, the Visual Basic run time sets **Err.Number** to the **HRESULT** value. Refer to the method reference for the meaning of the value. For more information on **HRESULT** values, see the COM documentation in the Microsoft Platform Software Development Kit (SDK).

DirectX Class Reference

This section contains references for the following classes that are common to all components of Microsoft® DirectX®:

- **DirectX8**
- **DirectXEvent8**

DirectX8

#

The **DirectX8** class serves as a starting point for the Microsoft® DirectX® component technologies. It contains methods that create the primary objects for the various DirectX components as well as some methods that are used by multiple components. This class is usually the first object created in your DirectX application.

The methods of the **DirectX8** class can be organized into the following groups:

Object Creation	**Direct3DCreate**
	DirectInputCreate

# IDH_DirectX8_vbintro

	DirectMusicComposerCreate
	DirectMusicLoaderCreate
	DirectMusicPerformanceCreate
	DirectPlayAddressCreate
	DirectPlayClientCreate
	DirectPlayLobbiedApplicationCreate
	DirectPlayLobbyClientCreate
	DirectPlayPeerCreate
	DirectPlayServerCreate
	DirectPlayVoiceClientCreate
	DirectPlayVoiceServerCreate
	DirectPlayVoiceSetupCreate
	DirectSoundCaptureCreate
	DirectSoundCreate
	DirectXFileCreate
Enumeration	**GetDSCaptureEnum**
	GetDSEnum
Event Handling	**CreateEvent**
	DestroyEvent
	SetEvent
GUID Creation	**CreateNewGuid**

DirectX8.CreateEvent

#

Creates a handle for the form's event object.

object.**CreateEvent(_**
  event **As DirectXEvent8 _**
) As Long

# IDH_DirectX8.CreateEvent_vbintro

Parts

object
Object expression that resolves to a **DirectX8** object.

event
The form's **DirectXEvent8** object.

Return Values

Returns a **Long** value set to the form's event handle.

Error Codes

If the method fails, an error is raised and **Err.Number** is set.

Remarks

A **DirectXEvent** object must be implemented by a form in order to receive notifications from Microsoft® DirectX®. This method creates an event handle for this object that is used by the notification methods of Microsoft DirectSound®, DirectMusic®, DirectPlay®, and DirectInput®.

DirectX automatically sets the event object to be signaled when an appropriate event occurs. You can also set the event manually by calling the **DirectX8.SetEvent** method. This procedure is normally used to test the event-handling code.

Applications must explicitly destroy all the events they create by calling the **DirectX8.DestroyEvent** method. Failure to do so will cause unpredictable results.

DirectX8.CreateNewGuid

#

Generates a new globally unique identifier (GUID).

object.**CreateNewGuid() As String**

Parts

object
Object expression that resolves to a **DirectX8** object.

Return Values

Returns a **String** value containing the new GUID.

# IDH_DirectX8.CreateNewGuid_vbintro

Error Codes

If the method fails, an error is raised and **Err.Number** is set.

DirectX8.DestroyEvent

#

Deletes an event handle created by the **DirectX8.CreateEvent** method.

object.**DestroyEvent(_**
  eventid **As Long)**

Parts

object
  Object expression that resolves to a **DirectX8** object.

eventid
  Long value set to the event handle to be destroyed. This handle must have been created by calling **DirectX8.CreateEvent**.

Error Codes

If the method fails, an error is raised and **Err.Number** is set.

DirectX8.Direct3DCreate

#

Creates a **Direct3D8** object.

object.**Direct3DCreate() As Direct3D8**

Parts

object
  Object expression that resolves to a **DirectX8** object.

Return Values

Returns a **Direct3D8** object.

Error Codes

If the method fails, an error is raised and **Err.Number** is set.

# IDH_DirectX8.DestroyEvent_vbintro
# IDH_DirectX8.Direct3DCreate_vbintro

Remarks

This function succeeds in all cases except out-of-memory conditions. It creates a **Direct3D8** object that supports enumeration and enables the creation of **Direct3DDevice8** objects.

Note that calling this method samples the current set of active display adapters. If the user dynamically adds adapters, either by adding devices to the desktop or by hot-docking a laptop, then those devices will not be enumerated for the lifetime of this Direct3D8 object. Creating a new Direct3D8 object will expose the new devices.

DirectX8.DirectInputCreate

#

Creates a **DirectInput8** object.

object.**DirectInputCreate() As DirectInput8**

Parts

object
Object expression that resolves to a **DirectX8** object.

Return Values

Returns a **DirectInput8** object.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following Microsoft® DirectInput® error codes:

- DIERR_BETADIRECTINPUTVERSION
- DIERR_INVALIDPARAM
- DIERR_OLDDIRECTINPUTVERSION
  - DIERR_OUTOFMEMORY

DirectX8.DirectMusicComposerCreate

#

Creates a **DirectMusicComposer8** object.

object.**DirectMusicComposerCreate() As DirectMusicComposer8**

# IDH_DirectX8.DirectInputCreate_vbintro
# IDH_DirectX8.DirectMusicComposerCreate_vbintro

Parts

object

Object expression that resolves to a **DirectX8** object.

Return Values

Returns a **DirectMusicComposer8** object.

Error Codes

If the method fails, an error is raised and **Err.Number** is set.

DirectX8.DirectMusicLoaderCreate

#[1]

Creates a **DirectMusicLoader8** object.

object.**DirectMusicLoaderCreate() As DirectMusicLoader8**

Parts

object

Object expression that resolves to a **DirectX8** object.

Return Values

Returns a **DirectMusicLoader8** object.

Error Codes

If the method fails, an error is raised and **Err.Number** is set.

DirectX8.DirectMusicPerformanceCreate

#[2]

Creates a **DirectMusicPerformance8** object.

```
object.DirectMusicPerformanceCreate() _
    As DirectMusicPerformance8
```

Parts

object

Object expression that resolves to a **DirectX8** object.

# IDH_DirectX8.DirectMusicLoaderCreate_vbintro

# IDH_DirectX8.DirectMusicPerformanceCreate_vbintro

Return Values

Returns a **DirectMusicPerformance8** object.

Error Codes

If the method fails, an error is raised and **Err.Number** is set.

DirectX8.DirectPlayAddressCreate

#

Creates a **DirectPlay8Address** object.

object.**DirectPlayAddressCreate() As DirectPlay8Address**

Parts

object
  Object expression that resolves to a **DirectX8** object.

Return Values

Returns a **DirectPlay8Address** object.

Error Codes

If the method fails, an error is raised and **Err.Number** is set.

DirectX8.DirectPlayClientCreate

#

Creates a **DirectPlay8Client** object.

object.**DirectPlayClientCreate() As DirectPlay8Client**

Parts

object
  Object expression that resolves to a **DirectX8** object.

Return Values

Returns a **DirectPlay8Client** object.

# IDH_DirectX8.DirectPlayAddressCreate_vbintro
# IDH_DirectX8.DirectPlayClientCreate_vbintro

Error Codes

If the method fails, an error is raised and **Err.Number** is set.

DirectX8.DirectPlayLobbiedApplicationCreate

#

Creates a **DirectPlay8LobbiedApplication** object.

object.**DirectPlayLobbiedApplicationCreate() _**
As DirectPlay8LobbiedApplication

Parts

object
Object expression that resolves to a **DirectX8** object.

Return Values

Returns a **DirectPlay8LobbiedApplication** object.

Error Codes

If the method fails, an error is raised and **Err.Number** is set.

DirectX8.DirectPlayLobbyClientCreate

#

Creates a **DirectPlay8LobbyClient** object.

object.**DirectPlayLobbyClientCreate() _**
As DirectPlay8LobbyClient

Parts

object
Object expression that resolves to a **DirectX8** object.

Return Values

Returns a **DirectPlay8LobbyClient** object.

# IDH_DirectX8.DirectPlayLobbiedApplicationCreate_vbintro
# IDH_DirectX8.DirectPlayLobbyClientCreate_vbintro

Error Codes

If the method fails, an error is raised and **Err.Number** is set.

DirectX8.DirectPlayPeerCreate

#[#]

Creates a **DirectPlay8Peer** object.

object.**DirectPlayPeerCreate() As DirectPlay8Peer**

Parts

object
Object expression that resolves to a **DirectX8** object.

Return Values

Returns a **DirectPlay8Peer** object.

Error Codes

If the method fails, an error is raised and **Err.Number** is set.

DirectX8.DirectPlayServerCreate

#[#]

Creates a **DirectPlay8Server** object.

object.**DirectPlayServerCreate() As DirectPlay8Server**

Parts

object
Object expression that resolves to a **DirectX8** object.

Return Values

Returns a **DirectPlay8Server** object.

Error Codes

If the method fails, an error is raised and **Err.Number** is set.

[#] IDH_DirectX8.DirectPlayPeerCreate_vbintro
[#] IDH_DirectX8.DirectPlayServerCreate_vbintro

DirectX8.DirectPlayVoiceClientCreate

[#]

Creates a **DirectPlayVoiceClient8** object.

object.**DirectPlayVoiceClientCreate() _**
As DirectPlayVoiceClient8

Parts

object
Object expression that resolves to a **DirectX8** object.

Return Values

Returns a **DirectPlayVoiceClient8** object.

Error Codes

If the method fails, an error is raised and **Err.Number** is set.

DirectX8.DirectPlayVoiceServerCreate

[#]

Creates a **DirectPlayVoiceServer8** object.

object.**DirectPlayVoiceServerCreate() _**
As DirectPlayVoiceServer8

Parts

object
Object expression that resolves to a **DirectX8** object.

Return Values

Returns a **DirectPlayVoiceServer8** object.

Error Codes

If the method fails, an error is raised and **Err.Number** is set.

[#] IDH_DirectX8.DirectPlayVoiceClientCreate_vbintro
[#] IDH_DirectX8.DirectPlayVoiceServerCreate_vbintro

DirectX8.DirectPlayVoiceSetupCreate

[#]

Creates a **DirectPlayVoiceTest8** object.

object.**DirectPlayVoiceSetupCreate() As DirectPlayVoiceSetup8**

Parts

object
: **Object** expression that resolves to a **DirectX8** object.

Return Values

Returns a **DirectPlayVoiceSetup8** object.

Error Codes

If the method fails, an error is raised and **Err.Number** is set.

DirectX8.DirectSoundCaptureCreate

[#]

Creates a **DirectSoundCapture8** object.

object.**DirectSoundCaptureCreate(_**
    guid **As String _**
) As DirectSoundCapture8

Parts

object
: **Object** expression that resolves to a **DirectX8** object.

guid
: A GUID string that identifies the sound capture device. The value of this parameter must be one of the GUIDs returned by **DirectX8.getDSCaptureEnum**, **vbNullString** for the default device, or one of the following values:

  DSDEVID_DEFAULTCAPTURE
  : System-wide default audio capture device.

  DSDEVID_DEFAULTVOICECAPTURE
  : Default voice capture device.

[#] IDH_DirectX8.DirectPlayVoiceSetupCreate_vbintro
[#] IDH_DirectX8.DirectSoundCaptureCreate_vbintro

Return Values

Returns a **DirectSoundCapture8** object.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following Microsoft® DirectSound® error codes:

- DSERR_INVALIDPARAM
- DSERR_NOAGGREGATION
    - DSERR_OUTOFMEMORY

Remarks

This method could fail on some sound cards that do not support full duplex. Calling **DirectX8.DirectSoundCreate** may set the capture device to an unsupported capture frequency, which will cause **DirectX8.DirectSoundCaptureCreate** to fail.

DirectX8.DirectSoundCreate

#

Creates a **DirectSound8** object.

object.**DirectSoundCreate(_**
  guid **As String _**
) As DirectSound8

Parts

object
  Object expression that resolves to a **DirectX8** object.

guid
  A GUID string that identifies the sound capture device. The value of this parameter must be one of the GUIDs returned by **DirectSoundEnum8.GetGuid**, **vbNullString** for the default device, or one of the following values:

  DSDEVID_DEFAULTPLAYBACK
    System-wide default audio playback device.

  DSDEVID_DEFAULTVOICEPLAYBACK
    Default voice playback device.

Return Values

Returns a **DirectSound8** object.

# IDH_DirectX8.DirectSoundCreate_vbintro

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following Microsoft® DirectSound® error codes:

- DSERR_ALLOCATED
- DSERR_INVALIDPARAM
- DSERR_NOAGGREGATION
- DSERR_NODRIVER
  - DSERR_OUTOFMEMORY

DirectX8.DirectXFileCreate

#

Creates a **DirectXFile** object.

object.**DirectXFileCreate() As DirectXFile**

Parts

object

Object expression that resolves to a **DirectX8** object.

Return Values

Returns a **DirectXFile** object.

Error Codes

If the method fails, an error is raised and **Err.Number** is set.

DirectX8.GetDSCaptureEnum

#

Creates a **DirectSoundEnum8** object.

object.**GetDSCaptureEnum() As DirectSoundEnum8**

Parts

object

Object expression that resolves to a **DirectX8** object.

# IDH_DirectX8.DirectXFileCreate_vbintro

# IDH_DirectX8.GetDSCaptureEnum_vbintro

Return Values

Returns a **DirectSoundEnum8** object.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_OUTOFMEMORY.

Remarks

The **DirectSoundEnum8** object returned by this method is used to enumerate the **DirectSoundCapture8** objects installed on the system.

See Also

DirectX8.GetDSEnum

DirectX8.GetDSEnum

#

Creates a **DirectSoundEnum8** object.

object.**GetDSEnum() As DirectSoundEnum8**

Parts

object
    Object expression that resolves to a **DirectX8** object.

Return Values

Returns a **DirectSoundEnum8** object.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_OUTOFMEMORY.

Remarks

The **DirectSoundEnum8** object returned by this method is used to enumerate the Microsoft® DirectSound® drivers installed on the system.

See Also

DirectX8.GetDSCaptureEnum

# IDH_DirectX8.GetDSEnum_vbintro

DirectX8.SetEvent

#

Sets the state of an event object to signaled.

object.**SetEvent(_**
    eventid **As Long)**

Parts

object
: **Object** expression that resolves to a **DirectX8** object.

eventid
: **Long** value containing the event handle. This handle must have been created by **DirectX8.CreateEvent**.

Error Codes

If the method fails, an error is raised and **Err.Number** is set.

Remarks

This method is used primarily for testing purposes. Microsoft® DirectX® automatically sets all created event objects to be signaled when an appropriate event occurs.

DirectXEvent8

#

The **DirectXEvent8** class creates event objects used in Microsoft® DirectSound®, DirectMusic®, DirectPlay, and DirectInput®. The purpose of this class is to provide a callback method for handling notification events. **DirectXEvent8** objects are created by implementing the class in a form.

The **DirectXEvent8** class has one method, **DirectXEvent8.DXCallback**.

DirectXEvent8.DXCallback

#

A callback routine for event handles.

object.**DXCallback(_**
    eventid **As Long)**

# IDH_DirectX8.SetEvent_vbintro
# IDH_DirectXEvent8_vbintro
# IDH_DirectXEvent8.DXCallback_vbintro

Parts

object

Object expression that resolves to a **DirectXEvent8** object.

eventid

Long value containing the event handle. This handle must have been created by **DirectX8.CreateEvent**.

Error Codes

If the method fails, an error is raised and **Err.Number** is set.

Remarks

When you implement **DirectXEvent8** in a form, you must also supply an implementation of the **DirectXEvent.DXCallback** method. The declaration and end of the method are automatically placed in the Code window when you select **DirectXEvent8** in the **Object** box and **DXCallback** in the **Procedures/Events** box. The method takes the following form:

```
Private Sub DirectXEvent8_DXCallBack(ByVal eventid as Long)
 ' Add your own event-handling code
 .
 .
 .
End Sub
```

The event handle passed to the method must have been created by **DirectX8.CreateEvent**.

See Also

DirectX8.SetEvent

Further Information

You can find further explanations of the graphics and multimedia concepts and terms discussed throughout the Microsoft® DirectX® documentation, as well as information on Microsoft Windows® programming in general, in the following sources:

- Bargen, Bradley and Peter Donnelly, *Inside DirectX*, Microsoft Press®, 1998.
- Begault, Durand R., *3-D Sound for Virtual Reality and Multimedia*, Academic Press, 1994.

- Blinn, James, *Jim Blinn's Corner: A Trip Down the Graphics Pipeline*, Morgan Kaufmann, 1996.
- Dodge, Charles and Thomas A. Jerse, *Computer Music: Synthesis, Composition, and Performance*, Schirmer Books, 1997 (2nd edition).
- Foley, James D., *Computer Graphics: Principles and Practice*, Addison-Wesley, 1991 (2nd edition).
- Hearn, Donald and M. Pauline Baker, *Computer Graphics*, Prentice-Hall, 1986.
- Kientzle, Tim, *A Programmer's Guide to Sound*, Addison-Wesley Developers Press, 1998.
- Kovach, Peter J., *Inside Direct3D,* Microsoft Press, 2000.
- Petzold, Charles, *Programming Windows 98*, Microsoft Press, 1998 (5th edition).
- Thompson, Nigel, *3D Graphics Programming for Windows 95*, Microsoft Press, 1996.
- Watt, Alan H., and Mark Watt, *Advanced Animation and Rendering Techniques*, Addison-Wesley, 1992.

Additional sources for the concepts and terms associated with COM can be found in the following sources:

- Brockschmidt, Kraig, *Inside OLE 2*, Microsoft Press, 1995 (2nd edition).
- Rogerson, Dale E., *Inside COM,* Microsoft Press, 1997.