

# DirectInput

This section provides information about the DirectInput® component of the Microsoft® DirectX® application programming interface (API). The information is divided into the following topics:

- About DirectInput
- Why Use DirectInput?
- DirectInput Architecture
- DirectInput Essentials
- DirectInput Tutorials
- DirectInput Reference
- DirectInput Tools and Samples

## About DirectInput

DirectInput is an API for input devices including the mouse, keyboard, joystick, and other game controllers, as well as for force-feedback (input/output) devices.

## Why Use DirectInput?

Aside from providing new services for devices not supported by the Microsoft® Win32® API, DirectInput gives faster access to input data by communicating directly with the hardware drivers, rather than relying on Microsoft® Windows® messages.

DirectInput enables an application to retrieve data from input devices even when the application is in the background. It also provides full support for any type of input device, as well as for force feedback.

The extended services and improved performance of DirectInput make it a valuable tool for games, simulations, and other real-time interactive applications running under Windows.

## DirectInput Architecture

This section covers the basic structure of DirectInput and how it works with both the Windows operating system and input hardware. For practical information on how to implement the elements of DirectInput introduced here, see DirectInput Essentials.

- Architectural Overview of DirectInput

- Integration with Windows
- Human Interface Device

## Architectural Overview of DirectInput

---

### [C++]

The basic architecture of a DirectInput implementation consists of the DirectInput object, which supports the **IDirectInput7** COM interface, and an object for each input device that provides data. Each device in turn has device object instances or more simply device objects, which are individual controls or switches such as keys, buttons, or axes.

### Note

The word object is used to describe both a code object and one of the individual controls on an input device. In this documentation, device object means an input device control, rather than a code object that instantiates the **IDirectInputDevice7** interface. Code objects representing whole devices are referred to as DirectInputDevice objects.

Each DirectInputDevice object represents one input device, such as a mouse, keyboard, or joystick. A piece of hardware that is really a combination of different types of input devices, such as a keyboard with a touchpad, can be represented by two or more DirectInputDevice objects. A force-feedback device is represented by a single joystick object that handles both input and output.

DirectInputDevice objects instantiate the **IDirectInputDevice7** interface.

The application ascertains the number and type of device objects available by using the **IDirectInputDevice7::EnumObjects** method. Individual device objects are not encapsulated as code objects, but are described in **DIDEVICEOBJECTINSTANCE** structures.

Force-feedback effects are represented by the **IDirectInputEffect** interface. Methods of this interface are used to create, modify, start, and stop effects.

All DirectInput interfaces are available in ANSI and Unicode versions. If "UNICODE" is defined during compilation, the Unicode versions are used.

---

### [Visual Basic]

The basic architecture of a DirectInput implementation consists of a single **DirectInput** object and a **DirectInputDevice** object for each input device that is being used by the application. The **DirectInputDevice** object is used for retrieving the input data.

Any available device, regardless of whether it is being used by DirectInput, can also be represented by a **DirectInputDeviceInstance** object, which can be used to retrieve miscellaneous information about that device.

Each **DirectInputDevice** object represents one input device, such as a mouse, keyboard, or joystick. (In this documentation, the term joystick includes all game controllers other than a mouse or keyboard.) A piece of hardware that is really a combination of different types of input devices, such as a keyboard with a touchpad, can be represented by two or more **DirectInputDevice** objects. A force-feedback device is represented by a single joystick object that handles both input and output.

Each device in turn has device objects, which are individual controls or switches, such as keys, buttons, or axes. Each device object is represented by an instance of the **DirectInputDeviceObjectInstance** class, whose methods can be used to retrieve information about the device object. (The input data, however, is always retrieved by **DirectInputDevice**.)

#### Note

The word object is used to describe both a code object and one of the individual controls on an input device. In this documentation, device object means an input device control, rather than a **DirectInputDevice** object.

Devices and device objects can be enumerated, and the resulting collections are represented by **DirectInputEnumDevices** and **DirectInputEnumDeviceObjects** objects.

---

## Integration with Windows

Because DirectInput works directly with the device drivers, it either suppresses or ignores mouse and keyboard messages. When using the mouse in exclusive mode, DirectInput suppresses mouse messages, and therefore Windows is unable to show the standard cursor.

DirectInput also ignores mouse and keyboard settings made by the user in Control Panel.

For the keyboard, character repeat settings are not used by DirectInput. When using buffered data, DirectInput interprets each press and release as a single event, with no repetition. When using immediate data, DirectInput is concerned only with the present physical state of the keys, not with keyboard events as interpreted by Windows.

For the mouse, DirectInput ignores Control Panel settings such as acceleration and swapped buttons. Again, DirectInput works directly with the mouse driver, bypassing the subsystem of Windows that interprets mouse data for windowed applications.

#### Note

Settings in the driver itself are recognized by DirectInput. For example, if the user has a three-button mouse and uses the driver-utility software to make the middle button a double-click shortcut, DirectInput reports a click of the middle button as two clicks of the primary button.

For a joystick or other game controller, DirectInput does use the calibrations set by the user in Control Panel.

## Human Interface Device

Human Interface Device (HID) is a class under the universal serial bus (USB) standard. DirectInput provides full support for devices that conform with HID.

Getting data from HID devices is substantially the same as from traditional devices. In addition, HID devices can accept output: for example, you can turn the keyboard LEDs on and off.

By querying a HID device, DirectInput can determine its *usage page* and *usage*. These are predefined codes that give information about the type and subtype of the device.

HID controls are grouped in *collections*. Collections can be nested.

For more information about HID, see <http://www.usb.org>.

## DirectInput Essentials

This section is a practical guide to the concepts and components of DirectInput and provides enough information for you to get started in implementing the DirectInput system in your application. For a broader overview, see DirectInput Architecture.

The following topics are discussed:

- Creating DirectInput
- DirectInput Device Enumeration
- DirectInput Devices
- DirectInput Device Data
- Force Feedback
- Designing for Previous Versions of DirectInput

## Creating DirectInput

---

[C++]

The first step in any DirectInput application is obtaining the **IDirectInput7** interface. You can do this most easily by calling the **DirectInputCreateEx** function.

You should create a single DirectInput object and not release it until the application terminates.

---

[\[Visual Basic\]](#)

The first step in any DirectInput application is creating the **DirectInput** object. Do this by using the **DirectX7.DirectInputCreate** method.

You should create a single **DirectInput** object and not destroy it until the application terminates.

---

## DirectInput Device Enumeration

DirectInput can query the system for all available input devices, determine whether they are connected, and return information about them. This process is called enumeration.

---

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

If your application is using only the standard keyboard or mouse, or both, you do not need to enumerate the available input devices. As explained under Creating the DirectInput Device, you can use predefined global variables when calling the **IDirectInput7::CreateDeviceEx** method.

For all other input devices, and for systems with multiple keyboards or mouse devices, call **IDirectInput7::EnumDevices** to obtain at least the instance globally unique identifiers (GUIDs) so that DirectInputDevice objects can be created. You might also want to enumerate to give the user a choice of available devices.

The following is a sample implementation of the **IDirectInput7::EnumDevices** method:

```
/* lpdi is a valid IDirectInput7 interface pointer. */

GUID      KeyboardGUID = GUID_SysKeyboard;

lpdi->EnumDevices(DIDEVTYPE_KEYBOARD,
                  DIEnumDevicesCallback,
                  &KeyboardGUID,
                  DIEDFL_ATTACHEDONLY);
```

The first parameter determines what types of devices are to be enumerated. It is **NULL** if you want to enumerate all devices, regardless of type; otherwise, it is one of the **DIDEVTYPE\_\*** values described in **DIDeviceInstance**.

The second parameter is a pointer to a callback function to be called once for each device enumerated. This function can be called by any name; the documentation uses the placeholder name **DIEnumDevicesCallback**.

The third parameter to the **EnumDevices** method is any 32-bit value that you want to pass into the callback function. In this example, it is a pointer to a variable of type **GUID**, passed in so that the callback can assign a keyboard instance GUID.

The fourth parameter is a flag to request enumeration of either all devices or only those that are attached (DIEDFL\_ALLDEVICES or DIEDFL\_ATTACHEDONLY).

If your application is using more than one input device, the callback function is a good place to initialize each device as it is enumerated. (For an example, see Tutorial 3: Using the Joystick.) You obtain the instance GUID of the device from the callback function. You can also perform other processing here, such as looking for particular subtypes of devices or adding the device name to a list box.

The following code example checks for the presence of an enhanced keyboard and stops the enumeration as soon as it finds one. It assigns the instance GUID of the last keyboard found to the *KeyboardGUID* variable (passed in as *pvRef* by the previous example of a **EnumDevices** call), which can then be used in a call to

**IDirectInput7::CreateDeviceEx.**

```
BOOL      hasEnhanced;

BOOL CALLBACK DIEnumKbdCallback(LPCDIDEVICEINSTANCE lpddi,
                                LPVOID pvRef)
{
    *(GUID*) pvRef = lpddi->guidInstance;
    if (GET_DIDEVICE_SUBTYPE(lpddi->dwDevType) ==
        DIDEVTYPEKEYBOARD_PCENH)
    {
        hasEnhanced = TRUE;
        return DIENUM_STOP;
    }
    return DIENUM_CONTINUE;
} // End of callback
```

The first parameter points to a structure containing information about the device. This structure is created for you by **DirectInput**.

The second parameter points to data passed in from **EnumDevices**. In this case, it is a pointer to the variable *KeyboardGUID*. This variable was assigned a default value earlier, but it is given a new value each time that a device is enumerated. It is not important what instance GUID you use for a single keyboard, but the code example illustrates a technique for retrieving an instance GUID from the callback.

The return value in this case indicates that enumeration is to stop if the sought-for device has been found, or otherwise that it is to continue. Enumeration automatically stops as soon as all devices have been enumerated.

---

[\[Visual Basic\]](#)

If your application is using only the standard keyboard or mouse, or both, you do not need to enumerate the available input devices. As explained under **Creating a DirectInput Device**, you can use predefined GUID aliases when calling the **DirectInput.CreateDevice** method.

For all other input devices, and for systems with multiple keyboards or mouse devices, call **DirectInput.GetDIEnumDevices** to build a collection of available devices. This method returns a **DirectInputEnumDevices** object representing the collection. Each device in the collection can be retrieved as a **DirectInputDeviceInstance** object by using the **DirectInputEnumDevices.GetItem** method.

At the very least, you must retrieve the unique identifier for a nonstandard device by calling **DirectInputDeviceInstance.GetGuidInstance** before you can create a **DirectInputDevice** object for that device. You might also want to enumerate devices to look for particular types and subtypes (by using **DirectInputDeviceInstance.GetDevType**) or to populate a list box that allows the user to select a game controller.

You might even want to search for a device with particular capabilities. To do this, you must create a **DirectInputDevice** object for each candidate to examine it further by using the **DirectInputDevice.GetCapabilities** method.

---

## DirectInput Devices

This section contains information about the code objects that represent devices such as a mouse, keyboards, and joysticks. The following topics are covered:

- Device Setup
- Creating a DirectInput Device
- Device Capabilities
- Cooperative Levels
- Device Object Enumeration
- Device Data Formats
- Device Properties
- Acquiring Devices
- Recognizing Device Changes

For information on how to retrieve and interpret data from devices, see DirectInput Device Data.

### Device Setup

Your application must create an object for each device from which it expects input. It must also prepare each device for use, which requires, at the very least, setting the data format and acquiring the device. You might also want to carry out other preparatory tasks, such as getting information about the devices and changing their properties.

The following tasks are part of the setup process. Certain steps are always required; others are only necessary if you need further information about devices or need to change default values.

1. Create the **DirectInput** device (required). See **Creating a DirectInput Device**.
2. Get the device capabilities (optional).
3. Enumerate the keys, buttons, and axes on the device (optional). See **Device Object Enumeration**.
4. Set the cooperative level (highly recommended).
5. Set the data format (required).
6. Set the device properties (you must at least set the buffer size if you intend to get buffered data).
7. When ready to read data, acquire the device (required). See **Acquiring Devices**.

## Creating a DirectInput Device

To get input data from a device, you first have to create an object to represent that device.

---

### [C++]

The **IDirectInput7::CreateDeviceEx** method is used to obtain a pointer to the **IDirectInputDevice7** interface. Methods of this interface are then used to manipulate the device and obtain data.

The code following example, where *lpdi* is a pointer to the **IDirectInput7** interface, creates a keyboard device:

```
LPDIRECTINPUTDEVICE7 lpdiKeyboard;  
lpdi->CreateDeviceEx(GUID_SysKeyboard, IID_IDirectInputDevice7,  
    (void**)&lpdiKeyboard, NULL);
```

The first parameter in **IDirectInput7::CreateDeviceEx** is an instance GUID that identifies the instance of the device for which the interface is to be created.

**DirectInput** has two predefined GUIDs, *GUID\_SysMouse* and *GUID\_SysKeyboard*, which represent the system mouse and keyboard. You can pass these identifiers into the **CreateDeviceEx** method. The global variable *GUID\_Joystick* should not be used as a parameter for **CreateDeviceEx** because it is a product GUID, not an instance GUID.

### Note

If the computer has more than one mouse, input from all of them is combined to form the system device. The same is true for multiple keyboards.

**DirectInput** provides four other predefined GUIDs primarily for testing. These are *GUID\_SysKeyboardEm*, *GUID\_SysKeyboardEm2*, *GUID\_SysMouseEm*, and *GUID\_SysMouseEm2*. Passing one of these GUIDs to **CreateDeviceEx** grants access to the system keyboard or mouse through an emulation layer, at



either level 1 or level 2. These GUIDs always represent the system mouse or keyboard. They are aliases for GUID\_SysKeyboard and GUID\_SysMouse, so they are not enumerated by **IDirectInput7::EnumDevices** unless the DIEDFL\_INCLUDEALIASES flag is passed.

For devices other than the system mouse or keyboard, use the instance GUID for the device returned by **IDirectInput7::EnumDevices**. The instance GUID for a device is always the same. You can allow the user to select a device from a list of those enumerated, then save the GUID to a configuration file and use it again in future sessions.

---

#### [Visual Basic]

The **DirectInput.CreateDevice** method is used to obtain a **DirectInputDevice** object. Methods of this interface are then used to manipulate the device and obtain data.

The following code example, where *di* is the **DirectInput** object, creates a keyboard device:

```
Dim diDev As DirectInputDevice
Set diDev = di.CreateDevice("GUID_SysKeyboard")
```

The parameter is an alias for a GUID that identifies the instance of the device for which the interface is to be created. DirectInput provides two predefined GUIDs, GUID\_SysMouse and GUID\_SysKeyboard, which represent the system mouse and keyboard, and you can pass either of these to the **CreateDevice** method.

#### Note

If the computer has more than one mouse, input from all of them is combined to form the system device. The same is true for multiple keyboards.

For devices other than the system mouse or keyboard, use the instance GUID for the device obtained from **DirectInputDeviceInstance.GetGuidInstance**. The instance GUID for a device is always the same. You can allow the user to select a device from a list of those enumerated, then save the GUID to a configuration file and use it again in future sessions.

In the following code example, it is presumed that the application has enumerated devices and found a suitable one, *diDevInstance*, which is to be created as a **DirectInputDevice**:

```
Dim guid As String
guid = diDevInstance.GetGuidInstance
Set diDev = di.CreateDevice(guid)
```

For more information on obtaining the **DirectInputDeviceInstance** object, see DirectInput Device Enumeration.

---

## Device Capabilities

Before you begin asking for input from a device, you need to find out something about its capabilities. Does the joystick have a point-of-view hat? Is the mouse currently attached to the user's computer?

---

### [C++]

Such questions are answered with a call to the **IDirectInputDevice7::GetCapabilities** method, which returns the data in a **DIDEVCAPS** structure. As with other such structures in DirectX, you must initialize the **dwSize** member before passing this structure to the method.

### Note

To optimize speed or memory usage, you can use the smaller **DIDEVCAPS\_DX3** structure instead.

The following code example checks whether the mouse is attached and whether it has a third axis (presumably a wheel):

```
// LPDIRECTINPUTDEVICE7 lpdiMouse; // initialized previously

DIDEVCAPS DIMouseCaps;
HRESULT hr;
BOOLEAN WheelAvailable;

DIMouseCaps.dwSize = sizeof(DIDEVCAPS);
hr = lpdiMouse->GetCapabilities(&DIMouseCaps);
WheelAvailable = ((DIMouseCaps.dwFlags & DIDC_ATTACHED)
    && (DIMouseCaps.dwAxes > 2));
```

Another way to check for a button or axis is to call **IDirectInputDevice7::GetObjectInfo** for that object. If the call returns **DIERR\_OBJECTNOTFOUND**, the object is not present. The following code example determines whether there is a z-axis, even if it is not the third axis:

```
DIDeviceObjectInstance didoi;

didoi.dwSize = sizeof(DIDeviceObjectInstance);
hr = lpdiMouse->GetObjectInfo(&didoi, DIMOFS_Z, DIPH_BYOFFSET);
WheelAvailable = SUCCEEDED(hr);
```

---

### [Visual Basic]

Such questions are answered with a call to the **DirectInputDevice.GetCapabilities** method, which returns the data in a **DIDEVCAPS** type.

The following code example checks whether the mouse is attached and whether it has a third axis (presumably a wheel):

```
/* diMouse is a valid DirectInputDevice object. */
```

```
Dim WheelAvailable As Boolean
```

```
Dim dicaps as DIDEVCAPS
```

```
Call diDev.GetCapabilities(dicaps)
```

```
WheelAvailable = ((dicaps.IFlags And DIDC_ATTACHED) _  
    And (dicaps.IAxes > 2))
```

Another way to check for a certain button or axis is to call **DirectInputDevice.GetObjectInfo** for that object. If the call raises error **DIERR\_NOTFOUND**, the object is not present. The following code example determines whether there is a z-axis, even if it is not the third axis:

```
Dim didoi As DirectInputDeviceObjectInstance
```

```
On Error GoTo NOTFOUND
```

```
Set didoi = diDev.GetObjectInfo(DIMOFS_Z, DIPH_BYOFFSET)
```

```
On Error GoTo 0
```

```
.  
.   
. 
```

```
NOTFOUND:
```

```
If Err.Number = DIERR_NOTFOUND Then MsgBox "No z-axis found."
```

```
End If
```

---

## Cooperative Levels

---

### [C++]

The cooperative level of a device determines how the input is shared with other applications and with the Windows system. You set it by using the **IDirectInputDevice7::SetCooperativeLevel** method, as in the following code example:

```
/* hwnd is the top-level window handle. */
```

```
lpdiDevice->SetCooperativeLevel(hwnd,  
    DISCL_NONEXCLUSIVE | DISCL_FOREGROUND)
```

The parameters are the handle to the top-level window associated with the device (generally the application window) and one or more flags.

---

### [Visual Basic]

The cooperative level of a device determines how the input is shared with other applications and with the Windows system. You set it by using the **DirectInputDevice.SetCooperativeLevel** method, as in the following code example:

```
diDevice.SetCooperativeLevel(hWnd,  
    DISCL_NONEXCLUSIVE Or DISCL_FOREGROUND)
```

The parameters are the handle to the top-level window associated with the device (generally the application window) and one or more flags. The **hWnd** property of a form does not become valid until the form is shown. If you are initializing the DirectInput device in the **Load** method of the application's main form, you must call **Show** before attempting to set the cooperative level.

---

### Note

Although DirectInput provides a default setting, you should still explicitly set the cooperative level because this is the only way to give DirectInput the window handle. Without this handle, DirectInput cannot react to situations that involve window messages, such as joystick recalibration.

The valid flag combinations are shown in the following table:

Flags	Notes
DISCL_NONEXCLUSIVE DISCL_BACKGROUND	The default setting
DISCL_NONEXCLUSIVE DISCL_FOREGROUND	
DISCL_EXCLUSIVE DISCL_FOREGROUND	Not valid for keyboard or mouse
DISCL_EXCLUSIVE DISCL_BACKGROUND	

---

### [C++]

For the keyboard, you can also include DISCL\_NOWINKEY in combination with DISCL\_NONEXCLUSIVE. This flag disables the Windows key so that users cannot inadvertently break out of the application. In exclusive mode, the Windows key is always disabled.

---

The cooperative level has two main components: whether the device is being used in the foreground or the background, and whether it is being used exclusively or nonexclusively.

## Foreground vs. Background

A foreground cooperative level means that the input device is available only when the application is in the foreground or, in other words, has the input focus. If the application moves to the background, the device is automatically unacquired, or made unavailable.

A background cooperative level really means foreground and background. A device with a background cooperative level can be acquired and used by an application at any time.

You will usually want to have foreground access only, since most applications are not interested in input that takes place when another program is in the foreground.

While developing an application, it is useful to employ conditional compilation so that the background cooperative level is always set for debugging. This prevents your application from losing access to the device every time that it moves to the background as you switch to the debugging environment.

## Exclusive vs. Nonexclusive

The fact that your application is using a device at the exclusive level does not mean that other applications cannot get data from the device. However, it does mean that no other application can also acquire the device exclusively.

Take the example of a music player that accepts input from a hand-held remote-control device, even when the application is running in the background. If you run a similar application that plays movies in response to signals from the same remote control, what happens when the user presses **Play**? Both programs start playing, which is probably not what the user wants. To prevent this from happening, each application should have the DISCL\_EXCLUSIVE flag set so that only one of them can be running at a time.

To use force-feedback effects, an application must have exclusive access to the device.

Windows itself requires exclusive access to the mouse because mouse events such as a click on an inactive window could force an application to unacquire the device, with potentially harmful results, such as a loss of data from the input buffer. Therefore, when an application has exclusive access to the mouse, Windows is not allowed any access at all. No mouse messages are generated. A further side effect is that the cursor disappears.

When an application has exclusive access to the keyboard, DirectInput suppresses all keyboard messages except CTRL+ALT+DEL and, on Windows 95 and Windows 98, ALT+TAB.

## Device Object Enumeration

It might be necessary for your application to determine what buttons or axes are available on a given device. To do this, you enumerate the device objects in much the same way that you enumerate devices.

### [C++]

To some extent, **IDirectInputDevice7::EnumObjects** overlaps the functionality of **IDirectInputDevice7::GetCapabilities**. Either method can be used to determine how many buttons or axes are available. However, **EnumObjects** is intended for cataloging all the available objects, rather than checking for a particular one. The DirectInput QuickTest application provided with the DirectX SDK, for example, uses **EnumObjects** to populate the list on the **Objects** tabbed page for the selected device.

Like **IDirectInput7::EnumDevices**, the **EnumObjects** method has a callback function that allows you to do other processing on each object—for example, adding it to a list or creating a corresponding element on a user interface.

The following code example extracts the name of each object so that it can be added to a string list or array. This standard callback is documented under the placeholder name **DIEnumDeviceObjectsCallback**, but you can give it any name that you like. Remember, this function is called once for each object enumerated.

```
char szName[MAX_PATH];

BOOL CALLBACK DIEnumDeviceObjectsCallback(
    LPCDIDEVICEOBJECTINSTANCE lpddoi,
    LPVOID pvRef)
{
    lstrcpy(szName, lpddoi->tszName);
    // Now, add szName to a list or array.
    .
    .
    .
    return DIENUM_CONTINUE;
}
```

The first parameter points to a structure containing information about the object. This structure is created for you by DirectInput.

The second parameter is an application-defined pointer to data, equivalent to the second parameter to **EnumObjects**. In the example, this parameter is not used.

The return value in this case indicates that enumeration is to continue as long as there are objects to be enumerated.

The following code example calls the **EnumObjects** method, which puts the callback function to work.

```
lpdiMouse->EnumObjects(DIEnumDeviceObjectsCallback,
```

```
NULL, DIDFT_ALL);
```

The first parameter is the address of the callback function.

The second parameter can be a pointer to any data that you want to use or modify in the callback. The example does not use this parameter and, therefore, passes NULL.

The third parameter is a flag to indicate which type or types of objects are to be included in the enumeration. In the example, all objects are to be enumerated. To restrict the enumeration, you can use one or more of the other DIDFT\_\* flags listed in **IDirectInput7::EnumDevices**.

#### Note

Some of the DIDFT\_\* flags are combinations of others; for example, DIDFT\_AXIS is equivalent to DIDFT\_ABSAXIS | DIDFT\_RELAXIS.

---

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

You enumerate device objects by calling

**DirectInputDevice.GetDeviceObjectsEnum**, which returns an instance of the **DirectInputEnumDeviceObjects** class representing the collection of available device objects that match the requested parameters.

The following code example enumerates axes on a device:

```
' diDev is a DirectInputDevice object.
```

```
Dim diEnumObjects As DirectInputEnumDeviceObjects  
Set diEnumObjects = diDev.GetDeviceObjectsEnum(DIDFT_AXIS)
```

The parameter is a flag to indicate which type or types of objects are to be included in the enumeration.

#### Note

Some of the **CONST\_DIDFTFLAGS** flags are combinations of others; for example, DIDFT\_AXIS is equivalent to DIDFT\_ABSAXIS **OR** DIDFT\_RELAXIS.

To obtain information about a particular device object, call the methods of a **DirectInputDeviceObjectInstance** object obtained by calling **DirectInputEnumDeviceObjects.GetItem**. Information available for a device object includes its name, its type, and its offset in the data structure for the device.

The following code example lists the names of the axes enumerated in the previous example:

```
Dim diDevEnumObjects As DirectInputEnumDeviceObjects  
Set diDevEnumObjects = diDev.GetDeviceObjectsEnum(DIDFT_AXIS)
```

```
Dim diDevObjInstance As DirectInputDeviceObjectInstance
Dim i As Integer
For i = 1 To diEnumObjects.GetCount
    Set diDevObjInstance = diEnumObjects.GetItem(i)
    Call List1.AddItem(diDevObjInstance.GetName)
Next i
```

---

## Device Data Formats

Setting the data format for a device is an essential step before you can acquire and begin using the device. This is true even if you do not intend to retrieve immediate (state) data from the device. DirectInput uses the data format in many methods to identify particular device objects.

---

### [C++]

The **IDirectInputDevice7::SetDataFormat** method tells DirectInput what device objects will be used and how the data will be arranged.

The examples in the reference for the **DIDATAFORMAT** and **DIOBJECTDATAFORMAT** structures show how to set up custom data formats for nonstandard devices. Fortunately, this step is not necessary for the joystick, keyboard, and mouse. DirectInput provides five global variables, *c\_dfDIJoystick*, *c\_dfDIJoystick2*, *c\_dfDIKeyboard*, *c\_dfDIMouse*, and *c\_dfDIMouse2*, which can be passed in to **SetDataFormat** to create a standard data format for these devices.

In the following code example, *lpdiMouse* is an initialized pointer to the mouse DirectInputDevice object:

```
lpdiMouse->SetDataFormat(&c_dfDIMouse);
```

### Note

You cannot change the **dwFlags** member in the predefined **DIDATAFORMAT** global variables (for example, to change the property of an axis), because they are **const** variables. To change properties, use the **IDirectInputDevice7::SetProperty** method after setting the data format, but before acquiring the device.

---

### [Visual Basic]

The **DirectInputDevice.SetCommonDataFormat** and **DirectInputDevice.SetDataFormat** methods tell DirectInput what device objects will be used and how the data will be arranged.

For standard devices—the mouse, keyboard, and any game controller whose input data can be described in a **DIJOYSTATE** or **DIJOYSTATE2** type—you can set the



data format by calling the **SetCommonDataFormat** method, passing in a constant from the **CONST\_DICOMMONDATAFORMATS** enumeration. The common data formats are adequate for most applications.

For specialized devices, you must pass a description of the data format to the **SetDataFormat** method. The following code example sets the data format for a device with two axes, both of which require a **Long** for their data, and no buttons:

```
Dim dx As New DirectX7
Dim di As DirectInput
Dim did As DirectInputDevice
Dim fD As DIDATAFORMAT
Dim fDA(1) As DIOBJECTDATAFORMAT

Private Sub Form_Load()
    Set di = dx.DirectInputCreate()

    Set did = di.CreateDevice("GUID_SysMouse")

    fDA(0).IFlags = DIDOI_POLLED
    fDA(0).IOfs = 0
    fDA(0).IType = DIDFT_RELAXIS
    fDA(0).strGuid = "GUID_XAxis"

    fDA(1).IFlags = DIDOI_POLLED
    fDA(1).IOfs = 4
    fDA(1).IType = DIDFT_RELAXIS
    fDA(1).strGuid = "GUID_YAxis"

    fD.dataSize = 8
    fD.IFlags = DIDF_RELAXIS
    fD.IObjSize = 4
    fD.numObjs = 2

    did.SetDataFormat fD, fDA()

End Sub
```

---

## Device Properties

Properties of DirectInput devices include the size of the data buffer, the range and granularity of values returned from an axis, whether axis data is relative or absolute, and the dead zone and saturation values for a joystick axis, which affect the relationship between the physical position of the stick and the reported data. Specialized devices can have other properties, as well.

With one exception—the gain property of a force-feedback device—properties can be changed only when the device is in an unacquired state.

---

#### [C++]

Before calling the **IDirectInputDevice7::SetProperty** or the **IDirectInputDevice7::GetProperty** method, set up a property structure, which consists of a **DIPROPHEADER** structure and one or more elements for data. There are potentially a great variety of properties for input devices, and **SetProperty** must be able to work with all sorts of structures defining those properties. The purpose of the **DIPROPHEADER** structure is to define the size of the property structure and how the data is to be interpreted.

DirectInput includes the following predefined property structures:

- **DIPROPDWORD** defines a structure containing a **DIPROPHEADER** and a **DWORD** data member for properties that require a single value, such as a buffer size.
- **DIPROPRange** is for range properties, which require two values (maximum and minimum). It consists of a **DIPROPHEADER** and two **LONG** data members.
- **DIPROPGUIDANDPATH** is a specialized property structure allowing applications to perform operations on an HID that are not supported by DirectInput. The structure consists of a **DIPROPHEADER**, a **GUID**, and a Unicode string for the path.
- **DIPROPSTRING** is for Unicode string properties. The structure comprises a **DIPROPHEADER** and a Unicode string.

For **SetProperty**, the data members of the property structure are the values that you want to set. For **GetProperty**, the current value is returned in these members.

Before the call to **GetProperty** or **SetProperty**, the **DIPROPHEADER** structure must be initialized with the following:

- The size of the property structure.
- The size of the **DIPROPHEADER** structure itself.
- An object identifier.
- A **dwHow** member indicating the way that the object identifier should be interpreted.

When getting or setting properties for a whole device, the object identifier **dwObj** is 0, and the **dwHow** member is **DIPH\_DEVICE**. If you want to get or set properties for a device object (for example, a particular axis), the combination of **dwObj** and **dwHow** values identifies the object. For details, see **DIPROPHEADER** structure.

After setting up the property structure, pass the address of its header into **GetProperty** or **SetProperty**, along with an identifier for the property that you want to obtain or change.

The following values are used to identify the property passed to **SetProperty** and **GetProperty**. For more information, see **IDirectInputDevice7::GetProperty**.

- **DIPROP\_BUFFERSIZE**. See also Buffered and Immediate Data.
- **DIPROP\_AXISMODE**. See also Relative and Absolute Axis Coordinates.
- **DIPROP\_CALIBRATIONMODE**
- **DIPROP\_GRANULARITY**
- **DIPROP\_FFGAIN**
- **DIPROP\_FFLOAD**
- **DIPROP\_AUTOCENTER**
- **DIPROP\_RANGE**
- **DIPROP\_DEADZONE**
- **DIPROP\_SATURATION**

For more information about the last three properties, see also Interpreting Joystick Axis Data.

The following code example sets the buffer size for a device to hold 10 data items:

```
DIPROPDWORD dipdw;  
HRESULT hres;  
dipdw.diph.dwSize = sizeof(DIPROPDWORD);  
dipdw.diph.dwHeaderSize = sizeof(DIPROPHEADER);  
dipdw.diph.dwObj = 0;  
dipdw.diph.dwHow = DIPH_DEVICE;  
dipdw.dwData = 10;  
hres = lpdiDevice->SetProperty(DIPROP_BUFFERSIZE, &dipdw.diph);
```

---

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

The **DirectInputDevice.SetProperty** or **DirectInputDevice.GetProperty** methods take two parameters: a GUID alias in string form that identifies the property being set, and data of type **Any**. The data is passed in one of the following types:

- **DIPROPLONG** is for properties that require a single value, such as a buffer size. The property data consists of a single **Long**.
- **DIPROP RANGE** is for range properties, which require two values (maximum and minimum). The property data consists of two **Long** data members.

For **SetProperty**, the data members of the property types are the values that you want to set. For **GetProperty**, the current value is returned in these members.

In addition to the actual property data, both these types contain three other members: **IHow**, **IObj**, and **ISize**.

The values in **IHow** and **IObj** work together, with **IHow** signifying the system that is used to identify the device object whose property is being set or retrieved, and **IObj** identifying the device object.

If **IHow** is **DIPH\_BYID**, the device object is described by a unique numerical identifier in **IObj**. This ID can be extracted from the value returned by **DirectInputDeviceObjectInstance.GetType** after device objects have been enumerated.

For most applications, it is simpler to identify the device object by its offset within the data structure established by **DirectInputDevice.SetCommonDataFormat** or **DirectInputDevice.SetDataFormat**. In this case, **IHow** is **DIPH\_BYOFFSET**, and **IObj** is the offset, in bytes. For the keyboard, mouse, and any game controller whose data can be returned in a **DIJOYSTATE** type, the device object can be identified by a predefined constant. See **CONST\_DIKEYFLAGS**, **CONST\_DIMOUSEOFS**, and **CONST\_DIJOYSTICKOFS**.

The **IHow** member can also contain **DIPH\_DEVICE**, which means that the property belongs to the entire device, rather than a single device object. Buffer size is an example of such a property. When **IHow** is **DIPH\_DEVICE**, **IObj** is 0.

Finally, the **ISize** member of the property type must be initialized to the size of the type. This step is necessary because **GetProperty** or **SetProperty** do not know what type is being passed.

The following strings are used to identify the property passed to **SetProperty** and **GetProperty**. For more information, see **DirectInputDevice.GetProperty**.

- **DIPROP\_AXISMODE**. See also Relative and Absolute Axis Coordinates.
- **DIPROP\_BUFFERSIZE**. See also Buffered and Immediate Data.
- **DIPROP\_DEADZONE**
- **DIPROP\_GRANULARITY**
- **DIPROP\_RANGE**
- **DIPROP\_SATURATION**

For more information about dead zone, range, and saturation, see also Interpreting Joystick Axis Data.

The following code example sets the buffer size for a device:

' diDev is a DirectInputDevice whose data format has been set.

```
Dim diProp As DIPROPLONG
diProp.IHow = DIPH_DEVICE
diProp.IObj = 0
diProp.IData = 10
diProp.ISize = Len(diProp)
Call diDev.SetProperty("DIPROP_BUFFERSIZE", diProp)
```

---

## Acquiring Devices

Acquiring a DirectInput device means giving your application access to it. As long as a device is acquired, DirectInput is making its data available to your application. If the device is not acquired, you can manipulate its characteristics, but not obtain any data.

Acquisition is not permanent. Your application can acquire and unacquire a device many times.

In certain cases, depending on the cooperative level, a device can be unacquired automatically whenever the application moves to the background. The mouse is automatically unacquired when the user clicks on a menu, because at this point Windows takes over the device.

You must unacquire a device before changing its properties. The only exception is that you can change the gain for a force-feedback device while it is in an acquired state.

The acquisition mechanism is needed for two reasons:

First, DirectInput must be able to tell the application when the flow of data from the device has been interrupted by the system. For instance, if the user has switched to another application with ALT+TAB and used the input device in that application, your application needs to know that the input no longer belongs to it and that the state of the buffers might have changed. Consider an application with the DISCL\_FOREGROUND cooperative level. The user presses the SHIFT key, and while continuing to press it, switches to another application. Then, the user releases the key and switches back to the first application. As far as the first application is concerned, the SHIFT key is still down. The acquisition mechanism, by telling the application that input was lost, allows it to recover from these conditions.

Second, because your application can alter the properties of the device, without safeguards DirectInput would have to check the properties each time that you wanted to retrieve data. This would be very inefficient. Even worse, this could cause a hardware interrupt accessing a data buffer when the buffer size is being changed. Therefore, DirectInput requires your application to unacquire the device before changing properties. When you reacquire it, DirectInput checks the properties and decides on the optimal way of transferring data from the device to the application. This is done only once, so the data retrieval methods can be very fast.

---

### [C++]

Since the most common cause of losing a device is that your application moves to the background, you might want to reacquire devices whenever your application is activated. Be careful, however, about relying on a WM\_ACTIVATE handler at startup time. The first WM\_ACTIVATE message will probably arrive when your window is being initialized, before DirectInput has been set up. To ensure that the device is acquired at startup, call **IDirectInputDevice7::Acquire** as soon as the device has been initialized.

Even acquiring the device on activation of your program window might not cover all cases in which a device is unacquired, especially for devices other than the standard mouse or keyboard. Because your application might unacquire a device unexpectedly, you need a mechanism for checking the acquisition state before attempting to get data from the device. The Scrawl sample application does this in the `Scrawl_OnMouseInput` function, in which a `DIERR_INPUTLOST` error triggers a message to reacquire the mouse. (See also Tutorial 2: Using the Mouse.)

---

#### [Visual Basic]

Because your application might unacquire a device unexpectedly, especially if you have set the exclusive cooperative level, ensure that the application tracks the state of acquisition. One technique is to check for the `DIERR_INPUTLOST` error after attempting to retrieve data. If this error is raised, you know the device has been unacquired. If your application is getting input in response to event notification, an event is signaled when acquisition is lost.

See the `ScrawlB` sample for more information about how to manage device acquisition in exclusive mode.

---

Attempting to reacquire a device that is already acquired does no harm. Redundant calls to **Acquire** are ignored, and the device can always be unacquired with a single call to **Unacquire**.

---

#### [C++]

Windows does not have access to the mouse when your application is using it in exclusive mode. If you want to let Windows have the mouse, you must release it. There is an example in the `Scrawl` sample that responds to a click of the right button by unacquiring the mouse, putting the Windows cursor in the same spot as its own, popping up a context menu, and letting Windows handle the input until a menu choice is made.

---

#### [Visual Basic]

Windows does not have access to the mouse when your application is using it in exclusive mode. If you want to let Windows have the mouse, you must release it. There is an example in the `ScrawlB` sample that responds to a click of the right button by unacquiring the mouse, putting the Windows cursor in the same spot as its own, popping up a context menu, and letting Windows handle the input until a menu choice is made.

---

---

## Recognizing Device Changes

---

### [Visual Basic]

This topic pertains only to applications written in C++.

---

### [C++]

Because universal serial bus (USB) devices can be added to and removed from the system without rebooting, you might want your application to be able to respond to a new configuration of input devices. For example, you might allow a new player to join a game in progress by plugging in another controller.

To receive a message when a device is changed, you must first register for notification, as in the following code example:

```
PVOID hNotifyDevNode;

void RegisterForDevChange(HWND hDlg, PVOID *hNotifyDevNode)
{
    DEV_BROADCAST_DEVICEINTERFACE *pFilterData =
        (DEV_BROADCAST_DEVICEINTERFACE*)
        _alloca(sizeof(DEV_BROADCAST_DEVICEINTERFACE));
    ASSERT (pFilterData);

    ZeroMemory(pFilterData, sizeof(DEV_BROADCAST_DEVICEINTERFACE));

    pFilterData->dbcc_size = sizeof(DEV_BROADCAST_DEVICEINTERFACE);
    pFilterData->dbcc_devicetype = DBT_DEVTYP_DEVICEINTERFACE;
    pFilterData->dbcc_classguid = GUID_CLASS_INPUT;

    *hNotifyDevNode = RegisterDeviceNotification(hDlg, pFilterData,
        DEVICE_NOTIFY_WINDOW_HANDLE);
}
```

Then, in your main window procedure, check for messages announcing that a device has been attached, is about to be removed, or has been removed, as follows:

```
MyWindowProc(HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM lParam)
{
    switch (nMsg)
    {
        case WM_DEVICECHANGE:
        {
            switch (wParam)
            {
                case DBT_DEVICEARRIVAL:
                    // Handle device arrival
            }
        }
    }
}
```

```
        break;

        case DBT_DEVICEQUERYREMOVE:
            // Handle device removal request
            break;

        case DBT_DEVICEREMOVECOMPLETE:
            // Handle device removal
            break;
    }
}
.
.
.
}
```

In response to a `DBT_DEVICEARRIVAL` event, obtain the instance GUID of the device by using **IDirectInput7::FindDevice**, and pass this value to **IDirectInput7::CreateDeviceEx**.

For more information, see Device Management in the the Platform SDK.

---

## DirectInput Device Data

This section covers the basic concepts of getting data from DirectInput devices.

- Buffered and Immediate Data
- Time Stamps and Sequence Numbers
- Polling and Events
- Relative and Absolute Axis Coordinates

Specific details about mouse, keyboard, and joystick input data, and about output data, are given in the following sections:

- Mouse Data
- Keyboard Data
- Joystick Data
- Output Data



## Buffered and Immediate Data

DirectInput supplies two types of data: buffered and immediate. Buffered data is a record of events that are stored until an application retrieves them. Immediate data is a snapshot of the current state of a device.

You might use immediate data in an application that is concerned only with the current state of a device—for example, a flight combat simulation that responds to the current position of the joystick and the state of one or more buttons. Buffered data might be the better choice where events are more important than states—for example, in an application that responds to movement of the mouse and button clicks. You can also use both types of data, as you might for example if you wanted to get immediate data for joystick axes but buffered data for the buttons.

The DirectInput QuickTest application supplied with the DirectX SDK lets you see both immediate and buffered data from a device. After you create the device in the application window, set its properties on the Mode tabbed page. On the Data tabbed page, you then see immediate data on the left and buffered data on the right.

---

### [C++]

An application retrieves immediate data by calling the **IDirectInputDevice7::GetDeviceState** method. As the name implies, this method returns the current state of the device: for example, whether each button is up or down. The method provides no data about what has happened with the device since the last call, apart from implicit information that you can derive by comparing the current state with the last one. If the user has pressed and released a button between two calls to **GetDeviceState**, your application does not know anything about it. On the other hand, if the user is holding a button down, **GetDeviceState** continues reporting button down until the user releases it.

This way of reporting the device state is different from the way that Windows reports events with one-time messages such as `WM_LBUTTONDOWN`; it is more like the results from the Win32 **GetKeyboardState** function. If you are polling a device with **GetDeviceState**, you are responsible for determining what constitutes a button click, a double-click, a single keystroke, and so on, and for ensuring that your application does not keep responding to a button-down or key-down state when it is not appropriate to do so.

With buffered data, events are stored until you are ready to deal with them. Every time a button or key is pressed or an axis is moved, information about the event is placed in a **DIDEVICEOBJECTDATA** structure in the buffer. If the buffer overflows, new data is lost. Your application reads the buffer with a call to **IDirectInputDevice7::GetDeviceData**. You can read any number of items at a time.

Reading an item normally deletes it from the buffer, but you also have the choice of peeking without deleting.

To get buffered data, you must first set the buffer size with the **IDirectInputDevice7::SetProperty** method. (See the example in Device Properties.) Set the buffer size before acquiring the device for the first time. For reasons of

efficiency, the default size of the buffer is 0, and you cannot obtain buffered data unless you change this value. The size of the buffer is measured in items of data for that type of device, not in bytes or **WORDS**.

Check the value of the *pdwInOut* parameter after a call to the **GetDeviceData** method. The number of items retrieved from the buffer is returned in this variable.

#### Note

For devices that do not generate interrupts, such as analog joysticks, **DirectInput** does not obtain any data until you call the **IDirectInputDevice7::Poll** method. For more information, see Polling and Events.

For examples of retrieving buffered data, see **IDirectInputDevice7::GetDeviceData**.

---

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

An application retrieves immediate data by calling one of the following methods:

- **DirectInputDevice.GetDeviceStateKeyboard**. For devices that retrieve data in a **DIKEYBOARDSTATE** type. (For this and the following three methods, the data format must have been set by using **DirectInputDevice.SetCommonDataFormat**.)
- **DirectInputDevice.GetDeviceStateMouse**. For devices that retrieve data in a **DIMOUSESTATE** type.
- **DirectInputDevice.GetDeviceStateJoystick**. For devices that retrieve data in a **DIJOYSTATE** type.
- **DirectInputDevice.GetDeviceStateJoystick2**. For devices that retrieve data in a **DIJOYSTATE2** type.
- **DirectInputDevice.GetDeviceState**. For devices that use custom data formats, as set by using **DirectInputDevice.SetDataFormat**.

As the names imply, each of these methods returns the current state of the device: for example, whether each button is up or down. The method provides no data about what has happened with the device since the last call, apart from implicit information that you can derive by comparing the current state with the last one. If the user has pressed and released a button between two calls to the method, your application does not know anything about it. On the other hand, if the user is holding a button down, the method continues reporting button down until the user releases it.

This way of reporting the device state is different from the way Visual Basic reports events with one-time events like **Click** and **Keydown**. If you are polling a device with one of the **GetDeviceState** methods, you are responsible for determining what constitutes a button click, a double-click, a single keystroke, and so on, and for ensuring that your application does not keep responding to a button-down or key-down state when it is not appropriate to do so.

With buffered data, events are stored until you are ready to deal with them. Every time a button or key is pressed or an axis is moved, information about the event is

placed in a **DIDeviceObjectData** type in the buffer. Your application reads the buffer with a call to **DirectInputDevice.GetDeviceData**. You can read any number of items at a time.

Reading an item normally deletes it from the buffer, but you also have the choice of retrieving without deleting by setting the **DIGDD\_PEEK** flag.

To get buffered data, you must first set the buffer size by using the **DirectInputDevice.SetProperty** method. (See the example under Device Properties.) Set the buffer size before acquiring the device for the first time. For reasons of efficiency, the default size of the buffer is 0, and you cannot obtain buffered data unless you change this value. The size of the buffer is measured in items of data for that type of device, not in bytes or **WORDS**.

The return value of **GetDeviceData** tells you the number of items retrieved from the buffer. If the buffer has overflowed, no data is returned, and **GetDeviceData** raises an error, which the application should trap.

#### Note

For devices that do not generate interrupts, such as analog joysticks, **DirectInput** does not obtain any data until you call the **DirectInputDevice.Poll** method. For more information, see Polling and Events.

---

See also:

- Time Stamps and Sequence Numbers
- Mouse Data
- Keyboard Data
- Joystick Data

## Time Stamps and Sequence Numbers

---

### [C++]

When **DirectInput** input data is buffered (see Buffered and Immediate Data), each **DIDeviceObjectData** structure contains not only information about the type of event and the device object associated with it, but also a time stamp and a sequence number.

The **dwTimeStamp** member contains the system time, in milliseconds, at which the event took place. This is equivalent to the value that would have been returned by the Win32 **GetTickCount** function, but at a higher resolution.

The **dwSequence** member contains a sequence number assigned by **DirectInput**. The **DirectInput** system keeps a single sequence counter, which is incremented by each nonsimultaneous buffered event from any device. Use this number to compare events from different devices and see which came first. The **DISEQUENCE\_COMPARE** macro takes wrap around into account.

### [Visual Basic]

When DirectInput input data is buffered (see Buffered and Immediate Data), each **DIDEVICEOBJECTDATA** type contains not only information about the type of event and the device object associated with it, but also a time stamp and a sequence number.

The **ITimeStamp** member contains the system time, in milliseconds, at which the event took place. This is equivalent to the value that would have been returned by the Win32 **GetTickCount** function, but at a higher resolution.

The **ISequence** member contains a sequence number assigned by DirectInput. The DirectInput system keeps a single sequence counter, which is incremented by each nonsimultaneous buffered event from any device. Use this number to compare events from different devices and see which came first.

---

Simultaneous events are assigned the same sequence number. If a mouse or joystick is moved diagonally, for example, the changes in the x-axis and the y-axis have the same sequence number.

### Note

Events are always placed in the buffer in chronological order, so you do not need to check the sequence numbers to sort the events from a single device.

## Polling and Events

There are two ways to find out whether input data is available: polling and event notification.

Polling a device means regularly getting the current state of the device objects or checking the contents of the event buffer. Polling is typically used by real-time games that are never idle, but are constantly updating and rendering the game world.

---

### [C++]

In a C++ application, polling would typically be done within the message loop.

Event notification is suitable for applications like the Scrawl sample that wait for input before doing anything. To use event notification, set up a thread-synchronization object with the Win32 **CreateEvent** function, and then associate this event with the device by passing its handle to the

**IDirectInputDevice7::SetEventNotification** method. The event is then signaled by DirectInput whenever the state of the device changes. Your application can receive notification of the event with a Win32 function such as **WaitForSingleObject**, and then respond by checking the input buffer to find out what the event was. For code examples, see the Scrawl sample and **IDirectInputDevice7::SetEventNotification**.

Some joysticks and other game devices, or particular objects on them, do not generate hardware interrupts and do not return any data or signal any events until you call the **IDirectInputDevice7::Poll** method. (This behind-the-scenes polling is not to be confused with the kind of application polling just discussed. **Poll** does not retrieve any data, but merely makes data available.)

To find out whether it is necessary to call **Poll** each time that you want to retrieve data, first set the data format for the device, then call the **IDirectInputDevice7::GetCapabilities** method, and check for the **DIDC\_POLLEDDATAFORMAT** flag in the **DIDEVCAPS** structure.

Do not confuse the **DIDC\_POLLEDDATAFORMAT** flag with the **DIDC\_POLLEDDEVICE** flag. The latter is set if any object on the device requires polling. You can then find out whether this is the case for a particular object by calling the **IDirectInputDevice7::GetObjectInfo** method and checking for the **DIDOI\_POLLED** flag in the **DIDeviceObjectInstance** structure.

The **DIDC\_POLLEDDEVICE** flag describes the worst case for the device, not the actual situation. For example, an HID mouse with software-controllable resolution might be marked as **DIDC\_POLLEDDEVICE** because reading the resolution information requires polling. Polling the device under these conditions is pointless if all that you want is the standard button and axis data.

Nevertheless, it does not hurt to call the **IDirectInputDevice7::Poll** method for any input device. If the call is unnecessary, it has no effect and is very fast.

---

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

In a Visual Basic application, polling would typically be done in the **Sub Main** procedure.

Event notification is suitable for applications such as the ScrawlB sample that wait for input before doing anything. To use event notification, implement **DirectXEvent** in the form or module in which you want to retrieve data. Then, create an event handle by using **DirectX7.CreateEvent**, and pass this handle to **DirectInputDevice.SetEventNotification**. Now, whenever an input event occurs on the device, the **DirectXEvent.DXCallback** method is called, and in your implementation of this method you can retrieve either the device state or buffered data as you would if you were doing so in **Sub Main**.

Some joysticks and other game devices, or particular objects on them, do not generate hardware interrupts and do not return any data or signal any events until you call the **DirectInputDevice.Poll** method. (This behind-the-scenes polling is not to be confused with the kind of application polling just discussed. **Poll** does not retrieve any data, but merely makes data available.)

To find out whether it is necessary to call **Poll** each time that you want to retrieve data, first set the data format for the device, then call the **DirectInputDevice.GetCapabilities** method, and check for the **DIDC\_POLLEDDATAFORMAT** flag in the **DIDEVCAPS** type.

Do not confuse the `DIDC_POLLEDDATAFORMAT` flag with the `DIDC_POLLEDDEVICE` flag. The latter is set if any object on the device requires polling. You can then find out whether this is the case for a particular object by calling the **IDirectInputDevice7::GetObjectInfo** method to get a **DirectInputDeviceObjectInstance** object, and then checking for the `DIDOI_POLLED` flag in the value returned by **DirectInputDeviceObjectInstance.GetFlags**.

The `DIDC_POLLEDDEVICE` flag describes the worst case for the device, not the actual situation. For example, an HID mouse with software-controllable resolution might be marked as `DIDC_POLLEDDEVICE` because reading the resolution information requires polling. Polling the device under these conditions is pointless if all that you want is the standard button and axis data.

Nevertheless, it does not hurt to call the **Poll** method for any input device. If the call is unnecessary, it has no effect and is very fast.

---

## Relative and Absolute Axis Coordinates

Axis coordinates can be returned as relative values—that is, the amount by which they have changed since the application last retrieved the device state or, in the case of buffered input, since the last item was put in the buffer.

Absolute axis coordinates are a running total of all the relative coordinates returned by the system since the device was acquired; in other words, they show the position of the axis in relation to a fixed point.

By default, mouse axes are reported as relative coordinates and joystick axes as absolute coordinates. You can change the coordinate system for a device by setting a property. For more information, see *Device Properties*.

## Mouse Data

---

[C++]

To set up the mouse device for data retrieval, first call the **IDirectInputDevice7::SetDataFormat** method with the `c_dfDIMouse` or `c_dfDIMouse2` global variable as the parameter value. Use `c_dfDIMouse2` if you want to support more than four mouse buttons.

For maximum performance in a full-screen application, set the cooperative level to `DISCL_EXCLUSIVE | DISCL_FOREGROUND`. The exclusive setting causes the Windows cursor to disappear. The `DISCL_FOREGROUND` setting causes the application to lose access to the mouse when you switch to a debugging window. Changing to `DISCL_BACKGROUND` allows you to debug the application more easily, at a cost in performance.

---

#### [Visual Basic]

To set up the mouse device for data retrieval, first call the **DirectInputDevice.SetCommonDataFormat** method with `DIFORMAT_MOUSE` as the parameter value.

For maximum performance in a full-screen application, set the cooperative level to `DISCL_EXCLUSIVE` **Or** `DISCL_FOREGROUND`. The exclusive setting causes the Windows cursor to disappear. The `DISCL_FOREGROUND` setting causes the application to lose access to the mouse when you switch to the Visual Basic development environment. Changing to `DISCL_BACKGROUND` allows you to debug the application more easily, at a cost in performance.

---

The following sections give more information about getting and interpreting immediate and buffered mouse data:

- Immediate Mouse Data
- Buffered Mouse Data
- Interpreting Mouse Axis Data
- Checking for Lost Mouse Input

### See also

Device Data Formats, Cooperative Levels

#### Immediate Mouse Data

---

#### [C++]

To retrieve the current state of the mouse, call **IDirectInputDevice7::GetDeviceState** with a pointer to a **DIMOUSESTATE** or a **DIMOUSESTATE2** structure, depending on the data format. The mouse state returned in the structure includes axis data and the state of each of the buttons.

The first three members of the structure hold the axis coordinates. (See Interpreting Mouse Axis Data.)

The **rgbButtons** member is an array of bytes, one for each of four or eight buttons. For a traditional mouse, the first element in the array is generally the left button, the second is the right button, and the third is the middle button. The high bit is set if the button is down, and clear if the button is up or not present.

---

#### [Visual Basic]

To retrieve the current state of the mouse, call **DirectInputDevice.GetDeviceStateMouse**, passing in a **DIMOUSESTATE** type. The mouse state returned in the type includes axis data and the state of each of the buttons.

The **x**, **y**, and **z** members of the **DIMOUSESTATE** type hold the axis coordinates. (See Interpreting Mouse Axis Data.) The **buttons** member is an array of bytes, one for each of four buttons. The first element in the array is generally the left button, the second is the right button, the third is the middle button, and the fourth is any other button. The high bit is set if the button is down, and clear if the button is up or not present.

---

## See also

Buffered and Immediate Data

### Buffered Mouse Data

To retrieve buffered data from the mouse, you must first set the buffer size (see Device Properties). The default size of the buffer is 0, so this step is essential.

---

#### [C++]

You must also declare an array of **DIDEVICEOBJECTDATA** structures. This array can have up to the same number of elements as the buffer size. You do not have to retrieve the entire contents of the buffer with a single call; if you want, you can have just one element in the array and retrieve events one at a time until the buffer is empty.

After acquiring the device, you can examine and flush events in the buffer at any time by using the **IDirectInputDevice7::GetDeviceData** method. (See Buffered and Immediate Data.) On return, each element in the **DIDEVICEOBJECTDATA** array represents a change in state for a single object on the mouse. For example, if the user presses button 0 and moves the mouse diagonally, the array passed to **GetDeviceData** (if it has at least three elements, and *pdwInOut* is at least 3) has three elements filled in—an element for button 0 being pressed, an element for the change in the x-axis, and an element for the change in the y-axis—and the value of *pdwInOut* is set to 3.

You can determine which object an element in the array refers to by checking the **dwOfs** member of the **DIDEVICEOBJECTDATA** structure against the values returned by the following macros:

- **DIMOFS\_BUTTON0** to **DIMOFS\_BUTTON3**
- **DIMOFS\_X**
- **DIMOFS\_Y**
- **DIMOFS\_Z**

Each of these values is derived from the offset of the data for the object in a **DIMOUSESTATE** or **DIMOUSESTATE2** structure. For example, **DIMOFS\_BUTTON0** is equivalent to the offset of **rgbButtons[0]** in the **DIMOUSESTATE** structure. With the macros you can use simple comparisons to determine which device object is associated with an item in the buffer. For example:



```
DIDeviceObjectData *lpdidod;
int n;
.
.
.
/* MouseBuffer is an array of DIDeviceObjectData structures
   that has been set by a call to GetDeviceData.
   n is incremented in a loop that examines all filled elements
   in the array. */
lpdidod = &MouseBuffer[n];
if ((int) lpdidod->dwOfs == DIMOFS_BUTTON0)
    && (lpdidod->dwData & 0x80)
{
    ; // Do something in response to left button press.
}
```

The data for the change of state of the device object is located in the **dwData** member of the **DIDeviceObjectData** structure. For axes, the coordinate value is returned in this member. For button objects, only the low byte of **dwData** is significant; the high bit of this byte is set if the button was pressed, and clear if the button was released. In other words, the button was pressed if (**dwData & 0x80**) is nonzero.

For more information on the other members of the **DIDeviceObjectData** structure, see Time Stamps and Sequence Numbers.

---

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

You must also declare an array of **DIDeviceObjectData** types. This array can have up to the same number of elements as the buffer size. You do not have to retrieve the entire contents of the buffer with a single call; if you want, you can have just one element in the array and retrieve events one at a time until the buffer is empty.

After acquiring the device, you can examine and flush the buffer at any time by using the **DirectInputDevice.GetDeviceData** method. (See Buffered and Immediate Data.) On return, each element in the **DIDeviceObjectData** array represents a change in state for a single object on the mouse. For instance, if the user presses button 0 and moves the mouse diagonally, the array passed to **GetDeviceData** (if it has at least three elements) has three elements filled in—an element for button 0 being pressed, an element for the change in the x-axis, and an element for the change in the y-axis—and the return value of the method is 3.

You can determine which object an element in the array refers to by checking the **IOfs** member of the **DIDeviceObjectData** type against the constants in the **CONST\_DIMOUSEOFS** enumeration. Each of these values is derived from the offset of the data for the object in a **DIMOUSESTATE** type. For example,

DIMOFS\_BUTTON0 is equivalent to the offset of **buttons(0)** in the **DIMOUSESTATE** type.

The data for the change of state of the device object is located in the **IData** member of the **DIDEVICEOBJECTDATA** type. For axes, the coordinate value is returned in this member. For button objects, only the low byte of **IData** is significant; the high bit of this byte is set if the button was pressed, and clear if the button was released. In other words, the button was pressed if (**IData & 0x80**) is nonzero.

For more information on the other members of the **DIDEVICEOBJECTDATA** type, see Time Stamps and Sequence Numbers.

The following code example retrieves the entire contents of the buffer (which contains *BufferSize* elements) and responds to various events:

```
' objDIDev is a DirectInputDevice object.
Dim diDeviceData(1 To BufferSize) As DIDEVICEOBJECTDATA
Dim NumEvents As Integer
Dim i As Integer

NumEvents = objDIDev.GetDeviceData(diDeviceData, 0)
For i = 1 To NumEvents
    Select Case diDeviceData(i).IData
        Case DIMOFS_X
            ' Respond to x-axis movement.

        Case DIMOFS_Y
            ' Respond to y-axis movement.

        Case DIMOFS_BUTTON0
            If diDeviceData(i).IData And &H80 Then
                ' Respond to left button pressed.
            Else
                ' Respond to left button released.
            End If
        End Select
    End Select
Next i
```

---

## Interpreting Mouse Axis Data

The data returned for the x-axis and y-axis of a mouse indicates the movement of the mouse itself, not the cursor. The units of measurement are based on the values returned by the mouse hardware and have nothing to do with pixels or any other form of screen measurement. Because DirectInput communicates directly with the mouse

driver, the values for mouse speed and acceleration set by the user in Control Panel do not affect this data.

Axis data returned from the mouse can be either relative or absolute. (See Relative and Absolute Axis Coordinates.) Because a mouse is a relative device—unlike a joystick, it does not have a home position—relative data is returned by default.

---

#### [C++]

The axis mode, which specifies whether relative or absolute data should be returned, is a property that can be changed before the device is acquired. (See Device Properties.) To set the axis mode to absolute, call

**IDirectInputDevice7::SetProperty** with the `DIPROP_AXISMODE` value in the *rguidProp* parameter and with `DIPROPAXISMODE_ABS` in the **dwData** member of the **DIPROPDWORD** structure.

---

#### [Visual Basic]

The axis mode, which specifies whether relative or absolute data should be returned, is a property that can be changed before the device is acquired. (See Device Properties.) To set the axis mode to absolute, call **DirectInputDevice.SetProperty** with "DIPROP\_AXISMODE" in the *guid* parameter and with `DIPROPAXISMODE_ABS` in the **IData** member of the **DIPROPLONG** type.

---

When the axis mode for the mouse is set to relative, the axis coordinate represents the number of units that the device has been moved along the axis since the last value was returned. A negative value indicates that the mouse was moved to the left for the x-axis, or away from the user for the y-axis, or that the z-axis (the wheel) was rotated toward the user. Positive values indicate movement in the opposite direction.

When the axis mode is set to absolute, the coordinates are simply a running total of all relative motions received by DirectInput. The axis coordinates are not initialized to any particular value when the device is acquired, so your application should treat absolute values as relative to an unknown origin. You can record the current absolute position whenever the device is acquired and save it as the virtual origin. This virtual origin can then be subtracted from subsequent absolute coordinates retrieved from the device to compute the relative distance that the mouse has moved from the point of acquisition.

The data returned for the axis coordinates is also affected by the granularity property of the device. For the x-axis and y-axis of the mouse, granularity is normally 1, meaning that the minimum change in value is 1. For the wheel axis, it can be larger.

## Checking for Lost Mouse Input

---

#### [C++]

Because Windows might force your application to unacquire the mouse when you have set the cooperative level to `DISCL_FOREGROUND` and the focus switches to

another application, or even to the menu in your own application, you should check for the `DIERR_INPUTLOST` return value from the **IDirectInputDevice7::GetDeviceData** or the **IDirectInputDevice7::GetDeviceState** method, and attempt to reacquire the mouse if necessary. (See Acquiring Devices.)

**Note**

You should not attempt to reacquire the mouse on getting a `DIERR_NOTACQUIRED` error. If you do, you could get caught in an infinite loop: acquisition would fail, you would get another `DIERR_NOTACQUIRED` error, and so on.

---

[\[Visual Basic\]](#)

Because Windows might force your application to unacquire the mouse when you have set the cooperative level to `DISCL_FOREGROUND` and the focus switches to another application, or even to the menu in your own application, you should check for the `DIERR_INPUTLOST` return value from the **DirectInputDevice.GetDeviceData** or the **DirectInputDevice.GetDeviceStateMouse** method, and attempt to reacquire the mouse if necessary. (See Acquiring Devices.)

**Note**

You should not attempt to reacquire the mouse on getting a `DIERR_NOTACQUIRED` error. If you do, you could get caught in an infinite loop: acquisition would fail, you would get another `DIERR_NOTACQUIRED` error, and so on.

---

## Keyboard Data

As far as DirectInput is concerned, the keyboard is not a text input device, but a game pad with many buttons. When your application requires text input, do not use DirectInput methods; it is far easier to retrieve the data from the normal Windows messages, in which you can take advantage of services such as character repeat and translation of physical keys to virtual keys. This is particularly important for languages other than English, which can require special translation of key presses.

---

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

To set up the keyboard device for data retrieval, you must first call the **IDirectInputDevice7::SetDataFormat** method with the `c_dfDIKeyboard` global variable as the parameter. (See Device Data Formats.)

---

[\[Visual Basic\]](#)

To set up the keyboard device for data retrieval, you must first call the **DirectInputDevice.SetCommonDataFormat** method with **DIFORMAT\_KEYBOARD** as the parameter.

---

The following sections give more information about obtaining and interpreting keyboard data:

- Immediate Keyboard Data
- Buffered Keyboard Data
- Interpreting Keyboard Data
- Checking for Lost Keyboard Input

## Immediate Keyboard Data

---

### [C++]

To retrieve the current state of the keyboard, call the **IDirectInputDevice7::GetDeviceState** method with a pointer to an array of 256 bytes to hold the returned data.

The **GetDeviceState** method behaves in the same way as the Win32 **GetKeyboardState** function, returning a snapshot of the current state of the keyboard. Each key is represented by a byte in the array of 256 bytes whose address was passed as the *lpvData* parameter. If the high bit of the byte is set, the key is down. The array is most conveniently indexed with the DirectInput Keyboard Device Constants. (See also Interpreting Keyboard Data.)

The following code example does something in response to the fact that the ESC key is down:

```
// LPDIRECTINPUTDEVICE lpdiKeyboard; // previously initialized
// and acquired

BYTE diKeys[256];
if (lpdiKeyboard->GetDeviceState(256, diKeys) == DI_OK)
{
    if (diKeys[DIK_ESCAPE] & 0x80) DoSomething();
}
```

---

### [Visual Basic]

To retrieve the current state of the keyboard, call the **DirectInputDevice.GetDeviceStateKeyboard** method, passing a **DIKEYBOARDSTATE** type.

The **GetDeviceState** method returns a snapshot of the current state of the keyboard. Each key is represented by an element in the array of 256 bytes that makes up the

**DIKEYBOARDSTATE** type. If the high bit of the byte is set, the key is down. The array is most conveniently indexed with the members of the **CONST\_DIKEYFLAGS** enumeration. (See also Interpreting Keyboard Data.)

The following code example determines whether the ESC key is currently being pressed:

```
' objDIDev is a DirectInputDevice object.  
Dim dev As DirectInputDevice  
Dim KeyState As DIKEYBOARDSTATE  
  
Call objDIDev.GetDeviceStateKeyboard(KeyState)  
If (KeyState.Key(DIK_ESCAPE) And &H80) Then  
    ' Key is down.  
End If
```

---

## Buffered Keyboard Data

To retrieve buffered data from the keyboard, you must first set the buffer size (see Device Properties). This step is essential because the default size of the buffer is 0.

---

### [C++]

You must also declare an array of **DIDeviceObjectData** structures. This array can have up to the same number of elements as the buffer size. You do not have to retrieve the entire contents of the buffer with a single call; if you want, you can have just one element in the array and retrieve events one at a time until the buffer is empty.

After acquiring the keyboard device, you can examine and flush the buffer at any time by using the **IDirectInputDevice7::GetDeviceData** method. (See Buffered and Immediate Data.)

Each element in the **DIDeviceObjectData** array represents a change in state for a single key; that is, a press or release. Because DirectInput gets the data directly from the keyboard, any settings for character repeat in Control Panel are ignored. This means that a keystroke is counted only once, no matter how long the key is held down.

You can determine which key an element in the array refers to by checking the **dwOfs** member of the **DIDeviceObjectData** structure against the DirectInput Keyboard Device Constants. (See also Interpreting Keyboard Data.)

The data for the change of state of the key is located in the **dwData** member of the **DIDeviceObjectData** structure. Only the low byte of **dwData** is significant; the high bit of this byte is set if the key was pressed and clear if it was released. In other words, the key was pressed if (**dwData & 0x80**) is nonzero.

---

#### [Visual Basic]

You must also declare an array of **DIDEVICEOBJECTDATA** types. This array can have up to the same number of elements as the buffer size. You do not have to retrieve the entire contents of the buffer with a single call; if you want, you can have just one element in the array and retrieve events one at a time until the buffer is empty.

After acquiring the keyboard device, you can examine and flush the buffer at any time by using the **DirectInputDevice.GetDeviceData** method. (See Buffered and Immediate Data.)

Each element in the **DIDEVICEOBJECTDATA** array represents a change in state for a single key; that is, a press or release. Because DirectInput gets the data directly from the keyboard, any settings for character repeat in Control Panel are ignored. This means that a keystroke is counted only once, no matter how long the key is held down.

You can determine which key an element in the array refers to by checking the **lof**s member of the **DIDEVICEOBJECTDATA** type against the constants in the **CONST\_DIKEYFLAGS** enumeration. (See also Interpreting Keyboard Data.)

The data for the change of state of the key is located in the **IData** member of the **DIDEVICEOBJECTDATA** type. Only the low byte of **IData** is significant; the high bit of this byte is set if the key was pressed and clear if it was released. In other words, the key was pressed if (**IData And &H80**) is nonzero.

---

## Interpreting Keyboard Data

---

#### [C++]

This section covers the identification of keys for which data is reported by the **IDirectInputDevice7::GetDeviceState** and **IDirectInputDevice7::GetDeviceData** methods. For more information on interpreting the data from **GetDeviceData**, see Time Stamps and Sequence Numbers.

---

#### [Visual Basic]

This section covers the identification of keys for which data is reported by the **DirectInputDevice.GetDeviceState** and **DirectInputDevice.GetDeviceData** methods. For more information on interpreting the data from **GetDeviceData**, see Time Stamps and Sequence Numbers.

---

In one important respect, DirectInput differs from other ways of reading the keyboard in Windows. Keyboard data refers not to virtual keys but to the actual physical keys—that is, the scan codes. **DIK\_ENTER**, for example, refers to the ENTER key on the main keyboard, but not to the one on the numerical keypad.

DirectInput defines a constant for each key on the enhanced keyboard, as well as the additional keys found on international keyboards. Because NEC keyboards support different scan codes than the PC-enhanced keyboards, DirectInput translates NEC key scan codes into PC-enhanced scan codes where possible.

Not all PC-enhanced keyboards have the Windows keys (DIK\_LWIN, DIK\_RWIN, and DIK\_APPS). There is no way to determine whether the keys are physically available.

Laptops and other small computers often do not implement a full set of keys. Instead, some keys (typically numeric keypad keys) are multiplexed with other keys, selected by an auxiliary mode key, which does not generate a separate scan code.

If the keyboard subtype indicates a PC XT or PC AT keyboard, the following keys are not available: DIK\_F11, DIK\_F12, and all the extended keys (DIK\_\* values greater than 0x7F). Furthermore, the PC XT keyboard lacks DIK\_SYSRQ.

Japanese keyboards, particularly the NEC PC-98 keyboards, contain a substantially different set of keys than U.S. keyboards. For more information, see DirectInput and Japanese Keyboards.

## Checking for Lost Keyboard Input

---

### [C++]

Because Windows might force your application to unacquire the keyboard when you have set the cooperative level to DISCL\_FOREGROUND and the focus switches to another application, you should check for the DIERR\_INPUTLOST return value from the **IDirectInputDevice7::GetDeviceData** or **IDirectInputDevice7::GetDeviceState** methods and attempt to reacquire the keyboard, if necessary. (See Acquiring Devices.)

### Note

You should not attempt to reacquire the keyboard on getting a DIERR\_NOTACQUIRED error. If you do, you could get caught in an infinite loop: acquisition would fail, you would get another DIERR\_NOTACQUIRED error, and so on.

---

### [Visual Basic]

Because Windows might force your application to unacquire the keyboard when you have set the cooperative level to DISCL\_FOREGROUND and the focus switches to another application, you should check for the DIERR\_INPUTLOST return value from the **DirectInputDevice.GetDeviceData** or the **DirectInputDevice.GetDeviceStateKeyboard** method and attempt to reacquire the keyboard, if necessary. (See Acquiring Devices.)

### Note



You should not attempt to reacquire the keyboard on getting a `DIERR_NOTACQUIRED` error. If you do, you could get caught in an infinite loop: acquisition would fail, you would get another `DIERR_NOTACQUIRED` error, and so on.

---

## Joystick Data

---

### [C++]

To set up the joystick device for data retrieval, first call the **IDirectInputDevice7::SetDataFormat** method with the `c_dfDIJoystick` or the `c_dfDIJoystick2` global variable as the parameter value. (See Device Data Formats.)

Because some device drivers do not notify `DirectInput` of changes in state until explicitly asked to do so, you should always call the **IDirectInputDevice7::Poll** method before attempting to retrieve data from the joystick. For more information, see Polling and Events.

---

### [Visual Basic]

To set up the joystick device for data retrieval, first call the **DirectInputDevice.SetCommonDataFormat** method with `DIFORMAT_JOYSTICK` or `DIFORMAT_JOYSTICK2` as the parameter value. (See Device Data Formats.)

Because some device drivers do not notify `DirectInput` of changes in state until explicitly asked to do so, you should always call the **DirectInputDevice.Poll** method before attempting to retrieve data from the joystick. For more information, see Polling and Events.

---

The following sections cover getting and interpreting data from a joystick or other similar input device, such as a game pad or steering wheel:

- Immediate Joystick Data
- Buffered Joystick Data
- Interpreting Joystick Axis Data
- Checking for Lost Joystick Input

## Immediate Joystick Data

---

### [C++]

To retrieve the current state of the joystick, call the **IDirectInputDevice7::GetDeviceState** method with a pointer to a **DIJOYSTATE** or a **DIJOYSTATE2** structure, depending on whether the data format was set with

*c\_dfDIJoystick* or *c\_dfDIJoystick2*. (See Device Data Formats.) The joystick state returned in the structure includes the coordinates of the axes, the state of the buttons, and the state of the point-of-view controllers.

The first seven members of the **DIJOYSTATE** structure hold the axis coordinates. The last of these seven, **rglSlider**, is an array of two values. (See Interpreting Joystick Axis Data.)

The **rgdwPOV** member contains the position of up to four point-of-view controllers in hundredths of a degree clockwise from north (or forward). The center position is reported as  $-1$ . For controllers that have only five positions, the position is one of the following values:

- $-1$
- $0$
- $90 * DI\_DEGREES$
- $180 * DI\_DEGREES$
- $270 * DI\_DEGREES$ .

Some drivers report a value of 65,535, instead of  $-1$ , for the neutral position. You should check for a centered POV indicator as follows:

```
BOOL POVCentered = (LOWORD(dwPOV) == 0xFFFF);
```

The **rgbButtons** member is an array of bytes, one for each of 32 or 128 buttons, depending on the data format. For each button, the high bit is set if the button is down, and clear if the button is up or not present.

The **DIJOYSTATE2** structure has additional members for information about the velocity, acceleration, force, and torque of the axes.

---

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

To retrieve the current state of the joystick, call the **DirectInputDevice.GetDeviceStateJoystick** or the **DirectInputDevice.GetDeviceStateJoystick2** method, depending on whether the data format was set with **DIFORMAT\_JOYSTICK** or **DIFORMAT\_JOYSTICK2**. (See Device Data Formats.) The joystick state returned in the *state* parameter includes the coordinates of the axes, the state of the buttons, and the state of the point-of-view controllers.

The **POV** member of the **DIJOYSTATE** or the **DIJOYSTATE2** type contains the position of up to four point-of-view controllers in hundredths of a degree clockwise from north (or forward). The center position is reported as  $-1$ . For controllers that have only five positions, the position is one of the following values:

- $-1$
- $0$
- $9000$

- 18000
- 27000

Some drivers report a value of 65,535, instead of –1, for the neutral position. You should check for a centered POV indicator as follows:

```
Dim POVCentered As Boolean
POVCentered = (dwPOV And &HFFFF) = &HFFFF;
```

The **buttons** member is an array of bytes, one for each of 32 or 128 buttons, depending on the data type. For each button, the high bit is set if the button is down, and clear if the button is up or not present.

The **DIJOYSTATE2** type has additional members for information about the velocity, acceleration, force, and torque of the axes.

For more information, see Interpreting Joystick Axis Data.

---

## See also

Buffered and Immediate Data

### Buffered Joystick Data

---

#### [C++]

To retrieve buffered data from the joystick, first set the buffer size (see Device Properties), and declare an array of **DIDEVICEOBJECTDATA** structures. This array can have up to the same number of elements as the buffer size. You do not have to retrieve the entire contents of the buffer with a single call; if you want, you can have just one element in the array and retrieve events one at a time until the buffer is empty.

After acquiring the device, you can examine and flush the buffer at any time with the **IDirectInputDevice7::GetDeviceData** method. (See Buffered and Immediate Data.)

Each element in the **DIDEVICEOBJECTDATA** array represents a change in state for a single object on the joystick. For instance, if the user presses button 0 and moves the stick diagonally, the array passed to **GetDeviceData** (if it has at least three elements, and *pdwInOut* is at least 3) has three elements filled in—an element for button 0 being pressed, an element for the change in the x-axis, and an element for the change in the y-axis—and the value of *pdwInOut* is set to 3.

You can determine which object an element in the array refers to by checking the **dwOfs** member of the **DIDEVICEOBJECTDATA** structure against the following values:

- DIJOFS\_X
- DIJOFS\_Y

- DIJOFS\_Z
- DIJOFS\_Rx
- DIJOFS\_Ry
- DIJOFS\_Rz
- DIJOFS\_BUTTON0 to DIJOFS\_BUTTON31 or DIJOFS\_BUTTON(*n*)
- DIJOFS\_POV(*n*)
- DIJOFS\_SLIDER(*n*)

Each of these values is equivalent to the offset of the data for the object in a **DIJOYSTATE** structure. For example, DIJOFS\_BUTTON0 is equivalent to the offset of **rgbButtons[0]** in the **DIJOYSTATE** structure. You can use simple comparisons to determine which device object is associated with an item in the buffer. For example:

```
DIDEVICEOBJECTDATA *lpdidod;
int                n;
.
.
.
/* JoyBuffer is an array of DIDEVICEOBJECTDATA structures
   that has been set by a call to GetDeviceData.
   n is incremented in a loop that examines all filled elements
   in the array. */
lpdidod = &JoyBuffer[n];
if ((int) lpdidod->dwOfs == DIJOFS_BUTTON0)
    && (lpdidod->dwData & 0x80))
{
    ; // Do something in response to press of primary button.
}
```

If the data format was set with *c\_dfDIJoystick2*, you can use the predefined offsets for all the device objects that exist in **DIJOYSTATE**, but you must supply your own offsets for device objects represented in the extra members of **DIJOYSTATE2**.

The data for the change of state of the device object is located in the **dwData** member of the **DIDEVICEOBJECTDATA** structure. For axes, the coordinate value is returned in this member. For button objects, only the low byte of **dwData** is significant; the high bit of this byte is set if the button is pressed, and clear if the button is released.

For the other members, see Time Stamps and Sequence Numbers.

---

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

To retrieve buffered data from the joystick, first set the buffer size (see Device Properties), and declare an array of **DIDEVICEOBJECTDATA** types. This array

can have up to the same number of elements as the buffer size. You do not have to retrieve the entire contents of the buffer with a single call; if you want, you can have just one element in the array and retrieve events one at a time until the buffer is empty.

After acquiring the device, you can examine and flush the buffer at any time with the **DirectInputDevice.GetDeviceData** method. (See Buffered and Immediate Data.)

Each element in the **DIDEVICEOBJECTDATA** array represents a change in state for a single object on the joystick. For instance, if the user presses button 0 and moves the stick diagonally, the array passed to **GetDeviceData** (if it has at least three elements) has three elements filled in—an element for button 0 being pressed, an element for the change in the x-axis, and an element for the change in the y-axis—and the return value of the method is 3.

You can determine which object an element in the array refers to by checking the **IOfs** member of the **DIDEVICEOBJECTDATA** type against the constants of the **CONST\_DIJOYSTICKOFS** enumeration. Each of these values is equivalent to the offset of the data for the object in a **DIJOYSTATE** type. For example, **DIJOFS\_BUTTON0** is equivalent to the offset of **buttons(0)** in the **DIJOYSTATE** type.

If the data format was set with **DIFORMAT\_JOYSTICK2**, you can use the offset constants for all the device objects that exist in **DIJOYSTATE**, but you must supply your own offsets for device objects represented in the extra members of **DIJOYSTATE2**. The internal organization of this type is the same as that of the equivalent C++ structure in the **Dinput.h** header file.

The data for the change of state of the device object is located in the **IData** member of the **DIDEVICEOBJECTDATA** type. For axes, the coordinate value is returned in this member. For button objects, only the low byte of **IData** is significant; the high bit of this byte is set if the button is pressed, and clear if the button is released.

For the other members, see Time Stamps and Sequence Numbers.

---

## Interpreting Joystick Axis Data

Axis values for the joystick are like those for the mouse: the value returned for the x-axis is greater as the stick moves to the right, and the value for the y-axis increases as the stick moves toward the user.

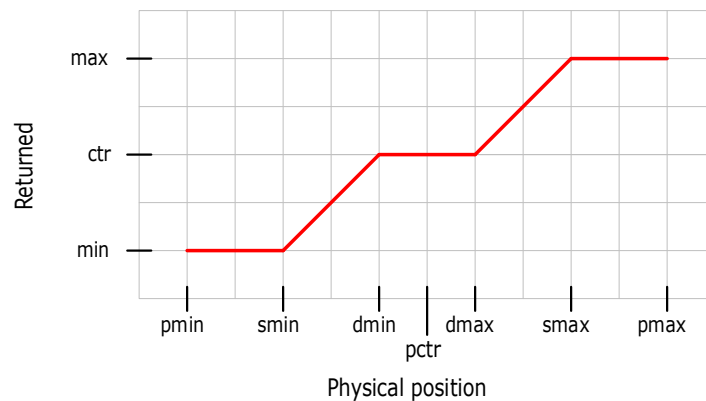
Data is in arbitrary units determined by the range property of the axis. For example, if the range for the stick's x-axis is from 0 through 10,000, a unit is one ten-thousandth of the stick's left-right travel, and the center position is 5,000. For some axes, the granularity property might be greater than 1, in which case values are rounded off. For example, if the granularity is 10, values are reported as 0, 10, 20, and so on.

Axis data is also affected by the dead zone, a region around the center position in which motion is ignored. The dead zone provides tolerance for a slight deviation from

the true center position for either or both axes of the stick. An axis value within the range of the dead zone is reported as true center.

The saturation property of an axis is a zone of tolerance at the minimum and maximum of the range. An axis value within this zone is reported as the minimum or maximum value. The purpose of the saturation property is to allow for slight differences between, for example, the minimum x-axis value reported at the top left and bottom left positions of the stick.

The following illustration shows the effect of the dead zone and the saturation zones. The vertical axis represents the returned axis values, where *min* and *max* are the lower and upper limits of the reported range and *ctr* is the reported center. The horizontal axis shows the physical position of the stick, where *pmin* and *pmax* are the extremes of the physical range, *pctr* is neutral position of the axis, *dmin* and *dmax* are the limits of the dead zone, and *smin* and *smax* are the boundaries of the lower and upper saturation zones. The lower saturation zone lies between *pmin* and *smin*, the upper saturation zone lies between *smax* and *pmax*, and the dead zone lies between *dmin* and *dmax*.




---

#### [C++]

For more information on joystick properties, see the following:

- **IDirectInputDevice7::GetProperty**
  - **IDirectInputDevice7::SetProperty**
  - **DIPROP\_RANGE**
- 

#### [Visual Basic]

For more information on joystick properties, see the following:

- **DirectInputDevice.GetProperty**
- **DirectInputDevice.SetProperty**
- **DIPROP\_RANGE**

Axis coordinates from the joystick can be either relative or absolute. (See Relative and Absolute Axis Coordinates.) Because a joystick is an absolute device—unlike a mouse, it cannot travel infinitely far along any axis—absolute data is returned by default. When the axis mode for the joystick is set to relative, the axis coordinate represents the number of units of movement along the axis since the last value was returned.

---

#### [C++]

The axis mode, which specifies whether relative or absolute data should be returned, is a property that can be changed before the device is acquired. (See Device Properties.) To set the axis mode to relative, call the **IDirectInputDevice7::SetProperty** method with the `DIPROP_AXISMODE` value in the *rguidProp* parameter and with `DIPROPAXISMODE_REL` in the **dwData** member of the **DIPROPDWORD** structure.

---

#### [Visual Basic]

The axis mode, which specifies whether relative or absolute data should be returned, is a property that can be changed before the device is acquired. (See Device Properties.) To set the axis mode to absolute, call **DirectInputDevice.SetProperty** with "DIPROP\_AXISMODE" in the *guid* parameter and with `DIPROPAXISMODE_ABS` in the **IData** member of the **DIPROPLONG** type.

---

## Checking for Lost Joystick Input

If you are using the joystick in foreground mode (see Cooperative Levels), you might lose the device when the focus shifts to another application.

---

#### [C++]

You should check for the `DIERR_INPUTLOST` return value from the **IDirectInputDevice7::GetDeviceData** or the **IDirectInputDevice7::GetDeviceState** method and attempt to reacquire the joystick, if necessary. (See Acquiring Devices.)

#### Note

You should not attempt to reacquire the joystick on getting a `DIERR_NOTACQUIRED` error. If you do, you could get caught in an infinite loop: acquisition would fail, you would get another `DIERR_NOTACQUIRED` error, and so on.

Because access to the joystick is not lost except when your application moves to the background—unlike the mouse and keyboard, the joystick is never used by the Windows system—an alternative method is to reacquire the device in response to a `WM_ACTIVATE` message.

#### [Visual Basic]

You should check for the DIERR\_INPUTLOST return value from the **DirectInputDevice.GetDeviceData** or the **DirectInputDevice.GetDeviceStateKeyboard** method and attempt to reacquire the keyboard, if necessary. (See Acquiring Devices.)

#### Note

You should not attempt to reacquire the keyboard on getting a DIERR\_NOTACQUIRED error. If you do, you could get caught in an infinite loop: acquisition would fail, you would get another DIERR\_NOTACQUIRED error, and so on.

---

## Output Data

---

#### [C++]

Human Interface Devices can accept output, as well as generating input. The **IDirectInputDevice7::SendDeviceData** method is used to send packets of data to such devices.

**SendDeviceData** can be viewed as **IDirectInputDevice7::GetDeviceData** in reverse. Like that method, it uses the **DIDEVICEOBJECTDATA** structure as the basic unit of data. In this case, however, the **dwOfs** member contains the instance ID of the device object associated with the data, rather than its offset in the data format for the device. (Because offset identifiers exist only for device objects that provide input in the selected data format, an object that accepts only output might not even have an offset.) The **dwData** member contains whatever data is appropriate for the object. The **dwTimeStamp** and **dwSequence** members are not used and must be set to 0.

To send data to the device, first set up an array of **DIDEVICEOBJECTDATA** structures, fill the required number of elements with data, and then pass its address and the number of elements used to **SendDeviceData**. Data for different device objects is combined into a single packet that is then sent to the device.

The form of the data packet is specific to the device, as is the treatment of unused fields in the packet. Some devices treat fields as optional, meaning that if no data is supplied, the state of the object remains unchanged. More commonly, all fields are significant, even when you do not specifically supply data for them. For example, if you send data to a single keyboard LED, it is assumed that the data for the other two LEDs is 0, and they are turned off. However, you can override this behavior by using the DISDD\_CONTINUE flag, in which case the data for the other two LEDs is the value you most recently sent them.

The following code example, when run repeatedly, causes the LEDs on the keyboard, as represented by the **IDirectInputDevice7** interface *pdev*, to flash in a recurring



pattern. The device object identifiers, *NumLockID*, *CapsLockID*, and *ScrollLockID*, have previously been obtained from the **dwType** member of the **DIDEVICEOBJECTINSTANCE** structure, either during enumeration of device objects or by calling **IDirectInputDevice7::GetObjectInfo**. It is assumed that the high bit of the data byte determines the state of the LED.

```
void FlashLEDs(void)
{
    static int    rgiBits[] = { 1, 2, 4, 2 };
    static int    iLooper = 0;
    DWORD        cdod = 3;           // Number of items
    DIDEVICEOBJECTDATA rgdod[3];
    HRESULT       hres;

    // Must clear dwTimeStamp and dwSequence
    ZeroMemory(rgdod, sizeof(rgdod));

    rgdod[0].dwOfs = NumLockID;
    rgdod[1].dwOfs = CapsLockID
    rgdod[2].dwOfs = ScrollLockID;

    rgdod[0].dwData = (rgiBits[iLooper] & 1) ? 0x80 : 0;
                        // 1,0,0,0,...
    rgdod[1].dwData = (rgiBits[iLooper] & 2) ? 0x80 : 0;
                        // 0,1,0,1,...
    rgdod[2].dwData = (rgiBits[iLooper] & 4) ? 0x80 : 0;
                        // 0,0,1,0,...

    iLooper = (iLooper + 1) % ARRAYSIZE(rgiBits); // Loops from 0 to 3

    hres = IDirectInputDevice7_SendDeviceData(pdev,
        sizeof(DIDEVICEOBJECTDATA),
        rgdod, &cdod, 0);
}
```

#### [Visual Basic]

Human Interface Devices can accept output, as well as generating input. The **DirectInputDevice.SendDeviceData** method is used to send packets of data to such devices.

**SendDeviceData** can be viewed as **DirectInputDevice.GetDeviceData** in reverse. Like that method, it uses the **DIDEVICEOBJECTDATA** type as the basic unit of data. In this case, however, the **IOfs** member contains the instance ID of the device object associated with the data, rather than its offset in the data format for the device. This ID can be extracted from the value returned by

**DirectInputDeviceObjectInstance.GetType** after device objects have been enumerated. (Because offset identifiers exist only for device objects that provide input in the selected data format, an object that accepts only output might not even have an offset.) The **IData** member contains whatever data is appropriate for the object. The **ITimeStamp** and **ISequence** members are not used and must be set to 0.

To send data to the device, first set up an array of **DIDEVICEOBJECTDATA** types, fill the required number of elements with data, and then pass its address and the number of elements used to **SendDeviceData**. Data for different device objects is combined into a single packet that is then sent to the device.

The form of the data packet is specific to the device, as is the treatment of unused fields in the packet. Some devices treat fields as optional, meaning that if no data is supplied, the state of the object remains unchanged. More commonly, all fields are significant, even when you do not specifically supply data for them. For example, if you send data to a single keyboard LED, it is assumed that the data for the other two LEDs is 0, and they are turned off. However, you can override this behavior by using the **DISDD\_CONTINUE** flag, in which case the data for the other two LEDs is the value you most recently sent them.

---

## Force Feedback

Force feedback is the generation of push or resistance in an input/output device, for example by motors mounted in the base of a joystick. DirectInput allows you to generate force-feedback effects for devices that have compatible drivers.

The following sections introduce the elements of force feedback:

- Basic Concepts of Force Feedback
- Effect Enumeration
- Loading Effects from a File
- Information About a Supported Effect
- Creating an Effect
- Effect Direction
- Envelopes and Offsets
- Effect Playback
- Downloading and Unloading Effects
- Changing an Effect
- Gain
- Force-Feedback State
- Effect Object Enumeration
- Effect Types

## Basic Concepts of Force Feedback

A particular instance of force feedback is called an *effect*, and the push or resistance is called the *force*. Most effects fall into one of the following categories:

- Constant force. A steady force in a single direction.
- Ramp force. A force that steadily increases or decreases in magnitude.
- Periodic effect. A force that pulsates according to a defined wave pattern.
- Condition. A force that occurs only in response to input by the user. Two examples are a friction effect that generates resistance to movement of the joystick, and a spring effect that tends to move the stick back to a certain position after it has been moved from that position.

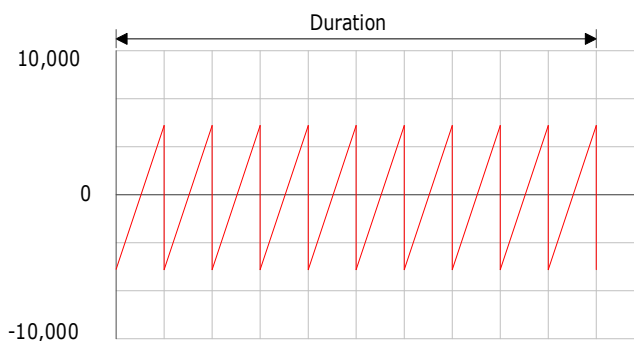
The strength of the force is called its *magnitude*. Magnitude is measured in units ranging from 0 (no force) through 10,000 (maximum force for the device, defined for C/C++ in Dinput.h as DI\_FFNOMINALMAX). A negative value indicates force in the opposite direction. Magnitudes are linear: a force of 10,000 is twice as great as one of 5,000.

Ramp forces have a beginning and ending magnitude. For a periodic effect, the basic magnitude is the force at the peak of the wave.

The *direction* of a force is the direction from which it comes; a positive force on a given axis pushes from the positive toward the negative.

Effects also have *duration*, measured in microseconds. Periodic effects have a *period*, or the duration of one cycle, also measured in microseconds. The *phase* of a periodic effect is the point along the wave at which playback begins.

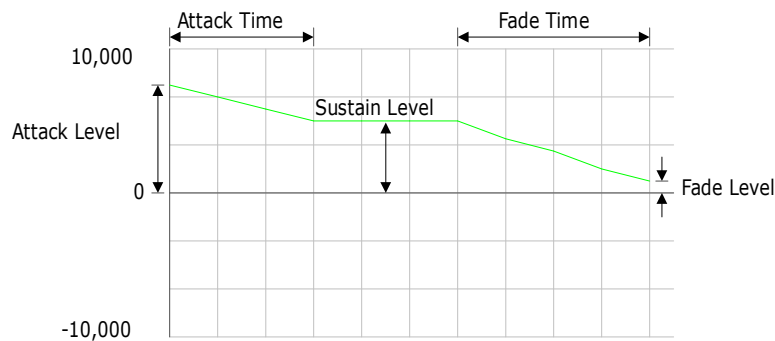
The following illustration represents a sawtooth periodic effect with a magnitude of 5,000, or half the maximum force for the device. The horizontal axis represents the duration of the effect, and the vertical axis represents the magnitude. Points above the center line represent positive force in the direction defined for the effect, and points below the center line represent negative force, or force in the opposite direction.



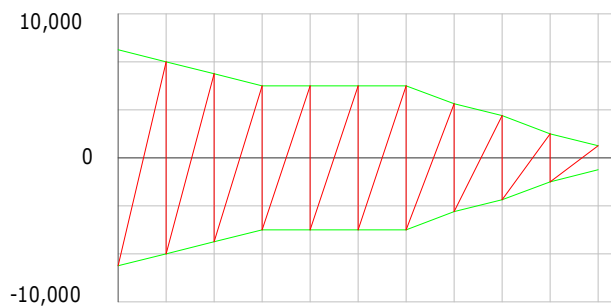
A force can be further shaped by an *envelope*. An envelope defines an *attack value* and a *fade value*, which modify the beginning and ending magnitude of the effect.

Attack and fade also have duration, which determines how long the magnitude takes to reach or fall away from the *sustain value*, the magnitude in the middle portion of the effect.

The following illustration shows an envelope. The attack level is set to 8,000, and the fade level to 1,000. The sustain level is defined by the basic magnitude of the force to which the envelope is being applied; in the example, it is 5,000. In this case, the attack is greater than the sustain, giving the effect an initial strong kick. Both the attack and the fade level can be either greater or less than the sustain level.



The following illustration shows the result of the envelope being applied to the periodic effect in the first illustration. The envelope is mirrored on the negative side of the magnitude. An attack value of 8,000 means that the initial magnitude of the force in either direction is 80 percent of the maximum possible.



Periodic effects and conditions can also be modified by the addition of an *offset*, which defines the amount by which the waveform is shifted up or down from the base level. The practical effect of applying a positive offset to the sawtooth example would be to strengthen the positive force and weaken the negative one—in other words, the force would peak more strongly in one direction than in the other.

Finally, the overall magnitude of an effect can be scaled by *gain*, which is analogous to a volume control in audio. A single gain value can be applied to all effects for a device; you might want to do this to compensate for stronger or weaker forces on different hardware or to accommodate the user's preferences.

## Effect Enumeration

[C++]

The **IDirectInputDevice7::EnumEffects** method returns information about the support offered by the device for various kinds of effects.

It is important to distinguish between *supported effects* and *created effects*, or *effect objects*. A supported effect might be a constant force that can be shaped by an envelope. However, this effect has no properties such as magnitude, direction, duration, attack, or fade. You set these properties when you create an effect object in your application. A supported effect can be represented by many effect objects, each with different parameters—for example, several constant forces, each with different duration, magnitude, and direction.

For information on enumerating created effects, see Effect Object Enumeration.

Like other DirectInput enumerations, the **IDirectInputDevice7::EnumEffects** method requires a callback function; this is documented with the placeholder name **DIEnumEffectsCallback**, but you can use a different name if you want. This function is called for each effect enumerated. Within the function, you can obtain the GUID for each effect, get information about the extent of hardware support, and create one or more effect objects whose methods you can use to manipulate the effect.

The following code example calls the **IDirectInputDevice7::EnumEffects** method that sets the enumeration in motion. The *pvRef* parameter of the callback can be any 32-bit value; in this case, it is a pointer to the device interface, used for getting information about effects supported by the device and for creating effect objects.

```
HRESULT hr;
// LPDIRECTINPUTDEVICE lpdid2; // already initialized

BOOL CALLBACK DIEnumEffectsCallback(LPCDIEFFECTINFO pdei,
    LPVOID pvRef)
{
    LPDIRECTINPUTDEVICE2 lpdid = pvRef; // Pointer to calling device
    LPDIRECTINPUTEFFECT lpdiEffect; // Pointer to created effect
    DIEFFECT diEffect; // Params for created effect
    DICONSTANTFORCE diConstantForce; // Type-specific parameters
    // for diEffect

    if (DIEF_GETTYPE(pdei->dwEffType) == DIEFFT_CONSTANTFORCE)
    {
        /* Here you can extract information about support for the
        effect type (from pdei), and tailor your effects
        accordingly. For example, the device might not support
        envelopes for this type of effect. */
        .
        .
    }
}
```

```
.
// Create one or more constant force effects.
// For each, you have to initialize a DICONSTANTFORCE
// and a DIEFFECT structure.
// See detailed example at Creating an Effect.
.
.
.
hr = pdid->CreateEffect(pdei->guid,
                      &diEffect,
                      &lpdiEffect,
                      NULL);
.
.
.
}
// And so on for other types of effect
.
.
.

return DIENUM_CONTINUE;
} // End of callback
.
.
.
// Set the callback in motion.
hr = lpdid2->EnumEffects(&EnumEffectsCallback,
                       lpdid2, DIEFT_ALL);
```

For more information on how to initialize an effect, see [Creating an Effect](#).

---

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

The **DirectInputDevice.GetEffectsEnum** method enumerates effects supported by the device. It returns a **DirectInputEnumEffects** object representing the collection of supported effects. Methods of **DirectInputEnumEffects** can be used to get information about a particular effect.

It is important to distinguish between *supported effects* and *created effects*, or *effect objects*. A supported effect might be a constant force that can be shaped by an envelope. However, this effect has no properties such as magnitude, direction, duration, attack, or fade. You set these properties when you create an effect object in your application. A supported effect can be represented by many effect objects, each with different parameters—for example, several constant forces, each with different duration, magnitude, and direction.

The following code example shows how an application could enumerate hardware-specific effects, looking for a particular one supported by the Microsoft® SideWinder® joystick. If the desired effect is not found, the application substitutes one of the standard effects:

```
Const BasketballDribble = _
    "{E84CD1AC-81FA-11D0-94AB-0080C74C7E95}"
Dim diEnumEffects as DirectInputEnumEffects
Dim EffGuid As String
Dim i as Integer

' diDev is a DirectInputDevice object.
Set diEnumEffects = diDev.GetEffectsEnum(DIEFT_HARDWARE)
For i = 1 To diEnumEffects.GetCount
    If diEnumEffects.GetEffectGuid(i) = BasketballDribble Then
        EffGuid = BasketballDribble
        Exit For
    End If
Next i
If EffGuid <> BasketballDribble Then
    EffGuid = "GUID_Sine"
    ' Set parameters for emulated dribble here
End If

' Ultimately pass EffGuid to DirectInputDevice.CreateEffect.
```

---

## Loading Effects from a File

---

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

This topic pertains only to applications developed in C++. DirectX for Visual Basic does not support effect files.

---

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

Using the Force Editor supplied with the DirectX SDK, or another application that uses the same file format, you can design effects and save them in a file. You can then use these effects in any application by loading them at run time.

To load effects, call the **IDirectInputDevice7::EnumEffectsInFile** method. By default, the enumeration is limited to the standard DirectInput effects, but you can enumerate other effects by setting the DIFEF\_INCLUDENONSTANDARD flag. By setting the DIFEF\_MODIFYIFNEEDED flag, you can also instruct DirectInput to modify the parameters of effects, if necessary, so that they work on the device. (For

example, an effect authored for two axes can be made to work on a single-axis steering wheel.)

In the following code example, the first three standard effects are loaded from a file and created as DirectInputEffect objects.

```
// g_lpdid7 is a valid IDirectInputDevice7 pointer to a
// force feedback device.

// The array of effect pointers is declared globally.
LPDIRECTINPUTEFFECT pEff[3];
.
.
.
g_lpdid7->EnumEffectsInFile("FEdit1.ffe", EnumEffectsInFileProc,
    NULL, DIFEF_MODIFYIFNEEDED);
.
.
.
```

The following callback procedure is called once for each effect in the file or until it returns DIENUM\_STOP.

```
BOOL CALLBACK EnumEffectsInFileProc(LPCDIFILEEFFECT lpdife,
    LPVOID pvRef)

{
    HRESULT hr;
    static int i;

    // Because the DIFEF_MODIFYIFNEEDED flag was passed, the
    // effect parameters might be modified in lpdife->lpDiEffect.

    hr = g_lpdid7->CreateEffect(lpdife->GuidEffect,
        lpdife->lpDiEffect,
        &pEff[i], NULL);
    if (FAILED(hr))
    {
        // Error handling
    }
    if (++i > 2) return DIENUM_STOP;
    else return DIENUM_CONTINUE;
}
```

---



---

## Information About a Supported Effect

---

### [C++]

The **IDirectInputDevice7::GetEffectInfo** method can be used to retrieve information about the device's support for an effect whose GUID is known. It retrieves the same information that is returned in the **DIEFFECTINFO** structure during enumeration. For more information, see Effect Enumeration.

The following code example fetches information about an effect whose GUID is stored in the *EffectGuid* variable and determines whether the direction of the effect can be changed without stopping and restarting it:

```
DIEFFECTINFO diEffectInfo;
diEffectInfo.dwSize = sizeof(DIEFFECTINFO);
lpdid2->GetEffectInfo(&diEffectInfo, EffectGuid);
if (diEffectInfo.dwDynamicParams & DIEP_DIRECTION)
{
    // Can reset parameter dynamically
}
```

---

### [Visual Basic]

In DirectX for Visual Basic, information about a supported effect must be obtained from the **DirectInputEnumEffects** enumeration object. See Effect Enumeration.

The following code example gets information about the first enumerated effect and determines whether the direction of the effect can be changed without stopping and restarting it:

```
' diEnumEffects is an initialized DirectInputEnumEffects object.
Dim params As Long
params = diEnumEffects.GetDynamicParams(1)
If params And DIEP_DIRECTION Then
    Debug.Print "Direction is dynamic."
End If
```

---

## Creating an Effect

---

### [C++]

Create an effect object by using the **IDirectInputDevice7::CreateEffect** method, as in the following code example, where *pdev2* points to an instance of the interface. This example creates a very simple effect that pulls the joystick away from the user at full force for half a second.

```
HRESULT hr;
```

---

```

LPDIRECTINPUTEFFECT lpdiEffect; // receives pointer to created effect
DIEFFECT diEffect;             // parameters for created effect

DWORD   dwAxes[2] = { DIJOFS_X, DIJOFS_Y };
LONG    lDirection[2] = { 18000, 0 };

DICONSTANTFORCE diConstantForce;

diConstantForce.lMagnitude = DI_FFNOMINALMAX; // Full force

diEffect.dwSize = sizeof(DIEFFECT);
diEffect.dwFlags = DIEFF_POLAR | DIEFF_OBJECTOFFSETS;
diEffect.dwDuration = 0.5 * DI_SECONDS;
diEffect.dwSamplePeriod = 0;           // = default
diEffect.dwGain = DI_FFNOMINALMAX;    // No scaling
diEffect.dwTriggerButton = DIEB_NOTRIGGER; // Not a button response
diEffect.dwTriggerRepeatInterval = 0;  // Not applicable
diEffect.cAxes = 2;
diEffect.rgdwAxes = &dwAxes;
diEffect.rglDirection = &lDirection;
diEffect.lpEnvelope = NULL;
diEffect.cbTypeSpecificParams = sizeof(DICONSTANTFORCE);
diEffect.lpvTypeSpecificParams = &diConstantForce;

hr = pdev2->CreateEffect(GUID_ConstantForce,
                        &diEffect,
                        &lpdiEffect,
                        NULL);

```

In the method call, the first parameter identifies the supported effect with which the created effect is to be associated. The example uses one of the predefined GUIDs found in `Dinput.h`. If you use a predefined GUID, the call fails if the device does not support the effect.

The second parameter sets the parameters as specified in the **DIEFFECT** structure.

The third parameter receives a pointer to the effect object if the call is successful.

The **DIEFF\_POLAR** flag specifies the type of coordinates used for the direction of the force. (See **Effect Direction**.) It is combined with **DIEFF\_OBJECTOFFSETS**, which indicates that any buttons or axes used in other members are identified by their offsets within the **DIDATAFORMAT** structure for the device. The alternative is to use the **DIEFF\_OBJECTIDS** flag, signifying that buttons and axes are identified by the **dwType** member of the **DIDeviceObjectInstance** structure returned for the object when it was enumerated with the **IDirectInputDevice7::EnumObjects** method.

For more information on the members of the **DIEFFECT** structure, see Effect Direction.

---

#### [Visual Basic]

You create an effect object by using the **DirectInputDevice.CreateEffect** method, as in the following code example, where *didev* is a **DirectInputDevice** object. This example creates a very simple effect that pulls the joystick away from the user at full force for half a second.

```
Dim effectInfo As DIEFFECT
Dim objDIEffect As DirectInputEffect

With effectInfo

    .constantForce.IMagnitude = 10000
    .IGain = 10000
    .IDuration = 500000
    .x = 18000
    .ITriggerButton = -1 ' No trigger button
End With

didev.Acquire
Set objDIEffect = didev.CreateEffect("GUID_ConstantForce", effectInfo)
```

The first parameter can be one of the predefined GUID aliases, or an actual GUID in string form known from hardware documentation or retrieved for an enumerated effect by using the **DirectInputEnumEffects.GetEffectGuid** method.

The relevant members of the **DIEFFECT** type vary according to the kind of effect. Constant forces are the simplest kind, requiring only a single type-specific parameter. The **IGain**, **IDuration**, and **ITriggerButton** members should be set for all effects, since the default values of 0 are not usually suitable.

By default, the direction of the effect is expressed in polar coordinates, meaning that **DIEFFECT.x** holds the direction from which the force comes, in hundredths of a degree, and **DIEFFECT.y** must be 0.

---

Effects are automatically downloaded to the device when created, provided the device is not full and is acquired at the exclusive cooperative level.

## Effect Direction

---

#### [C++]

Directions can be defined for one or more axes. As with the mouse and joystick, the x-axis increases from left to right, and the y-axis increases from far to near. For three-dimensional devices, the z-axis increases from up to down.

The direction of an effect is the direction from which it comes. An effect with a direction along the negative y-axis tends to push the stick along the positive y-axis (toward the user). It is somewhat easier to visualize the axis values of a direction if you imagine the user exerting a counteracting force on the device. If the user must push the stick toward the left to counteract an effect, the effect has a left direction; that is, it lies on the negative x-axis.

Direction can be expressed in *polar*, *spherical*, or *Cartesian* coordinates.

Polar coordinates are expressed as a single angle, in hundredths of a degree clockwise from whatever zero-point, or true north, has been established for the effect. Normally this is the negative y-axis; that is, away from the user. Thus an effect with a polar coordinate of 9,000 normally has a direction of east, or to the user's right, and the user must exert force to the right to counteract it.

Spherical coordinates are also in hundredths of a degree, but can contain two or more angles; for each angle, the direction is rotated in the positive direction of the next axis. For a 3-D device, the first angle would normally be rotated from the positive x-axis toward the positive y-axis (clockwise from east), and the second angle would be rotated toward the positive z-axis (down). Thus a force with a direction of (0, 0) would be to the user's right and parallel to the tabletop. A direction of 27,000 for the first angle and 4,500 for the second would be directly away from the user (270 degrees clockwise from east) and angling toward the floor (45 degrees downward from the tabletop); to counteract a force with this direction, the user would have to push forward and down.

Cartesian coordinates are similar to 3-D vectors. If you draw a straight line on graph paper with an origin of (0, 0) at the center of the page, the direction of the line can be defined by the coordinates of any intersection that it crosses, regardless of the distance from the origin. A direction of (1, -2) and a direction of (5, -10) are exactly the same.

### Note

The coordinates used in creating force-feedback effects define only direction, not magnitude or distance.

When an effect is created or modified, the **cAxes**, **rgdwAxes**, and **rglDirection** members of the **DIEFFECT** structure are used to specify the direction of the force.

The **cAxes** member specifies the number of elements in the arrays pointed to by the next two members.

The array pointed to by **rgdwAxes** identifies the axes. If the **DIEFF\_OBJECTOFFSETS** flag has been set, the axes are identified by the offsets within the data format structure. These offsets are most readily identified by using the **DIJOFS\_\*** defines. (For a list of these values, see Joystick Device Constants.)

Finally, the **rglDirection** member specifies the direction of the force.

**Note**

The **cAxes** and **rgdwAxes** members cannot be modified once they have been set. An effect always has the same axis list.

Regardless of whether you are using Cartesian, polar, or spherical coordinates, you must provide exactly as many elements in **rglDirection** as there are axes in the array pointed to by **rgdwAxes**.

In the polar coordinate system, north (0 degrees) lies along the vector (0, -1), where the elements of the vector correspond to the elements in the axis list pointed to by **rgdwAxes**. Normally those axes are x and y, so north is directly along the negative y-axis; that is, away from the user. The last element of *rglDirection* must be 0.

In the example in Creating an Effect, the direction of a two-dimensional force is defined in polar coordinates. The force has a south direction—it comes from the direction of the user, so the user has to pull the stick to counteract it. The direction is 180 degrees clockwise from north, and can be assigned as follows:

```
LONG IDirection[2] = { 18000, 0 };
```

For greater clarity, the assignment could also be expressed as follows:

```
LONG IDirection[2] = { 180 * DI_DEGREES, 0 };
```

For spherical coordinates, presuming that you are working with a three-axis device, the same direction is assigned as follows:

```
LONG IDirection[3] = { 90 * DI_DEGREES, 0, 0 }
```

In the **DIEFFECT** structure, the first angle is measured in hundredths of a degree from the (1, 0) direction, rotated in the direction of (0, 1); the second angle is measured in hundredths of a degree toward (0, 0, 1). The elements of the vector notation again correspond to elements in the array pointed to by the **rgdwAxes** member. Assume that the elements of this array represent the x, y, and z axes, in that order. The point of origin is at x = 1 and y = 0; that is, to the user's right. The direction of rotation is toward the positive y-axis (0, 1); that is, toward the user, or clockwise. The force in the example is 90 degrees clockwise from the right; that is, south. Because the second element of *IDirection* is 0, there is no rotation on the third axis.

How do you accomplish the same thing with Cartesian coordinates? Presuming that you have used the **DIEFF\_CARTESIAN** flag in the **dwFlags** member, you would specify the direction as follows:

```
LONG IDirection[2] = { 0, 1 };
```

Here again, the elements of the array correspond to the axes listed in the array pointed to by **rgdwAxes**. The example sets the x-axis to 0 and the y-axis to 1; that is, the direction lies directly along the positive y-axis, or to the south.

The theory of effect directions can be difficult to grasp, but the practice is fairly straightforward. For code examples, see Examples of Setting Effect Direction.

---

#### [Visual Basic]

DirectX for Visual Basic supports two-axis effects on the x-axis and y-axis.

The direction of an effect is the direction from which it comes. An effect with a direction along the negative y-axis tends to push the stick along the positive y-axis (toward the user). It is somewhat easier to visualize the axis values of a direction if you imagine the user exerting a counteracting force on the device. If the user must push the stick toward the left in order to counteract an effect, the effect has a left direction; that is, it lies on the negative x-axis.

Direction can be expressed in *polar* or *Cartesian* coordinates. By default, polar coordinates are used, and the direction is specified in the **x** member of the **DIEFFECT** type, with **y** always 0. To use Cartesian coordinates, you must specify **DIEFF\_CARTESIAN** in the **IFlags** member and supply values in both **x** and **y**.

Polar coordinates are expressed as a single angle, in hundredths of a degree clockwise from whatever zero-point, or true north, has been established for the effect. Normally, this is the negative y-axis; that is, away from the user. Thus an effect with a polar coordinate of 9,000 normally has a direction of east, or to the user's right, and the user must exert force to the right to counteract it.

Cartesian coordinates are similar to 3-D vectors and are most useful for matching a force to the user's orientation in a 3-D environment. If you draw a straight line on graph paper with an origin of (0, 0) at the center of the page, the direction of the line can be defined by the coordinates of any intersection that it crosses, regardless of the distance from the origin. A direction of (1, -2) and a direction of (5, -10) are exactly the same.

#### Note

Cartesian coordinates used in creating force-feedback effects define only direction, not magnitude or distance.

The following code example reverses the direction of a force represented by the **DirectInputEffect** object *dieff*, which was created using the global **DIEFFECT** type *effectinfo*:

```
effectinfo.IFlags = DIEFF_CARTESIAN
effectinfo.X = effectinfo.X * -1
effectinfo.Y = effectinfo.Y * -1
Call dieff.SetParameters(effectinfo, DIEP_DIRECTION)
```

DirectX for Visual Basic does not support setting direction for conditions.

---

---

## Examples of Setting Effect Direction

---

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

This topic pertains only to applications developed in C++.

---

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

#### Single-Axis Effects

Setting up the direction for a single-axis effect is extremely simple because there is really nothing to specify. Put the **DIEFF\_CARTESIAN** flag in the **dwFlags** member of the **DIEFFECT** structure, and set **rglDirection** to point to a single **LONG** containing the value 0.

The following code example sets up the direction and axis parameters for an x-axis effect:

```
DIEFFECT eff;
LONG    lZero = 0;           // No direction
DWORD   dwAxis = DIJOFS_X;   // X-axis effect

ZeroMemory(&eff, sizeof(DIEFFECT));
eff.cAxes = 1;               // One axis
eff.dwFlags =
    DIEFF_CARTESIAN | DIEFF_OBJECTOFFSETS; // Flags
eff.rglDirection = &lZero;    // Direction
eff.rgdwAxes = &dwAxis;       // Axis for effect
```

#### Two-Axis Effects with Polar Coordinates

Setting up the direction for a polar two-axis effect is only a little more complicated. Set the **DIEFF\_POLAR** flag in **dwFlags**, and set **rglDirection** to point to an array of two **LONGs**. The first element in this array is the direction from which you want the effect to come. The second element in the array must be 0.

The following code example sets up the direction and axis parameters for a two-axis effect coming from the east:

```
DIEFFECT eff;
LONG    rglDirection = { 90 * DI_DEGREES, 0 }; // 90 degrees from
                                                // north, that is, east
DWORD   rgdwAxes[2] = { DIJOFS_X, DIJOFS_Y }; // X- and y-axis

ZeroMemory(&eff, sizeof(DIEFFECT));
eff.cAxes = 2;               // Two axes
eff.dwFlags =
    DIEFF_POLAR | DIEFF_OBJECTOFFSETS;       // Flags
eff.rglDirection = rglDirection;              // Direction
eff.rgdwAxes = rgdwAxes;                     // Axis for effect
```

## Two-Axis Effects with Cartesian Coordinates

Setting up the direction for a Cartesian two-axis effect is a little more complicated. Set the **DIEFF\_CARTESIAN** flag in **dwFlags**, and again set **rglDirection** to point to an array of two **LONGs**. This time the first element in the array is the x-coordinate of the direction vector, and the second is the y-coordinate.

The following code example sets up the direction and axis parameters for a two-axis effect coming from the east:

```
DIEFFECT eff;
LONG    rglDirection = { 1, 0 };           // Positive x = east
DWORD   rgdwAxes[2] = { DIJOFS_X, DIJOFS_Y }; // X- and y-axis

ZeroMemory(&eff, sizeof(DIEFFECT));
eff.cAxes = 2;                             // Two axes
eff.dwFlags =
    DIEFF_CARTESIAN | DIEFF_OBJECTOFFSETS;  // Flags
eff.rglDirection = rglDirection;           // Direction
eff.rgdwAxes = rgdwAxes;                  // Axis for effect
```

---

## Envelopes and Offsets

You can modify the basic magnitude of some effects by applying an envelope and an offset. For an overview, see Basic Concepts of Force Feedback.

---

### [C++]

To apply an envelope when creating or modifying an effect, initialize a **DIENVELOPE** structure, and put a pointer to it in the **lpEnvelope** member of the **DIEFFECT** structure.

The device driver determines which effects support envelopes. Typically, you can apply an envelope to a constant force, a ramp force, or a periodic effect, but not to a condition. To determine whether a particular effect supports an envelope, call the **IDirectInputDevice7::GetEffectInfo** method, and check for the **DIEP\_ENVELOPE** flag in the **dwStaticParams** member of the **DIEFFECTINFO** structure.

To apply an offset, set the **lOffset** member of the **DIPERIODIC** or the **DICONDITION** structure pointed to by the **lpvTypeSpecificParams** member of the **DIEFFECT** structure. For periodic effects, the absolute value of the offset plus the magnitude of the effect must not exceed **DI\_FFNOMINALMAX**.

You cannot apply an offset to a constant force or ramp force. In these cases, the same effect can be achieved by altering the magnitude.

---



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

To apply an envelope when creating or modifying an effect, set the **bUseEnvelope** member of the **DIEFFECT** type to True, and initialize the **envelope** member, which is a **DIENVELOPE** type.

The device driver determines which effects support envelopes. Typically, you can apply an envelope to a constant force, a ramp force, or a periodic effect, but not to a condition. To determine whether a particular effect supports an envelope, call the **DirectInputEnumEffects.GetStaticParams** method, and check for the **DIEP\_ENVELOPE** flag in the returned value.

To apply an offset, set the **IOffset** member of the **DIPERIODICFORCE** or the **DICONDITION** type used in the **periodic** or **condition** member of **DIEFFECT**. For periodic effects, the absolute value of the offset plus the magnitude of the effect must not exceed 10,000.

You cannot apply an offset to a constant force or ramp force. In these cases, the same effect can be achieved by altering the magnitude.

---

## Effect Playback

---

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

There are two principal ways to start playback of an effect: manually by a call to the **IDirectInputEffect::Start** method, or automatically in response to a button press. Playback also starts when you change an effect by calling the **IDirectInputEffect::SetParameters** method with the **DIEP\_START** flag.

Passing **INFINITE** in the *dwIterations* parameter has the effect of playing the effect repeatedly, with the envelope being applied each time. If you want to repeat an effect without repeating the envelope—for example, to begin with a strong kick, then settle down to a steady throb—set *dwIterations* to 1, and set the **dwDuration** member of the **DIEFFECT** structure to **INFINITE**. (This is the structure passed to the **IDirectInputDevice7::CreateEffect** method.)

### Note

Some devices do not support multiple iterations of an effect and accept only the value 1 in the *dwIterations* parameter to the **Start** method. Always check the return value from **Start** to be sure the effect played successfully.

To associate an effect with a button press, set the **dwTriggerButton** member of the **DIEFFECT** structure. Also set the **dwTriggerRepeatInterval** member to the desired delay between playbacks when the button is held down; this is the interval, in microseconds, between the end of one playback and the start of the next.

### Note

On some devices, multiple effects cannot be triggered by the same button; if you associate more than one effect with a button; the last effect downloaded is the one triggered. Also, trigger repeat interval might not be supported.

To dissociate an effect from its trigger button, either call the **IDirectInputEffect::Unload** method, or set the parameters for the effect with **dwTriggerButton** set to **DIEB\_NOTRIGGER**.

Triggered effects, like all others, are lost when the application loses access to the device. To make them active again, download them as soon as the application reacquires the device. This step is not necessary for effects not associated with a trigger because they are automatically downloaded if necessary whenever the **Start** method is called.

If an effect has a finite duration and is started by a call to the **Start** method, it stops playing when the time has elapsed. If its duration was set to **INFINITE**, playback ends only when the **IDirectInputEffect::Stop** method is called. An effect associated with a trigger button starts when the button is pressed, and stops when the button is released or the duration has elapsed, whichever comes sooner.

---

#### [Visual Basic]

There are two principal ways to start playback of an effect: manually by a call to the **DirectInputEffect.Start** method, or automatically in response to a button press. Playback also starts when you change an effect by calling the **DirectInputEffect.SetParameters** method with the **DIEP\_START** flag.

Passing **-1** in the *iterations* parameter of **Start** has the effect of playing the effect repeatedly, with the envelope being applied each time. If you want to repeat an effect without repeating the envelope—for example, to begin with a strong kick, then settle down to a steady throb—set *iterations* to **1**, and set the **IDuration** member of the **DIEFFECT** type to **-1**. (This is the type passed to the **DirectInputDevice.CreateEffect** method.)

#### Note

Some devices do not support multiple iterations of an effect and accept only the value **1** in the *iterations* parameter to the **Start** method.

To associate an effect with a button press, set the **ITriggerButton** member of the **DIEFFECT** type. Also set the **ITriggerRepeatInterval** member to the desired delay between playbacks when the button is held down; this is the interval, in microseconds, between the end of one playback and the start of the next.

#### Note

On some devices, multiple effects cannot be triggered by the same button; if you associate more than one effect with a button; the last effect downloaded is the one triggered. Also, trigger repeat interval might not be supported.

To dissociate an effect from its trigger button, either call the **DirectInputEffect.Unload** method, or set the parameters for the effect with **ITriggerButton** set to -1.

Triggered effects, like all others, are lost when the application loses access to the device. To make them active again, download them as soon as the application reacquires the device. This step is not necessary for effects not associated with a trigger because they are automatically downloaded if necessary whenever the **Start** method is called.

If an effect has a finite duration and is started by a call to the **Start** method, it stops playing when the time has elapsed. If its duration was set to -1, playback ends only when the **DirectInputEffect.Stop** method is called. An effect associated with a trigger button starts when the button is pressed, and stops when the button is released or the duration has elapsed, whichever comes sooner.

---

## Downloading and Unloading Effects

Before an effect can be played, it must be downloaded to the device. Downloading an effect means telling the driver to prepare the effect for playback. It is entirely up to the driver to determine how this is done. Generally the driver places the parameters of the effect in hardware memory to minimize the subsequent transfer of data between the device and the system. The consequent reduction in latency is particularly important for conditions and for effects played in response to a trigger, such as a fire button. Ideally the device does not have to communicate with the system at all to respond to axis movements and button presses.

Downloading is done automatically when you create an effect, provided the device is not full and is acquired at the exclusive cooperative level. By default, it is also done when you start the effect or change its parameters.

---

### [C++]

If you specify the **DIEP\_NODOWNLOAD** flag when changing parameters, you must subsequently use the **IDirectInputEffect::Download** method to download or update the effect.

When the device is unacquired—for example, when it has been acquired with the exclusive foreground cooperative level and the application moves to the background—effects are unloaded and must be downloaded again when the application regains the foreground. As previously stated, this is done automatically when you call the **IDirectInputEffect::Start** method, but you can choose to download all effects immediately on reacquiring the device. You always have to download effects associated with a trigger button, since the **Start** method is not normally called for such effects.

If your application gets the **DIERR\_DEVICEFULL** error when downloading an effect, you must make room for the new effect by unloading an old one. You can remove an effect from the device by calling the **IDirectInputEffect::Unload** method.

---

You can also remove all effects by resetting the device through a call to the **IDirectInputDevice7::SendForceFeedbackCommand** method.

---

#### [Visual Basic]

If you specify the **DIEP\_NODOWNLOAD** flag when changing parameters, you must subsequently use the **DirectInputEffect.Download** method to download or update the effect.

When the device is unacquired—for example, when it has been acquired with the exclusive foreground cooperative level and the application moves to the background—effects are unloaded and must be downloaded again when the application regains the foreground. As previously stated, this is done automatically when you call the **DirectInputEffect.Start** method, but you can choose to download all effects immediately on reacquiring the device. You always have to download effects associated with a trigger button, since the **Start** method is not normally called for such effects.

If your application gets the **DIERR\_DEVICEFULL** error when downloading an effect, you must make room for the new effect by unloading an old one. You can remove an effect from the device by calling the **DirectInputEffect.Unload** method. You can also remove all effects by resetting the device through a call to the **DirectInputDevice.SendForceFeedbackCommand** method.

---

When you create a force-feedback device, the hardware and driver are reset, so any existing effects are cleared.

## Changing an Effect

---

#### [C++]

You can modify the parameters of an effect, in some cases even while the effect is playing. You do this by using the **IDirectInputEffect::SetParameters** method.

The **dwDynamicParams** member of the **DIEFFECTINFO** structure tells you which effect parameters can be changed while an effect is playing. If you attempt to modify an effect parameter that cannot be modified while the effect is playing and the effect is still playing, DirectInput normally stops the effect, updates the parameters, and restarts the effect. You can override this default behavior by passing the **DIEP\_NORESTART** flag.

The following code example changes the magnitude of the constant force that was set in the example under Creating an Effect.

```
DIEFFECT    diEffect;    // Parameters for effect
DICONSTANTFORCE diConstantForce;
// type-specific parameters.
```

```
diConstantForce.IMagnitude = 5000;  
diEffect.dwSize = sizeof(DIEFFECT);  
diEffect.cbTypeSpecificParams = sizeof(DICONSTANTFORCE);  
diEffect.lpvTypeSpecificParams = &diConstantForce;  
hr = lpdiEffect->SetParameters(&diEffect, DIEP_TYPESPECIFICPARAMS);
```

The flag ensures that the transfer of data from the **DIEFFECT** structure is restricted to the relevant members so that you do not have to initialize the entire structure and so that the minimum possible amount of data needs to be sent to the device.

---

#### [Visual Basic]

You can modify the parameters of an effect, in some cases even while the effect is playing. You do this by using the **DirectInputEffect.SetParameters** method.

To find out which effect parameters can be changed while an effect is playing, call the **DirectInputEnumEffects.GetDynamicParams** method.

If you attempt to modify an effect parameter that cannot be modified while the effect is playing and the effect is still playing, DirectInput normally stops the effect, updates the parameters, and restarts the effect. You can override this default behavior by passing the DIEP\_NOESTART flag to **SetParameters**.

The following code example changes the magnitude of a constant force represented by the **DirectInputEffect** object *dieff*, which was created using the global **DIEFFECT** type *effectinfo*:

```
effectinfo.constantForce.IMagnitude = 5000  
Call dieff.SetParameters(effectinfo, DIEP_TYPESPECIFICPARAMS)
```

You must set the DIEP\_TYPESPECIFICPARAMS flag if you are changing the **condition**, **constantforce**, **periodicforce**, or **rampforce** members of **DIEFFECT**.

---

## Gain

You might want to scale the force of your effects according to the force exerted by different devices. For example, if an application's effects feel right on a device that puts out a maximum force of *n* Newtons on a given axis, you might want to adjust the gain for a device that puts out more force. (You cannot use the gain to increase the maximum force of the axis, so you should set the basic effect magnitudes to values suitable for devices that put out less force.)

Gain can also be used to decrease the magnitude of a hardware-defined effect.

---

#### [C++]

The force generated by a device object such as an axis or button is returned in the **dwFFMaxForce** member of the **DIDeviceObjectInstance** structure when objects are enumerated. (See Device Object Enumeration.)

You can set the gain for the entire device by using the **IDirectInputDevice7::SetProperty** method.

You must set the gain for individual effects when creating them by putting a value in the **dwGain** member of the **DIEFFECT** structure. (If **dwGain** is 0, the effect is not felt.) You can change this value later by using **IDirectInputEffect::SetParameters**, passing **DIEP\_GAIN** in the *dwFlags* parameter.

---

#### [Visual Basic]

You can set the gain for the entire device by using the **DirectInputDevice.SetProperty** method.

You must set the gain for individual effects when creating them by putting a value in the **lGain** member of the **DIEFFECT** type. (If **lGain** is 0, the effect is not felt.) You can change this value later by using **DirectInputEffect.SetParameters**, passing **DIEP\_GAIN** in the *flags* parameter.

---

The purpose of setting the device gain is to allow your application to have control over the strength of all effects all at once. For example, you might have a slider control in your application to allow the user to specify how strong the force-feedback effects should be, like the master volume control on a sound mixer. When the device gain is set, your application does not need to adjust the gain of each individual effect to suit the user's preferences.

A gain value can be in the range from 0 through 10,000, where 10,000 indicates that magnitudes are not to be scaled, 7,500 means that forces are to be scaled to 75 percent of their nominal magnitudes, and so on.

## Force-Feedback State

---

#### [C++]

The **IDirectInputDevice7::SendForceFeedbackCommand** method allows you to turn off the device's actuators (effectively causing it to ignore any effects that are being played), pause or stop playback of effects, and reset the device so that all downloaded effects are removed.

To retrieve the current force-feedback state, use the **IDirectInputDevice7::GetForceFeedbackState** method. This method returns information about whether the actuators are active, whether playback is paused, and whether the device has been reset. It also retrieves information about various switches and about whether the device is currently powered.

---

[\[Visual Basic\]](#)

The **DirectInputDevice.SendForceFeedbackCommand** method allows you to turn off the device's actuators (effectively causing it to ignore any effects that are being played), pause or stop playback of effects, and reset the device so that all downloaded effects are removed.

To retrieve the current force-feedback state, use the **DirectInputDevice.GetForceFeedbackState** method. This method returns information about whether the actuators are active, whether playback is paused, and whether the device has been reset. It also retrieves information about various switches and about whether the device is currently powered.

---

## Effect Object Enumeration

---

[\[Visual Basic\]](#)

This topic pertains only to applications developed in C++. DirectX for Visual Basic does not support enumeration of created effects.

---

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

Whenever you need to examine or manipulate all the effects you have created, use the **IDirectInputDevice7::EnumCreatedEffectObjects** method. Since no flags are currently defined for this method, you cannot restrict the enumeration to particular kinds of effects; all effects are enumerated.

### Note

This method enumerates created effects, not effects supported by a device. For more information on the distinction between the two, see Effect Enumeration.

Like other DirectInput enumerations, the **EnumCreatedEffectObjects** method requires a callback function. This standard callback is documented with the placeholder name **DIEnumCreatedEffectObjectsCallback**, but you can use a different name. The function is called for each effect enumerated. Within the function you can perform any processing that you want; however, it is not safe to create a new effect while enumeration is going on.

The following is a code example of the callback function and the call to the **EnumCreatedEffectObjects** method. The *pvRef* parameter of the callback can be any 32-bit value; in this case, it is a pointer to the device interface.

```
HRESULT hr;
// LPDIRECTINPUTDEVICE lpdid; // already initialized

BOOL CALLBACK DIEnumCreatedEffectObjectsCallback(
    LPDIRECTINPUTEFFECT peff, LPVOID pvRef);
{
```

---

```

LPDIRECTINPUTDEVICE pdid = pvRef; // Pointer to calling device
DIEFFECT             diEffect;    // Params for created effect

diEffect.dwSize = sizeof(DIEFFECTINFO);
peff->GetParameters(&diEffect, DIEP_ALLPARAMS);
// Check or set parameters, or do anything else.
.
.
.
} // End of callback

// Set the callback in motion.
hr = lpdid->EnumCreatedEffectObjects(&EnumCreatedEffectObjectsCallback,
                                     &lpdid, 0);

```

---

## Effect Types

This section contains information specific to the following types of force-feedback effects:

- Constant Forces
- Ramp Forces
- Periodic Effects
- Conditions
- Custom Forces
- Device-Specific Effects

### Constant Forces

A constant force is a force with a defined magnitude and duration.

You can apply an envelope to a constant force to give it shape. For example, suppose you have an effect with a nominal magnitude of 2,000 and a duration of 2 seconds. Then you apply an envelope with the following values:

Attack time	0.5 second
Initial attack level	5,000
Fade time	1 second
Fade level	0

When you play the effect, you get the following:

Elapsed time	Magnitude
--------------	-----------



---

0.0	5,000
0.1	4,400
0.2	3,800
0.3	3,200
0.4	2,600
0.5	2,000
(Duration of sustain)	2,000
1.0:	2,000
1.1	1,800
1.2	1,600
1.3	1,400
1.4	1,200
1.5	1,000
1.6	800
1.7	600
1.8	400
1.9	200
2.0	0

You cannot apply an offset to a constant force.

---

#### [C++]

To create a constant force, pass `GUID_ConstantForce` to the **IDirectInputDevice7::CreateEffect** method. You can also pass any other GUID obtained by the **IDirectInputDevice7::EnumEffects** method, provided the low byte of the **dwEffType** member of the **DIEFFECTINFO** structure (**DIEFT\_GETTYPE(dwEffType)**) is equal to **DIEFT\_CONSTANTFORCE**. In this way, you can use hardware-specific forces designed by the manufacturer, such as a "constant" force that actually varies in magnitude in a seemingly random fashion to simulate turbulence.

A constant force uses a **DICONSTANTFORCE** structure to define the magnitude. A negative magnitude has the effect of reversing the direction of the force.

---

#### [Visual Basic]

To create a constant force, pass "GUID\_ConstantForce" to the **DirectInputDevice.CreateEffect** method. You can also pass any other GUID obtained by the **DirectInputEnumEffects.GetEffectGuid** method, provided the low byte of the value returned by **DirectInputEnumEffects.GetType** is equal to **DIEFT\_CONSTANTFORCE**. In this way, you can use hardware-specific forces designed by the manufacturer, such as a "constant" force that actually varies in magnitude in a seemingly random fashion to simulate turbulence.

---

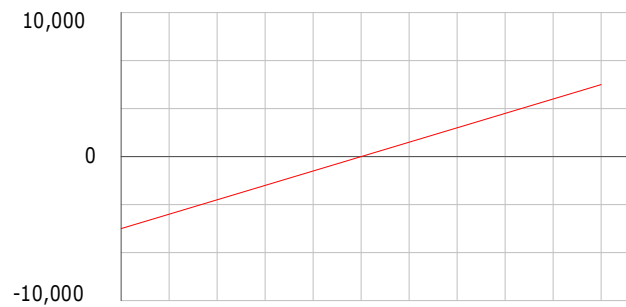
The magnitude of a constant force is contained in a **DICONSTANTFORCE** type in the **constantForce** member of **DIEFFECT**. A negative magnitude has the effect of reversing the direction of the force.

---

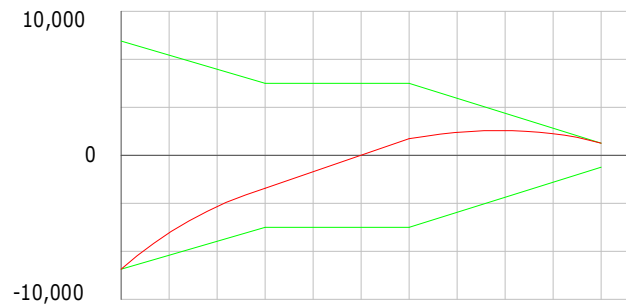
## Ramp Forces

A ramp force is a force with defined starting and ending magnitudes and a finite duration. A ramp force can continue in a single direction, or it can start as a strong push in one direction, weaken, stop, and then strengthen in the opposite direction.

The following illustration shows a ramp force that starts at a magnitude of -5,000 and ends at 5,000:



You can apply an envelope to a ramp force to shape it further. The following diagram shows the effect of applying an envelope, shown in green, to the ramp force in the previous diagram.



During the sustain portion of the envelope, the magnitude of the effect follows the same straight line as before the envelope was applied. For the duration of the attack and fade, the slope of the ramp is modified by the attack and fade levels.

You cannot apply an offset to a ramp force.

To create a ramp force, pass `GUID_RampForce` to the **IDirectInputDevice7::CreateEffect** method. You can also pass any other GUID obtained by the **IDirectInputDevice7::EnumEffects** method, provided the low byte of the **dwEffType** member of the **DIEFFECTINFO** structure (**DIEFT\_GETTYPE(dwEfftype)**) is equal to `DIEFT_RAMPFORCE`. In this way, you can use hardware-specific ramp forces designed by the manufacturer.

A ramp force uses a **DIRAMPFORCE** structure to define the starting and ending magnitude of the force, whereas the duration is taken from the **DIEFFECT** structure. Duration must never be set to `INFINITE`.

---

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

To create a ramp force, pass "`GUID_RampForce`" to the **DirectInputDevice.CreateEffect** method. You can also pass any other GUID obtained by the **DirectInputEnumEffects.GetEffectGuid** method, provided the low byte of the value returned by **DirectInputEnumEffects.GetType** is equal to `DIEFT_RAMPFORCE`. In this way, you can use hardware-specific ramp forces designed by the manufacturer.

The starting and ending magnitude of the force are defined in a **DIRAMPFORCE** type in the **rampForce** member of **DIEFFECT**. The duration cannot be infinite, and therefore **DIEFFECT.IDuration** must be a positive value.

---

## Periodic Effects

Periodic effects are waveform effects. DirectInput defines the following forms:

- Square.
- Sine.
- Cosine.
- Triangle.
- SawtoothUp. The waveform drops vertically after it reaches maximum positive force. See the example in Basic Concepts of Force Feedback.
- SawtoothDown. The waveform rises vertically after it reaches maximum negative force.

An envelope can be applied to periodic effects. See the example at Basic Concepts of Force Feedback.

The phase of a periodic effect is the point along the waveform where the effect begins. Phase is measured in hundredths of a degree, from 0 to 35,999. The following table indicates where selected phase values (in degrees) lie along the various waveforms. *Max* is the top (+) or bottom (–) of the wave and *Mid* is the midpoint, at which no force is applied in either direction.

---

	0	90	180	270
Square	+Max	+Max	–Max	–Max
Sine	Mid	+Max	Mid	–Max
Triangle	+Max	Mid	–Max	Mid
SawtoothUp	–Max	–Max/2	Mid	+Max/2 (reaches +Max just before the cycle repeats)
SawtoothDown	+Max	+Max/2	Mid	–Max/2 (reaches –Max just before the cycle repeats)

A driver can round off a phase value to the nearest supported value. For example, for a sine effect some drivers support only values in the range from 0 through 9,000 (to create a cosine); for other effects, only values in the range from 0 through 18,000 are supported.

---

#### [C++]

To create a periodic force, pass one of the following values in the *rguid* parameter of the **IDirectInputDevice7::CreateEffect** method:

- GUID\_Square
- GUID\_Sine
- GUID\_Triangle
- GUID\_SawtoothUp
- GUID\_SawtoothDown

You can also pass any other GUID obtained by the **IDirectInputDevice7::EnumEffects** method, provided the low byte of the **dwEffType** member of the **DIEFFECTINFO** structure (**DIEFT\_GETTYPE(dwEfftype)**) is equal to **DIEFT\_PERIODIC**. In this way, you can use hardware-specific forces designed by the manufacturer. For example, a hardware device might support a periodic effect that rotates the stick in a small circle.

The type-specific structure for periodic effects is **DIPERIODIC**.

Do not confuse the *period* of a periodic effect (**DIPERIODIC.dwPeriod**) with the *sample period* (**DIEFFECT.dwSamplePeriod**). The period is the length of time that it takes to go through a complete wave cycle. The sample period, as for all effects, is the minimum time between actual adjustments of magnitude.

---

#### [Visual Basic]

To create a periodic force, pass one of the following values in the *guid* parameter of the **DirectInputDevice.CreateEffect** method:

- "GUID\_Square"

- "GUID\_Sine"
- "GUID\_Triangle"
- "GUID\_SawtoothUp"
- "GUID\_SawtoothDown"

You can also pass any other GUID obtained by the **DirectInputEnumEffects.GetEffectGuid** method, provided the low byte of the value returned by **DirectInputEnumEffects.GetType** is equal to **DIEFT\_PERIODIC**. In this way, you can use hardware-specific periodic forces designed by the manufacturer.

The type-specific parameters for periodic effects are contained in a **DIPERIODICFORCE** type in the **periodicForce** member of **DIEFFECT**.

Do not confuse the period of a periodic effect (**DIEFFECT.periodic.IPeriod**) with the sample period (**DIEFFECT.ISamplePeriod**). The period is the length of time it takes to go through a complete wave cycle. The sample period, as for all effects, is the minimum time between actual adjustments of magnitude.

---

## Conditions

Conditions are forces applied in response to current sensor values within the device. In other words, conditions require information about device motion, such as position or velocity of a joystick handle.

In general, conditions are not associated with individual events during a game or other application. They represent ambient phenomena, such as the stiffness or looseness of a flight stick, or the tendency of a steering wheel to return to a straight-ahead position.

A condition does not have a predefined magnitude; the magnitude is scaled in proportion to the movement or position of the input object.

DirectInput defines the following types of condition effects:

- Friction. The force is applied when the axis is moved and depends on the defined friction coefficient.
- Damper. The force increases in proportion to the velocity with which the user moves the axis.
- Inertia. The force increases in proportion to the acceleration of the axis.
- Spring. The force increases in proportion to the distance of the axis from a defined neutral point.

Most hardware devices do not support the application of envelopes to conditions.

Conditions have the following type-specific parameters:

*Offset*

The offset from the 0 reading of the appropriate sensor value at which the condition begins to be applied. For a spring effect, the neutral point—that is, the point along the axis at which the spring would be considered at rest—would be defined by the offset for the condition. For a damper, the offset would define the greatest velocity value for which damping force is 0. Offset is not normally used for inertia or friction effects.

#### *Coefficient*

A multiplier that scales the effect. For some devices, you can set separate coefficients for the positive and negative direction along the axis associated with the condition. For example, a flight stick controlling a damaged aircraft might move more easily to the right than to the left.

#### *Saturation*

In force feedback, saturation is an expression of the maximum possible force for an effect. For example, suppose a flight stick has a spring condition on the x-axis. The offset is 0, and the coefficient is 10,000, so the maximum force is normally exerted when the stick is furthest from the center. But if you define a positive and negative saturation of 5,000, the force does not increase after the stick has been moved halfway to the right or left.

Not all devices support saturation.

#### *Deadband*

The deadband is a zone around the offset of an axis at which the condition is not active. In the case of a spring that is at rest in the middle of an axis, the deadband enlarges this area of rest.

Not all devices support saturation.

Conditions can have duration, although in most cases, you would probably want to set the duration to infinite (–1) and stop the effect only in response to some event in the application.

---

#### [C++]

To create a condition, pass one of the following values in the *rguid* parameter of the **IDirectInputDevice7::CreateEffect** method:

- GUID\_Spring
- GUID\_Damper
- GUID\_Inertia
- GUID\_Friction

You can also pass any other GUID obtained by the **IDirectInputDevice7::EnumEffects** method, provided the low byte of the **dwEffType** member of the **DIEFFECTINFO** structure (**DIEFT\_GETTYPE(dwEffType)**) is equal to **DIEFT\_CONDITION**. In this way, you can use hardware-specific conditions designed by the manufacturer.

The type-specific structure for conditions is **DICONDITION**. For multiple-axis conditions, you can provide an array of such structures, one for each axis, or a single

structure that defines the condition in the specified direction. In either case, you must set the **cbTypeSpecificParams** member of the **DIEFFECT** structure to the number of bytes used; that is, to **sizeof(DICONDITION) \* n**, where *n* is the number of structures provided. For more information on how to use either single or multiple structures, see the Remarks for the **DICONDITION** structure.

An application should call the **IDirectInputDevice7::GetEffectInfo** method or the **IDirectInputDevice7::EnumEffects** method and examine the **dwEffectType** member of the **DIEFFECTINFO** structure to determine whether both a positive and a negative coefficient and saturation for the effect are supported on the device. If the effect does not return the **DIEFT\_POSNEGCOEFFICIENTS** flag, it ignores the value in the **INegativeCoefficient** member, and the value in **IPositiveCoefficient** is applied to the entire axis. Likewise, if the effect does not return the **DIEFT\_POSNEGSATURATION** flag, it ignores the value in **dwNegativeSaturation**, and the value in **dwPositiveSaturation** is used as the negative saturation level. Finally, if the effect does not return the **DIEFT\_SATURATION** flag, it ignores both the **dwPositiveSaturation** and **dwNegativeSaturation** values, and no saturation is applied.

You can set a coefficient to a negative value, and this has the effect of generating the force in the opposite direction. For example, for a spring effect, it would cause the spring to push away from the offset point, rather than pulling toward it.

You should also check **DIEFFECTINFO.dwEffectType** for the **DIEFT\_DEADBAND** flag, to see if deadband is supported for the condition. If it is not supported, the value in the **IDeadBand** member of the **DICONDITION** structure is ignored.

---

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

To create a condition, pass one of the following values in the *rguid* parameter of the **IDirectInputDevice7::CreateEffect** method:

- "GUID\_Spring"
- "GUID\_Damper"
- "GUID\_Inertia"
- "GUID\_Friction"

You can also pass any other GUID obtained by the **DirectInputEnumEffects.GetEffectGuid** method, provided the low byte of the value returned by **DirectInputEnumEffects.GetType** is equal to **DIEFT\_CONDITION**. In this way, you can use hardware-specific conditions designed by the manufacturer.

The type-specific parameters for a condition are described in a **DICONDITION** type in the **condition** member of the **DIEFFECT** type.

An application should call the **DirectInputEnumEffects.GetType** method and examine the return value member to determine whether both a positive and a negative

coefficient and saturation for the effect are supported on the device. If the effect does not return the **DIEFT\_POSNEGCOEFFICIENTS** flag, it ignores the value in **DICONDITION.INegativeCoefficient**, and the value in **IPositiveCoefficient** is applied to the entire axis. Likewise, if the effect does not return the **DIEFT\_POSNEGSATURATION** flag, it ignores the value in **INegativeSaturation**, and the value in **IPositiveSaturation** is used as the negative saturation level. Finally, if the effect does not return the **DIEFT\_SATURATION** flag, it ignores both the **IPositiveSaturation** and **INegativeSaturation** values, and no saturation is applied.

You can set a coefficient to a negative value, and this has the effect of generating the force in the opposite direction. For example, for a spring effect, it would cause the spring to push away from the offset point, rather than pulling toward it.

You should also check the value returned by **GetType** for the **DIEFT\_DEADBAND** flag, to see if deadband is supported for the condition. If it is not supported, the value in the **IDeadBand** member of the **DICONDITION** type is ignored.

To create a single-axis effect, set the coefficients for the unused axis to 0 in the **conditionX** or **conditionY** member of the **DIEFFECT** type.

DirectX for Visual Basic does not support rotation of conditions.

---

## Custom Forces

Application writers can create their own effects by creating a custom force. A custom force is an array of constant force values played back by the device.

---

### [C++]

The type-specific structure for custom waveform effects is **DICUSTOMFORCE**.

Set the **dwSamplePeriod** member of the **DICUSTOMFORCE** structure and the **dwSamplePeriod** member of the **DIEFFECT** structure to the same value. This is the length of time, in milliseconds, for which each element in the array of forces is played.

The custom force is played repeatedly until the time set in the **dwDuration** member of the **DIEFFECT** structure has elapsed.

---

### [Visual Basic]

To create a custom force, first define an array of magnitudes, all in the range from –10,000 through 10,000. Then pass this array to the **DirectInputDevice.CreateCustomEffect** method.

Set the *samplePeriod* parameter and the **ISamplePeriod** member of the **DIEFFECT** type to the same value. This is the length of time, in milliseconds, for which each element in the array of forces is played.



The custom force is played repeatedly until the time set in the **DIEFFECT.IDuration** has elapsed.

---

## Device-Specific Effects

Hardware drivers can support special effects that do not fit into the categories defined by **DirectInput**. The type-specific parameters for these effects can be hard-coded or modifiable by the application.

---

### [C++]

If type-specific parameters are modifiable, the application developer must obtain a header file declaring the data structure required by the effect.

The hardware vendor must provide a GUID identifying the device-specific effect and might provide a custom structure for the type-specific parameters of the effect. Your application then must initialize a **DIEFFECT** structure and a type-specific structure, as with any other effect. You then call the **IDirectInputDevice7::CreateEffect** method, passing the device-specific GUID and a pointer to the **DIEFFECT** structure.

When you obtain information about a device-specific effect in a **DIEFFECTINFO** structure, the low byte of the **dwEffType** member (**DIEFT\_GETTYPE(dwEfftype)**) indicates into which of the predefined **DirectInput** effect categories (constant force, ramp force, periodic, or condition) the effect falls. If it does not fall into any of the predefined categories, the value is **DIEFT\_HARDWARE**.

If a device-specific effect falls into one of the predefined categories, the **lpvTypeSpecificParams** member of the **DIEFFECT** structure must point to the corresponding **DICONSTANTFORCE**, **DIRAMPFORCE**, **DIPERIODIC**, or **DICONDITION** structure, and the **cbTypeSpecificParams** member must be equal to the size of that structure.

If the (**DIEFT\_GETTYPE(dwEfftype)** == **DIEFT\_HARDWARE**), the values of the **lpvTypeSpecificParams** and **cbTypeSpecificParams** members depend on whether the effect requires custom type-specific parameters. If it does, these values must refer to the appropriate structure defined in the manufacturer's header file and declared and initialized by your application. If the effect does not require custom parameters—that is, if the **dwStaticParams** member of the **DIEFFECTINFO** structure for the hardware effect does not have the **DIEP\_TYPESPECIFICPARAMS** flag—**lpvTypeSpecificParams** must be **NULL** and **cbTypeSpecificParams** must be 0.

**DirectInput** passes the GUID and the **DIEFFECT** structure to the device driver for verification. If the GUID is unknown, the device returns **DIERR\_DEVICENOTREG**. If the GUID is known but the type-specific data is incorrect for that effect, the device returns **DIERR\_INVALIDPARAM**.

---

### [Visual Basic]

DirectX for Visual Basic does not support hardware effects that require custom type-specific parameters.

The hardware vendor must provide a GUID identifying the device-specific effect. Your application then must initialize a **DIEFFECT** type, as with any other effect. You then pass this type, along with the GUID, to the **DirectInputDevice.CreateEffect** method.

When you obtain information about a device-specific effect in a **DirectInputEnumEffects** enumeration, the low byte of the return value of **DirectInputEnumEffects.GetType** indicates into which of the predefined DirectInput effect categories (constant force, ramp force, periodic, or condition) the effect falls. If it does not fall into any of the predefined categories, the value is **DIEFT\_HARDWARE**.

If a device-specific effect falls into one of the predefined categories, the **DIEFFECT** type must be initialized, as it would be for a standard force of that category. For example, if the effect type is **DIEFT\_CONSTANTFORCE**, the **constantForce** member must be initialized.

DirectInput passes the GUID and the **DIEFFECT** structure to the device driver for verification. If the GUID is unknown, the device returns **DIERR\_DEVICENOTREG**. If the GUID is known but the type-specific data is incorrect for that effect, the device returns **DIERR\_INVALIDPARAM**.

---

## Designing for Previous Versions of DirectInput

---

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

This topic pertains only to applications developed in C++.

---

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

In several places, DirectInput requires you to pass a version number to a method. This parameter specifies which version of DirectX the DirectInput subsystem should emulate.

Applications designed for the latest version of DirectInput should pass the value **DIRECTINPUT\_VERSION** as defined in **Dinput.h**.

Applications designed to run under previous versions should pass a value corresponding to the version of DirectInput for which they were designed, with the main version number in the high-order byte. For example, an application that was designed to run on DirectInput 3 should pass a value of 0x0300.

If you define `DIRECTINPUT_VERSION` as 0x0300 before including the `Dinput.h` header file, the header file generates structure definitions compatible with DirectInput 3.0.

If you do not define `DIRECTINPUT_VERSION` before including the `Dinput.h` header file, the header file generates structure definitions compatible with the current version of DirectInput. However, the DirectX 3-compatible structures are available under the same names with "`_DX3`" appended. For example, the DirectX 3-compatible **DIDEVCAPS** structure is called **DIDEVCAPS\_DX3**.

You must also use the appropriate versions of the DirectInput and DirectInputDevice interfaces. For versions of DirectX prior to DirectX 7.0, you must use **IDirectInput** and either **IDirectInputDevice** or **IDirectInputDevice2** (the latter is available for DirectX 5.0 or later.)

---

## DirectInput Reference

This section contains reference information for the application programming interface (API) elements provided by Microsoft® DirectInput® in C/C++ and Microsoft® Visual Basic® development systems. Reference material is organized by language:

- DirectInput C/C++ Reference
- DirectInput Visual Basic Reference

## DirectInput C/C++ Reference

Reference material for the DirectInput C/C++ API is divided into the following categories:

- Interfaces
- Functions
- Callback Functions
- Macros
- Structures
- Device Constants
- Return Values

## Interfaces

This section contains references for methods of the following DirectInput interfaces:

- **IDirectInput7**
- **IDirectInputDevice7**
- **IDirectInputEffect**

**Note**

All DirectInput methods have corresponding macros that expand to C or C++ syntax depending on which language is defined. These macros are found in the Dinput.h header file and are not documented separately.

## IDirectInput7

Applications use the methods of the **IDirectInput7** interface to enumerate, create, and retrieve the status of DirectInput devices, initialize the DirectInput object, and invoke an instance of the Windows Control Panel.

The **IDirectInput7** interface is obtained by using the **DirectInputCreateEx** function.

**IDirectInput7** supersedes the **IDirectInput** interface used in earlier versions of DirectX.

The methods of the **IDirectInput7** interface can be organized into the following groups.

<b>Device Management</b>	<b>CreateDevice</b>
	<b>CreateDeviceEx</b>
	<b>EnumDevices</b>
	<b>FindDevice</b>
	<b>GetDeviceStatus</b>
<b>Miscellaneous</b>	<b>Initialize</b>
	<b>RunControlPanel</b>

The **IDirectInput** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

<b>IUnknown</b>	<b>AddRef</b>
	<b>QueryInterface</b>
	<b>Release</b>

The **LPDIRECTINPUT** type is defined as a pointer to the **IDirectInput7** interface:

```
typedef struct IDirectInput7 *LPDIRECTINPUT7;
```

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for

Windows 95.

**Header:** Declared in dinput.h.

**Import Library:** Use dinput.lib.

## IDirectInput7::CreateDevice

The **IDirectInput7::CreateDevice** method creates and initializes an instance of a device based on a given GUID, and obtains an **IDirectInputDevice** interface. This method has been superseded by **IDirectInput7::CreateDeviceEx**.

```
HRESULT CreateDevice(
    REFGUID rguid,
    LPDIRECTINPUTDEVICE *lplpDirectInputDevice,
    LPUNKNOWN pUnkOuter
);
```

### Parameters

*rguid*

Reference to (C++) or address of (C) the instance GUID for the desired input device (see Remarks). The GUID is retrieved through the **IDirectInput7::EnumDevices** method, or it can be one of the following predefined GUIDs:

*GUID\_SysKeyboard*

The default system keyboard.

*GUID\_SysMouse*

The default system mouse.

For the preceding GUID values to be valid, your application must define **INITGUID** before all other preprocessor directives at the beginning of the source file, or link to Dxguid.lib.

*lplpDirectInputDevice*

Address of a variable to receive the **IDirectInputDevice** interface pointer if successful.

*pUnkOuter*

Address of the controlling object's **IUnknown** interface for COM aggregation, or NULL if the interface is not aggregated. Most callers pass NULL.

### Return Values

If the method succeeds, the return value is **DI\_OK**.

If the method fails, the return value can be one of the following:

**DIERR\_DEVICENOTREG**

**DIERR\_INVALIDPARAM**

**DIERR\_NOINTERFACE**

DIERR\_NOTINITIALIZED  
DIERR\_OUTOFMEMORY

## Remarks

In C++ the *rguid* parameter must be passed by reference; in C, which does not permit passing by reference, it must be passed by address. The following is an example of a C++ call:

```
lpdi->CreateDevice(GUID_SysKeyboard, &pdev, NULL);
```

The following shows the same call in C:

```
lpdi->lpVtbl->CreateDevice(lpdi, &GUID_SysKeyboard, &pdev, NULL);
```

Calling this method with *pUnkOuter* = NULL is equivalent to creating the object by **CoCreateInstance**(&CLSID\_DirectInputDevice, NULL, CLSCTX\_INPROC\_SERVER, riid, lpplDirectInputDevice) and then initializing it with **Initialize**.

Calling this method with *pUnkOuter* != NULL is equivalent to creating the object by **CoCreateInstance**(&CLSID\_DirectInputDevice, punkOuter, CLSCTX\_INPROC\_SERVER, &IID\_IUnknown, lpplDirectInputDevice). The aggregated object must be initialized manually.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in dinput.h.

**Import Library:** Use dinput.lib.

## IDirectInput7::CreateDeviceEx

The **IDirectInput7::CreateDeviceEx** method creates and initializes an instance of a device based on a given GUID, and obtains an **IDirectInputDevice**, **IDirectInputDevice2**, or **IDirectInputDevice7** interface.

```
HRESULT CreateDeviceEx(
    REFGUID rguid,
    REFIID riid,
    LPVOID *pvOut,
    LPUNKNOWN pUnkOuter
);
```

## Parameters

### *rguid*

Reference to (C++) or address of (C) the instance GUID for the desired input device (see Remarks). The GUID is retrieved through the **IDirectInput7::EnumDevices** method, or it can be one of the following predefined GUIDs:

#### *GUID\_SysKeyboard*

The default system keyboard.

#### *GUID\_SysMouse*

The default system mouse.

For the preceding GUID values to be valid, your application must define **INITGUID** before all other preprocessor directives at the beginning of the source file, or link to **Dxguid.lib**.

### *riid*

Unique identifier for the desired interface. Currently accepted values are **IID\_IDirectInputDevice**, **IID\_IDirectInputDevice2**, and **IID\_IDirectInputDevice7**.

### *pvOut*

Address of a variable to receive the interface pointer if successful.

### *pUnkOuter*

Address of the controlling object's **IUnknown** interface for COM aggregation, or **NULL** if the interface is not aggregated. Most callers pass **NULL**.

## Return Values

If the method succeeds, the return value is **DI\_OK**.

If the method fails, the return value can be one of the following:

**DIERR\_DEVICENOTREG**

**DIERR\_INVALIDPARAM**

**DIERR\_NOINTERFACE**

**DIERR\_NOTINITIALIZED**

**DIERR\_OUTOFMEMORY**

## Remarks

In C++ the *rguid* parameter must be passed by reference; in C, which does not permit passing by-reference, it must be passed by address. The following is an example of a C++ call:

```
lpdi->CreateDevice(GUID_SysKeyboard, &pdev, NULL);
```

The following shows the same call in C:

```
lpdi->lpVtbl->CreateDevice(lpdi, &GUID_SysKeyboard, &pdev, NULL);
```

Most applications will want to pass IID\_IDirectInputDevice7 in the *riid* parameter to obtain the **IDirectInputDevice7** interface.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in dinput.h.

**Import Library:** Use dinput.lib.

## IDirectInput7::EnumDevices

The **IDirectInput7::EnumDevice** method enumerates devices that are either currently attached or could be attached to the computer.

```
HRESULT EnumDevices(
    DWORD dwDevType,
    LPDICALLBACK lpCallback,
    LPCVOID pvRef,
    DWORD dwFlags
);
```

### Parameters

*dwDevType*

Device type filter. If this parameter is 0, all device types are enumerated. Otherwise, it is a DIDEVTYPE\_\* value (see **DIDEVICEINSTANCE**), indicating the device type that should be enumerated.

*lpCallback*

Address of a callback function to be called with a description of each DirectInput device.

*pvRef*

Application-defined 32-bit value to be passed to the enumeration callback each time it is called.

*dwFlags*

Flag value that specifies the scope of the enumeration. This parameter can be one or more of the following values:

**DIEDFL\_ALLDEVICES**

All installed devices are enumerated. This is the default behavior.

**DIEDFL\_ATTACHEDONLY**

Only attached and installed devices.

**DIEDFL\_FORCEFEEDBACK**



Only devices that support force feedback.

DIEDFL\_INCLUDEALIASES

Include devices that are aliases for other devices.

DIEDFL\_INCLUDEPHANTOMS

Include phantom (placeholder) devices.

## Return Values

If the method succeeds, the return value is `DI_OK`.

If the method fails, the return value can be one of the following error values:

`DIERR_INVALIDPARAM`

`DIERR_NOTINITIALIZED`

## Remarks

All installed devices can be enumerated, even if they are not present. For example, a flight stick might be installed on the system, but not currently plugged into the computer. Set the *dwFlags* parameter to indicate whether only attached or all installed devices should be enumerated. If the `DIEDFL_ATTACHEDONLY` flag is not present, all installed devices are enumerated.

A preferred device type can be passed as a *dwDevType* filter so that only the devices of that type are enumerated.

The *lpCallback* parameter specifies the address of a callback function of the type documented as **DIEnumDevicesCallback**. DirectInput calls this function for every device that is enumerated. In the callback, the device type and friendly name, and the product GUID and friendly name, are given for each device. If a single input device can function as more than one DirectInput device type, it is returned for each device type that it supports. For example, a keyboard with a built-in mouse is enumerated as a keyboard and as a mouse. The product GUID would be the same for each device, however.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

**Import Library:** Use `dinput.lib`.

## IDirectInput7::FindDevice

The **IDirectInput7::FindDeviceStatus** method retrieves the instance GUID of a device that has been newly attached to the system. It is called in response to a Win32 device management notification.

```
HRESULT FindDevice(  
    REFGUID rguidClass,  
    LPCTSTR ptszName,  
    LPGUID pguidInstance  
);
```

## Parameters

*rguidClass*

Unique identifier of the device class for the device that the application is to locate. The application obtains the class GUID from the device arrival notification. (See the documentation for the DBT\_DEVICEARRIVAL event in the Platform SDK.)

*ptszName*

Name of the device. The application obtains the name from the device arrival notification.

*pguidInstance*

Address of a variable to receive the instance GUID for the device, if the device is found. This value can be passed to **IDirectInput7::CreateDeviceEx**.

## Return Values

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

If the method fails, the return value can be DIERR\_DEVICENOTREG. Failure results if the GUID and name do not correspond to a device class that is registered with DirectInput. For example, they might refer to a storage device, rather than an input device.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in dinput.h.

**Import Library:** Use dinput.lib.

## See Also

Recognizing Device Changes

## IDirectInput7::GetDeviceStatus

The **IDirectInput7::GetDeviceStatus** method retrieves the status of a specified device.

```
HRESULT GetDeviceStatus(  

```

```
REFGUID rguidInstance
);
```

## Parameters

*rguidInstance*  
Instance identifier of the device whose status is being checked.

## Return Values

If the method succeeds, the return value is `DI_OK` if the device is attached to the system, or `DI_NOTATTACHED` otherwise.

If the method fails, the return value can be one of the following error values:

```
DIERR_GENERIC
DIERR_INVALIDPARAM
DIERR_NOTINITIALIZED
```

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

**Import Library:** Use `dinput.lib`.

## IDirectInput7::Initialize

The **IDirectInput7::Initialize** method initializes a DirectInput object. The **DirectInputCreate** and **DirectInputCreateEx** functions automatically initialize the DirectInput object after creating it. Applications normally do not need to call this method.

```
HRESULT Initialize(
    HINSTANCE hinst,
    DWORD dwVersion
);
```

## Parameters

*hinst*  
Instance handle to the application or dynamic-link library (DLL) that is creating the DirectInput object. DirectInput uses this value to determine whether the application or DLL has been certified and to establish any special behaviors that might be necessary for backwards compatibility.

It is an error for a DLL to pass the handle of the parent application. For example, an ActiveX control embedded in a Web page that uses DirectInput must pass its own instance handle, and not the handle of the Web browser. This ensures that DirectInput recognizes the control and can enable any special behaviors that might be necessary.

#### *dwVersion*

Version number of DirectInput for which the application is designed. This value is normally `DIRECTINPUT_VERSION`. Passing the version number of a previous version causes DirectInput to emulate that version. For more information, see *Designing for Previous Versions of DirectInput*.

## Return Values

If the method succeeds, the return value is `DI_OK`.

If the method fails, the return value can be one of the following error values:

`DIERR_BETADIRECTINPUTVERSION`  
`DIERR_OLDDIRECTINPUTVERSION`

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

**Import Library:** Use `dinput.lib`.

## IDirectInput7::RunControlPanel

The **IDirectInput7::RunControlPanel** method runs Control Panel to allow the user to install a new input device or modify configurations.

```
HRESULT RunControlPanel(  
    HWND hwndOwner,  
    DWORD dwFlags  
);
```

## Parameters

#### *hwndOwner*

Handle of the window to be used as the parent window for the subsequent user interface. If this parameter is `NULL`, no parent window is used.

#### *dwFlags*

Currently not used and must be set to 0.

## Return Values

If the method succeeds, the return value is `DI_OK`.

If the method fails, the return value can be one of the following error values:

`DIERR_INVALIDPARAM`  
`DIERR_NOTINITIALIZED`

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

**Import Library:** Use `dinput.lib`.

## See Also

`IDirectInputDevice7::RunControlPanel`

# IDirectInputDevice7

Applications use the methods of the **IDirectInputDevice7** interface to gain and release access to DirectInput devices, manage device properties and information, set behavior, perform initialization, create and play force-feedback effects, and invoke a device's control panel.

The **IDirectInputDevice7** interface is obtained by using the **IDirectInput7::CreateDeviceEx** method. For an example, see [Creating a DirectInput Device](#).

**IDirectInputDevice7** supersedes the **IDirectInputDevice** and **IDirectInputDevice2** interfaces used in previous versions of DirectX.

The methods of the **IDirectInputDevice7** interface can be organized into the following groups.

Accessing input devices	<b>Acquire</b> <b>Unacquire</b>
Device information	<b>GetCapabilities</b> <b>GetDeviceData</b> <b>GetDeviceInfo</b> <b>GetDeviceState</b> <b>Poll</b> <b>SetDataFormat</b> <b>SetEventNotification</b>

---

<b>Device objects</b>	<b>EnumObjects</b>
	<b>GetObjectInfo</b>
<b>Device properties</b>	<b>GetProperty</b>
	<b>SetCooperativeLevel</b>
	<b>SetProperty</b>
<b>Force feedback</b>	<b>CreateEffect</b>
	<b>EnumCreatedEffectObjects</b>
	<b>EnumEffects</b>
	<b>EnumEffectsInFile</b>
	<b>Escape</b>
	<b>GetEffectInfo</b>
	<b>GetForceFeedbackState</b>
	<b>SendForceFeedbackCommand</b>
	<b>WriteEffectToFile</b>
<b>Miscellaneous</b>	<b>Initialize</b>
	<b>RunControlPanel</b>
	<b>SendDeviceData</b>

The **IDirectInputDevice7** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

<b>IUnknown</b>	<b>AddRef</b>
	<b>QueryInterface</b>
	<b>Release</b>

The **LPDIRECTINPUTDEVICE7** type is defined as a pointer to the **IDirectInputDevice** interface:

```
typedef struct IDirectInputDevice7 *LPDIRECTINPUTDEVICE7;
```

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in dinput.h.

**Import Library:** Use dinput.lib.

## See Also

**IDirectInputDevice7**

---

## IDirectInputDevice7::Acquire

The **IDirectInputDevice7::Acquire** method obtains access to the input device.

**HRESULT** Acquire();

### Parameters

None.

### Return Values

If the method succeeds, the return value is **DI\_OK** or **S\_FALSE**.

If the method fails, the return value can be one of the following error values:

**DIERR\_INVALIDPARAM**  
**DIERR\_NOTINITIALIZED**  
**DIERR\_OTHERAPPHASPRIO**

If the method returns **S\_FALSE**, the device has already been acquired.

### Remarks

Before a device can be acquired, a data format must be set by using the **IDirectInputDevice7::SetDataFormat** method.

Devices must be acquired before calling the **IDirectInputDevice7::GetDeviceState** or **IDirectInputDevice7::GetDeviceData** methods for that device.

Device acquisition does not use a reference count. Therefore, if an application calls the **IDirectInputDevice7::Acquire** method twice, then calls the **IDirectInputDevice7::Unacquire** method once, the device is unacquired.

### Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

**Import Library:** Use `dinput.lib`.

## IDirectInputDevice7::CreateEffect

The **IDirectInputDevice7::CreateEffect** method creates and initializes an instance of an effect identified by the effect GUID.

**HRESULT** CreateEffect(  
    **REFGUID** rguid,

```

LPCDIEFFECT lpeff,
LPDIRECTINPUTEFFECT * ppdeff,
LPUNKNOWN punkOuter
);

```

## Parameters

*rguid*

Identity of the effect to be created. This can be a predefined effect GUID, or it can be a GUID obtained from **IDirectInputDevice7::EnumEffects**.

The following effect GUIDs are defined:

- GUID\_ConstantForce
- GUID\_RampForce
- GUID\_Square
- GUID\_Sine
- GUID\_Triangle
- GUID\_SawtoothUp
- GUID\_SawtoothDown
- GUID\_Spring
- GUID\_Damper
- GUID\_Inertia
- GUID\_Friction
- GUID\_CustomForce

*lpeff*

**DIEFFECT** structure that provides parameters for the created effect. This parameter is optional. If it is NULL, the effect object is created without parameters. The application must then call the **IDirectInputEffect::SetParameters** method to set the parameters of the effect before it can download the effect.

*ppdeff*

Address of a variable to receive a pointer to the **IDirectInputEffect** interface if successful.

*punkOuter*

Controlling unknown for COM aggregation. The value is NULL if the interface is not aggregated. Most callers pass NULL.

## Return Values

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

If the method fails, the return value can be one of the following error values:



DIERR\_DEVICENOTREG  
 DIERR\_DEVICEFULL  
 DIERR\_INVALIDPARAM  
 DIERR\_NOTINITIALIZED

If the return value is S\_OK, the effect was created, and the parameters of the effect were updated, but the effect was not necessarily downloaded. For it to be downloaded, the device must be acquired in exclusive mode.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in dinput.h.

**Import Library:** Use dinput.lib.

## IDirectInputDevice7::EnumCreatedEffect Objects

The **IDirectInputDevice7::EnumCreatedEffectObjects** method enumerates all the currently created effects for this device. Effects created by **IDirectInputDevice7::CreateEffect** are enumerated.

```
HRESULT EnumCreatedEffectObjects(  

LPDIENUMCREATEDEFFECTOBJECTSCALLBACK lpCallback,  

LPVOID pvRef,  

DWORD fl  

);
```

### Parameters

*lpCallback*

Address of an application-defined callback function. DirectInput provides the prototype function **DIEnumCreatedEffectObjectsCallback**.

*pvRef*

Reference data (context) for callback.

*fl*

No flags are currently defined. This parameter must be 0.

### Return Values

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

If the method fails, the return value can be one of the following error values:

---

DIERR\_INVALIDPARAM  
DIERR\_NOTINITIALIZED

## Remarks

The results are unpredictable if you create or destroy an effect while an enumeration is in progress. However, the callback function can safely release the effect passed to it.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in dinput.h.

**Import Library:** Use dinput.lib.

## IDirectInputDevice7::EnumEffects

The **IDirectInputDevice7::EnumEffects** method enumerates all the effects supported by the force-feedback system on the device. The enumerated GUIDs can represent predefined effects, as well as effects peculiar to the device manufacturer.

```
HRESULT EnumEffects(  
    LPDIENUMEFFECTSCALLBACK lpCallback,  
    LPVOID pvRef,  
    DWORD dwEffType  
);
```

## Parameters

*lpCallback*

Address of an application-defined callback function. The declaration of this function must conform to that of the **DIEnumEffectsCallback** prototype.

*pvRef*

A 32-bit application-defined value to be passed to the callback function. This parameter can be any 32-bit type; it is declared as **LPVOID** for convenience.

*dwEffType*

Effect type filter. Use one of the **DIEFT\_\*** values to indicate the effect type to be enumerated, or **DIEFT\_ALL** to enumerate all effect types. For a list of these values, see **DIEFFECTINFO**.

## Return Values

If the method succeeds, the return value is **DI\_OK**.

If the method fails, the return value can be one of the following error values:

DIERR\_INVALIDPARAM  
DIERR\_NOTINITIALIZED

If the callback stops the enumeration prematurely, the enumeration is considered to have succeeded.

## Remarks

An application can use the **dwEffType** member of the **DIEFFECTINFO** structure to obtain general information about the effect, such as its type and which envelope and condition parameters are supported by the effect.

To exploit an effect to its fullest, contact the device manufacturer to obtain information on the semantics of the effect and its effect-specific parameters.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

**Import Library:** Use `dinput.lib`.

## IDirectInputDevice7::EnumEffectsInFile

The **IDirectInputDevice7::EnumEffectsInFile** method enumerates all the effects in a file created by the Force Editor utility or another application using the same file format.

```
HRESULT EnumEffectsInFile(
    LPCSTR lpszFileName,
    LPNUMEFFECTSINFILECALLBACK pec,
    LPVOID pvRef,
    DWORD dwFlags
);
```

## Parameters

*lpszFileName*  
Name of the RIFF file.

*pec*  
Address of an application-defined callback function. The declaration of this function must conform to that of the **DIEnumEffectsInFileCallback** prototype.

*pvRef*

Application-defined value to be passed to the callback function. This parameter can be any 32-bit type.

#### *dwFlags*

Can be DIFEF\_DEFAULT ( = 0) or one or both of the following values:

#### DIFEF\_INCLUDENONSTANDARD

Include effect types that are not defined by DirectInput.

#### DIFEF\_MODIFYIFNEEDED

Instruct DirectInput to modify the authored effect, if necessary, so that it plays on the current device. For example, by default, an effect authored for two axes does not play on a single-axis device. Setting this flag allows the effect to play on a single axis. The parameters are modified in the **DIEFFECT** structure pointed to by the **lpDiEffect** member of the **DIFILEEFFECT** structure passed to the callback.

## Return Values

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

If the method fails, the return value can be one of the following error values:

DIERR\_INVALIDPARAM

DIERR\_NOTINITIALIZED

If the callback stops the enumeration prematurely, the enumeration is considered to have succeeded.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in dinput.h.

**Import Library:** Use dinput.lib.

## See Also

**IDirectInputDevice7::WriteEffectToFile**, Loading Effects from a File

## IDirectInputDevice7::EnumObjects

The **IDirectInputDevice7::EnumObjects** method enumerates the input and force-feedback objects available on a device.

```
HRESULT EnumObjects(
    LPDIENUMDEVICEOBJECTSCALLBACK lpCallback,
    LPVOID pvRef,
    DWORD dwFlags
```

);

## Parameters

### *lpCallback*

Address of a callback function that receives DirectInputDevice objects. DirectInput provides a prototype of this function as **DIDEnumDeviceObjectsCallback**.

### *pvRef*

Reference data (context) for callback.

### *dwFlags*

Flags specifying the types of object to be enumerated. Each of the following values restricts the enumeration to objects of the described type:

**DIDFT\_ABSAXIS**

An absolute axis.

**DIDFT\_ALL**

All objects.

**DIDFT\_AXIS**

An axis, either absolute or relative.

**DIDFT\_BUTTON**

A push button or a toggle button.

**DIDFT\_COLLECTION**

An HID link collection. HID link collections do not generate data of their own.

**DIDFT\_ENUMCOLLECTION(*n*)**

An object that belongs to HID link collection number *n*.

**DIDFT\_FFACTUATOR**

An object that contains a force-feedback actuator. In other words, forces can be applied to this object.

**DIDFT\_FFEFFECTTRIGGER**

An object that can be used to trigger force-feedback effects.

**DIDFT\_NOCOLLECTION**

An object that does not belong to any HID link collection; in other words, an object for which the **wCollectionNumber** member of the **DIDDEVICEOBJECTINSTANCE** structure is 0.

**DIDFT\_NODATA**

An object that does not generate data.

**DIDFT\_OUTPUT**

An object to which data can be sent by using the **IDirectInputDevice7::SendDeviceData** method.

**DIDFT\_POV**

A point-of-view controller.

**DIDFT\_PSHBUTTON**

A push button. A push button is reported as down when the user presses it and as up when the user releases it.

DIDFT\_RELAXIS

A relative axis.

DIDFT\_TGLBUTTON

A toggle button. A toggle button is reported as down when the user presses it and remains so until the user presses the button a second time.

DIDFT\_VENDORDEFINED

An object of a type defined by the manufacturer.

## Return Values

If the method succeeds, the return value is `DI_OK`.

If the method fails, the return value can be one of the following error values:

`DIERR_INVALIDPARAM`

`DIERR_NOTINITIALIZED`

## Remarks

The `DIDFT_FFACTUATOR` and `DIDFT_FFEFFECTTRIGGER` flags in the *dwFlags* parameter restrict enumeration to objects that meet all the criteria defined by the included flags. For all the other flags, an object is enumerated if it meets the criterion defined by any included flag in this category. For example, (`DIDFT_FFACTUATOR` | `DIDFT_FFEFFECTTRIGGER`) restricts enumeration to force-feedback trigger objects, and (`DIDFT_FFEFFECTTRIGGER` | `DIDFT_TGLBUTTON` | `DIDFT_PSHBUTTON`) restricts enumeration to buttons of any kind that can be used as effect triggers.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

**Import Library:** Use `dinput.lib`.

## IDirectInputDevice7::Escape

The **IDirectInputDevice7::Escape** method sends a hardware-specific command to the force-feedback driver.

```
HRESULT Escape(
    LPDIEFFESCAPE pesc
);
```

## Parameters

*pesc*

**DIEFFESCAPE** structure that describes the command to be sent. On success, the **cbOutBuffer** member contains the number of bytes of the output buffer actually used.

## Return Values

If the method succeeds, the return value is **DI\_OK**.

If the method fails, the return value can be one of the following error values:

**DIERR\_DEVICEFULL**  
**DIERR\_NOTINITIALIZED**

Other device-specific error codes are also possible. Ask the hardware manufacturer for details.

## Remarks

Since each driver implements different escapes, it is the application's responsibility to ensure that it is sending the escape to the correct driver by comparing the value of the **guidFFDriver** member of the **DIDEVICEINSTANCE** structure against the value the application is expecting.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in **dinput.h**.

**Import Library:** Use **dinput.lib**.

## IDirectInputDevice7::GetCapabilities

The **IDirectInputDevice7::GetCapabilities** method obtains the capabilities of the **DirectInputDevice** object.

```
HRESULT GetCapabilities(  
    LPDIDEVCAPS lpDIDevCaps  
);
```

## Parameters

*lpDIDevCaps*

Address of a **DIDEVCAPS** structure to be filled with the device capabilities. The **dwSize** member of this structure must be initialized before calling this method.

## Return Values

If the method succeeds, the return value is `DI_OK`.

If the method fails, the return value can be one of the following error values:

`DIERR_INVALIDPARAM`  
`DIERR_NOTINITIALIZED`

## Remarks

For compatibility with DirectX 3, it is also valid to pass a **DIDEVCAPS\_DX3** structure with the **dwSize** member initialized to **sizeof(DIDEVCAPS\_DX3)**. For more information, see *Designing for Previous Versions of DirectInput*.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

**Import Library:** Use `dinput.lib`.

## IDirectInputDevice7::GetDeviceData

The **IDirectInputDevice7::GetDeviceData** method retrieves buffered data from the device.

```
HRESULT GetDeviceData(  
    DWORD cbObjectData,  
    LPDIDEVICEOBJECTDATA rgdod,  
    LPDWORD pdwInOut,  
    DWORD dwFlags  
);
```

## Parameters

*cbObjectData*

Size of the **DIDEVICEOBJECTDATA** structure, in bytes.

*rgdod*

Array of **DIDEVICEOBJECTDATA** structures to receive the buffered data.

The number of elements in this array must be equal to the value of the *pdwInOut* parameter. If this parameter is `NULL`, the buffered data is not stored anywhere, but all other side-effects take place.

*pdwInOut*



On entry, the number of elements in the array pointed to by the *rgdod* parameter.  
On exit, the number of elements actually obtained.

#### *dwFlags*

Flags that control the manner in which data is obtained. This value can be 0 or the following flag:

#### DIGDD\_PEEK

Do not remove the items from the buffer. A subsequent **IDirectInputDevice7::GetDeviceData** call will read the same data.  
Normally, data is removed from the buffer after it is read.

## Return Values

If the method succeeds, the return value is DI\_OK or DI\_BUFFEROVERFLOW.

If the method fails, the return value can be one of the following error values:

DIERR\_INPUTLOST  
DIERR\_INVALIDPARAM  
DIERR\_NOTACQUIRED  
DIERR\_NOTBUFFERED  
DIERR\_NOTINITIALIZED

## Remarks

Before device data can be obtained, you must set the data format by using the **IDirectInputDevice7::SetDataFormat** method, set the buffer size with the **IDirectInputDevice7::SetProperty** method, and acquire the device by using the **IDirectInputDevice7::Acquire** method.

The following code example reads up to ten buffered data elements, removing them from the device buffer as they are read.

```
DIDeviceObjectData rgdod[10];
DWORD dwItems = 10;
hres = IDirectInputDevice7_GetDeviceData(
    pdid,
    sizeof(DIDeviceObjectData),
    rgdod,
    &dwItems,
    0);
if (SUCCEEDED(hres)) {
    // dwItems = number of elements read (could be zero)
    if (hres == DI_BUFFEROVERFLOW) {
        // Buffer had overflowed.
    }
}
```

Your application can flush the buffer and retrieve the number of flushed items by specifying NULL for the *rgdod* parameter and a pointer to a variable containing INFINITE for the *pdwInOut* parameter. The following code example illustrates how this can be done:

```
dwItems = INFINITE;
hres = IDirectInputDevice7_GetDeviceData(
    pdid,
    sizeof(DIDeviceObjectData),
    NULL,
    &dwItems,
    0);
if (SUCCEEDED(hres)) {
    // Buffer successfully flushed
    // dwItems = number of elements flushed
    if (hres == DI_BUFFEROVERFLOW) {
        // Buffer had overflowed.
    }
}
```

Your application can query for the number of elements in the device buffer by setting the *rgdod* parameter to NULL, setting *pdwInOut* to INFINITE and setting *dwFlags* to DIGDD\_PEEK. The following code example illustrates how this can be done:

```
dwItems = INFINITE;
hres = IDirectInputDevice7_GetDeviceData(
    pdid,
    sizeof(DIDeviceObjectData),
    NULL,
    &dwItems,
    DIGDD_PEEK);
if (SUCCEEDED(hres)) {
    // dwItems = number of elements in buffer
    if (hres == DI_BUFFEROVERFLOW) {
        // Buffer overflow occurred; not all data
        // was successfully captured.
    }
}
```

To query about whether a buffer overflow has occurred, set the *rgdod* parameter to NULL and the *pdwInOut* parameter to 0. The following code example illustrates how this can be done:

```
dwItems = 0;
hres = IDirectInputDevice7_GetDeviceData(
    pdid,
    sizeof(DIDeviceObjectData),
```

```
        NULL,  
        &dwItems,  
        0);  
if (hres == DI_BUFFEROVERFLOW) {  
    // Buffer overflow occurred  
}
```

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in dinput.h.

**Import Library:** Use dinput.lib.

## See Also

**IDirectInputDevice7::Poll**, Polling and Events

## IDirectInputDevice7::GetDeviceInfo

The **IDirectInputDevice7::GetDeviceInfo** method obtains information about the device's identity.

```
HRESULT GetDeviceInfo(  
    LPDIDEVICEINSTANCE pdidi  
);
```

## Parameters

*pdidi*

Address of a **DIDEVICEINSTANCE** structure to be filled with information about the device's identity. An application must initialize the structure's **dwSize** member before calling this method.

## Return Values

If the method succeeds, the return value is **DI\_OK**.

If the method fails, the return value can be one of the following error values:

```
DIERR_INVALIDPARAM  
DIERR_NOTINITIALIZED
```

## Remarks

For compatibility with DirectX 3, it is also valid to pass a **DIDEVICEINSTANCE\_DX3** structure with the **dwSize** member initialized to **sizeof(DIDEVICEINSTANCE\_DX3)**. For more information, see Designing for Previous Versions of DirectInput.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in dinput.h.

**Import Library:** Use dinput.lib.

## IDirectInputDevice7::GetDeviceState

The **IDirectInputDevice7::GetDeviceState** method retrieves immediate data from the device.

```
HRESULT GetDeviceState(  
    DWORD cbData,  
    LPVOID lpvData  
);
```

## Parameters

*cbData*

Size of the buffer in the *lpvData* parameter, in bytes.

*lpvData*

Address of a structure that receives the current state of the device. The format of the data is established by a prior call to the **IDirectInputDevice7::SetDataFormat** method.

## Return Values

If the method succeeds, the return value is **DI\_OK**.

If the method fails, the return value can be one of the following error values:

```
DIERR_INPUTLOST  
DIERR_INVALIDPARAM  
DIERR_NOTACQUIRED  
DIERR_NOTINITIALIZED  
E_PENDING
```

## Remarks

Before device data can be obtained, set the cooperative level by using the **IDirectInputDevice7::SetCooperativeLevel** method, then set the data format by using **IDirectInputDevice7::SetDataFormat**, and acquire the device by using the **IDirectInputDevice7::Acquire** method.

The five predefined data formats require corresponding device state structures according to the following table:

Data format	State structure
<i>c_dfDIMouse</i>	<b>DIMOUSESTATE</b>
<i>c_dfDIMouse2</i>	<b>DIMOUSESTATE2</b>
<i>c_dfDIKeyboard</i>	array of 256 bytes
<i>c_dfDIJoystick</i>	<b>DIJOYSTATE</b>
<i>c_dfDIJoystick2</i>	<b>DIJOYSTATE2</b>

For example, if you passed the *c\_dfDIMouse* format to the **IDirectInputDevice7::SetDataFormat** method, you must pass a **DIMOUSESTATE** structure to the **IDirectInputDevice7::GetDeviceState** method.

## Requirements

**Windows NT/2000:** Requires Windows 2000.  
**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.  
**Header:** Declared in dinput.h.  
**Import Library:** Use dinput.lib.

## See Also

**IDirectInputDevice7::Poll**, Polling and Events, Buffered and Immediate Data

## IDirectInputDevice7::GetEffectInfo

The **IDirectInputDevice7::GetEffectInfo** method obtains information about an effect.

```
HRESULT GetEffectInfo(
    LPDIEFFECTINFO pdei,
    REFGUID rguid
);
```

## Parameters

*pdei*

**DIEFFECTINFO** structure that receives information about the effect. The caller must initialize the **dwSize** member of the structure before calling this method.

*rguid*

Identifier of the effect for which information is being requested.

## Return Values

If the method succeeds, the return value is **DI\_OK**.

If the method fails, the return value can be one of the following error values:

**DIERR\_DEVICENOTREG**  
**DIERR\_INVALIDPARAM**  
**DIERR\_NOTINITIALIZED**

## Remarks

In C++, the *rguid* parameter must be passed by reference; in C, which does not have pass-by-reference, it must be passed by address. The following is an example of a C++ call:

```
lpdev2->GetEffectInfo(&dei, GUID_Effect);
```

The following shows the same call in C:

```
lpdev2->lpVtbl->GetEffectInfo(lpdev2, &dei, &GUID_Effect);
```

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

**Import Library:** Use `dinput.lib`.

## IDirectInputDevice7::GetForceFeedbackState

The **IDirectInputDevice7::GetForceFeedbackState** method retrieves the state of the device's force-feedback system.

```
HRESULT GetForceFeedbackState(  
    LPDWORD pdwOut  
);
```

---

## Parameters

### *pdwOut*

Location for flags that describe the current state of the device's force-feedback system.

The value is a combination of the following constants:

#### DIGFFS\_ACTUATORSOFF

The device's force-feedback actuators are disabled.

#### DIGFFS\_ACTUATORSON

The device's force-feedback actuators are enabled.

#### DIGFFS\_DEVICELOST

The device suffered an unexpected failure and is in an indeterminate state. It must be reset either by unacquiring and reacquiring the device, or by sending a DISFFC\_RESET command.

#### DIGFFS\_EMPTY

The device has no downloaded effects.

#### DIGFFS\_PAUSED

Playback of all active effects has been paused.

#### DIGFFS\_POWEROFF

The force-feedback system is not currently available. If the device cannot report the power state, neither DIGFFS\_POWERON nor DIGFFS\_POWEROFF is returned.

#### DIGFFS\_POWERON

Power to the force-feedback system is currently available. If the device cannot report the power state, neither DIGFFS\_POWERON nor DIGFFS\_POWEROFF is returned.

#### DIGFFS\_SAFETYSWITCHOFF

The safety switch is currently off, meaning that the device cannot operate. If the device cannot report the state of the safety switch, neither DIGFFS\_SAFETYSWITCHON nor DIGFFS\_SAFETYSWITCHOFF is returned.

#### DIGFFS\_SAFETYSWITCHON

The safety switch is currently on, meaning that the device can operate. If the device cannot report the state of the safety switch, neither DIGFFS\_SAFETYSWITCHON nor DIGFFS\_SAFETYSWITCHOFF is returned.

#### DIGFFS\_STOPPED

No effects are playing, and the device is not paused.

#### DIGFFS\_USERFFSWITCHOFF

The user force-feedback switch is currently off, meaning that the device cannot operate. If the device cannot report the state of the user force-feedback switch, neither DIGFFS\_USERFFSWITCHON nor DIGFFS\_USERFFSWITCHOFF is returned.

#### DIGFFS\_USERFFSWITCHON

The user force-feedback switch is currently on, meaning that the device can operate. If the device cannot report the state of the user force-feedback switch, neither DIGFFS\_USERFFSWITCHON nor DIGFFS\_USERFFSWITCHOFF is returned.

Future versions of DirectInput can define additional flags. Applications should ignore any flags that are not currently defined.

## Return Values

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

If the method fails, the return value can be one of the following error values:

DIERR\_INPUTLOST  
 DIERR\_INVALIDPARAM  
 DIERR\_NOTEXCLUSIVEACQUIRED  
 DIERR\_NOTINITIALIZED  
 DIERR\_UNSUPPORTED

## Remarks

The device must be acquired at the exclusive cooperative level for this method to succeed.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in dinput.h.

**Import Library:** Use dinput.lib.

## IDirectInputDevice7::GetObjectInfo

The **IDirectInputDevice7::GetObjectInfo** method retrieves information about a device object, such as a button or axis.

```
HRESULT GetObjectInfo(
    LPDIDEVICEOBJECTINSTANCE pdidoi,
    DWORD dwObj,
    DWORD dwHow
);
```

## Parameters

*pdidoi*



Address of a **DIDeviceObjectInstance** structure to be filled with information about the object. The structure's **dwSize** member must be initialized before this method is called.

*dwObj*

Value that identifies the object whose information is to be retrieved. The value set for this parameter depends on the value specified in the *dwHow* parameter.

*dwHow*

Value specifying how the *dwObj* parameter should be interpreted. This value can be one of the following:

Value	Meaning
DIPH_BYOFFSET	The <i>dwObj</i> parameter is the offset into the current data format of the object whose information is being accessed.
DIPH_BYID	The <i>dwObj</i> parameter is the object type/instance identifier. This identifier is returned in the <b>dwType</b> member of the <b>DIDeviceObjectInstance</b> structure returned from a previous call to the <b>IDirectInputDevice7::EnumObjects</b> method.
DIPH_BYUSAGE	The <i>dwObj</i> parameter contains the HID Usage Page and Usage values of the object, combined by the <b>DIMAKEUSAGEDWORD</b> macro.

## Return Values

If the method succeeds, the return value is **DI\_OK**.

If the method fails, the return value can be one of the following error values:

**DIERR\_INVALIDPARAM**  
**DIERR\_NOTINITIALIZED**  
**DIERR\_OBJECTNOTFOUND**

## Remarks

For compatibility with DirectX 3, it is also valid to pass a **DIDeviceObjectInstance\_DX3** structure with the **dwSize** member initialized to **sizeof(DIDeviceObjectInstance\_DX3)**. For more information, see Designing for Previous Versions of DirectX.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in **dinput.h**.

**Import Library:** Use **dinput.lib**.

## IDirectInputDevice7::GetProperty

The **IDirectInputDevice7::GetProperty** method retrieves information about the input device.

```
HRESULT GetProperty(  
    REFGUID rguidProp,  
    LPDIPROPHEADER pdiph  
);
```

### Parameters

*rguidProp*

Identifier of the property to be retrieved. This can be one of the predefined values or a pointer to a GUID that identifies the property. The following properties are predefined for an input device:

**DIPROP\_AUTOCENTER**

Specifies whether device objects are self-centering. See **IDirectInputDevice7::SetProperty** for more information.

**DIPROP\_AXISMODE**

Retrieves the axis mode. The retrieved value (**DIPROPAXISMODE\_ABS** or **DIPROPAXISMODE\_REL**) is set in the **dwData** member of the associated **DIPROPDWORD** structure. See the description for the *pdiph* parameter for more information.

**DIPROP\_BUFFERSIZE**

Retrieves the input-buffer size. The retrieved value is set in the **dwData** member of the associated **DIPROPDWORD** structure. See the description for the *pdiph* parameter for more information.

The buffer size determines the amount of data that the buffer can hold between calls to the **IDirectInputDevice7::GetDeviceData** method before data is lost. This value can be set to 0 to indicate that the application does not be reading buffered data from the device. If the buffer size in the **dwData** member of the **DIPROPDWORD** structure is too large to be supported by the device, the largest possible buffer size is set. To determine whether the requested buffer size was set, retrieve the buffer-size property, and compare the result with the value that you previously attempted to set.

**DIPROP\_DEADZONE**

Retrieves a value for the dead zone of a joystick, in the range from 0 through 10,000, where 0 indicates that there is no dead zone, 5,000 indicates that the dead zone extends over 50 percent of the physical range of the axis on both sides of center, and 10,000 indicates that the entire physical range of the axis is dead. When the axis is within the dead zone, it is reported as being at the center of its range.

**DIPROP\_FFGAIN**

Retrieves the gain of the device. See **IDirectInputDevice7::SetProperty** for more information.

**DIPROP\_FFLOAD**

Retrieves the memory load for the device. This setting applies to the entire device, rather than to any particular object, so the **dwHow** member of the associated **DIPROPDWORD** structure must be **DIPH\_DEVICE**.

The **dwData** member contains a value in the range from 0 through 100, indicating the percentage of device memory in use.

**DIPROP\_GRANULARITY**

Retrieves the input granularity. The retrieved value is set in the **dwData** member of the associated **DIPROPDWORD** structure. See the description of the *pdiph* parameter for more information.

Granularity represents the smallest distance over which the object reports movement. Most axis objects have a granularity of one, meaning that all values are possible. Some axes have a larger granularity. For example, the wheel axis on a mouse can have a granularity of 20, meaning that all reported changes in position are multiples of 20. In other words, when the user turns the wheel slowly, the device reports a position of 0, then 20, then 40, and so on.

This is a read-only property; you cannot set its value by calling the **IDirectInputDevice7::SetProperty** method.

**DIPROP\_GUIDANDPATH**

Allows the application to access the class GUID and device interface (path) for the device. This property lets advanced applications perform operations on a HID that are not supported by DirectInput. For more information, see the reference for the **DIPROPGUIDANDPATH** structure.

**DIPROP\_INSTANCENAME**

Retrieves the friendly instance name of the device. For more information, see **IDirectInputDevice7::SetProperty**.

**DIPROP\_PRODUCTNAME**

Retrieves the friendly product name of the device. For more information, see **IDirectInputDevice7::SetProperty**.

**DIPROP\_RANGE**

Retrieves the range of values an object can possibly report. The retrieved minimum and maximum values are set in the **IMin** and **IMax** members of the associated **DIPROPRANGE** structure. See the description of the *pdiph* parameter for more information.

For some devices, this is a read-only property; you cannot set its value by calling the **IDirectInputDevice7::SetProperty** method.

**DIPROP\_SATURATION**

Retrieves a value for the saturation zones of a joystick, in the range from 0 through 10,000. The saturation level is the point at which the axis is considered to be at its most extreme position. For example, if the saturation level is set to 9,500, the axis reaches the extreme of its range when it has moved 95 percent of the physical distance from its center position (or from the dead zone).

*pdiph*

Address of the **DIPROPHEADER** portion of a larger property-dependent structure that contains the **DIPROPHEADER** structure as a member. When retrieving object range information, this value is the address of the **DIPROPHEADER** structure contained within the **DIPROPRange** structure. For most other properties, this value is the address of the **DIPROPHEADER** structure contained within the **DIPROPDWORD** structure.

## Return Values

If the method succeeds, the return value is **DI\_OK**.

If the method fails, the return value can be one of the following error values:

**DIERR\_INVALIDPARAM**  
**DIERR\_NOTINITIALIZED**  
**DIERR\_OBJECTNOTFOUND**  
**DIERR\_UNSUPPORTED**

## Remarks

The following C example shows how to obtain the value of the **DIPROP\_BUFFERSIZE** property:

```
DIPROPDWORD dipdw; // DIPROPDWORD contains a DIPROPHEADER structure.
HRESULT hr;
dipdw.diph.dwSize    = sizeof(DIPROPDWORD);
dipdw.diph.dwHeaderSize = sizeof(DIPROPHEADER);
dipdw.diph.dwObj      = 0; // device property
dipdw.diph.dwHow      = DIPH_DEVICE;

hr = IDirectInputDevice7_GetProperty(pdid, DIPROP_BUFFERSIZE, &dipdw.diph);
if (SUCCEEDED(hr)) {
    // The dipdw.dwData member contains the buffer size.
}
```

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in **dinput.h**.

**Import Library:** Use **dinput.lib**.

## See Also

**IDirectInputDevice7::SetProperty**

## IDirectInputDevice7::Initialize

The **IDirectInputDevice7::Initialize** method initializes a DirectInputDevice object. The **IDirectInput7::CreateDevice** method automatically initializes a device after creating it; applications normally do not need to call this method.

```
HRESULT Initialize(  
    HINSTANCE hinst,  
    DWORD dwVersion,  
    REFGUID rguid  
);
```

### Parameters

*hinst*

Instance handle to the application or DLL that is creating the DirectInput device object. DirectInput uses this value to determine whether the application or DLL has been certified and to establish any special behaviors that might be necessary for backward compatibility.

It is an error for a DLL to pass the handle to the parent application. For example, an ActiveX control embedded in a Web page that uses DirectInput must pass its own instance handle, and not the handle to the Web browser. This ensures that DirectInput recognizes the control and can enable any special behaviors that may be necessary.

*dwVersion*

Version number of DirectInput for which the application is designed. This value is normally **DIRECTINPUT\_VERSION**. Passing the version number of a previous version causes DirectInput to emulate that version. For more information, see Designing for Previous Versions of DirectInput.

*rguid*

Identifier for the instance of the device with which the interface should be associated. The **IDirectInput7::EnumDevices** method can be used to determine which instance GUIDs are supported by the system.

### Return Values

If the method succeeds, the return value is **DI\_OK** or **S\_FALSE**.

If the method fails, the return value can be one of the following error values:

```
DIERR_ACQUIRED  
DIERR_DEVICENOTREG
```

If the method returns **S\_FALSE**, the device had already been initialized with the instance GUID passed in though *rGUID*.

## Remarks

If this method fails, the underlying object should be considered to be in an indeterminate state and must be reinitialized before use.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in dinput.h.

**Import Library:** Use dinput.lib.

## IDirectInputDevice7::Poll

The **IDirectInputDevice7::Poll** method retrieves data from polled objects on a DirectInput device. If the device does not require polling, calling this method has no effect. If a device that requires polling is not polled periodically, no new data is received from the device. Calling this method causes DirectInput to update the device state, generate input events (if buffered data is enabled), and set notification events (if notification is enabled).

**HRESULT Poll()**

## Parameters

None.

## Return Values

If the method succeeds, the return value is **DI\_OK**, or **DI\_NOEFFECT** if the device does not require polling.

If the method fails, the return value can be one of the following error values:

**DIERR\_INPUTLOST**  
**DIERR\_NOTACQUIRED**  
**DIERR\_NOTINITIALIZED**

## Remarks

Before a device data can be polled, the data format must be set by using the **IDirectInputDevice7::SetDataFormat** method, and the device must be acquired by using the **IDirectInputDevice7::Acquire** method.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in dinput.h.

**Import Library:** Use dinput.lib.

## See Also

Polling and Events

## IDirectInputDevice7::RunControlPanel

The **IDirectInputDevice7::RunControlPanel** method runs the DirectInput control panel associated with this device. If the device does not have a control panel associated with it, the default device control panel is launched.

```
HRESULT RunControlPanel(  
    HWND hwndOwner,  
    DWORD dwFlags  
);
```

## Parameters

*hwndOwner*

Parent window handle. If this parameter is NULL, no parent window is used.

*dwFlags*

Not currently used. Zero is the only valid value.

## Return Values

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

If the method fails, the return value can be one of the following error values:

```
DIERR_INVALIDPARAM  
DIERR_NOTINITIALIZED
```

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in dinput.h.

**Import Library:** Use dinput.lib.

## IDirectInputDevice7::SendDeviceData

The **IDirectInputDevice7::SendDeviceData** method sends data to a device that accepts output. The device must be in an acquired state.

```
HRESULT SendDeviceData(
    DWORD cbObjectData,
    LPCDIDeviceObjectData rgdod,
    LPDWORD pdwInOut,
    DWORD fl
);
```

### Parameters

*cbObjectData*

Size, in bytes, of a single **DIDeviceObjectData** structure.

*rgdod*

Array of **DIDeviceObjectData** structures containing the data to send to the device. It must consist of *\*pdwInOut* elements.

The **dwOfs** field of each **DIDeviceObjectData** structure must contain the device object identifier (as obtained from the **dwType** field of the **DIDeviceObjectInstance** structure) for the device object to which the data is directed. The **dwTimeStamp** and **dwSequence** members must be 0.

*pdwInOut*

On entry, the variable pointed to by this parameter contains the number of elements in the array pointed to by *rgdod*. On exit, it contains the number of elements sent to the device.

*fl*

Flags controlling the manner in which data is sent. This can be 0 or the following value:

**DISDD\_CONTINUE**

The device data sent is overlaid on the previously sent device data. See Remarks.

### Return Values

If the method succeeds, the return value is **DI\_OK**.

If the method fails, the return value can be one of the following error values:

**DIERR\_INPUTLOST**

**DIERR\_NOTACQUIRED**

**DIERR\_REPORTFULL**

**DIERR\_UNPLUGGED**



## Remarks

There is no guarantee that the individual data elements will be sent in a particular order. However, data sent by successive calls to **IDirectInputDevice7::SendDeviceData** is not interleaved. Furthermore, if multiple pieces of data are sent to the same object with a single call, it is unspecified which piece of data is sent.

Consider, for example, a device that can be sent data in packets, each packet describing two pieces of information; call them A and B. Suppose the application attempts to send three data elements: B = 2, A = 1, and B = 0.

The actual device is sent a single packet. The A field of the packet contains the value 1, and the B field of the packet is either 2 or 0.

If the data must to be sent to the device exactly as specified, three calls to **IDirectInputDevice7::SendDeviceData** should be performed, each call sending one data element.

In response to the first call, the device is sent a packet in which the A field is blank and the B field contains the value 2.

In response to the second call, the device is sent a packet in which the A field contains the value 1, and the B field is blank.

Finally, in response to the third call, the device is sent a packet in which the A field is blank and the B field contains the value 0.

If the `DISDD_CONTINUE` flag is set, the device data sent is overlaid on the previously sent device data. Otherwise, the device data sent starts from scratch.

For example, suppose a device supports two button outputs, Button0 and Button1. If an application first calls **IDirectInputDevice7::SendDeviceData** passing "Button0 pressed", a packet of the form "Button0 pressed, Button1 not pressed" is sent to the device. If the application then makes another call, passing "Button1 pressed" and the `DISDD_CONTINUE` flag, a packet of the form "Button0 pressed, Button1 pressed" is sent to the device. However, if the application had not passed the `DISDD_CONTINUE` flag, the packet sent to the device would have been "Button0 not pressed, Button1 pressed".

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

**Import Library:** Use `dinput.lib`.

## IDirectInputDevice7::SendForceFeedbackCommand

The **IDirectInputDevice7::SendForceFeedbackCommand** method sends a command to the device's force-feedback system.

```
HRESULT SendForceFeedbackCommand(  
    DWORD dwFlags  
);
```

### Parameters

*dwFlags*

Single value indicating the desired change in state. The value can be one of the following:

**DISFFC\_CONTINUE**

Paused playback of all active effects is to be continued. It is an error to send this command when the device is not in a paused state.

**DISFFC\_PAUSE**

Playback of all active effects is to be paused. This command also stops the clock-on effects so that they continue playing to their full duration when restarted.

While the device is paused, new effects cannot be started, and existing ones cannot be modified. Doing so can cause the subsequent **DISFFC\_CONTINUE** command to fail to perform properly.

To abandon a pause and stop all effects, use the **DISFFC\_STOPALL** or **DISFCC\_RESET** commands.

**DISFFC\_RESET**

The device's force-feedback system is to be put in its startup state. All effects are removed from the device, are no longer valid, and must be recreated if they are to be used again. The device's actuators are disabled.

**DISFFC\_SETACTUATORSOFF**

The device's force-feedback actuators are to be disabled. While the actuators are off, effects continue to play but are ignored by the device. Using the analogy of a sound playback device, they are muted, rather than paused.

**DISFFC\_SETACTUATORSON**

The device's force-feedback actuators are to be enabled.

**DISFFC\_STOPALL**

Playback of any active effects is to be stopped. All active effects are reset, but are still being maintained by the device and are still valid. If the device is in a paused state, that state is lost.

This command is equivalent to calling the **IDirectInputEffect::Stop** method for each effect playing.

## Return Values

If the method succeeds, the return value is `DI_OK`.

If the method fails, the return value can be one of the following error values:

`DIERR_INPUTLOST`  
`DIERR_INVALIDPARAM`  
`DIERR_NOTEXCLUSIVEACQUIRED`  
`DIERR_NOTINITIALIZED`  
`DIERR_UNSUPPORTED`

## Remarks

The device must be acquired at the exclusive cooperative level for this method to succeed.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

**Import Library:** Use `dinput.lib`.

## IDirectInputDevice7::SetCooperativeLevel

The **IDirectInputDevice7::SetCooperativeLevel** method establishes the cooperative level for this instance of the device. The cooperative level determines how this instance of the device interacts with other instances of the device and the rest of the system.

```
HRESULT SetCooperativeLevel(
    HWND hwnd,
    DWORD dwFlags
);
```

## Parameters

*hwnd*

Window handle to be associated with the device. This parameter must be a valid top-level window handle that belongs to the process. The window associated with the device must not be destroyed while it is still active in a DirectInput device.

*dwFlags*

Flags that describe the cooperative level associated with the device. The following flags are defined:

**DISCL\_BACKGROUND**

The application requires background access. If background access is granted, the device can be acquired at any time, even when the associated window is not the active window.

**DISCL\_EXCLUSIVE**

The application requires exclusive access. If exclusive access is granted, no other instance of the device can obtain exclusive access to the device while it is acquired. However, nonexclusive access to the device is always permitted, even if another application has obtained exclusive access.

An application that acquires the mouse or keyboard device in exclusive mode should always unacquire the devices when it receives

WM\_ENTERSIZEMOVE and WM\_ENTERMENULOOP messages.

Otherwise, the user cannot manipulate the menu or move and resize the window.

**DISCL\_FOREGROUND**

The application requires foreground access. If foreground access is granted, the device is automatically unacquired when the associated window moves to the background.

**DISCL\_NONEXCLUSIVE**

The application requires nonexclusive access. Access to the device does not interfere with other applications that are accessing the same device.

**DISCL\_NOWINKEY**

Disable the Windows key. Setting this flag ensures that the user cannot inadvertently break out of the application

Applications must specify either DISCL\_FOREGROUND or DISCL\_BACKGROUND; it is an error to specify both or neither. Similarly, applications must specify either DISCL\_EXCLUSIVE or DISCL\_NONEXCLUSIVE.

## Return Values

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

If the method fails, the return value can be one of the following error values:

DIERR\_INVALIDPARAM

DIERR\_NOTINITIALIZED

E\_HANDLE

## Remarks

If the system mouse is acquired in exclusive mode, the pointer is removed from the screen until the device is unacquired.

Applications must call this method before acquiring the device by using the **IDirectInputDevice7::Acquire** method.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

**Import Library:** Use `dinput.lib`.

## See Also

Cooperative Levels

## IDirectInputDevice7::SetDataFormat

The **IDirectInputDevice7::SetDataFormat** method sets the data format for the DirectInput device.

```
HRESULT SetDataFormat(  
    LPCDIDATAFORMAT lpdf  
);
```

## Parameters

*lpdf*

Address of a structure that describes the format of the data that the DirectInputDevice should return. An application can define its own **DIDATAFORMAT** structure or use one of the following predefined global variables:

- *c\_dfDIKeyboard*
- *c\_dfDIMouse*
- *c\_dfDIMouse2*
- *c\_dfDIJoystick*
- *c\_dfDIJoystick2*

## Return Values

If the method succeeds, the return value is `DI_OK`.

If the method fails, the return value can be one of the following error values:

```
DIERR_ACQUIRED  
DIERR_INVALIDPARAM  
DIERR_NOTINITIALIZED
```

## Remarks

The data format must be set before the device can be acquired by using the **IDirectInputDevice7::Acquire** method. It is necessary to set the data format only once. The data format cannot be changed while the device is acquired.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

**Import Library:** Use `dinput.lib`.

## See Also

**IDirectInputDevice7::GetDeviceState**

## IDirectInputDevice7::SetEventNotification

The **IDirectInputDevice7::SetEventNotification** method sets the event notification status. This method specifies an event that is to be set when the device state changes. It is also used to turn off event notification.

```
HRESULT SetEventNotification(  
    HANDLE hEvent  
);
```

## Parameters

*hEvent*

Handle to the event that is to be set when the device state changes. DirectInput uses the Win32 **SetEvent** function on the handle when the state of the device changes. If the *hEvent* parameter is `NULL`, notification is disabled.

The application can create the handle as either a manual-reset or autoreset event by using the Win32 **CreateEvent** function. If the event is created as an autoreset event, the operating system automatically resets the event once a wait has been satisfied. If the event is created as a manual-reset event, it is the application's responsibility to call the Win32 **ResetEvent** function to reset it. DirectInput does not call the Win32 **ResetEvent** function for event notification handles. Most applications create the event as an automatic-reset event.

## Return Values

If the method succeeds, the return value is `DI_OK` or `DI_POLLEDDEVICE`.

If the method fails, the return value can be one of the following error values:

- `DIERR_ACQUIRED`
- `DIERR_HANDLEEXISTS`
- `DIERR_INVALIDPARAM`
- `DIERR_NOTINITIALIZED`

## Remarks

A device state change is defined as any of the following:

- A change in the position of an axis
- A change in the state (pressed or released) of a button
- A change in the direction of a POV control
- Loss of acquisition

Do not call the Win32 **CloseHandle** function on the event while it has been selected into a `DirectInputDevice` object. You must call this method with the *hEvent* parameter set to `NULL` before closing the event handle.

The event notification handle cannot be changed while the device is acquired. If the function is successful, the application can use the event handle like any other Win32 event handle.

The following code example checks whether the handle is currently set without blocking:

```
dwResult = WaitForSingleObject(hEvent, 0);
if (dwResult == WAIT_OBJECT_0) {
    // Event is set. If the event was created as
    // autoreset, it has also been reset.
}
```

The following code example illustrates blocking indefinitely until the event is set. This behavior is strongly discouraged because the thread does not respond to the system until the wait is satisfied. In particular, the thread does not respond to Windows messages.

```
dwResult = WaitForSingleObject(hEvent, INFINITE);
if (dwResult == WAIT_OBJECT_0) {
    // Event has been set. If the event was created
    // as autoreset, it has also been reset.
}
```

The following code example illustrates a typical message loop for a message-based application that uses two events:

```
HANDLE ah[2] = { hEvent1, hEvent2 };

while (TRUE) {

    dwResult = MsgWaitForMultipleObjects(2, ah, FALSE,
        INFINITE, QS_ALLINPUT);
    switch (dwResult) {
    case WAIT_OBJECT_0:
        // Event 1 has been set. If the event was created as
        // autoreset, it has also been reset.
        ProcessInputEvent1();
        break;

    case WAIT_OBJECT_0 + 1:
        // Event 2 has been set. If the event was created as
        // autoreset, it has also been reset.
        ProcessInputEvent2();
        break;

    case WAIT_OBJECT_0 + 2:
        // A Windows message has arrived. Process
        // messages until there aren't any more.
        while(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)){
            if (msg.message == WM_QUIT) {
                goto exitapp;
            }
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        break;

    default:
        // Unexpected error.
        Panic();
        break;
    }
}
```

The following code example illustrates a typical application loop for a non-message-based application that uses two events:

```
HANDLE ah[2] = { hEvent1, hEvent2 };
DWORD dwWait = 0;

while (TRUE) {
```



```
dwResult = MsgWaitForMultipleObjects(2, ah, FALSE,
                                     dwWait, QS_ALLINPUT);
dwWait = 0;

switch (dwResult) {
case WAIT_OBJECT_0:
    // Event 1 has been set. If the event was
    // created as autoreset, it has also
    // been reset.
    ProcessInputEvent1();
    break;

case WAIT_OBJECT_0 + 1:
    // Event 2 has been set. If the event was
    // created as autoreset, it has also
    // been reset.
    ProcessInputEvent2();
    break;

case WAIT_OBJECT_0 + 2:
    // A Windows message has arrived. Process
    // messages until there aren't any more.
    while(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)){
        if (msg.message == WM_QUIT) {
            goto exitapp;
        }
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    break;

default:
    // No input or messages waiting.
    // Do a frame of the game.
    // If the game is idle, tell the next wait
    // to wait indefinitely for input or a message.
    if (!DoGame()) {
        dwWait = INFINITE;
    }
    break;
}
}
```

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in dinput.h.

**Import Library:** Use dinput.lib.

## See Also

Polling and Events

## IDirectInputDevice7::SetProperty

The **IDirectInputDevice7::SetProperty** method sets properties that define the device behavior. These properties include input buffer size and axis mode.

```
HRESULT SetProperty(
    REFGUID rguidProp,
    LPCDIPROPHEADER pdiph
);
```

## Parameters

*rguidProp*

Identifier of the property to be set. This can be one of the predefined values, or a pointer to a GUID that identifies the property. The following property values are predefined for an input device:

**DIPROP\_AUTOCENTER**

Specifies whether device objects are self-centering. This setting applies to the entire device, rather than to any particular object, so the **dwHow** member of the associated **DIPROPDWORD** structure must be **DIPH\_DEVICE**.

The **dwData** member can be one of the following values:

**DIPROPAUTOCENTER\_OFF**: The device should not automatically center when the user releases the device. An application that uses force feedback should disable the auto-centering spring before playing effects.

**DIPROPAUTOCENTER\_ON**: The device should automatically center when the user releases the device. For example, in this mode, a joystick would engage the self-centering spring.

The use of force-feedback effects can interfere with the auto-centering spring. Some devices disable the auto-centering spring when a force-feedback effect is played.

Not all devices support the auto-center property.

**DIPROP\_AXISMODE**

Sets the axis mode. The value being set (**DIPROPAXISMODE\_ABS** or **DIPROPAXISMODE\_REL**) must be specified in the **dwData** member of the

associated **DIPROPDWORD** structure. See the description of the *pdiph* parameter for more information.

This setting applies to the entire device, so the **dwHow** member of the associated **DIPROPDWORD** structure must be set to **DIPH\_DEVICE**.

#### DIPROP\_BUFFERSIZE

Sets the input-buffer size. The value being set must be specified in the **dwData** member of the associated **DIPROPDWORD** structure. See the description of the *pdiph* parameter for more information.

This setting applies to the entire device, so the **dwHow** member of the associated **DIPROPDWORD** structure must be set to **DIPH\_DEVICE**.

#### DIPROP\_CALIBRATIONMODE

Allows the application to specify whether DirectInput should retrieve calibrated or uncalibrated data from an axis. By default, DirectInput retrieves calibrated data.

Setting the calibration mode for the entire device is equivalent to setting it for each axis individually.

The **dwData** member of the **DIPROPDWORD** structure can be one of the following values:

**DIPROPCALIBRATIONMODE\_COOKED**: DirectInput should return data after applying calibration information. This is the default mode.

**DIPROPCALIBRATIONMODE\_RAW**: DirectInput should return raw, uncalibrated data. This mode is typically used only by Control Panel-type applications.

Setting a device into raw mode causes the dead zone, saturation, and range settings to be ignored.

#### DIPROP\_DEADZONE

Sets the value for the dead zone of a joystick, in the range from 0 through 10,000, where 0 indicates that there is no dead zone, 5,000 indicates that the dead zone extends over 50 percent of the physical range of the axis on both sides of center, and 10,000 indicates that the entire physical range of the axis is dead. When the axis is within the dead zone, it is reported as being at the center of its range.

This setting can be applied to either the entire device or to a specific axis.

#### DIPROP\_FFGAIN

Sets the gain for the device. This setting applies to the entire device, rather than to any particular object, so the **dwHow** member of the associated **DIPROPDWORD** structure must be **DIPH\_DEVICE**.

The **dwData** member contains a gain value that is applied to all effects created on the device. The value is an integer in the range from 0 through 10,000, specifying the amount by which effect magnitudes should be scaled for the device. For example, a value of 10,000 indicates that all effect magnitudes are to be taken at face value. A value of 9,000 indicates that all effect magnitudes are to be reduced to 90% of their nominal magnitudes.

Setting a gain value is useful when an application wants to scale down the strength of all force-feedback effects uniformly, based on user preferences.

Unlike other properties, the gain can be set when the device is in an acquired state.

#### DIPROP\_INSTANCENAME

This property exists for advanced applications that want to change the friendly instance name of a device (as returned in the **tszInstanceName** member of the **DIDEVICEINSTANCE** structure) to distinguish it from similar devices that are plugged in simultaneously. Most applications should have no need to change the friendly name.

This setting applies to the entire device, so the **dwHow** member of the associated **DIPROPDWORD** structure must be set to **DIPH\_DEVICE**.

The *pdiph* parameter must be a pointer to the **diph** member of a **DIPROPSTRING** structure.

#### DIPROP\_PRODUCTNAME

This property exists for advanced applications that want to change the friendly product name of a device (as returned in the **tszProductName** member of the **DIDEVICEINSTANCE** structure) to distinguish it from similar devices which are plugged in simultaneously. Most applications should have no need to change the friendly name.

This setting applies to the entire device, so the **dwHow** member of the associated **DIPROPDWORD** structure must be set to **DIPH\_DEVICE**.

The *pdiph* parameter must be a pointer to the **diph** member of a **DIPROPSTRING** structure.

#### DIPROP\_RANGE

Sets the range of values an object can possibly report. The minimum and maximum values are taken from the **IMin** and **IMax** members of the associated **DIPROPDWORD** structure.

For some devices, this is a read-only property.

You cannot set a reverse range; **IMax** must be greater than **IMin**.

#### DIPROP\_SATURATION

Sets the value for the saturation zones of a joystick, in the range from 0 through 10,000. The saturation level is the point at which the axis is considered to be at its most extreme position. For example, if the saturation level is set to 9,500, the axis reaches the extreme of its range when it has moved 95 percent of the physical distance from its center position (or from the dead zone).

This setting can be applied to either the entire device or a specific axis.

*pdiph*

Address of the **DIPROPHEADER** structure contained within the type-specific property structure.

## Return Values

If the method succeeds, the return value is `DI_OK` or `DI_PROPNOEFFECT`.

If the method fails, the return value can be one of the following error values:

- `DIERR_INVALIDPARAM`
- `DIERR_NOTINITIALIZED`
- `DIERR_OBJECTNOTFOUND`
- `DIERR_UNSUPPORTED`

## Remarks

The buffer size determines the amount of data that the buffer can hold between calls to the **IDirectInputDevice7::GetDeviceData** method before data is lost. This value may be set to 0 to indicate that the application does not read buffered data from the device. If the buffer size in the **dwData** member of the **DIPROPDWORD** structure is too large to be supported by the device, the largest possible buffer size is set. To determine whether the requested buffer size was set, retrieve the buffer-size property, and compare the result with the value that you previously attempted to set.

## Requirements

**Windows NT/2000:** Requires Windows 2000.  
**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.  
**Header:** Declared in `dinput.h`.  
**Import Library:** Use `dinput.lib`.

## See Also

**IDirectInputDevice7::GetProperty**

## IDirectInputDevice7::Unacquire

The **IDirectInputDevice7::Unacquire** method releases access to the device.

**HRESULT Unacquire();**

## Parameters

None.

## Return Values

The return value is `DI_OK` if the device was unacquired, or `DI_NOEFFECT` if the device was not in an acquired state.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

**Import Library:** Use `dinput.lib`.

## IDirectInputDevice7::WriteEffectToFile

The **IDirectInputDevice7::WriteEffectToFile** method saves information about one or more force-feedback effects to a file that can be read by using **IDirectInputDevice7::EnumEffectsInFile**. This method is chiefly of interest to those wanting to write their own force-authoring applications.

```
HRESULT WriteEffectToFile(
    LPCSTR lpszFileName,
    DWORD dwEntries,
    LPCDIFILEEFFECT rgDiFileEft,
    DWORD dwFlags
);
```

### Parameters

*lpszFileName*

Name of the RIFF file.

*dwEntries*

Number of structures in the *rgDiFileEft* array.

*rgDiFileEft*

Array of **DIFILEEFFECT** structures.

*dwFlags*

Flags which control how the effect should be written. Can be **DIFEF\_DEFAULT** (= 0) or the following value:

**DIFEF\_INCLUDENONSTANDARD**

Includes effects that are not defined by DirectInput. If this flag is not specified, only effects with GUIDs defined in `Dinput.h`, such as `GUID_ConstantForce`, are written.

### Return Values

If the method succeeds, it returns **DI\_OK**.

If it fails, the return value can be **DIERR\_INVALIDPARAM**.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in dinput.h.

**Import Library:** Use dinput.lib.

## See Also

**IDirectInputDevice7::EnumEffectsInFile**

# IDirectInputEffect

Applications use the methods of the **IDirectInputEffect** interface to manage effects of force-feedback devices.

The interface is obtained by using the **IDirectInputDevice7::CreateEffect** method.

The methods of the **IDirectInputEffect** interface can be organized into the following groups.

<b>Effect information</b>	<b>GetEffectGuid</b>
	<b>GetEffectStatus</b>
	<b>GetParameters</b>
<b>Effect manipulation</b>	<b>Download</b>
	<b>Initialize</b>
	<b>SetParameters</b>
	<b>Start</b>
	<b>Stop</b>
	<b>Unload</b>
<b>Miscellaneous</b>	<b>Escape</b>

The **IDirectInputEffect** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

<b>IUnknown</b>	<b>AddRef</b>
	<b>QueryInterface</b>
	<b>Release</b>

The **LPDIRECTINPUTEFFECT** type is defined as a pointer to the **IDirectInputEffect** interface:

```
typedef struct IDirectInputEffect *LPDIRECTINPUTEFFECT;
```

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in dinput.h.

**Import Library:** Use dinput.lib.

## IDirectInputEffect::Download

The **IDirectInputEffect::Download** method places the effect on the device. If the effect is already on the device, the existing effect is updated to match the values set by the **IDirectInputEffect::SetParameters** method.

**HRESULT Download(void);**

## Parameters

None.

## Return Values

If the method succeeds, the return value is **DI\_OK** or **S\_FALSE**.

If the method fails, the return value can be one of the following error values:

**DIERR\_NOTINITIALIZED**  
**DIERR\_DEVICEFULL**  
**DIERR\_INCOMPLETEEFFECT**  
**DIERR\_INPUTLOST**  
**DIERR\_NOTEXCLUSIVEACQUIRED**  
**DIERR\_INVALIDPARAM**  
**DIERR\_EFFECTPLAYING**

If the method returns **S\_FALSE**, the effect has already been downloaded to the device.

## Remarks

It is valid to update an effect while it is playing. The semantics of such an operation are explained in the reference for **IDirectInputEffect::SetParameters**.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.



**Header:** Declared in dinput.h.

**Import Library:** Use dinput.lib.

## IDirectInputEffect::Escape

The **IDirectInputEffect::Escape** method sends a hardware-specific command to the driver.

```
HRESULT Escape(  
    LPDIEFFESCAPE pesc  
);
```

### Parameters

*pesc*

**DIEFFESCAPE** structure that describes the command to be sent. On success, the **cbOutBuffer** member contains the number of bytes of the output buffer used.

### Return Values

If the method succeeds, the return value is **DI\_OK**.

If the method fails, the return value can be one of the following error values:

**DIERR\_NOTINITIALIZED**  
**DIERR\_DEVICEFULL**

Other device-specific error codes are also possible. Ask the hardware manufacturer for details.

### Remarks

Since each driver implements different escapes, it is the application's responsibility to ensure that it is sending the escape to the correct driver by comparing the value of the **guidFFDriver** member of the **DIDEVICEINSTANCE** structure against the value the application is expecting.

### Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in dinput.h.

**Import Library:** Use dinput.lib.

---

## IDirectInputEffect::GetEffectGuid

The **IDirectInputEffect::GetEffectGuid** method retrieves the GUID for the effect represented by the **IDirectInputEffect** object.

```
HRESULT GetEffectGuid(  
    LPGUID pguid  
);
```

### Parameters

*pguid*  
GUID structure that is filled by the method.

### Return Values

If the method succeeds, the return value is **DI\_OK**.

If the method fails, the return value can be one of the following error values:

**DIERR\_INVALIDPARAM**  
**DIERR\_NOTINITIALIZED**

### Remarks

Additional information about the effect can be obtained by passing the GUID to **IDirectInputDevice7::GetEffectInfo**.

### Requirements

**Windows NT/2000:** Requires Windows 2000.  
**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.  
**Header:** Declared in `dinput.h`.  
**Import Library:** Use `dinput.lib`.

## IDirectInputEffect::GetEffectStatus

The **IDirectInputEffect::GetEffectStatus** method retrieves the status of an effect.

```
HRESULT GetEffectStatus(  
    LPDWORD pdwFlags  
);
```

### Parameters

*pdwFlags*

Status flags for the effect. The value can be 0 or one or more of the following constants:

**DIEGES\_PLAYING**

The effect is playing.

**DIEGES\_EMULATED**

The effect is emulated.

## Return Values

If the method succeeds, the return value is **DI\_OK**.

If the method fails, the return value can be one of the following error values:

**DIERR\_INVALIDPARAM**

**DIERR\_NOTINITIALIZED**

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

**Import Library:** Use `dinput.lib`.

## IDirectInputEffect::GetParameters

The **IDirectInputEffect::GetParameters** method retrieves information about an effect.

```
HRESULT GetParameters(  
    LPDIEFFECT peff,  
    DWORD dwFlags  
);
```

## Parameters

*peff*

Address of a **DIEFFECT** structure that receives effect information. The **dwSize** member must be filled in by the application before calling this method.

*dwFlags*

Flags specifying which parts of the effect information are to be retrieved. The value can be 0 or one or more of the following constants:

**DIEP\_ALLPARAMS**

The union of all other **DIEP\_\*** flags, indicating that all members of the **DIEFFECT** structure are being requested.

**DIEP\_ALLPARAMS\_DX5**

The union of all other DIEP\_\* flags except the DIEP\_STARTDELAY flag.

#### DIEP\_AXES

The **cAxes** and **rgdwAxes** members should receive data. The **cAxes** member on entry contains the size (in **DWORDs**) of the buffer pointed to by the **rgdwAxes** member. If the buffer is too small, the method returns **DIERR\_MOREDATA** and sets **cAxes** to the necessary size of the buffer.

#### DIEP\_DIRECTION

The **cAxes** and **rglDirection** members should receive data. The **cAxes** member on entry contains the size (in **DWORDs**) of the buffer pointed to by the **rglDirection** member. If the buffer is too small, the **GetParameters** method returns **DIERR\_MOREDATA** and sets **cAxes** to the necessary size of the buffer.

The **dwFlags** member must include at least one of the coordinate system flags (**DIEFF\_CARTESIAN**, **DIEFF\_POLAR**, or **DIEFF\_SPHERICAL**).

**DirectInput** returns the direction of the effect in one of the coordinate systems you specified, converting between coordinate systems as necessary. On exit, exactly one of the coordinate system flags is set in the **dwFlags** member, indicating which coordinate system **DirectInput** used. In particular, passing all three coordinate system flags retrieves the coordinates in exactly the same format in which they were set.

#### DIEP\_DURATION

The **dwDuration** member should receive data.

#### DIEP\_ENVELOPE

The **lpEnvelope** member points to a **DIENVELOPE** structure that should receive data. If the effect does not have an envelope associated with it, the **lpEnvelope** member is set to **NULL**.

#### DIEP\_GAIN

The **dwGain** member should receive data.

#### DIEP\_SAMPLEPERIOD

The **dwSamplePeriod** member should receive data.

#### DIEP\_STARTDELAY

The **dwStartDelay** member should receive data.

#### DIEP\_TRIGGERBUTTON

The **dwTriggerButton** member should receive data.

#### DIEP\_TRIGGERREPEATINTERVAL

The **dwTriggerRepeatInterval** member should receive data.

#### DIEP\_TYPESPECIFICPARAMS

The **lpvTypeSpecificParams** member points to a buffer whose size is specified by the **cbTypeSpecificParams** member. On return, the buffer is filled in with the type-specific data associated with the effect, and the **cbTypeSpecificParams** member contains the number of bytes copied. If the buffer supplied by the application is too small to contain all the type-specific data, the method returns **DIERR\_MOREDATA**, and the **cbTypeSpecificParams** member contains the required size of the buffer in bytes.

## Return Values

If the method succeeds, the return value is `DI_OK`.

If the method fails, the return value can be one of the following error values:

`DIERR_INVALIDPARAM`  
`DIERR_MOREDATA`  
`DIERR_NOTINITIALIZED`

## Remarks

Common errors resulting in a `DIERR_INVALIDPARAM` error include not setting the **dwSize** member of the **DIEFFECT** structure, passing invalid flags, or not setting up the members in the **DIEFFECT** structure properly in preparation for receiving the effect information. For example, if information is to be retrieved in the **dwTriggerButton** member, the **dwFlags** member must be set to either `DIEFF_OBJECTIDS` or `DIEFF_OBJECTOFFSETS` so that `DirectInput` knows how to describe the button.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

**Import Library:** Use `dinput.lib`.

## IDirectInputEffect::Initialize

The **IDirectInputEffect::Initialize** method initializes a `DirectInputEffect` object.

```
HRESULT Initialize(  
    HINSTANCE hinst,  
    DWORD dwVersion,  
    REFGUID rguid  
);
```

## Parameters

*hinst*

Instance handle to the application or DLL that is creating the `DirectInputEffect` object. `DirectInput` uses this value to determine whether the application or DLL has been certified and to establish any special behaviors that might be necessary for backward compatibility. It is an error for a DLL to pass the handle to the parent application.

*dwVersion*

Version number of DirectInput for which the application is designed. This value is normally `DIRECTINPUT_VERSION`. Passing the version number of a previous version causes DirectInput to emulate that version. For more information, see *Designing for Previous Versions of DirectInput*.

*rguid*

Identifier of the effect with which the interface is associated. The **IDirectInputDevice7::EnumEffects** method can be used to determine which effect GUIDs are supported by the device.

## Return Values

If the method succeeds, the return value is `DI_OK`.

If the method fails, the return value can be `DIERR_DEVICENOTREG`.

## Remarks

If this method fails, the underlying object should be considered to be an indeterminate state and needs to be reinitialized before it can be subsequently used.

The **IDirectInputDevice7::CreateEffect** method automatically initializes the effect after creating it. Applications normally do not need to call the **Initialize** method.

In C++ the *rguid* parameter must be passed by reference; in C, which does not have pass-by-reference, it must be passed by address. The following is an example of a C++ call:

```
lpEff->Initialize(g_hInstDll, DIRECTINPUT_VERSION, GUID_Effect);
```

The following shows the same call in C:

```
lpEff->lpVtbl->Initialize(lpEff, g_hInstDll,
    DIRECTINPUT_VERSION, &GUID_Effect);
```

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

**Import Library:** Use `dinput.lib`.

## IDirectInputEffect::SetParameters

The **IDirectInputEffect::SetParameters** method sets the characteristics of an effect.

```
HRESULT SetParameters(
    LPCDIEFFECT peff,
```

**DWORD** *dwFlags*

);

## Parameters

*peff*

**DIEFFECT** structure that contains effect information. The **dwSize** member must be filled in by the application before calling this method, as well as any members specified by corresponding bits in the *dwFlags* parameter.

*dwFlags*

Flags specifying which portions of the effect information are to be set and how the downloading of the parameters should be handled. The value can be 0 or one or more of the following constants:

**DIEP\_AXES**

The **cAxes** and **rgdwAxes** members contain data.

**DIEP\_DIRECTION**

The **cAxes** and **rglDirection** members contain data. The **dwFlags** member specifies (with **DIEFF\_CARTESIAN** or **DIEFF\_POLAR**) the coordinate system in which the values should be interpreted.

**DIEP\_DURATION**

The **dwDuration** member contains data.

**DIEP\_ENVELOPE**

The **lpEnvelope** member points to a **DIENVELOPE** structure that contains data. To detach any existing envelope from the effect, pass this flag and set the **lpEnvelope** member to NULL.

**DIEP\_GAIN**

The **dwGain** member contains data.

**DIEP\_NODOWNLOAD**

Suppress the automatic **IDirectInputEffect::Download** that is normally performed after the parameters are updated. See Remarks.

**DIEP\_NORESTART**

Suppress the stopping and restarting of the effect to change parameters. See Remarks.

**DIEP\_SAMPLEPERIOD**

The **dwSamplePeriod** member contains data.

**DIEP\_START**

The effect is to be started (or restarted if it is currently playing) after the parameters are updated. By default, the play state of the effect is not altered.

**DIEP\_STARTDELAY**

The **dwStartDelay** member contains data.

**DIEP\_TRIGGERBUTTON**

The **dwTriggerButton** member contains data.

**DIEP\_TRIGGERDELAY**

The **dwTriggerDelay** member contains data.

**DIEP\_TRIGGERREPEATINTERVAL**

The **dwTriggerRepeatInterval** member contains data.

**DIEP\_TYPESPECIFICPARAMS**

The **lpvTypeSpecificParams** and **cbTypeSpecificParams** members of the **DIEFFECT** structure contain the address and size of type-specific data for the effect.

## Return Values

If the method succeeds, the return value is one of the following:

DI\_OK  
DI\_EFFECTRESTARTED  
DI\_DOWNLOADSKIPPED  
DI\_TRUNCATED  
DI\_TRUNCATEDANDRESTARTED

If the method fails, the return value can be one of the following error values:

DIERR\_NOTINITIALIZED  
DIERR\_INCOMPLETEEFFECT  
DIERR\_INPUTLOST  
DIERR\_INVALIDPARAM  
DIERR\_EFFECTPLAYING

## Remarks

The **dwDynamicParams** member of the **DIEFFECTINFO** structure for the effect specifies which parameters can be dynamically updated while the effect is playing.

The **IDirectInputEffect::SetParameters** method automatically downloads the effect, but this behavior can be suppressed by setting the **DIEP\_NODOWNLOAD** flag. If automatic download has been suppressed, you can manually download the effect by invoking the **IDirectInputEffect::Download** method.

If the effect is playing while the parameters are changed, the new parameters take effect as if they were the parameters when the effect started.

For example, suppose a periodic effect with a duration of three seconds is started. After two seconds, the direction of the effect is changed. The effect then continues for one additional second in the new direction. The envelope, phase, amplitude, and other parameters of the effect continue smoothly, as if the direction had not changed.

In the same situation, if after two seconds, the duration of the effect were changed to 1.5 seconds, the effect would stop.

Normally, if the driver cannot update the parameters of a playing effect, the driver is permitted to stop the effect, update the parameters, and then restart the effect. Passing



the DIEP\_NORESTART flag suppresses this behavior. If the driver cannot update the parameters of an effect while it is playing, the error code DIERR\_EFFECTPLAYING is returned, and the parameters are not updated.

No more than one of the DIEP\_NODOWNLOAD, DIEP\_START, and DIEP\_NORESTART flags should be set. (It is also valid to pass none of them.)

These three flags control download and playback behavior as follows:

If DIEP\_NODOWNLOAD is set, the effect parameters are updated but not downloaded to the device.

If the DIEP\_START flag is set, the effect parameters are updated and downloaded to the device, and the effect is started just as if the **IDirectInputEffect::Start** method had been called with the *dwIterations* parameter set to 1 and with no flags. (Combining the update with DIEP\_START is slightly faster than calling **Start** separately, because it requires less information to be transmitted to the device.)

If neither DIEP\_NODOWNLOAD nor DIEP\_START is set and the effect is not playing, the parameters are updated and downloaded to the device.

If neither DIEP\_NODOWNLOAD nor DIEP\_START is set and the effect is playing, the parameters are updated if the device supports on-the-fly updating. Otherwise the behavior depends on the state of the DIEP\_NORESTART flag. If it is set, the error code DIERR\_EFFECTPLAYING is returned. If it is clear, the effect is stopped, the parameters are updated, and the effect is restarted.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

**Import Library:** Use `dinput.lib`.

## IDirectInputEffect::Start

The **IDirectInputEffect::Start** method begins playing an effect. If the effect is already playing, it is restarted from the beginning. If the effect has not been downloaded or has been modified since its last download, it is downloaded before being started. This default behavior can be suppressed by passing the DIEP\_NODOWNLOAD flag.

```
HRESULT Start(
    DWORD dwIterations,
    DWORD dwFlags
);
```

## Parameters

### *dwIterations*

Number of times to play the effect in sequence. The envelope is re-articulated with each iteration.

To play the effect exactly once, pass 1. To play the effect repeatedly until explicitly stopped, pass INFINITE. To play the effect until explicitly stopped without re-articulating the envelope, modify the effect parameters with the **IDirectInputEffect::SetParameters** method, and change the **dwDuration** member to INFINITE.

### *dwFlags*

Flags that describe how the effect should be played by the device. The value can be 0 or one or more of the following values:

#### DIES\_SOLO

All other effects on the device should be stopped before the specified effect is played. If this flag is omitted, the effect is mixed with existing effects already started on the device.

#### DIES\_NODOWNLOAD

Do not automatically download the effect.

## Return Values

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

If the method fails, the return value can be one of the following error values:

DIERR\_INVALIDPARAM

DIERR\_INCOMPLETEEFFECT

DIERR\_NOTEXCLUSIVEACQUIRED

DIERR\_NOTINITIALIZED

DIERR\_UNSUPPORTED

## Remarks

The device must be acquired at the exclusive cooperative level for this method to succeed.

Not all devices support multiple iterations.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in dinput.h.

**Import Library:** Use dinput.lib.

---

## IDirectInputEffect::Stop

The **IDirectInputEffect::Stop** method stops playing an effect.

**HRESULT Stop(void);**

### Return Values

If the method succeeds, the return value is **DI\_OK**.

If the method fails, the return value can be one of the following error values:

**DIERR\_NOTEXCLUSIVEACQUIRED**  
**DIERR\_NOTINITIALIZED**

### Remarks

The device must be acquired at the exclusive cooperative level for this method to succeed.

### Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in **dinput.h**.

**Import Library:** Use **dinput.lib**.

## IDirectInputEffect::Unload

The **IDirectInputEffect::Unload** method removes the effect from the device. If the effect is playing, it is automatically stopped before it is unloaded.

**HRESULT Unload(void);**

### Return Values

If the method succeeds, the return value is **DI\_OK**.

If the method fails, the return value can be one of the following error values:

**DIERR\_INPUTLOST**  
**DIERR\_INVALIDPARAM**  
**DIERR\_NOTEXCLUSIVEACQUIRED**  
**DIERR\_NOTINITIALIZED**

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in dinput.h.

**Import Library:** Use dinput.lib.

## Functions

This section is a reference for DirectInput functions other than COM interface methods and callback functions.

The following functions fall into this category:

- **DirectInputCreate** is used to create the DirectInput system and obtain an **IDirectInput** interface. This function is still supported but has been superseded by **DirectInputCreateEx**.
- **DirectInputCreateEx** is used to create the DirectInput system and obtain an **IDirectInput**, **IDirectInput2**, or **IDirectInput7** interface.

## DirectInputCreate

The **DirectInputCreate** function creates a DirectInput object that supports the **IDirectInput** COM interface.

```
HRESULT WINAPI DirectInputCreate(  
    HINSTANCE hinst,  
    DWORD dwVersion,  
    LPDIRECTINPUT *lplpDirectInput,  
    LPUNKNOWN punkOuter  
);
```

### Parameters

*hinst*

Instance handle to the application or DLL that is creating the DirectInput object. DirectInput uses this value to determine whether the application or DLL has been certified and to establish any special behaviors that may be necessary for backward compatibility.

It is an error for a DLL to pass the handle to the parent application. For example, an ActiveX control embedded in a Web page that uses DirectInput must pass its own instance handle, and not the handle to the Web browser. This ensures that DirectInput recognizes the control and can enable any special behaviors that might be necessary.

*dwVersion*

Version number of DirectInput for which the application is designed. This value is normally `DIRECTINPUT_VERSION`. Passing the version number of a previous version causes DirectInput to emulate that version. For more information, see *Designing for Previous Versions of DirectInput*.

*lplpDirectInput*

Address of a variable to receive a valid **IDirectInput** interface pointer if the call succeeds.

*punkOuter*

Pointer to the address of the controlling object's **IUnknown** interface for COM aggregation, or NULL if the interface is not aggregated. Most callers pass NULL. If aggregation is requested, the object returned in *\*lplpDirectInput* is a pointer to the **IUnknown**, rather than an **IDirectInput** interface, as required by COM aggregation.

## Return Values

If the function succeeds, the return value is `DI_OK`.

If the function fails, the return value can be one of the following error values:

`DIERR_BETADIRECTINPUTVERSION`  
`DIERR_INVALIDPARAM`  
`DIERR_OLDDIRECTINPUTVERSION`  
`DIERR_OUTOFMEMORY`

## Remarks

This function has been superseded by **DirectInputCreateEx**, which can be used to obtain any existing interface to the DirectInput object. It is recommended that you use that function to obtain the **IDirectInput7** interface, which has more functionality.

Calling the function with *punkOuter* = NULL is equivalent to creating the object through **CoCreateInstance**(&*CLSID\_DirectInput*, *punkOuter*, *CLSCTX\_INPROC\_SERVER*, &*IID\_IDirectInput*, *lplpDirectInput*), then initializing it with **Initialize**.

Calling the function with *punkOuter* != NULL is equivalent to creating the object through **CoCreateInstance**(&*CLSID\_DirectInput*, *punkOuter*, *CLSCTX\_INPROC\_SERVER*, &*IID\_IUnknown*, *lplpDirectInput*). The aggregated object must be initialized manually.

There are separate ANSI and Unicode versions of this service. The ANSI version creates an object that supports the **IDirectInputA** interface, whereas the Unicode version creates an object that supports the **IDirectInputW** interface. As with other system services that are sensitive to character-set issues, macros in the header file map **DirectInputCreate** to the appropriate character set variation.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

**Import Library:** Use `dinput.lib`.

## DirectInputCreateEx

The **DirectInputCreateEx** function creates a DirectInput object that supports the **IDirectInput**, **IDirectInput2**, or **IDirectInput7** COM interfaces.

```
HRESULT WINAPI DirectInputCreateEx(
    HINSTANCE hinst,
    DWORD dwVersion,
    REFIID riidIIf,
    LPVOID * ppvOut,
    LPUNKNOWN punkOuter
);
```

### Parameters

*hinst*

Instance handle to the application or DLL that is creating the DirectInput object. DirectInput uses this value to determine whether the application or DLL has been certified and to establish any special behaviors that might be necessary for backward compatibility.

It is an error for a DLL to pass the handle to the parent application. For example, an ActiveX control embedded in a Web page that uses DirectInput must pass its own instance handle, and not the handle to the Web browser. This ensures that DirectInput recognizes the control and can enable any special behaviors that might be necessary.

*dwVersion*

Version number of DirectInput for which the application is designed. This value is normally `DIRECTINPUT_VERSION`. Passing the version number of a previous version causes DirectInput to emulate that version. For more information, see *Designing for Previous Versions of DirectInput*.

*riidIIf*

Unique identifier of the desired DirectInput interface. Values supported are `IID_IDirectInput`, `IID_IDirectInput2`, and `IID_IDirectInput7`. If UNICODE is defined during compilation, the Unicode version of the interface is returned; otherwise, it is the ANSI version.

*ppvOut*

Address of a variable to receive the interface pointer if the call succeeds.

*punkOuter*

Pointer to the address of the controlling object's **IUnknown** interface for COM aggregation, or NULL if the interface is not aggregated. Most callers pass NULL. If aggregation is requested, the object returned in *\*ppvOut* is a pointer to **IUnknown**, rather than an **IDirectInput** interface, as required by COM aggregation.

## Return Values

If the function succeeds, the return value is **DI\_OK**.

If the function fails, the return value can be one of the following error values:

**DIERR\_BETADIRECTINPUTVERSION**  
**DIERR\_INVALIDPARAM**  
**DIERR\_OLDDIRECTINPUTVERSION**  
**DIERR\_OUTOFMEMORY**

## Remarks

Calling the function with *punkOuter* = NULL is equivalent to creating the object through **CoCreateInstance**(&*CLSID\_DirectInput*, *punkOuter*, *CLSCTX\_INPROC\_SERVER*, &*IID\_IDirectInput7W*, *lplpDirectInput*), then initializing it with **IDirectInput7::Initialize**.

Calling the function with *punkOuter* != NULL is equivalent to creating the object through **CoCreateInstance**(&*CLSID\_DirectInput*, *punkOuter*, *CLSCTX\_INPROC\_SERVER*, &*IID\_IUnknown*, *lplpDirectInput*). The aggregated object must be initialized manually.

There are separate ANSI and Unicode versions of this service. The ANSI version creates an object that supports the **IDirectInputA** interface, whereas the Unicode version creates an object that supports the **IDirectInputW** interface. As with other system services that are sensitive to character-set issues, macros in the header file map **DirectInputCreateEx** to the appropriate character set variation.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in *dinput.h*.

**Import Library:** Use *dinput.lib*.

## Callback Functions

The following four functions are prototype callback functions for use with various enumeration methods. Applications can declare one of these callback functions under

any name and define it in any way, but the parameter and return types must be the same as in the prototype.

- **DIEnumCreatedEffectObjectsCallback**
- **DIEnumDeviceObjectsCallback**
- **DIEnumDevicesCallback**
- **DIEnumEffectsCallback**
- **DIEnumEffectsInFileCallback**

## DIEnumCreatedEffectObjectsCallback

The **DIEnumCreatedEffectObjectsCallback** function is an application-defined callback function that receives DirectInputDevice effects as a result of a call to the **IDirectInputDevice7::EnumCreatedEffectObjects** method.

```
BOOL CALLBACK DIEnumCreatedEffectObjectsCallback(  
    LPDIRECTINPUTEFFECT peff,  
    LPVOID pvRef  
);
```

### Parameters

*peff*

Address of an effect object that has been created.

*pvRef*

Application-defined value given in the **IDirectInputDevice7::EnumCreatedEffectObjects** method.

### Return Values

Returns **DIENUM\_CONTINUE** to continue the enumeration or **DIENUM\_STOP** to stop the enumeration.

### Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

**Import Library:** User-defined.



## DIEnumDeviceObjectsCallback

The **DIEnumDeviceObjectsCallback** function is an application-defined callback function that receives DirectInputDevice objects as a result of a call to the **IDirectInputDevice7::EnumObjects** method.

```
BOOL CALLBACK DIEnumDeviceObjectsCallback(  
    LPCDIDeviceObjectInstance lpddoi,  
    LPVOID pvRef  
);
```

### Parameters

*lpddoi*

**DIDeviceObjectInstance** structure that describes the object being enumerated.

*pvRef*

Application-defined value given in the **EnumObjects** method.

### Return Values

Returns **DIENUM\_CONTINUE** to continue the enumeration or **DIENUM\_STOP** to stop the enumeration.

### Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

**Import Library:** User-defined.

## DIEnumDevicesCallback

The **DIEnumDevicesCallback** function is an application-defined callback function that receives DirectInput devices as a result of a call to the **IDirectInput7::EnumDevices** method.

```
BOOL CALLBACK DIEnumDevicesCallback(  
    LPCDIDeviceInstance lpddi,  
    LPVOID pvRef  
);
```

### Parameters

*lpddi*

Address of a **DIDeviceInstance** structure that describes the device instance.

*pvRef*

Application-defined value given in the **EnumDevices** method.

## Return Values

Returns **DIENUM\_CONTINUE** to continue the enumeration or **DIENUM\_STOP** to stop the enumeration.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

**Import Library:** User-defined.

# DIEnumEffectsCallback

The **DIEnumEffectsCallback** function is an application-defined callback function used with the **IDirectInputDevice7::EnumEffects** method.

```
BOOL CALLBACK DIEnumEffectsCallback(
    LPCDIEFFECTINFO pdei,
    LPVOID pvRef
);
```

## Parameters

*pdei*

**DIEFFECTINFO** structure that describes the enumerated effect.

*pvRef*

Address of application-defined data given to the **EnumEffects** method.

## Return Values

Returns **DIENUM\_CONTINUE** to continue the enumeration, or **DIENUM\_STOP** to stop it.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in dinput.h.

**Import Library:** User-defined.

## DIEnumEffectsInFileCallback

The **DIEnumEffectsInFileCallback** function is an application-defined callback function used with the **IDirectInputDevice7::EnumEffectsInFile** method.

```
BOOL CALLBACK DIEnumEffectsInFileCallback(  
    LPCDIFILEEFFECT lpDiFileEf,  
    LPVOID pvRef  
);
```

### Parameters

*lpDiFileEf*

**DIFILEEFFECT** structure that describes the enumerated effect.

*pvRef*

Address of application-defined data given to the **EnumEffectsInFile** method.

### Return Values

Returns **DIENUM\_CONTINUE** to continue the enumeration, or **DIENUM\_STOP** to stop it.

### Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in dinput.h.

**Import Library:** User-defined.

## Macros

This section describes the following macros used in DirectInput:

- **DIDFT\_GETINSTANCE**
- **DIDFT\_GETTYPE**
- **DIDFT\_MAKEINSTANCE**
- **DIEFT\_GETTYPE**
- **DIMAKEUSAGEDWORD**
- **DISEQUENCE\_COMPARE**
- **GET\_DIDEVICE\_SUBTYPE**

- **GET\_DIDEVICE\_TYPE**

Dinput.h also defines macros for C calls to all the methods of the **IDirectInput** and **IDirectInputDevice** interfaces. These macros eliminate the need for pointers to method tables. For example, the following is a C call to the **IDirectInputDevice7::Release** method:

```
lpdid->lpVtbl->Release(lpdid);
```

The equivalent macro call looks like this:

```
IDirectInputDevice7_Release(lpdid);
```

All these macros take the same parameters as the method calls themselves.

## DIDFT\_GETINSTANCE

The **DIDFT\_GETINSTANCE** macro extracts the object instance number code from a data format type.

```
DIDFT_GETINSTANCE(n) LOWORD((n) >> 8)
```

### Parameters

*n*

DirectInput data format type. The possible values for this parameter are identical to those found in the **dwType** member of the **DIOBJECTDATAFORMAT** structure.

### Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in dinput.h.

### See Also

**DIDFT\_MAKEINSTANCE**, **DIDFT\_GETTYPE**

## DIDFT\_GETTYPE

The **DIDFT\_GETTYPE** macro extracts the object type code from a data format type.

```
DIDFT_GETTYPE(n) LOBYTE(n)
```

## Parameters

*n*

DirectInput data format type. The possible values for this parameter are identical to those found in the **dwType** member of the **DIOBJECTDATAFORMAT** structure.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

## See Also

**DIDFT\_GETINSTANCE**

# DIDFT\_MAKEINSTANCE

The **DIDFT\_MAKEINSTANCE** macro creates an instance identifier of a device object for packing in the **dwType** member of the **DIOBJECTDATAFORMAT** structure.

`DIDFT_MAKEINSTANCE(n) ((WORD)(n) << 8)`

## Parameters

*n*

Instance of the object; for example, 1 for button 1 of a mouse.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

## See Also

**DIDFT\_GETINSTANCE**

# DIEFT\_GETTYPE

The **DIEFT\_GETTYPE** macro extracts the effect type code from an effect format type.

---

DIEFT\_GETTYPE(n) LOBYTE(n)

## Parameters

*n*

DirectInput effect format type. The possible values for this parameter are identical to those found in the **dwEffType** member of the **DIEFFECTINFO** structure.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in dinput.h.

# DIMAKEUSAGEDWORD

The **DIMAKEUSAGEDWORD** macro combines the Usage Page and Usage HID codes for a device object, for passing to the **IDirectInputDevice7::GetObjectInfo** method.

```
DWORD DIMAKEUSAGEDWORD(  
    WORD UsagePage,  
    WORD Usage  
);
```

## Parameters

*UsagePage*

Usage page of the device object.

*Usage*

Usage of the device object.

## Remarks

This macro is declared in Dinput.h as follows:

```
#define DIMAKEUSAGEDWORD(UsagePage, Usage) \  
    ((DWORD)MAKELONG(Usage, UsagePage))
```

## Requirements

**Windows NT/2000:** Requires Windows 2000.

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

**Header:** Declared in `dinput.h`.

# DISEQUENCE\_COMPARE

The **DISEQUENCE\_COMPARE** macro compares two DirectInput sequence numbers, compensating for wrap around.

```
DISEQUENCE_COMPARE(dwSequence1, cmp, dwSequence2) \
    ((int)((dwSequence1) - (dwSequence2)) cmp 0)
```

## Parameters

*dwSequence1*

First sequence number to compare.

*cmp*

One of the following comparison operators: `==`, `!=`, `<`, `>`, `<=`, or `>=`.

*dwSequence2*

Second sequence number to compare.

## Return Values

Returns a nonzero value if the result of the comparison specified by the *cmp* parameter is true, or 0 otherwise.

## Remarks

The following example checks whether the *dwSequence1* parameter value precedes the *dwSequence2* parameter value chronologically:

```
BOOL Sooner = (DISEQUENCE_COMPARE(dwSequence1, <, dwSequence2));
```

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

# GET\_DIDEVICE\_SUBTYPE

The **GET\_DIDEVICE\_SUBTYPE** macro extracts the device subtype code from a device type description code.

---

GET\_DIDEVICE\_SUBTYPE(dwDevType) HIBYTE(dwDevType)

## Parameters

*dwDevType*

DirectInput device type description code. The possible values for this parameter are identical to those found in the **dwDevType** member of the **DIDEVICEINSTANCE** structure.

## Remarks

The interpretation of the subtype code depends on the primary type.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

## See Also

GET\_DIDEVICE\_TYPE, DIDEVICEINSTANCE

# GET\_DIDEVICE\_TYPE

The **GET\_DIDEVICE\_TYPE** macro extracts the device primary type code from a device type description code.

GET\_DIDEVICE\_TYPE(dwDevType) LOBYTE(dwDevType)

## Parameters

*dwDevType*

DirectInput device type description code. Possible values for this parameter are identical to those found in the **dwDevType** member of the **DIDEVICEINSTANCE** structure.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.



## See Also

GET\_DIDEVICE\_SUBTYPE, DIDEVICEINSTANCE

## Structures

This section contains information on the following structures used with DirectInput:

- **DICONDITION**
- **DICONSTANTFORCE**
- **DICUSTOMFORCE**
- **DIDATAFORMAT**
- **DIDEVCAPS**
- **DIDEVICEINSTANCE**
- **DIDEVICEOBJECTDATA**
- **DIDEVICEOBJECTINSTANCE**
- **DIEFFECT**
- **DIEFFECTINFO**
- **DIEFFESCAPE**
- **DIENVELOPE**
- **DIFILEEFFECT**
- **DIJOYSTATE**
- **DIJOYSTATE2**
- **DIMOUSESTATE**
- **DIMOUSESTATE2**
- **DIOBJECTDATAFORMAT**
- **DIPERIODIC**
- **DIPROPDWORD**
- **DIPROPGUIDANDPATH**
- **DIPROPHEADER**
- **DIPROPRANGE**
- **DIPROPSTRING**
- **DIRAMPFORCE**

## Note

The memory for all DirectX structures must be initialized to 0 before use. In addition, all structures that contain a **dwSize** member should set the member to the size of the structure, in bytes, before use. The following example performs these tasks on a common structure, **DIDEVCAPS**:

```
DIDEVCAPS didevcaps; // Can't use this yet
```

```
ZeroMemory(&didevcaps, sizeof(didevcaps));
didevcaps.dwSize = sizeof(didevcaps);

// Now the structure can be used.
.
```

## DICONDITION

The **DICONDITION** structure contains type-specific information for effects that are marked as **DIEFT\_CONDITION**.

A pointer to an array of **DICONDITION** structures for an effect is passed in the **lpvTypeSpecificParams** member of the **DIEFFECT** structure. The number of elements in the array must be either one, or equal to the number of axes associated with the effect.

```
typedef struct DICONDITION {
    LONG IOffset;
    LONG IPositiveCoefficient;
    LONG INegativeCoefficient;
    DWORD dwPositiveSaturation;
    DWORD dwNegativeSaturation;
    LONG IDeadBand;
} DICONDITION, *LPDICONDITION;

typedef const DICONDITION *LPCDICONDITION;
```

### Members

#### **IOffset**

Offset for the condition, in the range from –10,000 through 10,000.

#### **IPositiveCoefficient**

Coefficient constant on the positive side of the offset, in the range from –10,000 through 10,000.

#### **INegativeCoefficient**

Coefficient constant on the negative side of the offset, in the range from –10,000 through 10,000.

If the device does not support separate positive and negative coefficients, the value of **INegativeCoefficient** is ignored, and the value of **IPositiveCoefficient** is used as both the positive and negative coefficients.

#### **dwPositiveSaturation**

Maximum force output on the positive side of the offset, in the range from 0 through 10,000.

If the device does not support force saturations, the value of this member is ignored.

#### **dwNegativeSaturation**

Maximum force output on the negative side of the offset, in the range from 0 through 10,000.

If the device does not support force saturation, the value of this member is ignored.

If the device does not support separate positive and negative saturation, the value of **dwNegativeSaturation** is ignored, and the value of **dwPositiveSaturation** is used as both the positive and negative saturation.

#### **IDeadBand**

Region around **IOffset** in which the condition is not active, in the range from 0 through 10,000. In other words, the condition is not active between **IOffset** minus **IDeadBand** and **IOffset** plus **IDeadBand**.

### **Remarks**

Different types of conditions interpret the parameters differently, but the basic idea is that force resulting from a condition is equal to  $A(q - q_0)$  where  $A$  is a scaling coefficient,  $q$  is some metric, and  $q_0$  is the neutral value for that metric.

The preceding simplified formula must be adjusted if a nonzero dead band is provided. If the metric is less than **IOffset** - **IDeadBand**, the resulting force is given by the following formula:

$$force = \text{INegativeCoefficient} * (q - (\text{IOffset} - \text{IDeadBand}))$$

Similarly, if the metric is greater than **IOffset** + **IDeadBand**, the resulting force is given by the following formula:

$$force = \text{IPositiveCoefficient} * (q - (\text{IOffset} + \text{IDeadBand}))$$

A spring condition uses axis position as the metric.

A damper condition uses axis velocity as the metric.

An inertia condition uses axis acceleration as the metric.

If the number of **DICONDITION** structures in the array is equal to the number of axes for the effect, the first structure applies to the first axis, the second applies to the second axis, and so on. For example, a two-axis spring condition with **IOffset** set to 0 in both **DICONDITION** structures would have the same effect as the joystick self-centering spring. When a condition is defined for each axis in this way, the effect must not be rotated.

If there is a single **DICONDITION** structure for an effect with more than one axis, the direction along which the parameters of the **DICONDITION** structure are in effect is determined by the direction parameters passed in the **rglDirection** field of the **DIEFFECT** structure. For example, a friction condition rotated 45 degrees (in

polar coordinates) would resist joystick motion in the northeast-southwest direction but would have no effect on joystick motion in the northwest-southeast direction.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in dinput.h.

## DICONSTANTFORCE

The **DICONSTANTFORCE** structure contains type-specific information for effects that are marked as **DIEFT\_CONSTANTFORCE**.

The structure describes a constant force effect.

A pointer to a single **DICONSTANTFORCE** structure for an effect is passed in the **lpvTypeSpecificParams** member of the **DIEFFECT** structure.

```
typedef struct DICONSTANTFORCE {
    LONG IMagnitude;
} DICONSTANTFORCE, *LPDICONSTANTFORCE;

typedef const DICONSTANTFORCE *LPCDICONSTANTFORCE;
```

## Members

### IMagnitude

The magnitude of the effect, in the range from –10,000 through 10,000. If an envelope is applied to this effect, the value represents the magnitude of the sustain. If no envelope is applied, the value represents the amplitude of the entire effect.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in dinput.h.

## DICUSTOMFORCE

The **DICUSTOMFORCE** structure contains type-specific information for effects that are marked as **DIEFT\_CUSTOMFORCE**.

The structure describes a custom or user-defined force.

A pointer to a **DICUSTOMFORCE** structure for an effect is passed in the **lpvTypeSpecificParams** member of the **DIEFFECT** structure.

```
typedef struct DICUSTOMFORCE {
    DWORD cChannels;
    DWORD dwSamplePeriod;
    DWORD cSamples;
    LPLONG rgfForceData;
} DICUSTOMFORCE, *LPDICUSTOMFORCE;

typedef const DICUSTOMFORCE *LPCDICUSTOMFORCE;
```

## Members

### cChannels

Number of channels (axes) affected by this force.

The first channel is applied to the first axis associated with the effect, the second to the second, and so on. If there are fewer channels than axes, nothing is associated with the extra axes.

If there is only a single channel, the effect is rotated in the direction specified by the **rglDirection** member of the **DIEFFECT** structure. If there is more than one channel, rotation is not allowed.

Not all devices support rotation of custom effects.

### dwSamplePeriod

Sample period, in microseconds.

### cSamples

Total number of samples in the **rgfForceData**. It must be an integral multiple of the **cChannels**.

### rgfForceData

Pointer to an array of force values representing the custom force. If multiple channels are provided, the values are interleaved. For example, if **cChannels** is 3, the first element of the array belongs to the first channel, the second to the second, and the third to the third.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

## DIDATAFORMAT

The **DIDATAFORMAT** structure carries information describing a device's data format. This structure is used with the **IDirectInputDevice7::SetDataFormat** method.

```
typedef struct DIDATAFORMAT {  
    DWORD dwSize;  
    DWORD dwObjSize;  
    DWORD dwFlags;  
    DWORD dwDataSize;  
    DWORD dwNumObjs;  
    LPDIOBJECTDATAFORMAT rgodf;  
} DIDATAFORMAT, *LPDIDATAFORMAT;
```

```
typedef const DIDATAFORMAT *LPCDIDATAFORMAT;
```

## Members

### dwSize

Size of this structure, in bytes.

### dwObjSize

Size of the **DIOBJECTDATAFORMAT** structure, in bytes.

### dwFlags

Flags describing other attributes of the data format. This value can be one of the following:

#### DIDF\_ABSAXIS

The axes are in absolute mode. Setting this flag in the data format is equivalent to manually setting the axis mode property, using the **IDirectInputDevice7::SetProperty** method. This cannot be combined with the DIDF\_RELAXIS flag.

#### DIDF\_RELAXIS

The axes are in relative mode. Setting this flag in the data format is equivalent to manually setting the axis mode property using the **IDirectInputDevice7::SetProperty** method. This cannot be combined with the DIDF\_ABSAXIS flag.

### dwDataSize

Size of a data packet returned by the device, in bytes. This value must be a multiple of 4 and must exceed the largest offset value for an object's data within the data packet.

### dwNumObjs

Number of objects in the **rgodf** array.

### rgodf

Address to an array of **DIOBJECTDATAFORMAT** structures. Each structure describes how one object's data should be reported in the device data. Typical errors include placing two pieces of information in the same location and placing one piece of information in more than one location.

## Remarks

Applications do not typically need to create a **DIDATAFORMAT** structure. An application can use one of the predefined global data format variables, *c\_dfDIMouse*, *c\_dfDIMouse2*, *c\_dfDIKeyboard*, *c\_dfDIJoystick*, or *c\_dfDIJoystick2*.

The following code example sets a data format that can be used by applications that need two axes (reported in absolute coordinates) and two buttons:

```
// Suppose an application uses the following
// structure to read device data.

typedef struct MYDATA {
    LONG  IX;           // X-axis goes here.
    LONG  IY;           // Y-axis goes here.
    BYTE  bButtonA;     // One button goes here.
    BYTE  bButtonB;     // Another button goes here.
    BYTE  bPadding[2];  // Must be DWORD multiple in size.
} MYDATA;
```

// Then, it can use the following data format.

```
DIOBJECTDATAFORMAT rgodf[ ] = {
    { &GUID_XAxis, FIELD_OFFSET(MYDATA, IX),
      DIDFT_AXIS | DIDFT_ANYINSTANCE, 0, },
    { &GUID_YAxis, FIELD_OFFSET(MYDATA, IY),
      DIDFT_AXIS | DIDFT_ANYINSTANCE, 0, },
    { &GUID_Button, FIELD_OFFSET(MYDATA, bButtonA),
      DIDFT_BUTTON | DIDFT_ANYINSTANCE, 0, },
    { &GUID_Button, FIELD_OFFSET(MYDATA, bButtonB),
      DIDFT_BUTTON | DIDFT_ANYINSTANCE, 0, },
};

#define numObjects (sizeof(rgodf) / sizeof(rgodf[0]))

DIDATAFORMAT df = {
    sizeof(DIDATAFORMAT),    // this structure
    sizeof(DIOBJECTDATAFORMAT), // size of object data format
    DIDE_ABSAXIS,           // absolute axis coordinates
    sizeof(MYDATA),         // device data size
    numObjects,             // number of objects
    rgodf,                  // and here they are
};
```

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for

Windows 95.

**Header:** Declared in dinput.h.

## DIDEVCAPS

The **DIDEVCAPS** structure contains information about a DirectInput device's capabilities. This structure is used with the **IDirectInputDevice7::GetCapabilities** method.

```
typedef struct DIDEVCAPS {  
    DWORD dwSize;  
    DWORD dwFlags;  
    DWORD dwDevType;  
    DWORD dwAxes;  
    DWORD dwButtons;  
    DWORD dwPOVs;  
    DWORD dwFFSamplePeriod;  
    DWORD dwFFMinTimeResolution;  
    DWORD dwFirmwareRevision;  
    DWORD dwHardwareRevision;  
    DWORD dwFFDriverVersion;  
} DIDEVCAPS, *LPDIDEVCAPS;
```

### Members

#### dwSize

Size of this structure, in bytes. This member must be initialized by the application before a call to the **IDirectInputDevice7::GetCapabilities** method.

#### dwFlags

Flags associated with the device. This value can be a combination of the following:

##### DIDC\_ALIAS

The device is a duplicate of another DirectInput device. Alias devices are by default not enumerated by **IDirectInput7::EnumDevices**.

##### DIDC\_PHANTOM

The device does not really exist. It is a placeholder for a device which might exist in the future. Phantom devices are by default not enumerated by **IDirectInput7::EnumDevices**.

##### DIDC\_ATTACHED

The device is physically attached.

##### DIDC\_DEADBAND

The device supports deadband for at least one force-feedback condition.

##### DIDC\_EMULATED



If this flag is set, the data is coming from a user mode device interface (such as HID) or by some other ring 3 means. If it is not set, the data is coming directly from a kernel mode driver.

#### DIDC\_FORCEFEEDBACK

The device supports force feedback.

#### DIDC\_FFFADE

The force-feedback system supports the fade parameter for at least one effect. If the device does not support fade, the fade level and fade time members of the **DIENVELOPE** structure are ignored by the device.

After a call to the **IDirectInputDevice7::GetEffectInfo** method, an individual effect sets the DIEFT\_FFFADE flag if fade is supported for that effect.

#### DIDC\_FFATTACK

The force-feedback system supports the attack parameter for at least one effect. If the device does not support attack, the attack level and attack time members of the **DIENVELOPE** structure are ignored by the device.

After a call to the **IDirectInputDevice7::GetEffectInfo** method, an individual effect sets the DIEFT\_FFATTACK flag if attack is supported for that effect.

#### DIDC\_POLLEDDATAFORMAT

At least one object in the current data format is polled, rather than interrupt-driven. For these objects, the application must explicitly call the **IDirectInputDevice7::Poll** method to obtain data.

#### DIDC\_POLLEDDEVICE

At least one object on the device is polled, rather than interrupt-driven. For these objects, the application must explicitly call the **IDirectInputDevice7::Poll** method to obtain data. HID devices can contain a mixture of polled and nonpolled objects.

#### DIDC\_POSNEGCOEFFICIENTS

The force-feedback system supports two coefficient values for conditions (one for the positive displacement of the axis and one for the negative displacement of the axis) for at least one condition. If the device does not support both coefficients, the negative coefficient in the **DICONDITION** structure is ignored.

After a call to the **IDirectInputDevice7::GetEffectInfo** method, an individual condition sets the DIEFT\_POSNEGCOEFFICIENTS flag if separate positive and negative coefficients are supported for that condition.

#### DIDC\_POSNEGSATURATION

The force-feedback system supports a maximum saturation for both positive and negative force output for at least one condition. If the device does not support both saturation values, the negative saturation in the **DICONDITION** structure is ignored.

After a call to the **IDirectInputDevice7::GetEffectInfo** method, an individual condition sets the DIEFT\_POSNEGSATURATION flag if separate positive and negative saturation are supported for that condition.

#### DIDC\_SATURATION

The force-feedback system supports the saturation of condition effects for at least one condition. If the device does not support saturation, the force generated by a condition is limited only by the maximum force that the device can generate.

After a call to the **IDirectInputDevice7::GetEffectInfo** method, an individual condition sets the **DIEFT\_SATURATION** flag if saturation is supported for that condition.

#### **DIDC\_STARTDELAY**

The force-feedback system supports the start delay parameter for at least one effect. If the device does not support start delays, the **dwStartDelay** member of the **DIEFFECT** structure is ignored.

#### **dwDevType**

Device type specifier. This member can contain values identical to those in the **dwDevType** member of the **DIDeviceInstance** structure.

#### **dwAxes**

Number of axes available on the device.

#### **dwButtons**

Number of buttons available on the device.

#### **dwPOVs**

Number of point-of-view controllers available on the device.

#### **dwFFSamplePeriod**

Minimum time between playback of consecutive raw force commands.

#### **dwFFMinTimeResolution**

Minimum time, in microseconds, that the device can resolve. The device rounds any times to the nearest supported increment. For example, if the value of **dwFFMinTimeResolution** is 1000, the device would round any times to the nearest millisecond.

#### **dwFirmwareRevision**

Specifies the firmware revision of the device.

#### **dwHardwareRevision**

Hardware revision of the device.

#### **dwFFDriverVersion**

Version number of the device driver.

## **Remarks**

The semantics of version numbers are left to the manufacturer of the device. The only guarantee is that newer versions have larger numbers.

## **Requirements**

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for

Windows 95.

**Header:** Declared in `dinput.h`.

## See Also

**DIDEVICEINSTANCE**

# DIDEVICEINSTANCE

The **DIDEVICEINSTANCE** structure contains information about an instance of a DirectInput device. This structure is used with the **IDirectInput7::EnumDevices** and **IDirectInputDevice7::GetDeviceInfo** methods.

```
typedef struct DIDEVICEINSTANCE {
    DWORD dwSize;
    GUID guidInstance;
    GUID guidProduct;
    DWORD dwDevType;
    TCHAR tszInstanceName[MAX_PATH];
    TCHAR tszProductName[MAX_PATH];
    GUID guidFFDriver;
    WORD wUsagePage;
    WORD wUsage;
} DIDEVICEINSTANCE, *LPDIDEVICEINSTANCE;

typedef const DIDEVICEINSTANCE *LPCDIDEVICEINSTANCE;
```

## Members

### **dwSize**

Size of this structure, in bytes. This member must be initialized before the structure is used.

### **guidInstance**

Unique identifier for the instance of the device. An application can save the instance GUID into a configuration file and use it at a later time. Instance GUIDs are specific to a particular computer. An instance GUID obtained from one computer is unrelated to instance GUIDs on another.

### **guidProduct**

Unique identifier for the product. This identifier is established by the manufacturer of the device.

### **dwDevType**

Device type specifier. The least-significant byte of the device type description code specifies the device type. The next-significant byte specifies the device subtype. This value can be one of the following types combined with their respective subtypes and optionally with **DIDEVTYPE\_HID**, which specifies a Human Interface Device.

**Device Types****DIDEVTYPE\_MOUSE**

A mouse or mouse-like device (such as a trackball).

**DIDEVTYPE\_KEYBOARD**

A keyboard or keyboard-like device.

**DIDEVTYPE\_JOYSTICK**

A joystick or similar device, such as a steering wheel.

**DIDEVTYPE\_DEVICE**

A device that does not fall into the previous categories.

**Mouse subtypes****DIDEVTYPEMOUSE\_UNKNOWN**

The subtype could not be determined.

**DIDEVTYPEMOUSE\_TRADITIONAL**

The device is a traditional mouse.

**DIDEVTYPEMOUSE\_FINGERSTICK**

The device is a fingerstick.

**DIDEVTYPEMOUSE\_TOUCHPAD**

The device is a touchpad.

**DIDEVTYPEMOUSE\_TRACKBALL**

The device is a trackball.

**Keyboard subtypes****DIDEVTYPEKEYBOARD\_UNKNOWN**

The subtype could not be determined.

**DIDEVTYPEKEYBOARD\_PCXT**

IBM PC/XT 83-key keyboard.

**DIDEVTYPEKEYBOARD\_OLIVETTI**

Olivetti 102-key keyboard.

**DIDEVTYPEKEYBOARD\_PCAT**

IBM PC/AT 84-key keyboard.

**DIDEVTYPEKEYBOARD\_PCENH**

IBM PC Enhanced 101/102-key or Microsoft Natural® keyboard.

**DIDEVTYPEKEYBOARD\_NOKIA1050**

Nokia 1050 keyboard.

**DIDEVTYPEKEYBOARD\_NOKIA9140**

Nokia 9140 keyboard.

**DIDEVTYPEKEYBOARD\_NEC98**

Japanese NEC PC98 keyboard.

**DIDEVTYPEKEYBOARD\_NEC98LAPTOP**

Japanese NEC PC98 laptop keyboard.

**DIDEVTYPEKEYBOARD\_NEC98106**

Japanese NEC PC98 106-key keyboard.

**DIDEVTYPEKEYBOARD\_JAPAN106**

Japanese 106-key keyboard.

**DIDEVTYPEKEYBOARD\_JAPANAX**

Japanese AX keyboard.

**DIDEVTYPEKEYBOARD\_J3100**

Japanese J3100 keyboard.

### **Joystick Subtypes**

**DIDEVTYPEJOYSTICK\_UNKNOWN**

The subtype could not be determined.

**DIDEVTYPEJOYSTICK\_TRADITIONAL**

A traditional joystick.

**DIDEVTYPEJOYSTICK\_FLIGHTSTICK**

A joystick optimized for flight simulation.

**DIDEVTYPEJOYSTICK\_GAMEPAD**

A device whose primary purpose is to provide button input.

**DIDEVTYPEJOYSTICK\_RUDDER**

A device for yaw control.

**DIDEVTYPEJOYSTICK\_WHEEL**

A steering wheel.

**DIDEVTYPEJOYSTICK\_HEADTRACKER**

A device that tracks the movement of the user's head.

### **Flags in the High Word**

**DIDEVTYPE\_HID**

The device uses the Human Interface Device (HID) protocol.

**tszInstanceName[MAX\_PATH]**

Friendly name for the instance. For example, "Joystick 1."

**tszProductName[MAX\_PATH]**

Friendly name for the product.

**guidFFDriver**

Unique identifier for the driver being used for force feedback. This identifier is established by the manufacturer of the driver.

**wUsagePage**

If the device is an HID device, this member contains the HID usage page code.

**wUsage**

If the device is an HID, this member contains the HID usage code.

## **Remarks**

For compatibility with previous versions of DirectX, a

**DIDEVICEINSTANCE\_DX3** structure is also defined, containing only the first six members of the **DIDEVICEINSTANCE** structure.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in dinput.h.

## DIDEVICEOBJECTDATA

The **DIDEVICEOBJECTDATA** structure contains raw buffered device information. This structure is used with the **IDirectInputDevice7::GetDeviceData** and the **IDirectInputDevice7::SendDeviceData** methods.

```
typedef struct DIDEVICEOBJECTDATA {
    DWORD dwOfs;
    DWORD dwData;
    DWORD dwTimeStamp;
    DWORD dwSequence;
} DIDEVICEOBJECTDATA, *LPDIDEVICEOBJECTDATA;

typedef const DIDEVICEOBJECTDATA *LPCDIDEVICEOBJECTDATA;
```

## Members

### dwOfs

For **GetDeviceData**, the offset into the current data format of the object whose data is being reported; that is, the location in which the **dwData** would have been stored if the data had been obtained by a call to the

**IDirectInputDevice7::GetDeviceState** method. If the device is accessed as a mouse, keyboard, or joystick, the **dwOfs** member is one of the mouse device constants, keyboard device constants, or joystick device constants. If a custom data format has been set, it is an offset relative to the custom data format.

For **SendDeviceData**, the instance ID of the object to which the data is being sent, as obtained from the **dwType** member of a **DIDEVICEOBJECTINSTANCE** structure.

### dwData

Data obtained from or sent to the device.

For axis input, if the device is in relative axis mode, the relative axis motion is reported. If the device is in absolute axis mode, the absolute axis coordinate is reported.

For button input, only the low byte of **dwData** is significant. The high bit of the low byte is set if the button went down; it is clear if the button went up.

### dwTimeStamp

Tick count at which the input event was generated, in milliseconds. The current system tick count (at a lower resolution) can be obtained by calling the Win32 **GetTickCount** function. This value wraps around approximately every 50 days.

When the structure is used with the **SendDeviceData** method, this member must be 0.

#### **dwSequence**

DirectInput sequence number for this event. All input events are assigned an increasing sequence number. This allows events from different devices to be sorted chronologically. Since this value can wrap around, care must be taken when comparing two sequence numbers. The **DISEQUENCE\_COMPARE** macro can be used to perform this comparison safely.

When the structure is used with the **SendDeviceData** method, this member must be 0.

### **Requirements**

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in dinput.h.

## **DIDeviceObjectInstance**

The **DIDeviceObjectInstance** structure contains information about a device object instance. This structure is used with the **IDirectInputDevice7::EnumObjects** method to provide the **DIEnumDeviceObjectsCallback** callback function with information about a particular object associated with a device, such as an axis or button. It is also used with the **IDirectInputDevice7::GetObjectInfo** method to retrieve information about a device object.

```
typedef struct DIDeviceObjectInstance {
    DWORD dwSize;
    GUID guidType;
    DWORD dwOfs;
    DWORD dwType;
    DWORD dwFlags;
    TCHAR tszName[MAX_PATH];
    DWORD dwFFMaxForce;
    DWORD dwFFForceResolution;
    WORD wCollectionNumber;
    WORD wDesignatorIndex;
    WORD wUsagePage;
    WORD wUsage;
    DWORD dwDimension;
    WORD wExponent;
    WORD wReportId;
} DIDeviceObjectInstance, *LPDIDeviceObjectInstance;
```

```
typedef const DIDEVICEOBJECTINSTANCE
*LPCDIDEVICEOBJECTINSTANCE;
```

## Members

### dwSize

Size of the structure, in bytes. During enumeration, the application can inspect this value to determine how many members of the structure are valid. When the structure is passed to the **IDirectInputDevice7::GetObjectInfo** method, this member must be initialized to **sizeof(DIDEVICEOBJECTINSTANCE)**.

### guidType

Unique identifier that indicates the object type. This member is optional. If present, it can be one of the following values:

#### GUID\_XAxis

The horizontal axis. For example, it can represent the left-right motion of a mouse.

#### GUID\_YAxis

The vertical axis. For example, it can represent the forward-backward motion of a mouse.

#### GUID\_ZAxis

The z-axis. For example, it can represent rotation of the wheel on a mouse, or movement of a throttle control on a joystick.

#### GUID\_RxAxis

Rotation around the x-axis.

#### GUID\_RyAxis

Rotation around the y-axis.

#### GUID\_RzAxis

Rotation around the z-axis (often a rudder control).

#### GUID\_Slider

A slider axis.

#### GUID\_Button

A button on a mouse.

#### GUID\_Key

A key on a keyboard.

#### GUID\_POV

A point-of-view indicator.

#### GUID\_Unknown

Unknown.

Other object types might be defined in the future.

### dwOfs

Offset within the data format at which data is reported for this object. This value can be used to identify the object in method calls and structures that accept the **DIPH\_BYOFFSET** flag.

### dwType



Device type that describes the object. It is a combination of **DIDFT\_\*** flags that describe the object type (axis, button, and so on) and contains the object instance number in the middle 16 bits. Use the **DIDFT\_GETINSTANCE** macro to extract the object instance number. For the **DIDFT\_\*** flags, see **IDirectInputDevice7::EnumObjects**.

#### **dwFlags**

Flags describing other attributes of the data format. This value can be one of the following:

**DIDOI\_ASPECTACCEL**

The object reports acceleration information.

**DIDOI\_ASPECTFORCE**

The object reports force information.

**DIDOI\_ASPECTMASK**

The bits that are used to report aspect information. An object can represent at most one aspect.

**DIDOI\_ASPECTPOSITION**

The object reports position information.

**DIDOI\_ASPECTVELOCITY**

The object reports velocity information.

**DIDOI\_FFACTUATOR**

The object can have force-feedback effects applied to it.

**DIDOI\_FFEFFECTTRIGGER**

The object can trigger playback of force-feedback effects.

**DIDOI\_GUIDISUSAGE**

The **pguid** member of the **DIOBJECTDATAFORMAT** structure is really a (suitably cast) **DIMAKEUSAGEDWORD** of the usage page and usage that is desired.

**DIDOI\_POLLED**

The object does not return data until the **IDirectInputDevice7::Poll** method is called.

#### **tszName[MAX\_PATH]**

Name of the object; for example, "X-Axis" or "Right Shift."

#### **dwFFMaxForce**

The magnitude of the maximum force that can be created by the actuator associated with this object. Force is expressed in newtons and measured in relation to where the hand would be during normal operation of the device.

#### **dwFFForceResolution**

The force resolution of the actuator associated with this object. The returned value represents the number of gradations, or subdivisions, of the maximum force that can be expressed by the force-feedback system from 0 (no force) to maximum force.

#### **wCollectionNumber**

The HID link collection to which the object belongs.

#### **wDesignatorIndex**

An index that refers to a designator in the HID physical descriptor. This number can be passed to functions in the HID parsing library (Hidpi.h) to obtain additional information about the device object.

**wUsagePage**

The HID usage page associated with the object, if known. Human Interface Devices always report a usage page. Non-HID devices can optionally report a usage page; if they do not, the value of this member is 0.

**wUsage**

The HID usage associated with the object, if known. Human Interface Devices always report a usage. Non-HID devices can optionally report a usage; if they do not, the value of this member is 0.

**dwDimension**

An HID code for the dimensional units in which the object's value is reported, if known, or 0 if not known.

**wExponent**

The exponent to associate with the dimension, if known. Dimensional units are always integral, so an exponent might be needed to convert them to nonintegral types.

**wReportId**

Reserved.

## Remarks

Applications can use the **wUsagePage** and **wUsage** members to obtain additional information about how the object was designed to be used. For example, if **wUsagePage** has the value 0x02 (vehicle controls) and **wUsage** has the value 0xB9 (elevator trim), the object was designed to be the elevator trim control on a flightstick. A flight simulator application can use this information to provide more reasonable defaults for objects on the device. HID usage codes are determined by the USB standards committee.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

## DIEFFECT

The **DIEFFECT** structure is used by the **IDirectInputDevice7::CreateEffect** method to initialize a new **IDirectInputEffect** object. It is also used by the **IDirectInputEffect::SetParameters** and **IDirectInputEffect::GetParameters** methods.

```
typedef struct DIEFFECT {
```

---

```

    DWORD dwSize;
    DWORD dwFlags;
    DWORD dwDuration;
    DWORD dwSamplePeriod;
    DWORD dwGain;
    DWORD dwTriggerButton;
    DWORD dwTriggerRepeatInterval;
    DWORD cAxes;
    LPDWORD rgdwAxes;
    LPLONG rgldirection;
    LPDIENVELOPE lpEnvelope;
    DWORD cbTypeSpecificParams;
    LPVOID lpvTypeSpecificParams;
    DWORD dwStartDelay;
} DIEFFECT, *LPDIEFFECT;

```

```
typedef const DIEFFECT *LPCDIEFFECT;
```

## Members

### dwSize

Specifies the size, in bytes, of the structure. This member must be initialized before the structure is used.

### dwFlags

Flags associated with the effect. This value can be a combination of one or more of the following values:

#### DIEFF\_CARTESIAN

The values of **rgldirection** are to be interpreted as Cartesian coordinates.

#### DIEFF\_OBJECTIDS

The values of **dwTriggerButton** and **rgdwAxes** are object identifiers as obtained by **IDirectInputDevice7::EnumObjects**.

#### DIEFF\_OBJECTOFFSETS

The values of **dwTriggerButton** and **rgdwAxes** are data format offsets, relative to the data format selected by **IDirectInputDevice7::SetDataFormat**.

#### DIEFF\_POLAR

The values of **rgldirection** are to be interpreted as polar coordinates.

#### DIEFF\_SPHERICAL

The values of **rgldirection** are to be interpreted as spherical coordinates.

### dwDuration

The total duration of the effect, in microseconds. If this value is INFINITE, the effect has infinite duration. If an envelope has been applied to the effect, the attack is applied, followed by an infinite sustain.

### dwSamplePeriod

The period at which the device should play back the effect, in microseconds. A value of 0 indicates that the default playback sample rate should be used.

If the device is not capable of playing back the effect at the specified rate, it chooses the supported rate that is closest to the requested value.

Setting a custom **dwSamplePeriod** can be used for special effects. For example, playing a sine wave at an artificially large sample period results in a rougher texture.

#### **dwGain**

The gain to be applied to the effect, in the range from 0 through 10,000. The gain is a scaling factor applied to all magnitudes of the effect and its envelope.

#### **dwTriggerButton**

The identifier or offset of the button to be used to trigger playback of the effect. The flags **DIEFF\_OBJECTIDS** and **DIEFF\_OBJECTOFFSETS** determine the semantics of the value. If this member is set to **DIEB\_NOTRIGGER**, no trigger button is associated with the effect.

#### **dwTriggerRepeatInterval**

The interval, in microseconds, between the end of one playback and the start of the next when the effect is triggered by a button press and the button is held down. Setting this value to **INFINITE** suppresses repetition.

Support for trigger repeat for an effect is indicated by the presence of the **DIEP\_TRIGGERREPEATINTERVAL** flag in the **dwStaticParams** member of the **DIEFFECTINFO** structure.

#### **cAxes**

Number of axes involved in the effect. This member must be filled in by the caller if changing or setting the axis list or the direction list.

The number of axes for an effect cannot be changed once it has been set.

#### **rgdwAxes**

Pointer to a **DWORD** array (of **cAxes** elements) containing identifiers or offsets identifying the axes to which the effect is to be applied. The flags **DIEFF\_OBJECTIDS** and **DIEFF\_OBJECTOFFSETS** determine the semantics of the values in the array.

The list of axes associated with an effect cannot be changed once it has been set.

No more than 32 axes can be associated with a single effect.

#### **rglDirection**

Pointer to a **LONG** array (of **cAxes** elements) containing either Cartesian coordinates or polar coordinates. The flags **DIEFF\_CARTESIAN**, **DIEFF\_POLAR**, and **DIEFF\_SPHERICAL** determine the semantics of the values in the array.

If Cartesian, each value in **rglDirection** is associated with the corresponding axis in **rgdwAxes**.

If polar, the angle is measured in hundredths of degrees from the (0, -1) direction, rotated in the direction of (1, 0). This usually means that north is away from the user, and east is to the user's right. The last element is not used.

If spherical, the first angle is measured in hundredths of a degree from the (1, 0) direction, rotated in the direction of (0, 1). The second angle (if the number of axes is three or more) is measured in hundredths of a degree toward (0, 0, 1). The third angle (if the number of axes is four or more) is measured in hundredths of a degree toward (0, 0, 0, 1), and so on. The last element is not used.

### Note

The **rglDirection** array must contain **cAxes** entries, even if polar or spherical coordinates are given. In these cases, the last element in the **rglDirection** array is reserved for future use and must be 0.

### lpEnvelope

Optional pointer to a **DIENVELOPE** structure that describes the envelope to be used by this effect. Not all effect types use envelopes. If no envelope is to be applied, the member should be set to NULL.

### cbTypeSpecificParams

Number of bytes of additional type-specific parameters for the corresponding effect type.

### lpvTypeSpecificParams

Pointer to type-specific parameters, or NULL if there are no type-specific parameters.

If the effect is of type **DIEFT\_CONDITION**, this member contains a pointer to an array of **DICONDITION** structures that define the parameters for the condition. A single structure may be used, in which case the condition is applied in the direction specified in the **rglDirection** array. Otherwise, there must be one structure for each axis, in the same order as the axes in **rgdwAxes** array. If a structure is supplied for each axis, the effect should not be rotated; you should use the following values in the **rglDirection** array:

- **DIEFF\_SPHERICAL**: 0, 0, ...
- **DIEFF\_POLAR**: 9000, 0, ...
- **DIEFF\_CARTESIAN**: 1, 0, ...

If the effect is of type **DIEFT\_CUSTOMFORCE**, this member contains a pointer to a **DICUSTOMFORCE** structure that defines the parameters for the custom force.

If the effect is of type **DIEFT\_PERIODIC**, this member contains a pointer to a **DIPERIODIC** structure that defines the parameters for the effect.

If the effect is of type **DIEFT\_CONSTANTFORCE**, this member contains a pointer to a **DICONSTANTFORCE** structure that defines the parameters for the constant force.

If the effect is of type **DIEFT\_RAMPFORCE**, this member contains a pointer to a **DIRAMPFORCE** structure that defines the parameters for the ramp force.

### dwStartDelay

Time (in microseconds) that the device should wait after a

**IDirectInputEffect::Start** call before playing the effect. If this value is 0, effect

playback begins immediately. This member is not present in versions prior to DirectX 7.0.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

## DIEFFECTINFO

The **DIEFFECTINFO** structure is used by the **IDirectInputDevice7::EnumEffects** and **IDirectInputDevice7::GetEffectInfo** methods to return information about a particular effect supported by a device.

```
typedef struct DIEFFECTINFO {
    DWORD dwSize;
    GUID guid;
    DWORD dwEffType;
    DWORD dwStaticParams;
    DWORD dwDynamicParams;
    TCHAR tszName[MAX_PATH];
} DIEFFECTINFO, *LPDIEFFECTINFO;

typedef const DIEFFECTINFO *LPCDIEFFECTINFO;
```

## Members

### dwSize

Size of the structure in bytes. During enumeration, the application can inspect this value to determine how many members of the structure are valid. This member must be initialized before the structure is passed to the **IDirectInputDevice7::GetEffectInfo** method.

### guid

Identifier of the effect.

### dwEffType

Zero or more of the following values:

**DIEFT\_ALL**

Valid only for **IDirectInputDevice7::EnumEffects**. Enumerate all effects, regardless of type. This flag cannot be combined with any of the other flags.

**DIEFT\_CONDITION**

The effect represents a condition. When creating or modifying a condition, the **lpvTypeSpecificParams** member of the **DIEFFECT** structure must point to an array of **DICONDITION** structures (one per axis), and the

**cbTypeSpecificParams** member must be set to **cAxis \* sizeof(DICONDITION)**.

Not all devices support all the parameters of conditions. Check the effect capability flags to determine which capabilities are available.

The flag can be passed to **IDirectInputDevice7::EnumEffects** to restrict the enumeration to conditions.

#### DIEFT\_CONSTANTFORCE

The effect represents a constant-force effect. When creating or modifying a constant-force effect, the **lpvTypeSpecificParams** member of the **DIEFFECT** must point to a **DICONSTANTFORCE** structure, and the **cbTypeSpecificParams** member must be set to **sizeof(DICONSTANTFORCE)**.

The flag can be passed to **IDirectInputDevice7::EnumEffects** to restrict the enumeration to constant-force effects.

#### DIEFT\_CUSTOMFORCE

The effect represents a custom-force effect. When creating or modifying a custom-force effect, the **lpvTypeSpecificParams** member of the **DIEFFECT** structure must point to a **DICUSTOMFORCE** structure, and the **cbTypeSpecificParams** member must be set to **sizeof(DICUSTOMFORCE)**.

The flag can be passed to **IDirectInputDevice7::EnumEffects** to restrict the enumeration to custom-force effects.

#### DIEFT\_DEADBAND

The effect generator for this condition effect supports the **IDeadBand** parameter.

#### DIEFT\_FFATTACK

The effect generator for this effect supports the attack envelope parameter. If the effect generator does not support attack, the attack level and attack time parameters of the **DIENVELOPE** structure are ignored by the effect.

If neither **DIEFT\_FFATTACK** nor **DIEFT\_FFFADE** is set, the effect does not support an envelope, and any provided envelope is ignored.

#### DIEFT\_FFFADE

The effect generator for this effect supports the fade parameter. If the effect generator does not support fade, the fade level and fade time parameters of the **DIENVELOPE** structure are ignored by the effect.

If neither **DIEFT\_FFATTACK** nor **DIEFT\_FFFADE** is set, the effect does not support an envelope, and any provided envelope is ignored.

#### DIEFT\_HARDWARE

The effect represents a hardware-specific effect. For additional information on using a hardware-specific effect, consult the hardware documentation.

The flag can be passed to the **IDirectInputDevice7::EnumEffects** method to restrict the enumeration to hardware-specific effects.

#### DIEFT\_PERIODIC

The effect represents a periodic effect. When creating or modifying a periodic effect, the **lpvTypeSpecificParams** member of the **DIEFFECT** structure must point to a **DIPERIODIC** structure, and the **cbTypeSpecificParams** member must be set to **sizeof(DIPERIODIC)**.

The flag can be passed to **IDirectInputDevice7::EnumEffects** to restrict the enumeration to periodic effects.

#### **DIEFT\_POSNEGCOEFFICIENTS**

The effect generator for this effect supports two coefficient values for conditions, one for the positive displacement of the axis and one for the negative displacement of the axis. If the device does not support both coefficients, the negative coefficient in the **DICONDITION** structure is ignored, and the positive coefficient is used in both directions.

#### **DIEFT\_POSNEGSATURATION**

The effect generator for this effect supports a maximum saturation for both positive and negative force output. If the device does not support both saturation values, the negative saturation in the **DICONDITION** structure is ignored, and the positive saturation is used in both directions.

#### **DIEFT\_RAMPFORCE**

The effect represents a ramp-force effect. When creating or modifying a ramp-force effect, the **lpvTypeSpecificParams** member of the **DIEFFECT** structure must point to a **DIRAMPFORCE** structure, and the **cbTypeSpecificParams** member must be set to **sizeof(DIRAMPFORCE)**.

The flag can be passed to **IDirectInputDevice7::EnumEffects** to restrict the enumeration to ramp-force effects.

#### **DIEFT\_SATURATION**

The effect generator for this effect supports the saturation of condition effects.

If the effect generator does not support saturation, the force generated by a condition is limited only by the maximum force that the device can generate.

#### **dwStaticParams**

Zero or more **DIEP\_\*** values describing the parameters supported by the effect. For example, if **DIEP\_ENVELOPE** is set, the effect supports an envelope. For a list of possible values, see **IDirectInputEffect::GetParameters**.

It is not an error for an application to attempt to use effect parameters that are not supported by the device. The unsupported parameters are ignored.

This information is provided to allow the application to tailor its use of force feedback to the capabilities of the specific device.

#### **dwDynamicParams**

Zero or more **DIEP\_\*** values denoting parameters of the effect that can be modified while the effect is playing. For a list of possible values, see **IDirectInputEffect::GetParameters**.

If an application attempts to change a parameter while the effect is playing and the driver does not support modifying that effect dynamically, the driver is permitted to stop the effect, update the parameters, then restart it. For more information, see **IDirectInputEffect::SetParameters**.



**tszName**[MAX\_PATH]

Name of the effect; for example, "Sawtooth up" or "Constant force".

## Remarks

Use the **DIEFT\_GETTYPE** macro to extract the effect type from the **dwEffType** flags.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

# DIEFFESCAPE

The **DIEFFESCAPE** structure is used by the **IDirectInputDevice7::Escape** and **IDirectInputEffect::Escape** methods to pass hardware-specific data directly to the device driver.

```
typedef struct DIEFFESCAPE {  
    DWORD dwSize;  
    DWORD dwCommand;  
    LPVOID lpvInBuffer;  
    DWORD cbInBuffer;  
    LPVOID lpvOutBuffer;  
    DWORD cbOutBuffer;  
} DIEFFESCAPE, *LPDIEFFESCAPE;
```

## Members

### **dwSize**

Size of the structure in bytes. This member must be initialized before the structure is used.

### **dwCommand**

Driver-specific command number. Consult the driver documentation for a list of valid commands.

### **lpvInBuffer**

Buffer containing the data required to perform the operation.

### **cbInBuffer**

Size, in bytes, of the **lpvInBuffer** buffer.

### **lpvOutBuffer**

Buffer in which the operation's output data is returned.

### **cbOutBuffer**

On entry, the size in bytes of the **lpvOutBuffer** buffer. On exit, the number of bytes actually produced by the command.

## Remarks

Since each driver implements different escapes, it is the application's responsibility to ensure that it is talking to the correct driver by comparing the **guidFFDriver** member in the **DIDEVICEINSTANCE** structure against the value the application is expecting.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

# DIENVELOPE

The **DIENVELOPE** structure is used by the **DIEFFECT** structure to specify the optional envelope parameters for an effect. The sustain level for the envelope is represented by the **dwMagnitude** member of the **DIPERIODIC** structure and the **IMagnitude** member of the **DICONSTANTFORCE** structure. The sustain time is represented by **dwDuration** member of the **DIEFFECT** structure.

```
typedef struct DIENVELOPE {
    DWORD dwSize;
    DWORD dwAttackLevel;
    DWORD dwAttackTime;
    DWORD dwFadeLevel;
    DWORD dwFadeTime;
} DIENVELOPE, *LPDIENVELOPE;

typedef const DIENVELOPE *LPCDIENVELOPE;
```

## Members

### dwSize

Size, in bytes, of the structure. This member must be initialized before the structure is used.

### dwAttackLevel

Amplitude for the start of the envelope, relative to the baseline, in the range from 0 through 10,000. If the effect's type-specific data does not specify a baseline, the amplitude is relative to 0.

### dwAttackTime

The time, in microseconds, to reach the sustain level.

### dwFadeLevel

Amplitude for the end of the envelope, relative to the baseline, in the range from 0 through 10,000. If the effect's type-specific data does not specify a baseline, the amplitude is relative to 0.

**dwFadeTime**

The time, in microseconds, to reach the fade level.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in dinput.h.

## DIFILEEFFECT

The **DIFILEEFFECT** structure describes data for a force-feedback effect stored in a file. It is used in conjunction with the **IDirectInputDevice7::EnumEffectsInFile** and **IDirectInputDevice7::WriteEffectToFile** methods.

```
typedef struct DIFILEEFFECT{
    DWORD    dwSize;
    GUID     GuidEffect;
    LPCDIEFFECT lpDiEffect;
    CHAR     szFriendlyName[MAX_PATH];
}DIFILEEFFECT, *LPDIFILEEFFECT;
```

## Members

**dwSize**

Size, in bytes, of the structure. This member must be initialized before the structure is used.

**GuidEffect**

Unique identifier of the effect type. This can be one of the standard GUIDs defined in Dinput.h, such as GUID\_ConstantForce, or one created by the designer.

**lpDiEffect**

Pointer to a **DIEFFECT** structure containing information about the effect.

**szFriendlyName**

Name of the effect.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in dinput.h.

# DIJOYSTATE

The **DIJOYSTATE** structure contains information about the state of a joystick device. This structure is used with the **IDirectInputDevice7::GetDeviceState** method.

```
typedef struct DIJOYSTATE {
    LONG    IX;
    LONG    IY;
    LONG    IZ;
    LONG    IRx;
    LONG    IRy;
    LONG    IRz;
    LONG    rglSlider[2];
    DWORD    rgdwPOV[4];
    BYTE    rgbButtons[32];
} DIJOYSTATE, *LPDIJOYSTATE;
```

## Members

### IX

Information about the joystick x-axis (usually the left-right movement of a stick).

### IY

Information about the joystick y-axis (usually the forward-backward movement of a stick).

### IZ

Information about the joystick z-axis (often the throttle control). If the joystick does not have this axis, the value is 0.

### IRx

Information about the joystick x-axis rotation. If the joystick does not have this, the value is 0.

### IRy

Information about the joystick y-axis rotation. If the joystick does not have this axis, the value is 0.

### IRz

Information about the joystick z-axis rotation (often called the rudder). If the joystick does not have this axis, the value is 0.

### rglSlider[2]

Two additional axis values (formerly called the u-axis and v-axis) whose semantics depend on the joystick. Use the **IDirectInputDevice7::GetObjectInfo** method to obtain semantic information about these values.

### rgdwPOV[4]

The current position of up to four direction controllers (such as point-of-view indicators). The position is indicated in hundredths of a degree clockwise from north (away from the user). The center position is normally reported as -1; but

see Remarks. For indicators that have only five positions, the value for a controller is -1, 0, 9,000, 18,000, or 27,000.

#### **rgbButtons[32]**

Array of button states. The high-order bit of the byte is set if the corresponding button is down, and clear if the button is up or does not exist.

## **Remarks**

You must prepare the device for joystick-style access by calling the **IDirectInputDevice7::SetDataFormat** method, passing the *c\_dfDIJoystick* global data format variable.

If an axis is in relative mode, the appropriate member contains the change in position. If it is in absolute mode, the member contains the absolute axis position.

Some drivers report the centered position of the POV indicator as 65,535. Determine whether the indicator is centered as follows:

```
BOOL POVCentered = (LOWORD(dwPOV) == 0xFFFF);
```

## **Requirements**

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

# **DIJOYSTATE2**

The **DIJOYSTATE2** structure contains information about the state of a joystick device with extended capabilities. This structure is used with the **IDirectInputDevice7::GetDeviceState** method.

```
typedef struct DIJOYSTATE2 {
    LONG    IX;
    LONG    IY;
    LONG    IZ;
    LONG    IRx;
    LONG    IRy;
    LONG    IRz;
    LONG    rglSlider[2];
    DWORD   rgdwPOV[4];
    BYTE    rgbButtons[128];
    LONG    IVX;
    LONG    IVY;
    LONG    IVZ;
    LONG    IVRx;
```

---

```

LONG   IVRy;
LONG   IVRz;
LONG   rgIVSlider[2];
LONG   IAX;
LONG   IAY;
LONG   IAZ;
LONG   IARx;
LONG   IARy;
LONG   IARz;
LONG   rgIASlider[2];
LONG   IFX;
LONG   IFY;
LONG   IFZ;
LONG   IFRx;
LONG   IFRy;
LONG   IFRz;
LONG   rgIFSslider[2];
} DIJOYSTATE2, *LPDIJOYSTATE2;

```

## Members

### IX

Information about the joystick x-axis (usually the left-right movement of a stick).

### IY

Information about the joystick y-axis (usually the forward-backward movement of a stick).

### IZ

Information about the joystick z-axis (often the throttle control). If the joystick does not have this axis, the value is 0.

### IRx

Information about the joystick x-axis rotation. If the joystick does not have this, the value is 0.

### IRy

Information about the joystick y-axis rotation. If the joystick does not have this axis, the value is 0.

### IRz

Information about the joystick z-axis rotation (often called the rudder). If the joystick does not have this axis, the value is 0.

### rglSlider[2]

Two additional axis values (formerly called the u-axis and v-axis) whose semantics depend on the joystick. Use the **IDirectInputDevice7::GetObjectInfo** method to obtain semantic information about these values.

### rgdwPOV[4]

The current position of up to four direction controllers (such as point-of-view hats). The position is indicated in hundredths of a degree clockwise from north (away from the user). The center position is normally reported as  $-1$ ; but see Remarks. For indicators that have only five positions, the value for a controller is  $-1$ ,  $0$ ,  $9,000$ ,  $18,000$ , or  $27,000$ .

**rgbButtons[128]**

Array of button states. The high-order bit of the byte is set if the corresponding button is down, and clear if the button is up or does not exist.

**IVX**

Information about the x-axis velocity.

**IVY**

Information about the y-axis velocity.

**IVZ**

Information about the z-axis velocity.

**IVRx**

Information about the x-axis angular velocity.

**IVRy**

Information about the y-axis angular velocity.

**IVRz**

Information about the z-axis angular velocity.

**rglVSlider[2]**

Information about extra axis velocities.

**IAX**

Information about the x-axis acceleration.

**IAY**

Information about the y-axis acceleration.

**IAZ**

Information about the z-axis acceleration.

**IARx**

Information about the x-axis angular acceleration.

**IARy**

Information about the y-axis angular acceleration.

**IARz**

Information about the z-axis angular acceleration.

**rglASlider[2]**

Information about extra axis accelerations.

**IFX**

Information about the x-axis force.

**IFY**

Information about the y-axis force.

**IFZ**

Information about the z-axis force.

**IFRx**

Information about the x-axis torque.

#### **IFRy**

Information about the y-axis torque.

#### **IFRz**

Information about the z-axis torque.

#### **rglFSlider[2]**

Information about extra axis forces.

## **Remarks**

You must prepare the device for access to a joystick with extended capabilities by calling the **IDirectInputDevice7::SetDataFormat** method, passing the *c\_dfDIJoystick2* global data format variable.

The **DIJOYSTATE2** structure has no special association with the **IDirectInputDevice7** interface. You can use either **DIJOYSTATE** or **DIJOYSTATE2** with either the **IDirectInputDevice** or the **IDirectInputDevice7** interface.

If an axis is in relative mode, the appropriate member contains the change in position. If it is in absolute mode, the member contains the absolute axis position.

Some drivers report the centered position of the POV indicator as 65,535. Determine whether the indicator is centered as follows:

```
BOOL POVCentered = (LOWORD(dwPOV) == 0xFFFF);
```

## **Requirements**

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in dinput.h.

# **DIMOUSESTATE**

The **DIMOUSESTATE** structure contains information about the state of a mouse device that has up to four buttons, or another device that is being accessed as if it were a mouse device. This structure is used with the

**IDirectInputDevice7::GetDeviceState** method.

```
typedef struct DIMOUSESTATE {
    LONG IX;
    LONG IY;
    LONG IZ;
    BYTE rgbButtons[4];
} DIMOUSESTATE, *LPDIMOUSESTATE;
```



## Members

### IX

Information about the mouse x-axis.

### IY

Information about the mouse y-axis.

### IZ

Information about the mouse z-axis (typically a wheel). If the mouse does not have a z-axis, the value is 0.

### rgbButtons[4]

Array of button states. The high-order bit of the byte is set if the corresponding button is down.

## Remarks

You must prepare the device for mouse-style access by calling the **IDirectInputDevice7::SetDataFormat** method, passing the *c\_dfDIMouse* global data format variable.

The mouse is a relative-axis device, so the absolute axis positions for mouse axes are simply accumulated relative motion. Therefore, the value of the absolute axis position is not meaningful except in comparison with other absolute axis positions.

If an axis is in relative mode, the appropriate member contains the change in position. If it is in absolute mode, the member contains the absolute axis position.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

## See Also

### DIMOUSESTATE2

## DIMOUSESTATE2

The **DIMOUSESTATE2** structure contains information about the state of a mouse device that has up to eight buttons, or another device that is being accessed as if it were a mouse device. This structure is used with the **IDirectInputDevice7::GetDeviceState** method.

```
typedef struct DIMOUSESTATE {  
    LONG IX;
```

```
    LONG IY;  
    LONG IZ;  
    BYTE rgbButtons[8];  
} DIMOUSESTATE, *LPDIMOUSESTATE;
```

## Members

### IX

Information about the mouse x-axis.

### IY

Information about the mouse y-axis.

### IZ

Information about the mouse z-axis (typically a wheel). If the mouse does not have a z-axis, the value is 0.

### rgbButtons[8]

Array of button states. The high-order bit of the byte is set if the corresponding button is down.

## Remarks

You must prepare the device for mouse-style access by calling the **IDirectInputDevice7::SetDataFormat** method, passing the *c\_dfDIMouse2* global data format variable.

The mouse is a relative-axis device, so the absolute axis positions for mouse axes are simply accumulated relative motion. Therefore, the value of the absolute axis position is not meaningful except in comparison with other absolute axis positions.

If an axis is in relative mode, the appropriate member contains the change in position. If it is in absolute mode, the member contains the absolute axis position.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in dinput.h.

## See Also

DIMOUSESTATE

# DIOBJECTDATAFORMAT

The **DIOBJECTDATAFORMAT** structure contains information about a device object's data format for use with the **IDirectInputDevice7::SetDataFormat** method.

```
typedef struct DIOBJECTDATAFORMAT {
    const GUID * pguid;
    DWORD      dwOfs;
    DWORD      dwType;
    DWORD      dwFlags;
} DIOBJECTDATAFORMAT, *LPDIOBJECTDATAFORMAT;
```

```
typedef const DIOBJECTDATAFORMAT *LPCDIOBJECTDATAFORMAT;
```

## Members

### pguid

Unique identifier for the axis, button, or other input source. When requesting a data format, making this member NULL indicates that any type of object is permissible.

### dwOfs

Offset within the data packet where the data for the input source is stored. This value must be a multiple of 4 for **DWORD** size data, such as axes. It can be byte-aligned for buttons.

### dwType

Device type that describes the object. It is a combination of the following flags describing the object type (axis, button, and so forth) and containing the object-instance number in the middle 16 bits. When requesting a data format, the instance portion must be set to DIDFT\_ANYINSTANCE to indicate that any instance is permissible, or to **DIDFT\_MAKEINSTANCE(*n*)** to restrict the request to instance *n*. See the examples under Remarks.

#### DIDFT\_ABSAXIS

The object selected by the **SetDataFormat** method must be an absolute axis.

#### DIDFT\_AXIS

The object selected by the **SetDataFormat** method must be an absolute or relative axis.

#### DIDFT\_BUTTON

The object selected by the **SetDataFormat** method must be a push button or a toggle button.

#### DIDFT\_FFACTUATOR

The object selected by the **SetDataFormat** method must contain a force-feedback actuator; in other words, it must be possible to apply forces to the object.

#### DIDFT\_FFEFFECTTRIGGER

The object selected by the **SetDataFormat** method must be a valid force-feedback effect trigger.

#### DIDFT\_POV

The object selected by the **SetDataFormat** method must be a point-of-view controller.

#### DIDFT\_PSHBUTTON

The object selected by the **SetDataFormat** method must be a push button.

DIDFT\_RELAXIS

The object selected by **SetDataFormat** must be a relative axis.

DIDFT\_TGLBUTTON

The object selected by **SetDataFormat** must be a toggle button.

DIDFT\_VENDORDEFINED

The object selected by **SetDataFormat** must be of a type defined by the manufacturer.

#### dwFlags

Zero or more of the following values:

DIDOI\_ASPECTACCEL

The object selected by **SetDataFormat** must report acceleration information.

DIDOI\_ASPECTFORCE

The object selected by **SetDataFormat** must report force information.

DIDOI\_ASPECTPOSITION

The object selected by **SetDataFormat** must report position information.

DIDOI\_ASPECTVELOCITY

The object selected by **SetDataFormat** must report velocity information.

#### Remarks

A data format is made up of several **DIOBJECTDATAFORMAT** structures, one for each object (axis, button, and so on). An array of these structures is contained in the **DIDATAFORMAT** structure that is passed to **IDirectInputDevice7::SetDataFormat**. An application typically does not need to create an array of **DIOBJECTDATAFORMAT** structures; rather, it can use one of the predefined data formats, *c\_dfDIMouse*, *c\_dfDIMouse2*, *c\_dfDIKeyboard*, *c\_dfDIJoystick*, or *c\_dfDIJoystick2*, which have predefined settings for **DIOBJECTDATAFORMAT**.

The following object data format specifies that DirectInput should choose the first available axis and report its value in the **DWORD** at offset 4 in the device data.

```
DIOBJECTDATAFORMAT dfAnyAxis = {
    0,                // Wildcard
    4,                // Offset
    DIDFT_AXIS | DIDFT_ANYINSTANCE, // Any axis is okay.
    0,                // Don't care about aspect
};
```

The following object data format specifies that the x-axis of the device should be stored in the **DWORD** at offset 12 in the device data. If the device has more than one x-axis, the first available one should be selected.

```
DIOBJECTDATAFORMAT dfAnyXAxis = {
    &GUID_XAxis,      // Must be an X axis
```

```

12,                // Offset
DIDFT_AXIS | DIDFT_ANYINSTANCE, // Any X axis is okay.
0,                // Don't care about aspect
};

```

The following object data format specifies that DirectInput should choose the first available button and report its value in the high bit of the byte at offset 16 in the device data.

```

DIOBJECTDATAFORMAT dfAnyButton = {
0,                // Wildcard
16,                // Offset
DIDFT_BUTTON | DIDFT_ANYINSTANCE, // Any button is okay.
0,                // Don't care about aspect
};

```

The following object data format specifies that button 0 of the device should be reported as the high bit of the byte stored at offset 18 in the device data.

If the device does not have a button 0, the attempt to set this data format fails.

```

DIOBJECTDATAFORMAT dfButton0 = {
0,                // Wildcard
18,                // Offset
DIDFT_BUTTON | DIDFT_MAKEINSTANCE(0), // Button zero
0,                // Don't care about aspect
};

```

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

## DIPERIODIC

The **DIPERIODIC** structure contains type-specific information for effects that are marked as **DIEFT\_PERIODIC**.

The structure describes a periodic effect.

A pointer to a single **DIPERIODIC** structure for an effect is passed in the **lpvTypeSpecificParams** member of the **DIEFFECT** structure.

```

typedef struct DIPERIODIC {
    DWORD dwMagnitude;
    LONG lOffset;
    DWORD dwPhase;
};

```

```

    DWORD dwPeriod;
} DIPERIODIC, *LPDIPERIODIC;

typedef const DIPERIODIC *LPCDIPERIODIC;

```

## Members

### dwMagnitude

Magnitude of the effect, in the range from 0 through 10,000. If an envelope is applied to this effect, the value represents the magnitude of the sustain. If no envelope is applied, the value represents the amplitude of the entire effect.

### lOffset

Offset of the effect. The range of forces generated by the effect is **lOffset** minus **dwMagnitude** to **lOffset** plus **dwMagnitude**. The value of the **lOffset** member is also the baseline for any envelope that is applied to the effect.

### dwPhase

Position in the cycle of the periodic effect at which playback begins, in the range from 0 through 35,999. See Remarks.

### dwPeriod

Period of the effect, in microseconds.

## Remarks

A device driver cannot provide support for all values in the **dwPhase** member. In this case, the value is rounded off to the nearest supported value.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

# DIPROPDWORD

The **DIPROPDWORD** is a generic structure used to access **DWORD** properties.

```

typedef struct DIPROPDWORD {
    DIPROPHEADER    diph;
    DWORD           dwData;
} DIPROPDWORD, *LPDIPROPDWORD;

typedef const DIPROPDWORD *LPCDIPROPDWORD;

```

## Members

### diph

**DIPROPHEADER** structure that must be initialized as follows:

Member	Value
<b>dwSize</b>	<b>sizeof(DIPROPDWORD)</b>
<b>dwHeaderSize</b>	<b>sizeof(DIPROPHEADER)</b>
<b>dwObj</b>	<p>If the <b>dwHow</b> member is <b>DIPH_DEVICE</b>, this member must be 0.</p> <p>If the <b>dwHow</b> member is <b>DIPH_BYID</b>, this member must be the identifier for the object whose property setting is to be set or retrieved.</p> <p>If the <b>dwHow</b> member is <b>DIPH_BYOFFSET</b>, this member must be a data format offset for the object whose property setting is to be set or retrieved. For example, if the <i>c_dfDIMouse</i> data format is selected, it must be one of the <b>DIIMOFs_*</b> values.</p>
<b>dwHow</b>	<p>Specifies how the <b>dwObj</b> member should be interpreted. If <b>dwObj</b> is <b>DIPROP_AXISMODE</b> or <b>DIPROP_BUFFERSIZE</b>, <b>dwHow</b> should be <b>DIPH_DEVICE</b>.</p>

**dwData**

Property-specific value being set or retrieved.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

## See Also

**IDirectInputDevice7::GetProperty**, **IDirectInputDevice7::SetProperty**

# DIPROPGUIDANDPATH

The **DIPROPGUIDANDPATH** structure is used to access properties whose values represent a GUID and a path.

```
typedef struct DIPROPGUIDANDPATH {
    DIPROPHEADER diph;
    GUID         guidClass;
    WCHAR        wszPath[MAX_PATH];
} DIPROPGUIDANDPATH, *LPDIPROPGUIDANDPATH;
```

## Members

### diph

**DIPROPHEADER** structure that must be initialized as follows:

Member	Value
<b>dwSize</b>	<b>sizeof(DIPROPGUIDANDPATH)</b>
<b>dwHeaderSize</b>	<b>sizeof(DIPROPHEADER)</b>
<b>dwObj</b>	Identifier of the object. For devices, must be 0
<b>dwHow</b>	How the <b>dwObj</b> member should be interpreted.

### guidClass

Class GUID for the object.

### wszPath

Returned path for the object. This is a Unicode string.

## Remarks

The **DIPROP\_GUIDANDPATH** property associated with the **DIPROPGUIDANDPATH** structure allows advanced applications to perform operations on a HID that are not supported by DirectInput.

The application calls the **IDirectInputDevice7::GetProperty** method with **DIPROP\_GUIDANDPATH** as the *rguidProp* parameter. The class GUID of the device is returned in the **guidClass** member of the **DIPROPGUIDANDPATH** structure, and the device interface path is returned in the **wszPath** member. The application can then call the **CreateFile** function on this path to access the device directly.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

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

**Header:** Declared in dinput.h.

# DIPROPHEADER

**DIPROPHEADER** is a generic structure placed at the beginning of all property structures.

```
typedef struct DIPROPHEADER {
    DWORD dwSize;
    DWORD dwHeaderSize;
    DWORD dwObj;
    DWORD dwHow;
} DIPROPHEADER, *LPDIPROPHEADER;
```



```
typedef const DIPROPHEADER *LPCDIPROPHEADER;
```

## Members

### dwSize

Size of the enclosing structure. This member must be initialized before the structure is used.

### dwHeaderSize

Size of the **DIPROPHEADER** structure.

### dwObj

Object for which the property is to be accessed. The value set for this member depends on the value specified in the **dwHow** member.

### dwHow

Value specifying how the **dwObj** member should be interpreted. This value can be one of the following:

Value	Meaning
DIPH_DEVICE	The <b>dwObj</b> member must be 0.
DIPH_BYOFFSET	The <b>dwObj</b> member is the offset into the current data format of the object whose property is being accessed.
DIPH_BYID	The <b>dwObj</b> member is the object type/instance identifier. This identifier is returned in the <b>dwType</b> member of the <b>DIDEVICEOBJECTINSTANCE</b> structure returned from a previous call to the <b>IDirectInputDevice7::EnumObjects</b> member.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

## DIPROP\_RANGE

The **DIPROP\_RANGE** structure contains information about the range of an object within a device. This structure is used with the **DIPROP\_RANGE** flag set in the **IDirectInputDevice7::GetProperty** and **IDirectInputDevice7::SetProperty** methods.

```
typedef struct DIPROP_RANGE {
    DIPROPHEADER diph;
    LONG         lMin;
    LONG         lMax;
} DIPROP_RANGE, *LPDIPROP_RANGE;
```

```
typedef const DIPOPRANGE *LPCDIPOPRANGE;
```

## Members

### diph

**DIPROPHEADER** structure that must be initialized as follows:

Member	Value
<b>dwSize</b>	<b>sizeof(DIPOPRANGE)</b>
<b>dwHeaderSize</b>	<b>sizeof(DIPROPHEADER)</b>
<b>dwObj</b>	Identifier of the object whose range is being retrieved or set.
<b>dwHow</b>	How the <b>dwObj</b> member should be interpreted.

### lMin

Lower limit of the range. If the range of the device is unrestricted, this value is **DIPOPRANGE\_NOMIN** when the **IDirectInputDevice7::GetProperty** method returns.

### lMax

Upper limit of the range. If the range of the device is unrestricted, this value is **DIPOPRANGE\_NOMAX** when the **IDirectInputDevice7::GetProperty** method returns.

## Remarks

The range values for devices whose ranges are unrestricted wrap around.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in `dinput.h`.

## See Also

**IDirectInputDevice7::GetProperty**, **IDirectInputDevice7::SetProperty**

# DIPROPSTRING

The **DIPROPSTRING** structure is used to access string properties.

```
typedef struct {
    DIPROPHEADER diph;
    WCHAR wsz;
} DIPROPSTRING;
```

## Members

### diph

**DIPROPHEADER** structure that must be initialized as follows:

Member	Value
<b>dwSize</b>	<b>sizeof(DIPROPSTRING)</b>
<b>dwHeaderSize</b>	<b>sizeof(DIPROPHEADER)</b>
<b>dwObj</b>	Identifier of the object whose property is being retrieved or set.
<b>dwHow</b>	How the <b>dwObj</b> member should be interpreted.

### wsz

String itself. This is a Unicode string.

## Remarks

The **DIPROP\_INSTANCENAME** and **DIPROP\_PRODUCTNAME** properties associated with the **DIPROPSTRING** structure allows advanced applications to perform operations on an HID that are not supported by DirectInput.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

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

**Header:** Declared in `dinput.h`.

## DIRAMPFORCE

The **DIRAMPFORCE** structure contains type-specific information for effects that are marked as **DIEFT\_RAMPFORCE**. The structure describes a ramp force effect.

A pointer to a single **DIRAMPFORCE** structure for an effect is passed in the **lpvTypeSpecificParams** member of the **DIEFFECT** structure.

```
typedef struct DIPROP_RANGE {
    LONG IStart;
    LONG IEnd;
} DIRAMPFORCE, *LPDIRAMPFORCE;

typedef const DIRAMPFORCE *LPCDIRAMPFORCE;
```

## Members

### IStart

Magnitude at the start of the effect, in the range from –10,000 through 10,000.

### IEnd

Magnitude at the end of the effect, in the range from –10,000 through 10,000.

## Remarks

The **dwDuration** for a ramp force effect cannot be INFINITE.

## Requirements

**Windows NT/2000:** Requires Windows 2000.

**Windows 95/98:** Requires Windows 95 or later. Available as a redistributable for Windows 95.

**Header:** Declared in dinput.h.

## Device Constants

This section is a reference for constants used to interpret data for keys, buttons, and axes.

- Keyboard Device Constants
- DirectInput and Japanese Keyboards
- Mouse Device Constants
- Joystick Device Constants

## Keyboard Device Constants

Keyboard device constants, defined in Dinput.h, represent offsets within a keyboard device's data packet, a 256-byte array. The data at a given offset is associated with a keyboard key. Typically, these values are used in the **dwOfs** member of the **DIDEVICEOBJECTDATA**, **DIOBJECTDATAFORMAT** or **DIDEVICEOBJECTINSTANCE** structures, or as indices when accessing data within the array using array notation.

The standard keyboard device constants are the following (in ascending order):

Constant	Note
DIK_ESCAPE	
DIK_1	On main keyboard
DIK_2	On main keyboard
DIK_3	On main keyboard
DIK_4	On main keyboard
DIK_5	On main keyboard
DIK_6	On main keyboard
DIK_7	On main keyboard
DIK_8	On main keyboard

DIK_9	On main keyboard
DIK_0	On main keyboard
DIK_MINUS	On main keyboard
DIK_EQUALS	On main keyboard
DIK_BACK	The BACKSPACE key
DIK_TAB	
DIK_Q	
DIK_W	
DIK_E	
DIK_R	
DIK_T	
DIK_Y	
DIK_U	
DIK_I	
DIK_O	
DIK_P	
DIK_LBRACKET	The [ key
DIK_RBRACKET	The ] key
DIK_RETURN	ENTER key on main keyboard
DIK_LCONTROL	Left CTRL key
DIK_A	
DIK_S	
DIK_D	
DIK_F	
DIK_G	
DIK_H	
DIK_J	
DIK_K	
DIK_L	
DIK_SEMICOLON	
DIK_APOSTROPHE	
DIK_GRAVE	Grave accent (') key
DIK_LSHIFT	Left SHIFT key
DIK_BACKSLASH	
DIK_Z	
DIK_X	
DIK_C	
DIK_V	

DIK_B	
DIK_N	
DIK_M	
DIK_COMMA	
DIK_PERIOD	On main keyboard
DIK_SLASH	Forward slash on main keyboard
DIK_RSHIFT	Right SHIFT key
DIK_MULTIPLY	The * key on numeric keypad
DIK_LMENU	Left ALT key
DIK_SPACE	SPACEBAR
DIK_CAPITAL	CAPS LOCK key
DIK_F1	
DIK_F2	
DIK_F3	
DIK_F4	
DIK_F5	
DIK_F6	
DIK_F7	
DIK_F8	
DIK_F9	
DIK_F10	
DIK_NUMLOCK	
DIK_SCROLL	SCROLL LOCK
DIK_NUMPAD7	
DIK_NUMPAD8	
DIK_NUMPAD9	
DIK_SUBTRACT	MINUS SIGN on numeric keypad
DIK_NUMPAD4	
DIK_NUMPAD5	
DIK_NUMPAD6	
DIK_ADD	PLUS SIGN on numeric keypad
DIK_NUMPAD1	
DIK_NUMPAD2	
DIK_NUMPAD3	
DIK_NUMPAD0	
DIK_DECIMAL	PERIOD (decimal point) on numeric keypad
DIK_OEM_102	On British and German keyboards.
DIK_F11	

---

DIK_F12	
DIK_F13	
DIK_F14	
DIK_F15	
DIK_KANA	On Japanese keyboard
DIK_ABNT_C1	On numeric pad of Brazilian keyboards
DIK_CONVERT	On Japanese keyboard
DIK_NOCONVERT	On Japanese keyboard
DIK_YEN	On Japanese keyboard
DIK_ABNT_C2	On numeric pad of Brazilian keyboards
DIK_NUMPADEQUALS	On numeric keypad (NEC PC98)
DIK_PREVTRACK	Previous track; circumflex on Japanese keyboard
DIK_AT	On Japanese keyboard
DIK_COLON	On Japanese keyboard
DIK_UNDERLINE	On Japanese keyboard
DIK_KANJI	On Japanese keyboard
DIK_STOP	On Japanese keyboard
DIK_AX	On Japanese keyboard
DIK_UNLABELED	On Japanese keyboard
DIK_NEXTTRACK	Next track
DIK_NUMPADENTER	
DIK_RCONTROL	Right CTRL key
DIK_MUTE	
DIK_CALCULATOR	
DIK_PLAYPAUSE	
DIK_MEDIASTOP	
DIK_VOLUMEDOWN	
DIK_VOLUMEUP	
DIK_WEBHOME	
DIK_NUMPADCOMMA	COMMA on NEC PC98 numeric keypad
DIK_DIVIDE	Forward slash on numeric keypad
DIK_SYSRQ	
DIK_RMENU	Right ALT key
DIK_PAUSE	
DIK_HOME	
DIK_UP	UP ARROW
DIK_PRIOR	PAGE UP
DIK_LEFT	LEFT ARROW

---

DIK_RIGHT	RIGHT ARROW
DIK_END	
DIK_DOWN	DOWN ARROW
DIK_NEXT	PAGE DOWN
DIK_INSERT	
DIK_DELETE	
DIK_LWIN	Left Windows key
DIK_RWIN	Right Windows key
DIK_APPS	Application key
DIK_POWER	
DIK_SLEEP	
DIK_WAKE	
DIK_WEBSEARCH	
DIK_WEBFAVORITES	
DIK_WEBREFRESH	
DIK_WEBSTOP	
DIK_WEBFORWARD	
DIK_WEBBACK	
DIK_MYCOMPUTER	
DIK_MAIL	
DIK_MEDIASELECT	

The following alternate names are available:

<b>Alternate name</b>	<b>Regular name</b>	<b>Note</b>
DIK_BACKSPACE	DIK_BACK	BACKSPACE
DIK_NUMPADSTAR	DIK_MULTIPLY	* key on numeric keypad
DIK_LALT	DIK_LMENU	Left ALT
DIK_CAPSLOCK	DIK_CAPITAL	CAPSLOCK
DIK_CIRCUMFLEX	DIK_PREVTRACK	On Japanese keyboards.
DIK_NUMPADMINUS	DIK__SUBTRACT	Minus key on numeric keypad
DIK_NUMPADPLUS	DIK_ADD	Plus key on numeric keypad
DIK_NUMPADPERIOD	DIK_DECIMAL	Period key on numeric keypad
DIK_NUMPADSLASH	DIK__DIVIDE	Forward slash on numeric keypad
DIK_RALT	DIK_RMENU	Right ALT
DIK_UPARROW	DIK_UP	On arrow keypad
DIK_PGUP	DIK_PRIOR	On arrow keypad



---

DIK_LEFTARROW	DIK_LEFT	On arrow keypad
DIK_RIGHTARROW	DIK_RIGHT	On arrow keypad
DIK_DOWNARROW	DIK_DOWN	On arrow keypad
DIK_PGDN	DIK_NEXT	On arrow keypad

For information on Japanese keyboards, see DirectInput and Japanese Keyboards.

## DirectInput and Japanese Keyboards

There are substantial differences between Japanese and U.S. keyboards. The following chart lists the additional keys that are available on each type of Japanese keyboard. It also lists the keys that are available on U.S. keyboards but are missing on the various Japanese keyboards.

On some NEC PC-98 keyboards, the DIK\_CAPSLOCK and DIK\_KANA keys are toggle buttons and not push buttons. These generate a down event when first pressed, then generate an up event when pressed a second time.

Keyboard	Additional keys	Missing keys
DOS/V 106 Keyboard, NEC PC-98 106 Keyboard	DIK_AT, DIK_CIRCUMFLEX, DIK_COLON, DIK_CONVERT, DIK_KANA, DIK_KANJI, DIK_NOCONVERT, DIK_YEN	DIK_APOSTROPHE, DIK_EQUALS, DIK_GRAVE
NEC PC-98 Standard Keyboard, NEC PC-98 Laptop Keyboard	DIK_AT, DIK_CIRCUMFLEX, DIK_COLON, DIK_F13, DIK_F14, DIK_F15, DIK_KANA, DIK_KANJI, DIK_NOCONVERT, DIK_NUMPADCOMMA, DIK_NUMPADEQUALS, DIK_STOP, DIK_UNDERLINE, DIK_YEN	DIK_APOSTROPHE, DIK_BACKSLASH, DIK_EQUALS, DIK_GRAVE, DIK_NUMLOCK, DIK_NUMPADENTER, DIK_RCONTROL, DIK_RMENU, DIK_RSHIFT, DIK_SCROLL
AX Keyboard	DIK_AX, DIK_CONVERT, DIK_KANJI, DIK_NOCONVERT, DIK_YEN	DIK_RCONTROL, DIK_RMENU

J-3100 Keyboard

DIK\_KANA, DIK\_KANJI, DIK\_RCONTROL,  
DIK\_NOLABEL, DIK\_YEN DIK\_RMENU

## Mouse Device Constants

Mouse device constants, defined in `Dinput.h`, represent offsets within a mouse device's data packet, the **DIMOUSESTATE** or **DIMOUSESTATE2** structure. The data at a given offset is associated with a device object (button or axis). Typically, these values are used in the **dwOfs** member of the **DIDEVICEOBJECTDATA**, **DIOBJECTDATAFORMAT** or **DIDEVICEOBJECTINSTANCE** structures.

The mouse device constants are the following:

DIMOFS_BUTTON0	Offset of the data representing the state of mouse button 0.
DIMOFS_BUTTON1	Offset of the data representing the state of mouse button 1.
DIMOFS_BUTTON2	Offset of the data representing the state of mouse button 2.
DIMOFS_BUTTON3	Offset of the data representing the state of mouse button 3.
DIMOFS_BUTTON4	Offset of the data representing the state of mouse button 4. ( <b>DIMOUSESTATE2</b> only.)
DIMOFS_BUTTON5	Offset of the data representing the state of mouse button 5. ( <b>DIMOUSESTATE2</b> only.)
DIMOFS_BUTTON6	Offset of the data representing the state of mouse button 6. ( <b>DIMOUSESTATE2</b> only.)
DIMOFS_BUTTON7	Offset of the data representing the state of mouse button 7. ( <b>DIMOUSESTATE2</b> only.)
DIMOFS_X	Offset of the data representing the mouse's position on the x-axis.
DIMOFS_Y	Offset of the data representing the mouse's position on the y-axis.
DIMOFS_Z	Offset of the data representing the mouse's position on the z-axis.

## Joystick Device Constants

Joystick device constants represent offsets within a joystick device's data packet, the **DIJOYSTATE** structure. The data at a given offset is associated with a device object; that is, a button or axis. Typically, these values are used in the **dwOfs** member of the **DIDEVICEOBJECTDATA**, **DIOBJECTDATAFORMAT** or **DIDEVICEOBJECTINSTANCE** structures.

The following macros return a constant indicating the offset of the data for a particular button or axis relative to the beginning of the **DIJOYSTATE** structure:

DIJOFS_BUTTON0 to DIJOFS_BUTTON31 or DIJOFS_BUTTON( <i>n</i> )	A button.
DIJOFS_POV( <i>n</i> )	A point-of-view indicator.
DIJOFS_RX	The x-axis rotation.
DIJOFS_RY	The y-axis rotation.
DIJOFS_RZ	The z-axis rotation (rudder).
DIJOFS_X	The x-axis.
DIJOFS_Y	The y-axis.
DIJOFS_Z	The z-axis.
DIJOFS_SLIDER( <i>n</i> )	A slider axis.

## Return Values

This table lists the **HRESULT** values that can be returned by DirectInput methods and functions. Errors are represented by negative values and cannot be combined.

For a list of the error values each method or function can return, see the individual descriptions. Lists of error codes in the documentation are necessarily incomplete. For example, any DirectInput method can return **DIERR\_OUTOFMEMORY** even though the error code is not explicitly listed as a possible return value in the documentation for that method.

### DI\_BUFFEROVERFLOW

The device buffer overflowed and some input was lost. This value is equal to the **S\_FALSE** standard COM return value.

### DI\_DOWNLOADSKIPPED

The parameters of the effect were successfully updated, but the effect could not be downloaded because the associated device was not acquired in exclusive mode.

### DI\_EFFECTRESTARTED

The effect was stopped, the parameters were updated, and the effect was restarted.

### DI\_NOEFFECT

The operation had no effect. This value is equal to the **S\_FALSE** standard COM return value.

### DI\_NOTATTACHED

The device exists but is not currently attached. This value is equal to the **S\_FALSE** standard COM return value.

### DI\_OK

The operation completed successfully. This value is equal to the **S\_OK** standard COM return value.

**DI\_POLLEDDEVICE**

The device is a polled device. As a result, device buffering does not collect any data and event notifications is not signaled until the **IDirectInputDevice7::Poll** method is called.

**DI\_PROPNOEFFECT**

The change in device properties had no effect. This value is equal to the **S\_FALSE** standard COM return value.

**DI\_TRUNCATED**

The parameters of the effect were successfully updated, but some of them were beyond the capabilities of the device and were truncated to the nearest supported value.

**DI\_TRUNCATEDANDRESTARTED**

Equal to **DI\_EFFECTRESTARTED** | **DI\_TRUNCATED**.

**DIERR\_ACQUIRED**

The operation cannot be performed while the device is acquired.

**DIERR\_ALREADYINITIALIZED**

This object is already initialized

**DIERR\_BADDRIVERVER**

The object could not be created due to an incompatible driver version or mismatched or incomplete driver components.

**DIERR\_BETADIRECTINPUTVERSION**

The application was written for an unsupported prerelease version of DirectInput.

**DIERR\_DEVICEFULL**

The device is full.

**DIERR\_DEVICENOTREG**

The device or device instance is not registered with DirectInput. This value is equal to the **REGDB\_E\_CLASSNOTREG** standard COM return value.

**DIERR\_EFFECTPLAYING**

The parameters were updated in memory but were not downloaded to the device because the device does not support updating an effect while it is still playing.

**DIERR\_HASEFFECTS**

The device cannot be reinitialized because there are still effects attached to it.

**DIERR\_GENERIC**

An undetermined error occurred inside the DirectInput subsystem. This value is equal to the **E\_FAIL** standard COM return value.

**DIERR\_HANDLEEXISTS**

The device already has an event notification associated with it. This value is equal to the **E\_ACCESSDENIED** standard COM return value.

**DIERR\_INCOMPLETEEFFECT**

The effect could not be downloaded because essential information is missing. For example, no axes have been associated with the effect, or no type-specific information has been supplied.

**DIERR\_INPUTLOST**

Access to the input device has been lost. It must be reacquired.

**DIERR\_INVALIDPARAM**

An invalid parameter was passed to the returning function, or the object was not in a state that permitted the function to be called. This value is equal to the E\_INVALIDARG standard COM return value.

**DIERR\_MOREDATA**

Not all the requested information fit into the buffer.

**DIERR\_NOAGGREGATION**

This object does not support aggregation.

**DIERR\_NOINTERFACE**

The specified interface is not supported by the object. This value is equal to the E\_NOINTERFACE standard COM return value.

**DIERR\_NOTACQUIRED**

The operation cannot be performed unless the device is acquired.

**DIERR\_NOTBUFFERED**

The device is not buffered. Set the DIPROP\_BUFFERSIZE property to enable buffering.

**DIERR\_NOTDOWNLOADED**

The effect is not downloaded.

**DIERR\_NOTEXCLUSIVEACQUIRED**

The operation cannot be performed unless the device is acquired in DISCL\_EXCLUSIVE mode.

**DIERR\_NOTFOUND**

The requested object does not exist.

**DIERR\_NOTINITIALIZED**

This object has not been initialized.

**DIERR\_OBJECTNOTFOUND**

The requested object does not exist.

**DIERR\_OLDDIRECTINPUTVERSION**

The application requires a newer version of DirectInput.

**DIERR\_OTHERAPPHASPRIO**

Another application has a higher priority level, preventing this call from succeeding. This value is equal to the E\_ACCESSDENIED standard COM return value. This error can be returned when an application has only foreground access to a device but is attempting to acquire the device while in the background.

**DIERR\_OUTOFMEMORY**

The DirectInput subsystem could not allocate sufficient memory to complete the call. This value is equal to the E\_OUTOFMEMORY standard COM return value.

**DIERR\_READONLY**

The specified property cannot be changed. This value is equal to the E\_ACCESSDENIED standard COM return value.

**DIERR\_REPORTFULL**

More information was requested to be sent than can be sent to the device.

**DIERR\_UNPLUGGED**

The operation could not be completed because the device is not plugged in.

**DIERR\_UNSUPPORTED**

The function called is not supported at this time. This value is equal to the E\_NOTIMPL standard COM return value.

**E\_HANDLE**

The HWND parameter is not a valid top-level window that belongs to the process.

**E\_PENDING**

Data is not yet available.

## DirectInput Visual Basic Reference

Reference material for the DirectInput Visual Basic application programming interface is divided into the following categories:

- Classes
- Types
- Enumerations
- Keyboard Keys
- Error Codes

### Classes

This section contains references for methods of the following DirectInput classes:

- **DirectInput**
- **DirectInputDevice**
- **DirectInputDeviceInstance**
- **DirectInputDeviceObjectInstance**
- **DirectInputEffect**
- **DirectInputEnumDeviceObjects**
- **DirectInputEnumDevices**
- **DirectInputEnumEffects**

## DirectInput

# The **DirectInput** class represents the DirectInput system. An application should have a single object of this class, which is used to enumerate available devices, create

# IDH\_DirectInput\_dinput\_vb

devices, and retrieve the status of devices, as well as to invoke an instance of the Windows Control Panel.

The **DirectInput** object is obtained by using the **DirectX7.DirectInputCreate** method.

The **DirectInput** class has the following methods:

<b>Device Management</b>	<b>CreateDevice</b>
	<b>GetDeviceStatus</b>
	<b>GetDIEnumDevices</b>
<b>Miscellaneous</b>	<b>RunControlPanel</b>

## DirectInput.CreateDevice

# The **DirectInput.CreateDevice** method creates and initializes an instance of a device based on a given GUID.

*object*.**CreateDevice(guid As String) As DirectInputDevice**

### Parameters

*object*

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

*guid*

The instance GUID for the desired input device. The GUID is retrieved from the **DirectInputDeviceInstance** object returned by the **DirectInputEnumDevices.GetItem** method, or it can be one of the following strings:

GUID\_SysKeyboard

The default system keyboard.

GUID\_SysMouse

The default system mouse.

### Return Values

If the method succeeds, a **DirectInputDevice** object is returned.

### Error Codes

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

DIERR\_DEVICENOTREG

DIERR\_INVALIDPARAM

DIERR\_NOINTERFACE

---

# IDH\_DirectInput.CreateDevice\_dinput\_vb

---

DIERR\_OUTOFMEMORY

## See Also

Using GUIDs

## DirectInput.GetDeviceStatus

# The **DirectInput.GetDeviceStatus** method determines whether a device is attached to the system.

*object*.**GetDeviceStatus**(*guid* As String) As Boolean

## Parameters

*object*

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

*guid*

The instance GUID for the desired input device. The GUID is retrieved by using the **DirectInputDeviceInstance.GetGuidInstance** method on the object returned by **DirectInputEnumDevices.GetItem** method, or it can be one of the following strings:

GUID\_SysKeyboard

The default system keyboard.

GUID\_SysMouse

The default system mouse.

## Return Values

The method returns True if the device is attached, and False otherwise.

## Error Codes

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

DIERR\_GENERIC

DIERR\_INVALIDPARAM

## See Also

Using GUIDs

---

# IDH\_DirectInput.GetDeviceStatus\_dinput\_vb



## DirectInput.GetDIEnumDevices

# The **DirectInput.GetDIEnumDevices** method returns a **DirectInputEnumDevices** object which is used to enumerate devices that are either currently attached or could be attached to the computer.

```
object.GetDIEnumDevices( _  
    deviceType As CONST_DIDEVTYPE, _  
    flags As CONST_DIENUMDEVICESFLAGS) _  
    As DirectInputEnumDevices
```

### Parameters

*object*

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

*deviceType*

Device type filter. If this parameter is 0, all device types are enumerated.

Otherwise, it is one of the following DIDEVTYPE\_\* constants of the **CONST\_DIDEVTYPE** enumeration, indicating the device type that should be enumerated.

DIDEVTYPE\_MOUSE

A mouse or mouse-like device (such as a trackball).

DIDEVTYPE\_KEYBOARD

A keyboard or keyboard-like device.

DIDEVTYPE\_JOYSTICK

A joystick or similar device, such as a steering wheel.

DIDEVTYPE\_DEVICE

A device that does not fall into the previous categories.

*flags*

Flag value that specifies the scope of the enumeration. This parameter can be one of the following constants of the **CONST\_DIENUMDEVICESFLAGS** enumeration. If this flag is 0 (DIEDFL\_ALLDEVICES), then all installed devices are enumerated. If it is DIEDFL\_ATTACHEDONLY, only devices actually attached to the system are enumerated.

### Return Values

If the method succeeds, the method returns a **DirectInputEnumDevices** object.

### Error Codes

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

---

# IDH\_DirectInput.GetDIEnumDevices\_dinput\_vb

## Remarks

All installed devices can be enumerated, even if they are not present. For example, a joystick may be installed on the system but not currently plugged into the computer.

If a single piece of hardware can function as more than one **DirectInput** device type, it will be returned for each device type it supports. For example, a keyboard with a built-in mouse will be enumerated as a keyboard and as a mouse. The product GUID would be the same for each device, however.

## DirectInput.RunControlPanel

# The **DirectInput.RunControlPanel** method runs the Windows Control Panel to allow the user to install a new input device or modify configurations.

*object*.**RunControlPanel**(*hwndOwner* As Long)

## Parameters

*object*

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

*hwndOwner*

Handle to the window to be used as the parent window for the subsequent user interface. If this parameter is 0, no parent window is used.

## Error Codes

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

## See Also

**DirectInputDevice.RunControlPanel**

## DirectInputDevice

# Applications use the methods of the **DirectInputDevice** class to gain and release access to **DirectInput** devices, manage device properties and information, set behavior, perform initialization, and invoke a device's property sheet.

The **DirectInputDevice** object is obtained by using the **DirectInput.CreateDevice** method.

The methods of the **DirectInputDevice** class can be organized into the following groups.

**Access**

**Acquire**

---

# IDH\_DirectInput.RunControlPanel\_dinput\_vb

# IDH\_DirectInputDevice\_dinput\_vb

---

	<b>SetCooperativeLevel</b>
	<b>Unacquire</b>
<b>Objects</b>	<b>GetDeviceObjectsEnum</b>
	<b>GetObjectInfo</b>
<b>Properties</b>	<b>GetCapabilities</b>
	<b>GetDeviceInfo</b>
	<b>GetProperty</b>
	<b>SetCommonDataFormat</b>
	<b>SetDataFormat</b>
	<b>SetProperty</b>
<b>Retrieving Data</b>	<b>GetDeviceData</b>
	<b>GetDeviceState</b>
	<b>GetDeviceStateJoystick</b>
	<b>GetDeviceStateJoystick2</b>
	<b>GetDeviceStateKeyboard</b>
	<b>GetDeviceStateMouse</b>
	<b>Poll</b>
	<b>SetEventNotification</b>
<b>Force Feedback</b>	<b>CreateCustomEffect</b>
	<b>CreateEffect</b>
	<b>GetEffectsEnum</b>
	<b>GetForceFeedbackState</b>
	<b>SendForceFeedbackCommand</b>
<b>Miscellaneous</b>	<b>RunControlPanel</b>
	<b>SendDeviceData</b>

## DirectInputDevice.Acquire

# The **DirectInputDevice.Acquire** method obtains access to the input device.

*object.Acquire()*

### Parameters

*object*

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

---

# IDH\_DirectInputDevice.Acquire\_dinput\_vb

## Error Codes

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

DIERR\_INVALIDPARAM  
DIERR\_OTHERAPPHASPRIO

## Remarks

Before a device can be acquired, a data format must be set by using the **DirectInputDevice.SetDataFormat** or **DirectInputDevice.SetCommonDataFormat** method.

A device must be acquired before input data can be retrieved from it.

## See Also

**DirectInputDevice.Unacquire**

## DirectInputDevice.CreateCustomEffect

# The **DirectInputDevice.CreateCustomEffect** method creates a force-feedback effect consisting of a series of constant forces of fixed duration.

*object*.**CreateCustomEffect**(*effectinfo* As **DIEFFECT**, \_  
    *channels* As Long, *samplePeriod* As Long, \_  
    *nSamples* As Long, *sampledata()* As Long) \_  
    As **DirectInputEffect**

## Parameters

*object*

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

*effectinfo*

**DIEFFECT** type containing general parameters of the effect.

*channels*

The number of channels (axes) affected by this force. Must be 1 or 2.

If there is only a single channel, then the effect will be rotated in the direction specified by the **x** member of the **DIEFFECT** type. Not all devices support rotation of custom effects.

If there is more than one channel, the first channel is applied to the x-axis and the second to the y-axis. Rotation is not allowed.

*samplePeriod*

The sample period in microseconds. See Remarks.

*nSamples*

# IDH\_DirectInputDevice.CreateCustomEffect\_dinput\_vb

Number of elements in the *sampledata* array.

#### *sampledata*

Array of magnitudes. If *channels* is greater than 1, then the values are interleaved. For example, if *channels* is 2, then the first element of the array is assigned to the x-axis, the second to the y-axis, the third to the x-axis, and so on.

## Remarks

In theory, *samplePeriod* is the length of time for which each magnitude in *sampledata* is valid, whereas **DIEFFECT.ISamplePeriod** is the length of time between samplings of the data (and the minimum time between changes in magnitude). Since each element in the array needs to be sampled exactly once on each iteration through the array, and some drivers ignore *samplePeriod* in any case, it is best to make the values of **ISamplePeriod** and *samplePeriod* identical.

## See Also

Custom Forces

# DirectInputDevice.CreateEffect

# The **DirectInputDevice.CreateEffect** method creates a force-feedback effect. If the device is currently acquired at the exclusive cooperative level, the effect is also downloaded.

*object*.**CreateEffect**(*effectGuid* As String, \_  
*effectInfo* As DIEFFECT) As DirectInputEffect

## Parameters

#### *object*

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

#### *effectGuid*

String representation of a GUID for an effect recognized by the hardware driver, or one of the following aliases for standard effects:

GUID\_ConstantForce  
 GUID\_RampForce  
 GUID\_Square  
 GUID\_Sine  
 GUID\_Triangle  
 GUID\_SawtoothUp  
 GUID\_SawtoothDown  
 GUID\_Spring  
 GUID\_Damper  
 GUID\_Inertia  
 GUID\_Friction

---

# IDH\_DirectInputDevice.CreateEffect\_dinput\_vb

*effectinfo*

**DIEFFECT** structure containing the parameters for the effect.

## Return Values

If the method succeeds, the method returns a **DirectInputEffect** object.

## Error Codes

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

DIERR\_DEVICENOTREG  
DIERR\_DEVICEFULL  
DIERR\_INVALIDPARAM  
DIERR\_NOTINITIALIZED

## Remarks

If no error is raised, the effect was created and the parameters of the effect were updated, but the effect was not necessarily downloaded. In order for it to be downloaded, the device must be acquired in exclusive mode.

## See Also

**DirectInputEffect.Download**, **DirectInputEffect.Start**

## DirectInputDevice.GetCapabilities

# The **DirectInputDevice.GetCapabilities** method obtains the capabilities of the **DirectInputDevice** object.

*object*.**GetCapabilities**(*caps* As **DIDEVCAPS**)

## Parameters

*object*

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

*caps*

A **DIDEVCAPS** type to be filled with the device capabilities.

## Error Codes

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

---

# IDH\_DirectInputDevice.GetCapabilities\_dinput\_vb

## DirectInputDevice.GetDeviceData

# The **DirectInputDevice.GetDeviceData** method retrieves buffered data from the device.

```
object.GetDeviceData( _  
    deviceObjectDataArray() As DIDEVICEOBJECTDATA, _  
    flags As CONST_DIDGDDFLAGS) As Long
```

### Parameters

*object*

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

*deviceObjectDataArray()*

Array of **DIDEVICEOBJECTDATA** types to receive the buffered data.

*flags*

Flags that control the manner in which data is obtained. This value can be 0 or one of the constants of the **CONST\_DIDGDDFLAGS** enumeration.

### Return Values

If it succeeds, the method returns the number of buffered data elements actually returned in *deviceObjectDataArray*.

### Error Codes

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

```
DI_BUFFEROVERFLOW  
DIERR_INPUTLOST  
DIERR_INVALIDPARAM  
DIERR_NOTACQUIRED  
DIERR_NOTBUFFERED
```

### Remarks

Before device data can be obtained, you must set the data format by using the **DirectInputDevice.SetDataFormat** method, set the buffer size by using **DirectInputDevice.SetProperty** method, and acquire the device by using the **DirectInputDevice.Acquire** method.

You can use this method to retrieve one or more input events from the buffer you created by using **SetProperty**. You do not have to retrieve all pending events with a single call. You can, for example, pass in a *deviceObjectDataArray()* consisting of a single element and loop on **GetDeviceData** till no more data is returned.

---

```
# IDH_DirectInputDevice.GetDeviceData_dinput_vb
```

If the buffer overflows, all pending data is lost and the `DI_BUFFEROVERFLOW` error is raised.

## See Also

**DirectInputDevice.Poll**, Polling and Events

## DirectInputDevice.GetDeviceInfo

# The **DirectInputDevice.GetDeviceInfo** method obtains information about the device's identity.

*object*.**GetDeviceInfo()** As **DirectInputDeviceInstance**

### Parameters

*object*

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

### Return Values

If it succeeds, the method returns a **DirectInputDeviceInstance** object

### Error Codes

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

## DirectInputDevice.GetDeviceObjectsEnum

# The **DirectInputDevice.GetDeviceObjectsEnum** method returns a **DirectInputEnumDeviceObjects** object which is used to enumerate the objects available on a device. A device object is typically an axis or a button.

*object*.**GetDeviceObjectsEnum**(  
    *flags* As **CONST\_DIDFTFLAGS**)  
    As **DirectInputEnumDeviceObjects**

### Parameters

*object*

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

*flags*

Flags specifying the type of object to be enumerated. Can be one or more of the members of the **CONST\_DIDFTFLAGS** enumeration.

---

# **IDH\_DirectInputDevice.GetDeviceInfo\_dinput\_vb**

# **IDH\_DirectInputDevice.GetDeviceObjectsEnum\_dinput\_vb**



## Return Values

If the method succeeds, the return value is a **DirectInputEnumDeviceObjects** object.

## Error Codes

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

## Remarks

The DIDFT\_FFACTUATOR and DIDFT\_FFEFFECTTRIGGER flags restrict enumeration to objects that meet all the criteria defined by the included flags. For all the other flags, an object is enumerated if it meets the criterion defined by any included flag in this category. For example, (DIDFT\_FFACTUATOR **Or** DIDFT\_FFEFFECTTRIGGER) restricts enumeration to force-feedback trigger objects, and (DIDFT\_FFEFFECTTRIGGER **Or** DIDFT\_TGLBUTTON **Or** DIDFT\_PSHBUTTON) restricts enumeration to buttons of any kind that can be used as effect triggers.

## DirectInputDevice.GetDeviceState

# The **DirectInputDevice.GetDeviceState** method retrieves immediate data for a device other than a standard keyboard, mouse, or joystick.

*object*.GetDeviceState(*cb As Long, state As Any*)

## Parameters

*object*

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

*cb*

Size of the array whose first element is passed as *state*.

*state*

First element of an array to receive device state information.

## Error Codes

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

DIERR\_INPUTLOST  
DIERR\_INVALIDPARAM  
DIERR\_NOTACQUIRED  
E\_PENDING

---

# IDH\_DirectInputDevice.GetDeviceState\_dinput\_vb

## Remarks

Before device data can be obtained, you must set the cooperative level by using the **DirectInputDevice.SetCooperativeLevel** method, then set the data format by using **DirectInputDevice.SetDataFormat**, and acquire the device by using the **DirectInputDevice.Acquire** method.

## See Also

**DirectInputDevice.GetDeviceStateJoystick**,  
**DirectInputDevice.GetDeviceStateJoystick2**,  
**DirectInputDevice.GetDeviceStateKeyboard**,  
**DirectInputDevice.GetDeviceStateMouse**, **DirectInputDevice.SetDataFormat**,  
 Buffered and Immediate Data

# DirectInputDevice.GetDeviceStateJoystick

# The **DirectInputDevice.GetDeviceStateJoystick** method retrieves instantaneous data from a joystick device.

*object*.GetDeviceStateJoystick(*state* As DIJOYSTATE)

## Parameters

*object*

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

*state*

A **DIJOYSTATE** type that receives the current state of the device.

## Error Codes

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

DIERR\_INPUTLOST  
 DIERR\_INVALIDPARAM  
 DIERR\_NOTACQUIRED  
 E\_PENDING

## Remarks

Before device data can be obtained, you must set the cooperative level by using the **DirectInputDevice.SetCooperativeLevel** method, then set the data format by using

---

# IDH\_DirectInputDevice.GetDeviceStateJoystick\_dinput\_vb

**DirectInputDevice.SetCommonDataFormat**, and acquire the device by using the **DirectInputDevice.Acquire** method.

## See Also

**DirectInputDevice.Poll**

## DirectInputDevice.GetDeviceStateJoystick2

# The **DirectInputDevice.GetDeviceStateJoystick2** method retrieves instantaneous data from a joystick device with extended capabilities.

*object*.**GetDeviceStateJoystick2**(*state* As **DIJOYSTATE2**)

## Parameters

*object*

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

*state*

A **DIJOYSTATE2** type that receives the current state of the device. The format of the data is established by a prior call to the **DirectInputDevice.SetDataFormat** method.

## Error Codes

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

DIERR\_INPUTLOST  
DIERR\_INVALIDPARAM  
DIERR\_NOTACQUIRED  
E\_PENDING

## Remarks

Before device data can be obtained, you must set the cooperative level by using the **DirectInputDevice.SetCooperativeLevel** method, then set the data format by using **DirectInputDevice.SetCommonDataFormat**, and acquire the device by using the **DirectInputDevice.Acquire** method.

## See Also

**DirectInputDevice.Poll**, Polling and Events

---

# IDH\_DirectInputDevice.GetDeviceStateJoystick2\_dinput\_vb

---

## DirectInputDevice.GetDeviceStateKeyboard

# The **DirectInputDevice.GetDeviceStateKeyboard** method retrieves instantaneous data from a keyboard device.

*object*.GetDeviceStateKeyboard(*state* As DIKEYBOARDSTATE)

### Parameters

*object*

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

*state*

A **DIKEYBOARDSTATE** type that receives the current state of the device.

### Error Codes

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

DIERR\_INPUTLOST  
DIERR\_INVALIDPARAM  
DIERR\_NOTACQUIRED  
E\_PENDING

### Remarks

Before device data can be obtained, you must set the cooperative level by using the **DirectInputDevice.SetCooperativeLevel** method, then set the data format by using **DirectInputDevice.SetCommonDataFormat**, and acquire the device by using the **DirectInputDevice.Acquire** method.

### See Also

**DirectInputDevice.Poll**

## DirectInputDevice.GetDeviceStateMouse

# The **DirectInputDevice.GetDeviceStateMouse** method retrieves instantaneous data from a mouse device.

*object*.GetDeviceStateMouse(*state* As DIMOUSESTATE)

---

# IDH\_DirectInputDevice.GetDeviceStateKeyboard\_dinput\_vb

# IDH\_DirectInputDevice.GetDeviceStateMouse\_dinput\_vb

## Parameters

*object*

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

*state*

A **DIMOUSESTATE** type that receives the current state of the device.

## Error Codes

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

DIERR\_INPUTLOST  
DIERR\_INVALIDPARAM  
DIERR\_NOTACQUIRED  
E\_PENDING

## Remarks

Before device data can be obtained, you must set the cooperative level by using the **DirectInputDevice.SetCooperativeLevel** method, then set the data format by using **DirectInputDevice.SetCommonDataFormat**, and acquire the device by using the **DirectInputDevice.Acquire** method.

## See Also

**DirectInputDevice.Poll**

## DirectInputDevice.GetEffectsEnum

# The **DirectInputDevice.GetEffectsEnum** method enumerates force-feedback effects supported by the device, including standard effects as well as effects designed by the device manufacturer.

*object*.GetEffectsEnum( \_  
    *effType* As CONST\_DIEFTFLAGS) As DirectInputEnumEffects

## Parameters

*object*

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

*effType*

One of the following flags from the **CONST\_DIEFTFLAGS** enumeration specifying the type of effect to be enumerated:

---

# IDH\_DirectInputDevice.GetEffectsEnum\_dinput\_vb

DIEFT\_ALL  
DIEFT\_CONDITION  
DIEFT\_CONSTANTFORCE  
DIEFT\_CUSTOMFORCE  
DIEFT\_HARDWARE  
DIEFT\_PERIODIC  
DIEFT\_RAMPFORCE

## Return Values

The method returns a **DirectInputEnumEffects** object whose methods can be used to retrieve information about the effects.

## Error Codes

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

# DirectInputDevice.GetForceFeedbackState

# The **DirectInputDevice.GetForceFeedbackState** method retrieves the state of the device's force-feedback system.

*object*.GetForceFeedbackState() As CONST\_DIGFFSFLAGS

## Parameters

*object*

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

## Return Values

If it succeeds, the method returns flags from the **CONST\_DIGFFSFLAGS** enumeration that describe the current state of the device's force-feedback system.

Future versions of DirectInput may define additional flags. Applications should ignore any flags that are not currently defined.

## Error Codes

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

DIERR\_INPUTLOST  
DIERR\_INVALIDPARAM  
DIERR\_NOTEXCLUSIVEACQUIRED

---

# IDH\_DirectInputDevice.GetForceFeedbackState\_dinput\_vb

DIERR\_NOTINITIALIZED  
DIERR\_UNSUPPORTED

## Remarks

The device must be acquired at the exclusive cooperative level for this method to succeed.

## DirectInputDevice.GetObjectInfo

# The **DirectInputDevice.GetObjectInfo** method retrieves information about a device object such as a button or axis.

```
object.GetObjectInfo( _  
    Obj As Long, _  
    how As CONST_DIPHFLAGS) _  
    As DirectInputDeviceObjectInstance
```

## Parameters

*object*

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

*Obj*

Value that identifies the object whose information will be retrieved. The interpretation of this parameter depends on the value specified in the *how* parameter.

*how*

Value specifying how the *Obj* parameter should be interpreted. This value can be one of the constants of the **CONST\_DIPHFLAGS** enumeration.

## Return Values

The method returns a **DirectInputDeviceObjectInstance** object whose methods can be used to retrieve information about the object.

## Error Codes

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

DIERR\_INVALIDPARAM  
DIERR\_OBJECTNOTFOUND

---

# IDH\_DirectInputDevice.GetObjectInfo\_dinput\_vb

## DirectInputDevice.GetProperty

# The **DirectInputDevice.GetProperty** method retrieves information about the input device.

*object*.GetProperty( \_  
     *guid* As String, \_  
     *propertyInfo* As Any)

### Parameters

*object*

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

*guid*

Identifier of the property to be retrieved. The following properties are defined for an input device and can be passed as strings:

DIPROP\_AUTOCENTER

Specifies whether device objects are self-centering. See **DirectInputDevice.SetProperty** for more information.

DIPROP\_AXISMODE

Retrieves the axis mode. The retrieved value can be DIPROPAXISMODE\_ABS or DIPROPAXISMODE\_REL. (See the **CONST\_DINPUT** enumeration.)

DIPROP\_BUFFERSIZE

Retrieves the input-buffer size. The buffer size determines the amount of data that the buffer can hold between calls to the **DirectInputDevice.GetDeviceData** method before data is lost. This value may be set to 0 to indicate that the application will not be reading buffered data from the device.

DIPROP\_DEADZONE

Retrieves a value for the dead zone of a joystick, in the range 0 to 10,000, where 0 indicates there is no dead zone, 5,000 indicates that the dead zone extends over 50 percent of the physical range of the axis on both sides of center, and 10,000 indicates that the entire physical range of the axis is dead. When the axis is within the dead zone, it is reported as being at the center of its range.

DIPROP\_FFGAIN

Retrieves the gain of the device. See **DirectInputDevice.SetProperty** for more information.

DIPROP\_FFLOAD

Retrieves the memory load for the device. This setting applies to the entire device, rather than to any particular object, so the **IHow** member of the associated **DIPROPLONG** type must be DIPH\_DEVICE.

The **IData** member contains a value in the range 0 to 100, indicating the percentage of device memory in use.

# IDH\_DirectInputDevice.GetProperty\_dinput\_vb



**DIPROP\_GRANULARITY**

Retrieves the input granularity. Granularity represents the smallest distance the object will report movement. Most axis objects have a granularity of 1, meaning that all values are possible. Some axes may have a larger granularity. For example, the wheel axis on a mouse may have a granularity of 20, meaning that all reported changes in position will be multiples of 20. In other words, when the user turns the wheel slowly, the device reports a position of 0, then 20, then 40, and so on.

This is a read-only property; you cannot set its value by calling the **DirectInputDevice.SetProperty** method.

**DIPROP\_RANGE**

Retrieves the range of values an object can possibly report. The retrieved minimum and maximum values are set in the **IMin** and **IMax** members of the associated **DIPROP\_RANGE** type.

For some devices, this is a read-only property; you cannot set its value by calling the **DirectInputDevice.SetProperty** method.

**DIPROP\_SATURATION**

Retrieves a value for the saturation zones of a joystick, in the range 0 to 10,000. The saturation level is the point at which the axis is considered to be at its most extreme position. For example, if the saturation level is set to 9,500, then the axis reaches the extreme of its range when it has moved 95 percent of the physical distance from its center position (or from the dead zone).

*propertyInfo*

A **DIPROPLONG** type to receive a single value, or a **DIPROP\_RANGE** type to receive a pair of values for the property. The **IObj**, **IHow**, and **ISize** members of this type must be initialized before the method is called.

**Error Codes**

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

DIERR\_INVALIDPARAM  
DIERR\_OBJECTNOTFOUND  
DIERR\_UNSUPPORTED

**See Also**

**DirectInputDevice.SetProperty**

## DirectInputDevice.Poll

# The **DirectInputDevice.Poll** method makes data available from polled objects on a DirectInput device. If the device does not require polling, then calling this method has no effect. If a device that requires polling is not polled periodically, no new data will be received from the device. Calling this method causes DirectInput to update the device state, generate input events (if buffered data is enabled), and set notification events (if notification is enabled).

*object.Poll()*

### Parameters

*object*

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

### Error Codes

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

DIERR\_INPUTLOST  
DIERR\_NOTACQUIRED

### Remarks

Before a device data can be polled, the data format must be set by using the **DirectInputDevice.SetDataFormat** or **DirectInputDevice.SetCommonDataFormat** method, and the device must be acquired by using the **DirectInputDevice.Acquire** method.

## DirectInputDevice.RunControlPanel

# The **DirectInputDevice.RunControlPanel** method opens the Control Panel property sheet associated with this device. If the device does not have a property sheet associated with it, the default device property sheet is displayed.

*object.RunControlPanel(hwnd As Long)*

### Parameters

*object*

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

*hwnd*

Handle to the parent window. If this parameter is 0, no parent window is used.

---

# IDH\_DirectInputDevice.Poll\_dinput\_vb  
# IDH\_DirectInputDevice.RunControlPanel\_dinput\_vb

## Error Codes

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

DIERR\_INVALIDPARAM

## DirectInputDevice.SendDeviceData

# The **DirectInputDevice.SendDeviceData** method sends data to a device that accepts output. The device must be in an acquired state.

```
object.SendDeviceData(count As Long, _  
    data() As DIDEVICEOBJECTDATA, _  
    flags As CONST_DESDDFLAGS) As Long
```

## Parameters

*object*

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

*count*

Number of elements in *data*.

*data*

Array of **DIDEVICEOBJECTDATA** types containing the data to send to the device.

The **IOfs** field of each **DIDEVICEOBJECTDATA** type must contain the instance identifier (not the data offset) for the device object to which the data is directed. (See **DirectInputDeviceObjectInstance.GetType**.) The **ITimeStamp** and **ISequence** members must be 0.

*flags*

Flags controlling the manner in which data is sent. This may be 0 or the following value:

DISDD\_CONTINUE

The device data sent will be overlaid on the previously sent device data. See Remarks.

## Error Codes

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

DIERR\_INPUTLOST

DIERR\_NOTACQUIRED

DIERR\_REPORTFULL

---

# IDH\_DirectInputDevice.SendDeviceData\_dinput\_vb

---

DIERR\_UNPLUGGED

## Remarks

There is no guarantee that the individual data elements will be sent in a particular order. However, data sent by successive calls to **SendDeviceData** will not be interleaved. Furthermore, if multiple pieces of data are sent to the same object with a single call, it is unspecified which piece of data is sent.

Consider, for example, a device that can be sent data in packets, each packet describing two pieces of information, call them A and B. Suppose the application attempts to send three data elements: B = 2, A = 1, and B = 0.

The actual device will be sent a single packet. The A field of the packet will contain the value 1, and the B field of the packet will be either 2 or 0.

If the data must to be sent to the device exactly as specified, then three calls to **SendDeviceData** should be performed, each call sending one data element.

In response to the first call, the device will be sent a packet where the A field is blank and the B field contains the value 2.

In response to the second call, the device will be sent a packet where the A field contains the value 1, and the B field is blank.

Finally, in response to the third call, the device will be sent a packet where the A field is blank and the B field contains the value 0.

If the DISDD\_CONTINUE flag is set, then the device data sent will be overlaid on the previously sent device data. Otherwise, the device data sent will start from scratch.

For example, suppose a device supports two button outputs, Button0 and Button1. If an application first calls **SendDeviceData** passing "Button0 pressed", then a packet of the form "Button0 pressed, Button1 not pressed" is sent to the device. If the application then makes another call, passing "Button1 pressed" and the DISDD\_CONTINUE flag, then a packet of the form "Button0 pressed, Button1 pressed" is sent to the device. However, if the application had not passed the DISDD\_CONTINUE flag, the packet sent to the device would have been "Button0 not pressed, Button1 pressed".

## DirectInputDevice.SendForceFeedbackCommand

# The **DirectInputDevice.SendForceFeedbackCommand** method sends a command to the device's force-feedback system.

*object*.**SendForceFeedbackCommand**(*flags As Long*)

---

# IDH\_DirectInputDevice.SendForceFeedbackCommand\_dinput\_vb

## Parameters

*object*

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

*flags*

A single value indicating the desired change in state. The value may be one of the members of the **CONST\_DISFFCFLAGS** enumeration.

## Error Codes

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

DIERR\_INPUTLOST  
DIERR\_NOTACQUIRED  
DIERR\_REPORTFULL  
DIERR\_UNPLUGGED

## DirectInputDevice.SetCommonDataFormat at

# The **DirectInputDevice.SetCommonDataFormat** method sets the input data format for standard devices.

*object*.**SetCommonDataFormat**(  
    *format* As **CONST\_DICOMMONDATAFORMATS**)

## Parameters

*object*

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

*format*

One of the **CONST\_DICOMMONDATAFORMATS** enumeration, identifying the data format to use for the device.

## Error Codes

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

DIERR\_ACQUIRED  
DIERR\_INVALIDPARAM

---

# IDH\_DirectInputDevice.SetCommonDataFormat\_dinput\_vb

## Remarks

The data format must be set before the device can be acquired by using the **DirectInputDevice.Acquire** method. It is necessary to set the data format only once. The data format cannot be changed while the device is acquired.

## See Also

**DirectInputDevice.SetDataFormat**

## DirectInputDevice.SetCooperativeLevel

# The **DirectInputDevice.SetCooperativeLevel** method establishes the cooperative level for this instance of the device. The cooperative level determines how this instance of the device interacts with other instances of the device and the rest of the system.

*object*.**SetCooperativeLevel**(*hwnd* As Long, \_  
*flags* As CONST\_DISCLFLAGS)

## Parameters

*object*

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

*hwnd*

Window handle to be associated with the device. This parameter must be a valid top-level window handle that belongs to the process. The window associated with the device must not be destroyed while it is still active in a DirectInput device.

*flags*

Flags that describe the cooperative level associated with the device. The flags are constants of the **CONST\_DISCLFLAGS** enumeration.

The following combinations of flags are valid:

Flags	Meaning	Valid for
DISCL_NONEXCLUSIVE Or DISCL_BACKGROUND	Others can acquire device in exclusive or nonexclusive mode; your application has access to data at all times.	All.
DISCL_NONEXCLUSIVE Or DISCL_FOREGROUND	Others can acquire device in exclusive or nonexclusive mode; your application has access to data only when in the foreground.	All.
DISCL_EXCLUSIVE Or DISCL_BACKGROUND	Others can acquire device in nonexclusive mode; your application has access to data	Joystick.

# IDH\_DirectInputDevice.SetCooperativeLevel\_dinput\_vb

DISCL\_EXCLUSIVE Or  
DISC\_FOREGROUND

at all times.

Others can acquire device in All. Valid for mouse nonexclusive mode; your but prevents application has access to data Windows from only when in the foreground. displaying the cursor.

## Error Codes

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

DIERR\_INVALIDPARAM  
DIERR\_INVALIDHANDLE

## Remarks

No two applications (or instances of the same application) can have a device acquired in exclusive mode at the same time. This is primarily a security feature; it prevents input intended for one application from going to another that may be running concurrently.

If the system mouse is acquired in exclusive mode, then the pointer will be removed from the screen until the device is unacquired.

Applications must call this method before acquiring the device by using the **DirectInputDevice.Acquire** method.

## See Also

Cooperative Levels

## DirectInputDevice.SetDataFormat

# The **DirectInputDevice.SetDataFormat** method sets the data format for a DirectInput device that is not a standard keyboard, mouse, or keyboard.

*object*.**SetDataFormat**(*format* As DIDATAFORMAT, \_  
*formatArray*() As DIOBJECTDATAFORMAT))

## Parameters

*object*

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

*format*

A **DIDATAFORMAT** type that describes the format of the data the device should return.

---

# IDH\_DirectInputDevice.SetDataFormat\_dinput\_vb

*formatArray*

Array of **DIOBJECTDATAFORMAT** types describing data formats for objects on the device.

## Error Codes

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

DIERR\_ACQUIRED

DIERR\_INVALIDPARAM

## Remarks

The data format must be set before the device can be acquired by using the **DirectInputDevice.Acquire** method. It is necessary to set the data format only once. The data format cannot be changed while the device is acquired.

## See Also

**DirectInputDevice.SetCommonDataFormat**

# DirectInputDevice.SetEventNotification

# The **DirectInputDevice.SetEventNotification** method sets the event notification status. This method specifies an event that is to be set when the device state changes. It is also used to turn off event notification.

*object*.SetEventNotification(*hEvent* As Long)

## Parameters

*object*

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

*hEvent*

Handle to the event that is to be set when the device state changes, or 0 to disable notification.

## Error Codes

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

DIERR\_ACQUIRED

DIERR\_HANDLEEXISTS

DIERR\_INVALIDPARAM

---

# IDH\_DirectInputDevice.SetEventNotification\_dinput\_vb



## Remarks

A device state change is defined as any of the following:

- A change in the position of an axis
- A change in the state (pressed or released) of a button
- A change in the direction of a POV control
- Loss of acquisition

You must call this method with the *hEvent* parameter set to 0 before destroying the event.

The event notification handle cannot be changed while the device is acquired.

## See Also

Polling and Events, **DirectXEvent**

## DirectInputDevice.SetProperty

# The **DirectInputDevice.SetProperty** method sets properties that define the device behavior.

```
object.SetProperty( _  
    guid As String, _  
    propertyInfo As Any)
```

## Parameters

*object*

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

*guid*

Identifier of the property to be set. The following property values are predefined for an input device and can be passed as strings:

DIPROP\_AXISMODE

Sets the axis mode. The value being set (either DIPROPAXISMODE\_ABS or DIPROPAXISMODE\_REL from the **CONST\_DINPUT** enumeration) must be specified in the **IData** member of the associated **DIPROPLONG** type.

This setting applies to the entire device, so the **IHow** member of the **DIPROPLONG** type must be set to DIPH\_DEVICE.

DIPROP\_BUFFERSIZE

Sets the input-buffer size. See Remarks.

---

# IDH\_DirectInputDevice.SetProperty\_dinput\_vb

This setting applies to the entire device, so the **IHow** member of the associated **DIPROPLONG** type must be set to **DIPH\_DEVICE**.

#### DIPROP\_CALIBRATIONMODE

Allows the application to specify whether DirectInput should retrieve calibrated or uncalibrated data from an axis. By default, DirectInput retrieves calibrated data.

Setting the calibration mode for the entire device is equivalent to setting it for each axis individually.

The **IData** member of the **DIPROPLONG** type may be one of the following values:

**DIPROPCALIBRATIONMODE\_COOKED**: DirectInput should return data after applying calibration information. This is the default mode.

**DIPROPCALIBRATIONMODE\_RAW**: DirectInput should return raw, uncalibrated data. This mode is typically used only by Control Panel-type applications.

Note that setting a device into raw mode causes the dead zone, saturation, and range settings to be ignored.

#### DIPROP\_DEADZONE

Sets the value for the dead zone of a joystick, in the range 0 to 10,000, where 0 indicates there is no dead zone, 5,000 indicates that the dead zone extends over 50 percent of the physical range of the axis on both sides of center, and 10,000 indicates that the entire physical range of the axis is dead. When the axis is within the dead zone, it is reported as being at the center of its range.

This setting can be applied to either the entire device or to a specific axis.

#### DIPROP\_RANGE

Sets the range of values an object can possibly report. The minimum and maximum values are taken from the **IMin** and **IMax** members of the associated **DIPROPRANGE** type.

For some devices, this is a read-only property.

You cannot set a reverse range; **IMax** must be greater than **IMin**.

#### DIPROP\_SATURATION

Sets the value for the saturation zones of a joystick, in the range 0 to 10,000. The saturation level is the point at which the axis is considered to be at its most extreme position. For example, if the saturation level is set to 9,500, then the axis reaches the extreme of its range when it has moved 95 percent of the physical distance from its center position (or from the dead zone).

This setting can be applied to either the entire device or to a specific axis.

#### *propertyInfo*

A **DIPROPLONG** type containing data for properties that take a single value, or a **DIPROPRANGE** type containing data for properties that take a pair of values.

## Error Codes

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

DIERR\_INVALIDPARAM  
DIERR\_OBJECTNOTFOUND  
DIERR\_UNSUPPORTED

## Remarks

The buffer size determines the amount of data that the buffer can hold between calls to the **DirectInputDevice.GetDeviceData** method before data is lost. This value may be set to 0 to indicate that the application will not be reading buffered data from the device. If the buffer size in the **IData** member of the **DIPROPLONG** type is too large to be supported by the device, the largest possible buffer size is set. To determine whether the requested buffer size was set, retrieve the buffer-size property and compare the result with the value you previously attempted to set.

## See Also

**DirectInputDevice.GetProperty**

## DirectInputDevice.Unacquire

# The **DirectInputDevice.Unacquire** method releases access to the device.

*object*.Unacquire()

## Parameters

*object*

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

## Error Codes

None.

## See Also

**DirectInputDevice.Acquire**

---

# IDH\_DirectInputDevice.Unacquire\_dinput\_vb

## DirectInputDeviceInstance

# The **DirectInputDeviceInstance** class is used to obtain information about an instance of a DirectInput device.

An object of this class is returned by the **DirectInputDevice.GetDeviceInfo** and **DirectInputEnumDevices.GetItem** method.

The **DirectInputDeviceInstance** class has the following methods:

<b>Information</b>	<b>GetDevType</b>
	<b>GetGuidFFDriver</b>
	<b>GetGuidInstance</b>
	<b>GetGuidProduct</b>
	<b>GetInstanceName</b>
	<b>GetProductName</b>
	<b>GetUsage</b>
	<b>GetUsagePage</b>

### DirectInputDeviceInstance.GetDevType

# The **DirectInputDeviceInstance.GetDevType** method retrieves the device type and subtype.

*object*.**GetDevType()** As Long

#### Parameters

*object*

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

#### Return Values

The method returns a device type specifier. This value is a combination of a type (in the least significant byte) and subtype (in the next most significant byte), optionally combined using **Or** with DIDEVTYPE\_HID, which specifies a Human Interface Device. The following constants are from the **CONST\_DIDEVICETYPE** enumeration.

##### Device Types

DIDEVTYPE\_MOUSE

A mouse or mouse-like device (such as a trackball).

DIDEVTYPE\_KEYBOARD

A keyboard or keyboard-like device.

DIDEVTYPE\_JOYSTICK

# IDH\_DirectInputDeviceInstance\_dinput\_vb

# IDH\_DirectInputDeviceInstance.GetDevType\_dinput\_vb

A joystick or similar device, such as a steering wheel.

DIDEVTYPE\_DEVICE

A device that does not fall into the previous categories.

**Mouse subtypes**

DIDEVTYPEMOUSE\_UNKNOWN

The subtype could not be determined.

DIDEVTYPEMOUSE\_TRADITIONAL

The device is a traditional mouse.

DIDEVTYPEMOUSE\_FINGERSTICK

The device is a fingerstick.

DIDEVTYPEMOUSE\_TOUCHPAD

The device is a touchpad.

DIDEVTYPEMOUSE\_TRACKBALL

The device is a trackball.

**Keyboard subtypes**

DIDEVTYPEKEYBOARD\_UNKNOWN

The subtype could not be determined.

DIDEVTYPEKEYBOARD\_PCXT

IBM PC/XT 83-key keyboard.

DIDEVTYPEKEYBOARD\_OLIVETTI

Olivetti 102-key keyboard.

DIDEVTYPEKEYBOARD\_PCAT

IBM PC/AT 84-key keyboard.

DIDEVTYPEKEYBOARD\_PCENH

IBM PC Enhanced 101/102-key or Microsoft Natural® keyboard.

DIDEVTYPEKEYBOARD\_NOKIA1050

Nokia 1050 keyboard.

DIDEVTYPEKEYBOARD\_NOKIA9140

Nokia 9140 keyboard.

DIDEVTYPEKEYBOARD\_NEC98

Japanese NEC PC98 keyboard.

DIDEVTYPEKEYBOARD\_NEC98LAPTOP

Japanese NEC PC98 laptop keyboard.

DIDEVTYPEKEYBOARD\_NEC98106

Japanese NEC PC98 106-key keyboard.

DIDEVTYPEKEYBOARD\_JAPAN106

Japanese 106-key keyboard.

DIDEVTYPEKEYBOARD\_JAPANAX

Japanese AX keyboard.

DIDEVTYPEKEYBOARD\_J3100

Japanese J3100 keyboard.

**Joystick Subtypes**

DIDEVTYPEJOYSTICK\_UNKNOWN

The subtype could not be determined.

DIDEVTYPEJOYSTICK\_TRADITIONAL

A traditional joystick.

DIDEVTYPEJOYSTICK\_FLIGHTSTICK

A joystick optimized for flight simulation.

DIDEVTYPEJOYSTICK\_GAMEPAD

A device whose primary purpose is to provide button input.

DIDEVTYPEJOYSTICK\_RUDDER

A device for yaw control.

DIDEVTYPEJOYSTICK\_WHEEL

A steering wheel.

DIDEVTYPEJOYSTICK\_HEADTRACKER

A device that tracks the movement of the user's head.

#### **Human Interface Device**

DIDEVTYPE\_HID

The device uses the Human Interface Device (HID) protocol.

## **Error Codes**

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

## **Remarks**

To look for a particular subtype of device such as a wheel, you should check both the type and the subtype.

## **DirectInputDeviceInstance.GetGuidFFDriver**

# The **DirectInputDeviceInstance.GetGuidFFDriver** method returns the unique identifier for the force-feedback driver.

*object*.GetGuidFFDriver() As String

## **Parameters**

*object*

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

## **Return Values**

The method returns the GUID for the force feedback driver, in string form. This identifier is established by the manufacturer of the driver.

# IDH\_DirectInputDeviceInstance.GetGuidFFDriver\_dinput\_vb

## Error Codes

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

## See Also

Using GUIDs

# DirectInputDeviceInstance.GetGuidInstance

# The **DirectInputDeviceInstance.GetGuidInstance** method returns the unique identifier for the instance of the device.

*object*.GetGuidInstance() As String

## Parameters

*object*

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

## Return Values

The method returns the GUID for the device instance, in string form.

## Error Codes

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

## Remarks

An application can save the instance GUID into a configuration file and use it at a later time. Instance GUIDs are specific to a particular computer. An instance GUID obtained from one computer is unrelated to instance GUIDs on another.

## See Also

**DirectInputDeviceInstance.GetGuidProduct**, Using GUIDs

# DirectInputDeviceInstance.GetGuidProduct

# The **DirectInputDeviceInstance.GetGuidProduct** method retrieves the manufacturer's unique identifier for the device.

---

# IDH\_DirectInputDeviceInstance.GetGuidInstance\_dinput\_vb

# IDH\_DirectInputDeviceInstance.GetGuidProduct\_dinput\_vb

*object*.GetGuidProduct() As String

## Parameters

*object*

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

## Return Values

The method returns the GUID for the product, in string form. This identifier is established by the manufacturer of the device.

## Error Codes

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

## See Also

**DirectInputDeviceInstance.GetGuidInstance**, Using GUIDs

# DirectInputDeviceInstance.GetInstanceName

# The **DirectInputDeviceInstance.GetInstanceName** method retrieves the name of the device instance.

*object*.GetInstanceName() As String

## Parameters

*object*

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

## Return Values

The method returns the friendly name for the instance—for example, "Joystick 1."

## Error Codes

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

## See Also

**DirectInputDeviceInstance.GetProductName**

---

# IDH\_DirectInputDeviceInstance.GetInstanceName\_dinput\_vb



---

## DirectInputDeviceInstance.GetProductName

# The **DirectInputDeviceInstance.GetProductName** method retrieves the product name of the device.

*object*.**GetProductName()** As String

### Parameters

*object*

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

### Return Values

The method returns the friendly name for the product—for example, "Microsoft SideWinder".

### Error Codes

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

### See Also

**DirectInputDeviceInstance.GetInstanceName**

## DirectInputDeviceInstance.GetUsage

# The **DirectInputDeviceInstance.GetUsage** method retrieves the usage code for Human Interface Devices.

*object*.**GetUsage()** As Integer

### Parameters

*object*

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

### Return Values

If the device is a HID, the method returns the usage code.

### Error Codes

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

---

# IDH\_DirectInputDeviceInstance.GetProductName\_dinput\_vb

# IDH\_DirectInputDeviceInstance.GetUsage\_dinput\_vb

## See Also

**DirectInputDeviceInstance.GetUsagePage**

# DirectInputDeviceInstance.GetUsagePage

# The **DirectInputDeviceInstance.GetUsagePage** method retrieves the usage page for Human Interface Devices.

*object*.**GetUsagePage()** As Integer

## Parameters

*object*

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

## Return Values

If the device is a HID, the method returns the HID usage page.

## Error Codes

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

## See Also

**DirectInputDeviceInstance.GetUsage**

# DirectInputDeviceObjectInstance

# The **DirectInputDeviceObjectInstance** class represents an object on a DirectInput device, such as a button or axis.

A **DirectInputDeviceObjectInstance** object is returned by the **DirectInputDevice.GetObjectInfo** and **DirectInputEnumDeviceObjects.GetItem** methods.

This class has the following methods:

### Information

**GetCollectionNumber**  
**GetDesignatorIndex**  
**GetDimension**  
**GetExponent**  
**GetFlags**

# IDH\_DirectInputDeviceInstance.GetUsagePage\_dinput\_vb

# IDH\_DirectInputDeviceObjectInstance\_dinput\_vb

**GetGuidType**  
**GetName**  
**GetOfs**  
**GetType**  
**GetUsage**  
**GetUsagePage**

## **DirectInputDeviceObjectInstance.GetCollectionNumber**

# The **DirectInputDeviceObjectInstance.GetCollectionNumber** method retrieves the number of the HID link collection to which the device object belongs.

*object*.**GetCollectionNumber()** As Integer

### **Parameters**

*object*

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

### **Return Values**

If the device is a Human Interface Device and the object belongs to a collection, the method returns the number of the collection.

### **Error Codes**

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

## **DirectInputDeviceObjectInstance.GetDesignatorIndex**

# The **DirectInputDeviceObjectInstance.GetDesignatorIndex** method retrieves the designator index for an object on a Human Interface Device.

*object*.**GetDesignatorIndex()** As Integer

### **Parameters**

*object*

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

---

# IDH\_DirectInputDeviceObjectInstance.GetCollectionNumber\_dinput\_vb

# IDH\_DirectInputDeviceObjectInstance.GetDesignatorIndex\_dinput\_vb

## Return Values

The method returns an index that refers to a designator in the HID physical descriptor. This number can be passed to functions in the HID parsing library to obtain additional information about the device object.

## Error Codes

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

# DirectInputDeviceObjectInstance.GetDimension

# The **DirectInputDeviceObjectInstance.GetDimension** method retrieves a HID code for the dimensional units in which the object's value is reported.

*object*.GetDimension() As Long

## Parameters

*object*

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

## Return Values

The method returns a code for the dimensional units in which the object's value is reported, if known, or 0 if not known.

## Error Codes

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

## See Also

**DirectInputDeviceObjectInstance.GetExponent**

# DirectInputDeviceObjectInstance.GetExponent

# The **DirectInputDeviceObjectInstance.GetExponent** method retrieves the exponent to associate with the dimensional units of the device object.

*object*.GetExponent() As Integer

---

# IDH\_DirectInputDeviceObjectInstance.GetDimension\_dinput\_vb

# IDH\_DirectInputDeviceObjectInstance.GetExponent\_dinput\_vb

## Parameters

*object*

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

## Return Values

The method returns the exponent to associate with the dimension, if known. Dimensional units are always integral, so an exponent may be needed in order to convert them to non-integral types.

## Error Codes

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

## See Also

**DirectInputDeviceObjectInstance.GetDimension**

# DirectInputDeviceObjectInstance.GetFlags

# The **DirectInputDeviceObjectInstance.GetFlags** method retrieves the flags associated with the device object.

*object*.GetFlags() As CONST\_DIDEVICEOBJINSTANCEFLAGS

## Parameters

*object*

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

## Return Values

The method retrieves flags describing miscellaneous attributes of the object. The return value may consist of one or more members of the **CONST\_DIDEVICEOBJINSTANCEFLAGS** enumeration.

## Error Codes

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

---

## DirectInputDeviceObjectInstance.GetGuidType

# The **DirectInputDeviceObjectInstance.GetGuidType** method retrieves the unique identifier of the object type.

*object*.GetGuidType() As String

### Parameters

*object*

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

### Return Values

The method may return one of the following string identifiers, representing the unique identifier for the object type. If the object type has a GUID not represented in the following list, a string representing the actual GUID will be returned. If the object type does not have a GUID, an empty string will be returned.

GUID\_XAxis

The horizontal axis. For example, it may represent the left-right motion of a mouse.

GUID\_YAxis

The vertical axis. For example, it may represent the forward-backward motion of a mouse.

GUID\_ZAxis

The z-axis. For example, it may represent rotation of the wheel on a mouse, or movement of a throttle control on a joystick.

GUID\_RxAxis

Rotation around the x-axis.

GUID\_RyAxis

Rotation around the y-axis.

GUID\_RzAxis

Rotation around the z-axis (often a rudder control).

GUID\_Slider

A slider axis.

GUID\_Button

A button on a mouse.

GUID\_Key

A key on a keyboard.

GUID\_POV

A point-of-view indicator or "hat".

GUID\_Unknown

---

# IDH\_DirectInputDeviceObjectInstance.GetGuidType\_dinput\_vb

Unknown.

## Error Codes

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

## DirectInputDeviceObjectInstance.GetName

# The **DirectInputDeviceObjectInstance.GetName** method retrieves the friendly name of the device object.

*object.GetName()* As String

### Parameters

*object*

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

### Return Values

The method retrieves the name of the object—for example, "X-Axis" or "Right Shift."

## Error Codes

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

## DirectInputDeviceObjectInstance.GetOfs

# The **DirectInputDeviceObjectInstance.GetOfs** method retrieves the offset of the device object's data within the data format for the device.

*object.GetOfs()* As Long

### Parameters

*object*

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

### Return Values

The method returns the offset within the data format at which data is reported for this object. This value can be used to identify the object in method calls and types that accept the DIPH\_BYOFFSET flag.

---

# IDH\_DirectInputDeviceObjectInstance.GetName\_dinput\_vb

# IDH\_DirectInputDeviceObjectInstance.GetOfs\_dinput\_vb

## Error Codes

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

## See Also

**DirectInputDevice.GetObjectInfo**, **DIPROPLONG**, **DIPROPRANGE**

# DirectInputDeviceObjectInstance.GetType

# The **DirectInputDeviceObjectInstance.GetType** method retrieves the type and instance identifier of the object.

*object*.GetType() As Long

## Parameters

*object*

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

## Return Values

Device type that describes the object. It is a combination of **CONST\_DIDFTFLAGS** flags that describe the object type (axis, button, and so forth) and contains the object instance number in the middle 16 bits.

## Error Codes

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

## Remarks

To extract the object instance ID, use the following operation:

```
Dim ObjID as Long
ObjID = (diObj.GetType And &HFFFF00) \ 256
```

# DirectInputDeviceObjectInstance.GetUsage

# The **DirectInputDeviceObjectInstance.GetUsage** method retrieves the Human Interface Device usage code for the device object.

*object*.GetUsage() As Integer

---

# IDH\_DirectInputDeviceObjectInstance.GetType\_dinput\_vb  
# IDH\_DirectInputDeviceObjectInstance.GetUsage\_dinput\_vb



## Parameters

*object*

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

## Return Values

The method returns the HID usage associated with the object, if known. Human Interface Devices will always report a usage. Non-HID devices may optionally report a usage; if they do not, then the value of this member will be zero.

## Error Codes

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

## See Also

**DirectInputDeviceObjectInstance.GetUsagePage**

# DirectInputDeviceObjectInstance.GetUsagePage

# The **DirectInputDeviceObjectInstance.GetUsagePage** method retrieves the Human Interface Device usage page for the device object.

*object*.**GetUsagePage()** As Integer

## Parameters

*object*

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

## Return Values

The method returns the HID usage page associated with the object, if known. Human Interface Devices will always report a usage page. Non-HID devices may optionally report a usage page; if they do not, then the value of this member will be zero.

## Error Codes

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

## See Also

**DirectInputDeviceObjectInstance.GetUsage**

---

# IDH\_DirectInputDeviceObjectInstance.GetUsagePage\_dinput\_vb

## DirectInputEffect

# An object of the **DirectInputEffect** class represents a force-feedback effect created by an application. The object is created by using the **DirectInputDevice.CreateEffect** method.

This class has the following methods:

<b>Information</b>	<b>GetEffectGuid</b>
	<b>GetEffectStatus</b>
	<b>GetParameters</b>
<b>Manipulation</b>	<b>Download</b>
	<b>SetParameters</b>
	<b>Start</b>
	<b>Stop</b>
	<b>Unload</b>

## DirectInputEffect.Download

# The **DirectInputEffect.Download** method places the effect on the device. If the effect is already on the device, then the existing effect is updated to match the values set by the **DirectInputEffect.SetParameters** method.

*object*.Download()

### Parameters

*object*

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

### Error Codes

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

```
DIERR_NOTINITIALIZED
DIERR_DEVICEFULL
DIERR_INCOMPLETEEFFECT
DIERR_INPUTLOST
DIERR_NOTEXCLUSIVEACQUIRED
DIERR_INVALIDPARAM
DIERR_EFFECTPLAYING
```

---

# IDH\_DirectInputEffect\_dinput\_vb

# IDH\_DirectInputEffect.Download\_dinput\_vb

## Remarks

The device must be acquired at the exclusive cooperative level for this method to succeed.

It is valid to update an effect while it is playing. The semantics of such an operation are explained in the reference for **DirectInputEffect.SetParameters**.

## DirectInputEffect.GetEffectGuid

# The **DirectInputEffect.GetEffectGuid** method retrieves the GUID or GUID alias for the effect represented by the **DirectInputEffect** object.

*object*.GetEffectGuid() As String

## Parameters

*object*

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

## Return Values

The method returns the GUID or alias that was passed to **DirectInputDevice.CreateEffect**—for example, "GUID\_ConstantForce".

## Error Codes

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

## See Also

Using GUIDs

## DirectInputEffect.GetEffectStatus

# The **DirectInputEffect.GetEffectStatus** method retrieves the status of an effect.

*object*.GetEffectStatus() As Long

## Parameters

*object*

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

---

# IDH\_DirectInputEffect.GetEffectGuid\_dinput\_vb

# IDH\_DirectInputEffect.GetEffectStatus\_dinput\_vb

## Return Values

The method returns status flags for the effect. The value may be zero, or one or more of the following constants of the **CONST\_DIEGESFLAGS** enumeration:

**DIEGES\_PLAYING**

The effect is playing.

**DIEGES\_EMULATED**

The effect is emulated.

## Error Codes

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

## DirectInputEffect.GetParameters

# The **DirectInputEffect.GetParameters** method retrieves information about an effect.

*object*.**GetParameters**(*effectinfo* As **DIEFFECT**)

### Parameters

*object*

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

*effectinfo*

**DIEFFECT** type to receive the effect parameters.

## Error Codes

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

**DIERR\_INVALIDPARAM**

**DIERR\_MOREDATA**

**DIERR\_NOTINITIALIZED**

## DirectInputEffect.SetParameters

# The **DirectInputEffect.SetParameters** method sets the characteristics of an effect.

*object*.**SetParameters**(*effectinfo* As **DIEFFECT**, *flags* As **CONST\_DIEPFLAGS**)

---

# IDH\_DirectInputEffect.GetParameters\_dinput\_vb

# IDH\_DirectInputEffect.SetParameters\_dinput\_vb

## Parameters

*object*

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

*effectinfo*

**DIEFFECT** type containing effect parameters.

*flags*

Flags from the **CONST\_DIEPFLAGS** enumeration specifying which portions of the effect information are to be set and how the downloading of the parameters should be handled.

## Error Codes

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

DIERR\_NOTINITIALIZED  
DIERR\_INCOMPLETEEFFECT  
DIERR\_INPUTLOST  
DIERR\_INVALIDPARAM  
DIERR\_EFFECTPLAYING

## Remarks

To determine which parameters can be dynamically updated while the effect is playing, use the **DirectInputEnumEffects.GetDynamicParams** method.

The **DirectInputEffect.SetParameters** method automatically downloads the effect, but this behavior can be suppressed by setting the **DIEP\_NODOWNLOAD** flag. If automatic download has been suppressed, then you can manually download the effect by calling **DirectInputEffect.Download**.

If the effect is playing while the parameters are changed, then the new parameters take effect as if they were the parameters when the effect started.

For example, suppose a periodic effect with a duration of three seconds is started. After two seconds, the direction of the effect is changed. The effect will then continue for one additional second in the new direction. The envelope, phase, amplitude, and other parameters of the effect continue smoothly as if the direction had not changed.

In the same scenario, if after two seconds the duration of the effect were changed to 1.5 seconds, then the effect would stop.

Normally, if the driver cannot update the parameters of a playing effect, the driver is permitted to stop the effect, update the parameters, and then restart the effect. Passing the **DIEP\_NORESTART** flag suppresses this behavior. If the driver cannot update the parameters of an effect while it is playing, the error code **DIERR\_EFFECTPLAYING** is returned and the parameters are not updated.

No more than one of the DIEP\_NODOWNLOAD, DIEP\_START, and DIEP\_NORESTART flags should be set. (It is also valid to pass none of them.)

These three flags control download and playback behavior as follows:

If DIEP\_NODOWNLOAD is set, the effect parameters are updated but not downloaded to the device.

If the DIEP\_START flag is set, the effect parameters are updated and downloaded to the device, and the effect is started just as if the **DirectInputEffect.Start** method had been called with the *iterations* parameter set to 1 and with no flags. (Combining the update with DIEP\_START is slightly faster than calling **Start** separately, because it requires less information to be transmitted to the device.)

If neither DIEP\_NODOWNLOAD nor DIEP\_START is set and the effect is not playing, then the parameters are updated and downloaded to the device.

If neither DIEP\_NODOWNLOAD nor DIEP\_START is set and the effect is playing, then the parameters are updated if the device supports on-the-fly updating. Otherwise the behavior depends on the state of the DIEP\_NORESTART flag. If it is set, the error code DIERR\_EFFECTPLAYING is returned. If it is clear, the effect is stopped, the parameters are updated, and the effect is restarted.

## DirectInputEffect.Start

# The **DirectInputEffect.Start** method begins playing an effect. If the effect is already playing, it is restarted from the beginning. If the effect has not been downloaded or has been modified since its last download, then it will be downloaded before being started. This default behavior can be suppressed by passing the DIEP\_NODOWNLOAD flag.

*object.Start(iterations As Long, flags As Long)*

### Parameters

*object*

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

*iterations*

Number of times to play the effect in sequence. The envelope is re-articulated with each iteration.

To play the effect exactly once, pass 1. To play the effect repeatedly until explicitly stopped, pass -1. To play the effect until explicitly stopped without re-articulating the envelope, modify the effect parameters with the

**DirectInputEffect.SetParameters** method and change the **IDuration** member of **DIEFFECT** to -1.

*flags*

---

# IDH\_DirectInputEffect.Start\_dinput\_vb

Flags from the **CONST\_DIESFLAGS** enumeration that describe how the effect should be played by the device. The value may be zero or one or more of the following values:

**DIES\_SOLO**

All other effects on the device should be stopped before the specified effect is played. If this flag is omitted, then the effect is mixed with existing effects already started on the device.

**DIES\_NODOWNLOAD**

Do not automatically download the effect.

## Error Codes

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

DIERR\_INVALIDPARAM

DIERR\_INCOMPLETEEFFECT

DIERR\_NOTEXCLUSIVEACQUIRED

DIERR\_NOTINITIALIZED

DIERR\_UNSUPPORTED

## Remarks

The device must be acquired at the exclusive cooperative level for this method to succeed.

Not all devices support multiple iterations.

## DirectInputEffect.Stop

# The **DirectInputEffect.Stop** method stops an effect that is playing.

*object*.Stop()

## Parameters

*object*

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

## Error Codes

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

DIERR\_NOTEXCLUSIVEACQUIRED

---

# IDH\_DirectInputEffect.Stop\_dinput\_vb

---

DIERR\_NOTINITIALIZED

## Remarks

The device must be acquired at the exclusive cooperative level for this method to succeed.

## DirectInputEffect.Unload

# The **DirectInputEffect.Unload** method removes the effect from the device. If the effect is playing, it is automatically stopped before it is unloaded.

*object.Unload()*

## Parameters

*object*

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

## Error Codes

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

DIERR\_INPUTLOST  
DIERR\_INVALIDPARAM  
DIERR\_NOTEXCLUSIVEACQUIRED  
DIERR\_NOTINITIALIZED

## Remarks

The device must be acquired at the exclusive cooperative level for this method to succeed.

## DirectInputEnumDeviceObjects

# The **DirectInputEnumDeviceObjects** class enumerates the **DirectInputDevice** objects installed on a system. This object is created and filled with data as a result of a call to **DirectInputDevice.GetDeviceObjectsEnum** method.

This class has the following methods:

Information	GetCount
-------------	----------

---

# IDH\_DirectInputEffect.Unload\_dinput\_vb

# IDH\_DirectInputEnumDeviceObjects\_dinput\_vb



---

## GetItem

# DirectInputEnumDeviceObjects.GetCount

# The **DirectInputEnumDeviceObjects.GetCount** method returns the number of items in the **DirectInputEnumDeviceObjects** collection.

*object*.GetCount() As Long

## Parameters

*object*

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

## Return Values

The method returns the number of device objects enumerated for the device.

## Error Codes

None.

# DirectInputEnumDeviceObjects.GetItem

# The **DirectInputEnumDeviceObjects.GetItem** method retrieves an object describing the specified device object.

*object*.GetItem(*index* As Long) \_  
As DirectInputDeviceObjectInstance

## Parameters

*object*

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

*index*

Index of the enumerated item to retrieve.

## Return Values

The method returns a **DirectInputDeviceObjectInstance** object whose methods can be used to retrieve information about the device object.

---

# IDH\_DirectInputEnumDeviceObjects.GetCount\_dinput\_vb

# IDH\_DirectInputEnumDeviceObjects.GetItem\_dinput\_vb

## Error Codes

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

## Remarks

To get the number of entries in the **DirectInputEnumDeviceObjects** first call the **DirectInputEnumDeviceObjects.GetCount** method.

# DirectInputEnumDevices

# The **DirectInputEnumDevices** enumerates the DirectInput devices installed on a system. This object is created and filled with data as a result of a call to **DirectInput.GetDIEnumDevices** method.

This class has the following methods:

<b>Information</b>	<b>GetCount</b>
	<b>GetItem</b>

## DirectInputEnumDevices.GetCount

# The **DirectInputEnumDevices.GetCount** method returns the number of DirectInput devices in the **DirectInputEnumDevices** collection.

*object*.**GetCount()** As Long

## Parameters

*object*

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

## Return Values

If the method succeeds, the number of DirectInput devices in the **DirectInputEnumDevices** collection is returned.

## Error Codes

None.

---

# IDH\_DirectInputEnumDevices\_dinput\_vb  
# IDH\_DirectInputEnumDevices.GetCount\_dinput\_vb

---

## DirectInputEnumDevices.GetItem

# The **DirectInputEnumDevices.GetItem** method returns information about an enumerated device.

*object*.**GetItem**(*index* As Long) As **DirectInputDeviceInstance**

### Parameters

*object*

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

*index*

The specific DirectInput device data entry in **DirectInputEnumDevices**.

### Return Values

The method returns a **DirectInputDeviceInstance** object whose methods can be used to retrieve information about the device.

### Error Codes

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

### Remarks

To get the number of entries in **DirectInputEnumDevices** first call the **DirectInputEnumDevices.GetCount** method.

## DirectInputEnumEffects

# An object of the **DirectInputEnumEffects** class represents a collection of supported force-feedback effects on a device, and is used for retrieving information about the effects. The object is created by using the **DirectInputDevice.GetEffectsEnum** method.

The class has the following methods:

<b>Information</b>	<b>GetCount</b>
	<b>GetDynamicParams</b>
	<b>GetEffectGuid</b>
	<b>GetName</b>
	<b>GetStaticParams</b>
	<b>GetType</b>

---

# IDH\_DirectInputEnumDevices.GetItem\_dinput\_vb

# IDH\_DirectInputEnumEffects\_dinput\_vb

---

## DirectInputEnumEffects.GetCount

# The **DirectInputEnumEffects.GetCount** method returns the number of enumerated effects in the collection.

*object*.GetCount() As Long

### Parameters

*object*

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

### Return Values

The method returns the number of items in the collection.

### Error Codes

If the method fails, it raises an error and **Err.Number** will be set.

## DirectInputEnumEffects.GetDynamicParams

# The **DirectInputEnumEffects.GetDynamicParams** method retrieves information about which effect parameters can be changed without stopping the effect, using the **DirectInputEffect.SetParameters** method.

*object*.GetDynamicParams(*i* As Long) As CONST\_DIEPFLAGS

### Parameters

*object*

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

*i*

Index (1-based) of the effect in the collection.

### Return Values

The method returns flags from the **CONST\_DIEPFLAGS** enumeration specifying which parameters can be dynamically updated while the effect is playing.

### Error Codes

If the method fails, it raises an error and **Err.Number** will be set.

---

# IDH\_DirectInputEnumEffects.GetCount\_dinput\_vb

# IDH\_DirectInputEnumEffects.GetDynamicParams\_dinput\_vb

## See Also

**DirectInputEffect.SetParameters**

## DirectInputEnumEffects.GetEffectGuid

# The **DirectInputEnumEffects.GetEffectGuid** method retrieves the GUID for the supported effect. This GUID can be passed to the **DirectInputDevice.CreateEffect** method.

*object*.GetEffectGuid(*i* As Long) As String

### Parameters

*object*

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

*i*

Index (1-based) of the effect in the collection.

### Return Values

The method returns the GUID in string form. For standard effects, the return value is an alias such as "GUID\_ConstantForce".

### Error Codes

If the method fails, it raises an error and **Err.Number** will be set.

## See Also

Using GUIDs

## DirectInputEnumEffects.GetName

# The **DirectInputEnumEffects.GetName** method retrieves the name of the effect.

*object*.GetName(*i* As Long) As String

### Parameters

*object*

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

*i*

Index (1-based) of the effect in the collection.

---

# IDH\_DirectInputEnumEffects.GetEffectGuid\_dinput\_vb

# IDH\_DirectInputEnumEffects.GetName\_dinput\_vb

## Return Values

The method returns the name of the effect, such as "Constant Force".

## Error Codes

If the method fails, it raises an error and **Err.Number** will be set.

# DirectInputEnumEffects.GetStaticParams

# The **DirectInputEnumEffects.GetStaticParams** method retrieves information about effect parameters supported on the device.

*object*.GetStaticParams(*i* As Long) As CONST\_DIEPFLAGS

## Parameters

*object*

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

*i*

Index (1-based) of the effect in the collection.

## Return Values

The method returns flags from the **CONST\_DIEPFLAGS** enumeration specifying which parameters are supported by the effect.

## Error Codes

If the method fails, it raises an error and **Err.Number** will be set.

# DirectInputEnumEffects.GetType

# The **DirectInputEnumEffects.GetType** method retrieves information about the type and capabilities of the effect.

*object*.GetType(*i* As Long) As CONST\_DIEFTFLAGS

## Parameters

*object*

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

*i*

Index (1-based) of the effect in the collection.

---

# IDH\_DirectInputEnumEffects.GetStaticParams\_dinput\_vb

# IDH\_DirectInputEnumEffects.GetType\_dinput\_vb

## Return Values

The method returns flags from the **CONST\_DIEFTFLAGS** enumeration giving information about the effect.

## Error Codes

If the method fails, it raises an error and **Err.Number** will be set.

## Remarks

The effect type is stored in the low byte and can be retrieved by a bitwise **And** with &HFF.

## Types

This section contains information on the following types used with DirectInput:

- **DICONDITION**
- **DICONSTANTFORCE**
- **DIDATAFORMAT**
- **DIDEVCAPS**
- **DIDEVICEOBJECTDATA**
- **DIEFFECT**
- **DIENVELOPE**
- **DIJOYSTATE**
- **DIJOYSTATE2**
- **DIKEYBOARDSTATE**
- **DIMOUSESTATE**
- **DIOBJECTDATAFORMAT**
- **DIPERIODICFORCE**
- **DIPROPLONG**
- **DIPROPRANGE**
- **DIRAMPFORCE**

## DICONDITION

# The **DICONDITION** type describes parameters for a force-feedback condition in the **DIEFFECT** type.

Type DICONDITION  
IDeadBand As Long

# IDH\_DICONDITION\_dinput\_vb

INegativeCoefficient As Long  
 INegativeSaturation As Long  
 IOffset As Long  
 IPositiveCoefficient As Long  
 IPositiveSaturation As Long  
 End Type

## Members

### IDeadBand

The region around **IOffset** where the condition is not active, in the range 0 to 10,000. In other words, the condition is not active between **IOffset - IDeadBand** and **IOffset + IDeadBand**.

### INegativeCoefficient

The coefficient constant on the negative side of the offset, in the range -10,000 to +10,000.

If the device does not support separate positive and negative coefficients, then the value of **INegativeCoefficient** is ignored and the value of **IPositiveCoefficient** is used as both the positive and negative coefficients.

### INegativeSaturation

The maximum force output on the negative side of the offset, in the range 0 to 10,000.

If the device does not support force saturations, then the value of this member is ignored.

If the device does not support separate positive and negative saturations, then the value of **INegativeSaturation** is ignored and the value of **IPositiveSaturation** is used as both the positive and negative saturations.

### IOffset

The offset for the condition, in the range -10,000 to +10,000.

### IPositiveCoefficient

The coefficient constant on the positive side of the offset, in the range -10,000 to +10,000.

### IPositiveSaturation

The maximum force output on the positive side of the offset, in the range 0 to 10,000.

If the device does not support force saturation, then the value of this member is ignored.

## Remarks

Different types of conditions will interpret the parameters differently, but the basic idea is that force resulting from a condition is equal to  $A(q - q_0)$  where  $A$  is a scaling coefficient,  $q$  is some metric, and  $q_0$  is the neutral value for that metric.



The preceding simplified formula must be adjusted if a nonzero dead band is provided. If the metric is less than **IOffset - IDeadBand**, then the resulting force is given by the following formula:

$$force = \text{INegativeCoefficient} * (q - (\text{IOffset} - \text{IDeadBand}))$$

Similarly, if the metric is greater than **IOffset + IDeadBand**, then the resulting force is given by the following formula:

$$force = \text{IPositiveCoefficient} * (q - (\text{IOffset} + \text{IDeadBand}))$$

A spring condition uses axis position as the metric.

A damper condition uses axis velocity as the metric.

An inertia condition uses axis acceleration as the metric.

## DICONSTANTFORCE

# The **DICONSTANTFORCE** type describes parameters for a constant force in the **DIEFFECT** type.

```
Type DICONSTANTFORCE
    IMagnitude As Long
End Type
```

### Members

#### IMagnitude

Magnitude of the effect, in the range -10,000 to +10,000. If an envelope is applied to this effect, then the value represents the magnitude of the sustain. If no envelope is applied, then the value represents the amplitude of the entire effect.

## DIDATAFORMAT

# The **DIDATAFORMAT** type carries information describing a device's data format. This type is used with the **DirectInputDevice.SetDataFormat** method.

```
Type DIDATAFORMAT
    dataSize As Long
    IFlags As Long
    IObjSize As Long
    numObjs As Long
End Type
```

### Members

#### dataSize

---

```
# IDH_DICONSTANTFORCE_dinput_vb
# IDH_DIDATAFORMAT_dinput_vb
```

Size of a data packet returned by the device, in bytes. This value must be a multiple of 4 and must exceed the largest offset value for an object's data within the data packet.

**IFlags**

Flags describing other attributes of the data format. This value can be one of the **CONST\_DIDATAFORMATFLAGS** enumeration.

**IObjSize**

Size of the **DIOBJECTDATAFORMAT** type, in bytes.

**numObjs**

Number of objects for which data is to be returned.

**Remarks**

Applications need to create a **DIDATAFORMAT** type only for nonstandard devices. For the mouse, keyboard, and joystick, you set the data format by using **DirectInputDevice.SetCommonDataFormat**.

## DIDEVCAPS

# The **DIDEVCAPS** type contains information about a DirectInput device's capabilities. This type is used with the **DirectInputDevice.GetCapabilities** method.

Type DIDEVCAPS

IAxes As Long

IButtons As Long

IDevType As CONST\_DIDEVICETYPE

IDriverVersion As Long

IFFMinTimeResolution As Long

IFFSamplePeriod As Long

IFirmwareRevision As Long

IFlags As CONST\_DIDEVCAPSFLAGS

IHardwareRevision As Long

IPOVs As Long

End Type

**Members****IAxes**

Number of axes available on the device.

**IButtons**

Number of buttons available on the device.

**IDevType**

---

# IDH\_DIDEVCAPS\_dinput\_vb

A packed value containing information about the type and subtype of the device. The value is identical to that returned by the **DirectInputDeviceInstance.GetDevType** method.

#### **IDriverVersion**

The version number of the device driver.

#### **IFFMinTimeResolution**

The minimum amount of time, in microseconds, that the device can resolve when playing force-feedback effects. The device rounds any times to the nearest supported increment. For example, if the value of **IFFMinTimeResolution** is 1000, then the device would round any times to the nearest millisecond.

#### **IFFSamplePeriod**

The minimum time between playback of consecutive raw force commands.

#### **IFirmwareRevision**

Specifies the firmware revision of the device.

#### **IFlags**

Flags associated with the device. This value can be a combination of the constants of the **CONST\_DIDEVCAPSFLAGS** enumeration.

#### **IHardwareRevision**

The hardware revision of the device.

#### **IPOVs**

Number of point-of-view controllers available on the device.

### **Remarks**

The semantics of version numbers are left to the manufacturer of the device. The only guarantee is that newer versions will have larger numbers.

## **DIDeviceObjectData**

# The **DIDeviceObjectData** type contains raw buffered device information. This type is used with the **DirectInputDevice.GetDeviceData** method.

Type **DIDeviceObjectData**

  IData As Long

  IOfs As Long

  ISequence As Long

  ITimeStamp As Long

End Type

### **Members**

#### **IData**

Data obtained from the device.

---

# IDH\_DIDeviceObjectData\_dinput\_vb

For axis input, if the device is in relative axis mode, then the relative axis motion is reported. If the device is in absolute axis mode, then the absolute axis coordinate is reported.

For button input, only the low byte of **IData** is significant. The high bit of the low byte is set if the button went down; it is clear if the button went up.

#### **IOfs**

Offset into the current data format of the object whose data is being reported—that is, the location where the data would have been stored if it had been obtained by a call to the **DirectInputDevice.GetDeviceStateX** method where **X** refers to the specific device, for instance **GetDeviceStateMouse**. If the device is accessed as a keyboard, you can determine which key generated the event by comparing this value with the members of the **CONST\_DIKEYFLAGS** enumeration. For the mouse and joystick, the value in **IOfs** is equivalent to the byte offset of the button or axis within the **DIMOUSESTATE**, **DIJOYSTATE**, or **DIJOYSTATE2** type (depending on the data format that was established by using **DirectInputDevice.SetCommonDataFormat**.) Constants for these offsets are contained in the **CONST\_DIMOUSEOFS** and **CONST\_DIJOYSTICKOFS** enumerations. If a custom data format has been set by using **DirectInputDevice.SetDataFormat**, then **IOfs** is the offset of the device object's place in the custom data format.

#### **ISequence**

DirectInput sequence number for this event. All input events are assigned an increasing sequence number. This allows events from different devices to be sorted chronologically. Since this value can wrap around, care must be taken when comparing two sequence numbers.

#### **ITimeStamp**

System time at which the input event was generated, in milliseconds. This value wraps around approximately every 50 days.

## **DIEFFECT**

# The **DIEFFECT** type describes a force-feedback effect. It is passed to the **DirectInputDevice.CreateEffect** and **DirectInputEffect.SetParameters** methods in order to set parameters for an effect. Existing parameters are retrieved in this type through the **DirectInputEffect.GetParameters** method.

Type **DIEFFECT**

bUseEnvelope As Long  
 conditionFlags As **CONST\_DICONDITIONFLAGS**  
 conditionX As **DICONDITION**  
 conditionY As **DICONDITION**  
 constantForce As **DICONSTANTFORCE**  
 envelope As **DIENVELOPE**  
 IDuration As Long

---

# **IDH\_DIEFFECT\_dinput\_vb**

---

IFlags As Long  
 IGain As Long  
 ISamplePeriod As Long  
 IStartDelay As Long  
 ITriggerButton As Long  
 ITriggerRepeatInterval As Long  
 periodicForce As DIPERIODICFORCE  
 rampForce As DIRAMPFORCE  
 x As Long  
 y As Long  
 End Type

## Members

### bUseEnvelope

True if the envelope described in the **envelope** member is to be applied to the effect.

### conditionFlags

Flags from the **CONST\_DICONDITIONFLAGS** enumeration. In the current version of DirectX for Visual Basic, this value should be 0.

### conditionX

**DICONDITION** type describing parameters of the condition on the x-axis. Ignored for other types of effects.

### conditionY

**DICONDITION** type describing parameters of the condition on the y-axis. Ignored for other types of effects.

### constantForce

**DICONSTANTFORCE** type describing parameters of a constant force. Ignored for other types of effects.

### envelope

**DIENVELOPE** type describing parameters of an envelope to be applied to the effect. Valid only if the **bUseEnvelope** member is True.

### IDuration

Duration of the effect, in microseconds. A value of -1 indicates infinite duration. If an envelope is applied to an effect of infinite duration, then the attack will be applied, followed by an infinite sustain.

### IFlags

Zero or more members of the **CONST\_DIEFFFLAGS** enumeration specifying how other members are to be interpreted.

### IGain

The gain to be applied to the effect, in the range 0 to 10,000. The gain is a scaling factor applied to all magnitudes of the effect and its envelope.

### ISamplePeriod

The period, in microseconds, at which the device samples the effect—in other words, the granularity of changes in force. A value of 0 indicates that the default playback sample rate should be used.

If the device is not capable of sampling the effect at the specified rate, it will choose the supported rate that is closest to the requested value.

Setting a custom sample period can be used for special effects. For example, playing a sine wave with an artificially large sample period results in a rougher texture.

#### **lStartDelay**

Time, in microseconds, the device should wait after a **DirectInputEffect.Start** call before playing the effect. If this value is 0, then effect playback begins immediately.

#### **lTriggerButton**

Offset value of the button that will trigger the effect. This should be one of the members of the **CONST\_DJOYSTICKOFS** enumeration, or -1 to indicate that the effect does not have a trigger button.

#### **lTriggerRepeatInterval**

The interval, in microseconds, between the end of one playback and the start of the next when the effect is triggered by a button press and the button is held down. Setting this value to -1 suppresses repetition.

Not all devices support trigger repeat.

#### **periodicForce**

**DIPERIODICFORCE** type describing parameters of a periodic effect. Ignored for other types of effects.

#### **rampForce**

**DIRAMPFORCE** type describing parameters of a ramp force. Ignored for other types of effects.

#### **x**

Direction of the effect. Normally, this is the amount by which the direction is rotated from north (usually the negative y-axis), in hundredths of degrees. Thus a value of 0 indicates a force pushing toward the user, a value of 9000 indicates a force pushing from the user's right, and so on. In this case, **y** should be 0.

If **lFlags** contains **DIEFF\_CARTESIAN**, this is a Cartesian value describing the relative amount of force on the x-axis. For example, if **x** = -1 and **y** = 1, the direction of the force is from the southwest. For more information, see **Effect Direction**.

#### **y**

If **lFlags** contains **DIEFF\_CARTESIAN**, this is a Cartesian value describing the relative amount of force on the y-axis. Otherwise it should be 0.

## DIENVELOPE

# The **DIENVELOPE** type describes parameters for an envelope in the **DIEFFECT** type.

```
Type DIENVELOPE
  IAttackLevel As Long
  IAttackTime As Long
  IFadeLevel As Long
  IFadeTime As Long
End Type
```

### Members

#### IAttackLevel

Amplitude for the start of the envelope, relative to the baseline (offset), in the range 0 to 10,000. If the effect's type-specific data does not specify a baseline, then the amplitude is relative to zero.

#### IAttackTime

The time, in microseconds, to reach the sustain level.

#### IFadeLevel

Amplitude for the end of the envelope, relative to the baseline, in the range 0 to 10,000. If the effect's type-specific data does not specify a baseline, then the amplitude is relative to zero.

#### IFadeTime

The time, in microseconds, to reach the fade level.

## DIJOYSTATE

# The **DIJOYSTATE** type contains information about the state of a joystick device. (This term includes other controllers such as game pads and steering wheels). This type is used with the **DirectInputDevice.GetDeviceStateJoystick** method.

```
Type DIJOYSTATE
  buttons(0 To 31) As Byte
  POV(0 To 3) As Long
  rx As Long
  ry As Long
  rz As Long
  slider(0 To 1) As Long
  x As Long
  y As Long
  z As Long
```

---

# IDH\_DIENVELOPE\_dinput\_vb

# IDH\_DIJOYSTATE\_dinput\_vb

End Type

## Members

### buttons

Array of button states. The high-order bit of the byte is set if the corresponding button is down and clear if the button is up or does not exist.

### POV

The current position of up to four direction controllers (such as point-of-view hats). The position is indicated in hundredths of degrees clockwise from north (away from the user). The center position is normally reported as -1; but see Remarks. For indicators that have only five positions, the value for a controller will be -1, 0, 9,000, 18,000, or 27,000.

### rx

Information about the joystick x-axis rotation. If the joystick does not have this, the value is 0.

### ry

Information about the joystick y-axis rotation. If the joystick does not have this axis, the value is 0.

### rz

Information about the joystick z-axis rotation (often called the rudder). If the joystick does not have this axis, the value is 0.

### slider

Two additional axis values whose semantics depend on the joystick. Use the **DirectInputDevice.GetObjectInfo** method to obtain semantic information about these values.

### x

Information about the joystick x-axis (usually the left-right movement of a stick).

### y

Information about the joystick y-axis (usually the forward-backward movement of a stick).

### z

Information about the joystick z-axis (often the throttle control). If the joystick does not have this axis, the value is zero.

## Remarks

You must prepare the device for joystick-style access by calling the **DirectInputDevice.SetCommonDataFormat** method, passing the `DIFORMAT_JOYSTICK` format constant.

If an axis is in relative mode, then the appropriate member contains the change in position. If it is in absolute mode, then the member contains the absolute axis position.

Some drivers report the centered position of the POV indicator as 65,535. You can determine whether an indicator is centered as follows:



---

Dim POVCentered as Boolean  
POVCentered = MyDijoystate.POV(0) And &HFFFF;

## See Also

### DIJOYSTATE2

# DIJOYSTATE2

# The **DIJOYSTATE2** type contains information about the state of a joystick device with extended capabilities. This type is used with the **DirectInputDevice.GetDeviceStateJoystick2** method.

Most applications do not need to use this type, which is for highly specialized controllers including force-feedback devices. For standard game controllers, use the **DIJOYSTATE** type and obtain data by calling **DirectInputDevice.GetDeviceStateJoystick**.

Type DIJOYSTATE2  
  buttons(0 To 31) As Byte  
  frx As Long  
  fry As Long  
  frz As Long  
  fslider(0 To 1) As Long  
  fx As Long  
  fy As Long  
  fz As Long  
  POV(0 To 3) As Long  
  rx As Long  
  ry As Long  
  rz As Long  
  slider(0 To 1) As Long  
  vrx As Long  
  vry As Long  
  vrz As Long  
  vslider(0 To 1) As Long  
  vx As Long  
  vy As Long  
  vz As Long  
  x As Long  
  y As Long  
  z As Long  
End Type

---

# IDH\_DIJOYSTATE2\_dinput\_vb

## Members

### **buttons**

Array of button states. The high-order bit of the byte is set if the corresponding button is down and clear if the button is up or does not exist.

### **frx**

Information about the x-axis torque.

### **fry**

Information about the y-axis torque.

### **frz**

Information about the z-axis torque.

### **fslider**

Information about extra axis forces.

### **fx**

Information about the x-axis force.

### **fy**

Information about the y-axis force.

### **fz**

Information about the z-axis force.

### **POV**

The current position of up to four direction controllers (such as point-of-view hats). The position is indicated in hundredths of degrees clockwise from north (away from the user). The center position is normally reported as -1; but see Remarks. For indicators that have only five positions, the value for a controller will be -1, 0, 9,000, 18,000, or 27,000.

### **rx**

Information about the joystick x-axis rotation. If the joystick does not have this, the value is 0.

### **ry**

Information about the joystick y-axis rotation. If the joystick does not have this axis, the value is 0.

### **rz**

Information about the joystick z-axis rotation (often called the rudder). If the joystick does not have this axis, the value is zero.

### **slider**

Two additional axis values whose semantics depend on the joystick. Use the **DirectInputDevice.GetObjectInfo** method to obtain semantic information about these values.

### **vrx**

Information about the x-axis angular velocity.

### **vry**

Information about the y-axis angular velocity.

### **vrz**

Information about the z-axis angular velocity.

**vslider[0 To 1]**

Information about extra axis velocities.

**vx**

Information about the x-axis velocity.

**vy**

Information about the y-axis velocity.

**vz**

Information about the z-axis velocity.

**x**

Information about the joystick x-axis (usually the left-right movement of a stick).

**y**

Information about the joystick y-axis (usually the forward-backward movement of a stick).

**z**

Information about the joystick z-axis (often the throttle control). If the joystick does not have this axis, the value is zero.

**Remarks**

You must prepare the device for access to a joystick with extended capabilities by calling the **DirectInputDevice.SetCommonDataFormat** method, passing the DIFORMAT\_JOYSTICK2 data format variable.

If an axis is in relative mode, then the appropriate member contains the change in position. If it is in absolute mode, then the member contains the absolute axis position.

Some drivers report the centered position of the POV indicator as 65,535. You can determine whether an indicator is centered as follows:

```
Dim POVCentered as Boolean
POVCentered = MyDijoystate2.POV(0) And &HFFFF;
```

**DIKEYBOARDSTATE**

# The **DIKEYBOARDSTATE** type contains information about the state of keyboard keys. This type is used with the **DirectInputDevice.GetDeviceStateKeyboard** method.

```
Type DIKEYBOARDSTATE
    key(0 To 255) As Byte
End Type
```

---

```
# IDH_DIKEYBOARDSTATE_dinput_vb
```

## Members

### key

Array of key states. The array can be indexed by using members of the **CONST\_DIKEYFLAGS** enumeration. For each key, the high bit is set if the key is down, and clear if the key is up or does not exist.

## Remarks

The following example checks to see if the **Esc** key is being pressed:

```
Dim keyState as DIKEYBOARDSTATE
' diDevice is a valid DirectInputDevice object.

Call diDevice.GetDeviceStateKeyboard(keyState)
If (keyState.key(DIK_ESCAPE) And &H80) Then
    ' Key is down
End If
```

# DIMOUSESTATE

# The **DIMOUSESTATE** type contains information about the state of a mouse device or another device that is being accessed as if it were a mouse device. This type is used with the **DirectInputDevice.GetDeviceStateMouse** method.

```
Type DIMOUSESTATE
    buttons(0 To 3) As Byte
    x As Long
    y As Long
    z As Long
End Type
```

## Members

### buttons

Array of button states. The high-order bit of the byte is set if the corresponding button is down.

### x

Information about the mouse x-axis.

### y

Information about the mouse y-axis.

### z

Information about the mouse z-axis (typically a wheel). If the mouse does not have a z-axis, then the value is 0.

---

# IDH\_DIMOUSESTATE\_dinput\_vb

## Remarks

Immediate data is returned in this type from a device that has been prepared by passing the `DIFORMAT_MOUSE` constant to the **DirectInputDevice.SetCommonDataFormat** method.

If an axis is in relative mode, then the appropriate member contains the change in position since the last call to this method. If the axis is in absolute mode, then the member contains the accumulated relative motion in relation to an arbitrary start point. The absolute axis position is not meaningful except in comparison with other absolute axis positions.

# DIOBJECTDATAFORMAT

# The **DIOBJECTDATAFORMAT** type contains information about a device object's data format for use with the **DirectInputDevice.SetDataFormat** method.

```
Type DIOBJECTDATAFORMAT {
    IFlags As CONST_DIDEVICEOBJINSTANCEFLAGS
    IOfs As Long
    IType As CONST_DIDFTFLAGS
    strGuid As String
End Type
```

## Members

### IFlags

Zero or more of the following values from the **CONST\_DIDEVICEOBJINSTANCEFLAGS** enumeration.

**DIDOI\_ASPECTACCEL**

The object selected by **DirectInput.SetDataFormat** must report acceleration information.

**DIDOI\_ASPECTFORCE**

The object selected by **DirectInput.SetDataFormat** must report force information.

**DIDOI\_ASPECTPOSITION**

The object selected by **DirectInput.SetDataFormat** must report position information.

**DIDOI\_ASPECTVELOCITY**

The object selected by **DirectInput.SetDataFormat** must report velocity information.

### IOfs

Byte offset within the data packet where the data for the input source will be stored. This value must be a multiple of four for **Long** size data, such as axes. It can be byte-aligned for buttons.

---

# `IDH_DIOBJECTDATAFORMAT_dinput_vb`

**IType**

Device type that describes the object. It is a combination of values from the **CONST\_DIDFTFLAGS** enumeration describing the object type (axis, button, and so forth) and containing the object-instance number in the middle 16 bits.

**strGuid**

Unique identifier for the type of input source. An empty string indicates that any type of object is permissible.

The following strings can be used in place of actual GUID strings to identify the type of device object:

GUID\_XAxis  
 GUID\_YAxis  
 GUID\_ZAxis  
 GUID\_RxAxis  
 GUID\_RyAxis  
 GUID\_RzAxis  
 GUID\_Slider  
 GUID\_Button  
 GUID\_Key  
 GUID\_POV

## DIPERIODICFORCE

# The **DIPERIODICFORCE** type describes parameters for a periodic force in the **DIEFFECT** type.

Type DIPERIODICFORCE

IMagnitude As Long

IOffset As Long

IPhase As Long

IPeriod As Long

End Type

**Members****IMagnitude**

The magnitude of the effect, in the range 0 to 10,000. If an envelope is applied to this effect, then the value represents the magnitude of the sustain. If no envelope is applied, then the value represents the amplitude of the entire effect.

**IOffset**

The offset of the effect. The range of forces generated by the effect will be **IOffset - IMagnitude** to **IOffset + IMagnitude**. The value of the **IOffset** member is also the baseline for any envelope that is applied to the effect.

**IPhase**

The position in the cycle of the periodic effect at which playback begins, in the range 0 to 35,999. See Remarks.

---

# IDH\_DIPERIODICFORCE\_dinput\_vb

**IPeriod**

The period of the effect, in microseconds.

**Remarks**

A device driver might not provide support for all values in the **IPhase** member. In this case the value will be rounded off to the nearest supported value.

## DIPROPLONG

# The **DIPROPLONG** type is used to store property information to be set on or retrieved from the input device by using the **DirectInputDevice.SetProperty** and **DirectInputDevice.GetProperty** methods, where the property is a single value.

Type DIPROPLONG

  IData As Long

  IHow As Long

  IObj As Long

  ISize As Long

End Type

**Members****IData**

The property-specific value being set or retrieved.

**IHow**

Value specifying how the *IObj* parameter should be interpreted. This value may be one of the members of the **CONST\_DIPHFLAGS** enumeration.

If **IObj** is **DIPROP\_AXISMODE** or **DIPROP\_BUFFERSIZE**, **IHow** should be **DIPH\_DEVICE**, because these properties cannot be set for an individual object.

**IObj**

Object for which the property is to be accessed.

If the **IHow** member is **DIPH\_BYID**, this member must be the identifier for the object whose property setting is to be set or retrieved. This value can be retrieved for the object by using the **DirectInputDeviceObjectInstance.GetType** method.

If the **IHow** member is **DIPH\_BYOFFSET**, this member must be a data format offset for the object whose property setting is to be set or retrieved. This value can be obtained by using the **DirectInputDeviceObjectInstance.GetOfs** method.

If **IHow** is **DIPH\_DEVICE**, this value should be 0.

**ISize**

Size of this type.

---

# IDH\_DIPROPLONG\_dinput\_vb

## Remarks

All members must be initialized with the proper values before the type is passed to the **SetProperty** method. All members except **IData** must be initialized with the proper values before the type is passed to **GetProperty**.

## See Also

DIPROPRANGE

# DIPROPRANGE

# The **DIPROPRANGE** type is used to store property information to be set on or retrieved from the input device by using the **DirectInputDevice.SetProperty** and **DirectInputDevice.GetProperty** methods, where the property is a range of values.

Type DIPROPRANGE

IHow As Long

IMax As Long

IMin As Long

IObj As Long

ISize As Long

End Type

## Members

### IHow

Value specifying how the **IObj** member should be interpreted. This value may be one of the members of the **CONST\_DIPHFLAGS** enumeration.

### IMax

Upper limit of the range. If the range of the device is unrestricted, this value will be **DIPROPRANGE\_NOMAX** (from the **CONST\_DINPUT** enumeration) when the **DirectInputDevice.GetProperty** method returns.

### IMin

Lower limit of the range. If the range of the device is unrestricted, this value will be **DIPROPRANGE\_NOMIN** (from the **CONST\_DINPUT** enumeration) when the **DirectInputDevice.GetProperty** method returns.

### IObj

Object for which the property is to be accessed.

If the **IHow** member is **DIPH\_BYID**, this member must be the identifier for the object whose property setting is to be set or retrieved. This value can be retrieved for the object by using the **DirectInputDeviceObjectInstance.GetType** method.

If the **IHow** member is **DIPH\_BYOFFSET**, this member must be a data format offset for the object whose property setting is to be set or retrieved. This value

# IDH\_DIPROPRANGE\_dinput\_vb



can be obtained by using the **DirectInputDeviceObjectInstance.GetOfs** method.

If **IHow** is **DIPH\_DEVICE**, this value should be 0.

**ISize**

Size of this type.

**See Also**

**DIPROPLONG**

## DIRAMPFORCE

# The **DIRAMPFORCE** type describes parameters for a ramp force in the **DIEFFECT** type.

Type DIRAMPFORCE  
IRangeEnd As Long  
IRangeStart As Long  
End Type

**Members****IRangeEnd**

The magnitude at the end of the effect, in the range -10,000 to +10,000.

**IRangeStart**

The magnitude at the start of the effect, in the range -10,000 to +10,000.

**Remarks**

The duration of a ramp force effect must be finite.

## Enumerations

DirectInput uses enumerations to group constants in order to take advantage of the statement completion feature of Visual Basic. The enumerations used in DirectInput are:

- **CONST\_DICOMMONDATAFORMATS**
- **CONST\_DICONDITIONFLAGS**
- **CONST\_DIDATAFORMATFLAGS**
- **CONST\_DIDEVCAPSFLAGS**
- **CONST\_DIDEVICEOBJINSTANCEFLAGS**
- **CONST\_DIDEVICETYPE**

---

# IDH\_DIRAMPFORCE\_dinput\_vb

- 
- **CONST\_DIDFTFLAGS**
  - **CONST\_DIDGDDFLAGS**
  - **CONST\_DIEFFFLAGS**
  - **CONST\_DIEFTFLAGS**
  - **CONST\_DIEGESFLAGS**
  - **CONST\_DIENUMDEVICESFLAGS**
  - **CONST\_DIEPFLAGS**
  - **CONST\_DIESFLAGS**
  - **CONST\_DIGFFSFLAGS**
  - **CONST\_DIJOYSTICKOFS**
  - **CONST\_DIKEYFLAGS**
  - **CONST\_DIMOUSEOFS**
  - **CONST\_DINPUT**
  - **CONST\_DINPUTERR**
  - **CONST\_DIPHFLAGS**
  - **CONST\_DISCLFLAGS**
  - **CONST\_DISDDFLAGS**
  - **CONST\_DISFFCFLAGS**

## CONST\_DICOMMONDATAFORMAT S

# Members of the **CONST\_DICOMMONDATAFORMATS** enumeration are used to specify the data format in the *format* parameter of the **DirectInputDevice.SetCommonDataFormat** method.

```
Enum CONST_DICOMMONDATAFORMATS
```

```
    DIFORMAT_JOYSTICK = 3
```

```
    DIFORMAT_JOYSTICK2 = 4
```

```
    DIFORMAT_KEYBOARD = 1
```

```
    DIFORMAT_MOUSE = 2
```

```
End Enum
```

```
DIFORMAT_JOYSTICK
```

Joystick whose state data can be received in a **DIJOYSTATE** type.

```
DIFORMAT_JOYSTICK2
```

Joystick with extended capabilities whose state data can be received in a **DIJOYSTATE2** type.

```
DIFORMAT_KEYBOARD
```

Keyboard whose state data can be received in a **DIKEYBOARDSTATE** type.

---

```
# IDH_CONST_DICOMMONDATAFORMATS_dinput_vb
```

**DIFORMAT\_MOUSE**

Mouse whose state data can be received in a **DIMOUSESTATE** type.

## **CONST\_DICONDITIONFLAGS**

# The **CONST\_DICONDITIONFLAGS** enumeration is used in the **conditionFlags** member of the **DIEFFECT** type to set the axes and direction of a force-feedback effect.

```
Enum CONST_DICONDITIONFLAGS
    DICONDITION_USE_BOTH_AXES = 1
    DICONDITION_USE_DIRECTION = 2
End Enum
```

**DICONDITION\_USE\_BOTH\_AXES**

Not currently supported.

**DICONDITION\_USE\_DIRECTION**

Not currently supported.

## **CONST\_DIDATAFORMATFLAGS**

# The **CONST\_DIDATAFORMATFLAGS** enumeration is used in the **IFlags** member of the **DIDATAFORMAT** type to describe additional attributes of the data format.

```
Enum CONST_DIDATAFORMATFLAGS
    DIDF_ABSAXIS = 1
    DIDF_RELAXIS = 2
End Enum
DIDF_ABSAXIS
```

The axes are in absolute mode. Setting this flag in the data format is equivalent to manually setting the axis mode property using the **DirectInputDevice.SetProperty** method. This may not be combined with DIDF\_RELAXIS flag.

**DIDF\_RELAXIS**

The axes are in relative mode. Setting this flag in the data format is equivalent to manually setting the axis mode property using the **DirectInputDevice.SetProperty** method. This may not be combined with the DIDF\_ABSAXIS flag.

---

# IDH\_CONST\_DICONDITIONFLAGS\_dinput\_vb

# IDH\_CONST\_DIDATAFORMATFLAGS\_dinput\_vb

## CONST\_DIDEVCAPSFLAGS

# The **CONST\_DIDEVCAPSFLAGS** enumeration is used in the **IFlags** member of the **DIDEVCAPS** type to describe the DirectInput device.

Enum **CONST\_DIDEVCAPSFLAGS**

```

DIDC_ATTACHED =          1
DIDC_DEADBAND =        16384 (&H4000)
DIDC_EMULATED =          4
DIDC_FFATTACK =         512 (&H200)
DIDC_FFFADE =         1024 (&H400)
DIDC_FORCEFEEDBACK =    256 (&H100)
DIDC_POLLEDDATAFORMAT =    8
DIDC_POLLEDDEVICE =      2
DIDC_POSNEGCOEFFICIENTS = 4096 (&H1000)
DIDC_POSNEGSATURATION =  8192 (&H2000)
DIDC_SATURATION =       2048 (&H800)

```

End Enum

**DIDC\_ATTACHED**

The device is physically attached.

**DIDC\_DEADBAND**

The device supports deadband for at least one force-feedback condition.

**DIDC\_EMULATED**

If this flag is set, the data is coming from a user mode device interface (such as HID) or by some other ring 3 means. If it is not set, the data is coming directly from a kernel mode driver.

**DIDC\_FFATTACK**

The force-feedback system supports the **attack** parameter for at least one effect. If the device does not support attack then the **IAttackLevel** and **IAttackTime** members of the **DIENVELOPE** type will be ignored by the device.

**DIDC\_FFFADE**

The force-feedback system supports the **fade** parameter for at least one effect. If the device does not support fade then the **IFadeLevel** and **IFadeTime** members of the **DIENVELOPE** type will be ignored by the device.

**DIDC\_FORCEFEEDBACK**

The device supports force feedback.

**DIDC\_POLLEDDATAFORMAT**

At least one object in the current data format is polled rather than interrupt-driven. For these objects, the application must explicitly call the **DirectInputDevice.Poll** method in order to obtain data.

**DIDC\_POLLEDDEVICE**

At least one object on the device is polled rather than interrupt-driven. For these objects, the application must explicitly call the **DirectInputDevice.Poll** method

---

# IDH\_CONST\_DIDEVCAPSFLAGS\_dinput\_vb

in order to obtain data. HID devices may contain a mixture of polled and non-polled objects.

#### DIDC\_POSNEGCOEFFICIENTS

The force-feedback system supports two coefficient values for conditions (one for the positive displacement of the axis and one for the negative displacement of the axis) for at least one condition. If the device does not support both coefficients, then the negative coefficient in the **DICONDITION** type will be ignored.

#### DIDC\_POSNEGSATURATION

The force-feedback system supports a maximum saturation for both positive and negative force output for at least one condition. If the device does not support both saturation values, then the negative saturation in the **DICONDITION** structure will be ignored.

#### DIDC\_SATURATION

The force-feedback system supports the saturation of condition effects for at least one condition. If the device does not support saturation, then the force generated by a condition is limited only by the maximum force which the device can generate.

### See Also

#### CONST\_DIEFTFLAGS

## CONST\_DIDEVICEOBJINSTANCEFL LAGS

# Members of the **CONST\_DIDEVICEOBJINSTANCEFLAGS** enumeration describe device object capabilities and are returned by the **DirectInputDeviceObjectInstance.GetFlags** method. They are also present in the **DIOBJECTDATAFORMAT** type passed to the **DirectInputDevice.SetDataFormat** method

Enum CONST\_DIDEVICEINSTANCEFLAGS

```
DIDOI_ASPECTACCEL = 768 (&H300)
DIDOI_ASPECTFORCE = 1024 (&H400)
DIDOI_ASPECTMASK = 3840 (&HF00)
DIDOI_ASPECTPOSITION = 256 (&H100)
DIDOI_ASPECTVELOCITY = 512 (&H200)
DIDOI_FFACTUATOR = 1
DIDOI_FFEFFECTTRIGGER = 2
DIDOI_POLLED = 32768 (&H8000)
```

End Enum

DIDOI\_ASPECTACCEL

---

# IDH\_CONST\_DIDEVICEOBJINSTANCEFLAGS\_dinput\_vb

The object reports acceleration information.

DIDOI\_ASPECTFORCE

The object reports force information.

DIDOI\_ASPECTMASK

The bits that are used to report aspect information. An object can represent at most one aspect.

DIDOI\_ASPECTPOSITION

The object reports position information.

DIDOI\_ASPECTVELOCITY

The object reports velocity information.

DIDOI\_FFACTUATOR

The object can have force-feedback effects applied to it.

DIDOI\_FFEFFECTTRIGGER

The object can trigger playback of force-feedback effects.

DIDOI\_POLLED

The object does not return data until the **DirectInputDevice.Poll** method is called.

## Remarks

The only one of these flags that is of interest for applications developed with DirectX for Visual Basic is DIDIO\_POLLED.

# CONST\_DIDEVICETYPE

# Members of the **CONST\_DIDEVICETYPE** enumeration are used to identify the input device type and subtype. A packed value representing the type and subtype is returned by the **DirectInputDeviceInstance.GetDevType** method and in the **IDevType** member of the **DIDEVCAPS** type returned by **DirectInputDevice.GetCapabilities**. A value representing a primary type is passed as the *deviceType* parameter to the **DirectInput.GetDIEnumDevices** method.

For a table of members listed by type and subtype, see **DirectInputDeviceInstance.GetDevType**.

```
Enum CONST_DIDEVICETYPE
    DIDEVTYPE_DEVICE          = 1
    DIDEVTYPE_HID             = 65536 (&H10000)
    DIDEVTYPE_JOYSTICK        = 4
    DIDEVTYPE_KEYBOARD        = 3
    DIDEVTYPE_MOUSE           = 2
    DIDEVTYPEJOYSTICK_FLIGHTSTICK = 3
    DIDEVTYPEJOYSTICK_GAMEPAD  = 4
    DIDEVTYPEJOYSTICK_HEADTRACKER = 7
    DIDEVTYPEJOYSTICK_RUDDER   = 5
```

# IDH\_CONST\_DIDEVICETYPE\_dinput\_vb

```

DIDEVTYPEJOYSTICK_TRADITIONAL = 2
DIDEVTYPEJOYSTICK_UNKNOWN     = 1
DIDEVTYPEJOYSTICK_WHEEL       = 6
DIDEVTYPEKEYBOARD_J3100       = 12
DIDEVTYPEKEYBOARD_JAPAN106    = 10
DIDEVTYPEKEYBOARD_JAPANAX     = 11
DIDEVTYPEKEYBOARD_NEC98       = 7
DIDEVTYPEKEYBOARD_NEC98106    = 9
DIDEVTYPEKEYBOARD_NEC98LAPTOP = 8
DIDEVTYPEKEYBOARD_NOKIA1050   = 5
DIDEVTYPEKEYBOARD_NOKIA9140   = 6
DIDEVTYPEKEYBOARD_OLIVETTI    = 2
DIDEVTYPEKEYBOARD_PCAT        = 3
DIDEVTYPEKEYBOARD_PCENH       = 4
DIDEVTYPEKEYBOARD_PCXT        = 1
DIDEVTYPEKEYBOARD_UNKNOWN     = 0
DIDEVTYPEMOUSE_FINGERSTICK    = 3
DIDEVTYPEMOUSE_TOUCHPAD       = 4
DIDEVTYPEMOUSE_TRACKBALL      = 5
DIDEVTYPEMOUSE_TRADITIONAL    = 2
DIDEVTYPEMOUSE_UNKNOWN        = 1

```

End Enum

**DIDEVTYPE\_DEVICE**

A device that does not fall into the other categories.

**DIDEVTYPE\_HID**

The device uses the Human Interface Device (HID) protocol.

**DIDEVTYPE\_JOYSTICK**

A joystick or similar device, such as a steering wheel.

**DIDEVTYPE\_KEYBOARD**

A keyboard or keyboard-like device.

**DIDEVTYPE\_MOUSE**

A mouse or mouse-like device (such as a trackball).

**DIDEVTYPEJOYSTICK\_FLIGHTSTICK**

A joystick optimized for flight simulation.

**DIDEVTYPEJOYSTICK\_GAMEPAD**

A device whose primary purpose is to provide button input.

**DIDEVTYPEJOYSTICK\_HEADTRACKER**

A device that tracks the movement of the user's head.

**DIDEVTYPEJOYSTICK\_RUDDER**

A device for yaw control.

**DIDEVTYPEJOYSTICK\_TRADITIONAL**

A traditional joystick.

**DIDEVTYPEJOYSTICK\_UNKNOWN**

The subtype could not be determined.

DIDEVTYPEJOYSTICK\_WHEEL

A steering wheel.

DIDEVTYPEKEYBOARD\_J3100

Japanese J3100 keyboard.

DIDEVTYPEKEYBOARD\_JAPAN106

Japanese 106-key keyboard.

DIDEVTYPEKEYBOARD\_JAPANAX

Japanese AX keyboard.

DIDEVTYPEKEYBOARD\_NEC98

Japanese NEC PC98 keyboard.

DIDEVTYPEKEYBOARD\_NEC98106

Japanese NEC PC98 106-key keyboard.

DIDEVTYPEKEYBOARD\_NEC98LAPTOP

Japanese NEC PC98 laptop keyboard.

DIDEVTYPEKEYBOARD\_NOKIA1050

Nokia 1050 keyboard.

DIDEVTYPEKEYBOARD\_NOKIA9140

Nokia 9140 keyboard.

DIDEVTYPEKEYBOARD\_OLIVETTI

Olivetti 102-key keyboard.

DIDEVTYPEKEYBOARD\_PCAT

IBM PC/AT 84-key keyboard.

DIDEVTYPEKEYBOARD\_PCENH

IBM PC Enhanced 101/102-key or Microsoft Natural® keyboard.

DIDEVTYPEKEYBOARD\_PCXT

IBM PC/XT 83-key keyboard.

DIDEVTYPEKEYBOARD\_UNKNOWN

The subtype could not be determined.

DIDEVTYPEMOUSE\_FINGERSTICK

The device is a fingerstick.

DIDEVTYPEMOUSE\_TOUCHPAD

The device is a touchpad.

DIDEVTYPEMOUSE\_TRACKBALL

The device is a trackball.

DIDEVTYPEMOUSE\_TRADITIONAL

The device is a traditional mouse.

DIDEVTYPEMOUSE\_UNKNOWN

The subtype could not be determined.



## CONST\_DIDFTFLAGS

# Members of the **CONST\_DIDFTFLAGS** enumeration are used in the *flags* parameter of the **DirectInputDevice.GetDeviceObjectsEnum** method to specify the type of device object to enumerate. These values are also returned by the **DirectInputDeviceObjectInstance.GetFlags** method to describe capabilities of the device object.

```
Enum CONST_DIDFTFLAGS
    DIDFT_ABSAXIS      =    2
    DIDFT_ALL          =    0
    DIDFT_ANYINSTANCE  = 16776960 (&HFFFF00)
    DIDFT_AXIS        =    3
    DIDFT_BUTTON       =   12
    DIDFT_COLLECTION   =   64 (&H40)
    DIDFT_FFACTUATOR   = 16777216 (&H1000000)
    DIDFT_FFEFFECTTRIGGER = 33554432 (&H2000000)
    DIDFT_INSTANCEMASK = 16776960 (&HFFFF00)
    DIDFT_NOCOLLECTION = 16776960 (&HFFFF00)
    DIDFT_NODATA       =   128 (&H80)
    DIDFT_POV          =   16 (&H10)
    DIDFT_PSHBUTTON    =    4
    DIDFT_RELAXIS      =    1
    DIDFT_TGLBUTTON    =    8
```

End Enum

**DIDFT\_ABSAXIS**

An absolute axis.

**DIDFT\_ALL**

All objects.

**DIDFT\_ANYINSTANCE**

Any instance of an object.

**DIDFT\_AXIS**

An axis, either absolute or relative.

**DIDFT\_BUTTON**

A push button or a toggle button.

**DIDFT\_COLLECTION**

A HID link collection. HID link collections do not generate data of their own.

**DIDFT\_FFACTUATOR**

An object that contains a force-feedback actuator. In other words, forces can be applied to this object.

**DIDFT\_FFEFFECTTRIGGER**

An object that can be used to trigger force-feedback effects.

**DIDFT\_INSTANCEMASK**

---

# IDH\_CONST\_DIDFTFLAGS\_dinput\_vb

Same as DIDFT\_ANYINSTANCE.

#### DIDFT\_NOCOLLECTION

An object that does not belong to any HID link collection; in other words, an object for which **DirectInputDeviceObjectInstance.GetCollectionNumber** returns 0.

#### DIDFT\_NODATA

An object that does not generate data.

#### DIDFT\_POV

A point-of-view controller.

#### DIDFT\_PSHBUTTON

A push button. A push button is reported as down when the user presses it and as up when the user releases it.

#### DIDFT\_RELAXIS

A relative axis.

#### DIDFT\_TGLBUTTON

A toggle button. A toggle button is reported as down when the user presses it and remains so until the user presses the button a second time.

## CONST\_DIDGDDFLAGS

# Members of the **CONST\_DIDGDDFLAGS** enumeration are used in the *flags* parameter of the **DirectInputDevice.GetDeviceData** method to control the manner in which data is obtained.

```
Enum CONST_DIDGDDFLAGS
```

```
    DIGDD_DEFAULT = 0
```

```
    DIGDD_PEEK    = 1
```

```
End Enum
```

#### DIGDD\_DEFAULT

Remove retrieved items from the buffer.

#### DIGDD\_PEEK

Do not remove retrieved items from the buffer. A subsequent **GetDeviceData** call will read the same data.

## CONST\_DIEFFFLAGS

# Members of the **CONST\_DIEFFFLAGS** enumeration are used in the **IFlags** member of the **DIEFFECT** type.

```
Enum CONST_DIEFFFLAGS
```

```
    DIEFF_CARTESIAN = 16 (&H10)
```

```
    DIEFF_OBJECTOFFSETS = 2
```

```
    DIEFF_POLAR = 32 (&H20)
```

```
# IDH_CONST_DIDGDDFLAGS_dinput_vb
```

```
# IDH_CONST_DIEFFFLAGS_dinput_vb
```

---

End Enum

#### DIEFF\_CARTESIAN

The direction of the effect is given in Cartesian coordinates. **DIEFFECT.x** and **DIEFFECT.y** contain valid values.

#### DIEFF\_OBJECTOFFSETS

The value of the **ITriggerButton** member of **DIEFFECT** is the offset of the button in the data structure for the device.

#### DIEFF\_POLAR

The direction of the effect is given in polar coordinates. **DIEFFECT.x** contains a valid value.

### Remarks

The default behavior of **DirectInputDevice.CreateEffect** and **DirectInputDevice.CreateCustomEffect** is to create the effect as if **DIEFFECT.IfFlags** contained (DIEFF\_OBJECTOFFSETS **Or** DIEFF\_POLAR). In other words, it is not necessary to specify these flags.

## CONST\_DIEFTFLAGS

# Members of the **CONST\_DIEFTFLAGS** enumeration describe types and capabilities of force-feedback effects. They are used to restrict effect enumeration in the **DirectInputDevice.GetEffectsEnum** method to one or more of the primary types. They are also returned by the **DirectInputEnumEffects.GetType** method.

Enum CONST\_DIEFTFLAGS

```

DIEFT_ALL = 0
DIEFT_CONDITION = 4
DIEFT_CONSTANTFORCE = 1
DIEFT_CUSTOMFORCE = 5
DIEFT_DEADBAND = 16384 (&H4000)
DIEFT_FFATTACK = 512 (&H200)
DIEFT_FFFADE = 1024 (&H400)
DIEFT_HARDWARE = 255 (&HFF)
DIEFT_PERIODIC = 3
DIEFT_POSNEGCOEFFICIENTS = 4096 (&H1000)
DIEFT_POSNEGSATURATION = 8192 (&H2000)
DIEFT_RAMPFORCE = 2
DIEFT_SATURATION = 2048 (&H800)

```

End Enum

#### DIEFT\_ALL

All effects are to be enumerated.

---

# IDH\_CONST\_DIEFTFLAGS\_dinput\_vb

**DIEFT\_CONDITION**

The effect is a condition, or conditions are to be enumerated.

**DIEFT\_CONSTANTFORCE**

The effect is a constant force, or constant forces are to be enumerated.

**DIEFT\_CUSTOMFORCE**

The effect is a custom force, or custom forces are to be enumerated.

**DIEFT\_DEADBAND**

The effect supports deadband.

**DIEFT\_FFATTACK**

The effect supports attack.

**DIEFT\_FFADE**

The effect supports fade.

**DIEFT\_HARDWARE**

The effect is specific to the hardware, or hardware effects are to be enumerated.

**DIEFT\_PERIODIC**

The effect is periodic, or periodic effects are to be enumerated.

**DIEFT\_POSNEGCOEFFICIENTS**

The condition supports different positive and negative coefficients.

**DIEFT\_POSNEGSATURATION**

The effect supports different positive and negative saturation [\\_dx\\_saturation\\_glos](#).

**DIEFT\_RAMPFORCE**

The effect is a ramp force, or ramp forces are to be enumerated.

**DIEFT\_SATURATION**

The effect supports saturation.

**See Also****CONST\_DIDEVCAPSFLAGS**

## **CONST\_DIEGESFLAGS**

# Members of the **CONST\_DIEGESFLAGS** enumeration are returned by the **DirectInputEffect.GetEffectStatus** method.

Enum **CONST\_DIEGESFLAGS**

**DIEGES\_EMULATED** = 2

**DIEGES\_PLAYING** = 1

End Enum

**DIEGES\_EMULATED**

The effect is emulated.

**DIEGES\_PLAYING**

The effect is playing.

---

# **IDH\_CONST\_DIEGESFLAGS\_dinput\_vb**

## CONST\_DIENUMDEVICESFLAGS

# Members of the **CONST\_DIENUMDEVICESFLAGS** enumeration are used in the *flags* parameter of the **DirectInput.GetDIEnumDevices** method to indicate whether all device, or only attached devices, are to be enumerated.

```
Enum CONST_DIENUMDEVICESFLAGS
```

```
    DIEDFL_ALLDEVICES = 0
```

```
    DIEDFL_ATTACHEDONLY = 1
```

```
End Enum
```

```
DIEDFL_ALLDEVICES
```

All installed devices will be enumerated. This is the default behavior.

```
DIEDFL_ATTACHEDONLY
```

Only attached and installed devices.

## CONST\_DIEPFLAGS

# Members of the **CONST\_DIEPFLAGS** enumeration are passed to the **DirectInputEffect.SetParameters** method in order to specify which parameters are being set, and the subsequent behavior of the effect. Some members are also returned by the **DirectInputEnumEffects.GetStaticParams** and **DirectInputEnumEffects.GetDynamicParams** methods, giving information about support for the parameter.

```
Enum CONST_DIEPFLAGS
```

```
    DIEP_ALLPARAMS = 511 (&H1FF)
```

```
    DIEP_AXES = 32 (&H20)
```

```
    DIEP_DIRECTION = 64 (&H40)
```

```
    DIEP_DURATION = 1
```

```
    DIEP_ENVELOPE = 128 (&H80)
```

```
    DIEP_GAIN = 4
```

```
    DIEP_NODOWNLOAD = -2147483648 (&H80000000)
```

```
    DIEP_NORESTART = 1073741824 (&H40000000)
```

```
    DIEP_SAMPLEPERIOD = 2
```

```
    DIEP_START = 536870912 (&H20000000)
```

```
    DIEP_TRIGGERBUTTON = 8
```

```
    DIEP_TRIGGERREPEATINTERVAL = 16 (&H10)
```

```
    DIEP_TYPESPECIFICPARAMS = 256 (&H100)
```

```
End Enum
```

```
DIEP_ALLPARAMS
```

Not used.

```
DIEP_AXES
```

Not used.

---

```
# IDH_CONST_DIENUMDEVICESFLAGS_dinput_vb
```

```
# IDH_CONST_DIEPFLAGS_dinput_vb
```

**DIEP\_DIRECTION**

The **x** and **y** members of **DIEFFECT** are valid, or the direction parameter is supported. If you are setting or requesting effect parameters, you can specify that the direction is supplied or is to be returned in either Cartesian or polar coordinates by specifying **DIEFF\_CARTESIAN** or **DIEFF\_POLAR** in **DIEFFECT.IfFlags**.

**DIEP\_DURATION**

The **IDuration** member of **DIEFFECT** is valid, or the effect supports changing of the duration.

**DIEP\_ENVELOPE**

The **envelope** member of **DIEFFECT** is valid, or the effect supports the application of an envelope.

**DIEP\_GAIN**

The **IGain** member is valid, or the effect supports the application of gain.

**DIEP\_NODOWNLOAD**

After setting parameters, the effect is not to be downloaded.

**DIEP\_NORESTART**

Suppress the stopping and restarting of the effect in order to change parameters.

**DIEP\_SAMPLEPERIOD**

The **ISamplePeriod** member is valid.

**DIEP\_START**

After setting parameters, start the effect immediately.

**DIEP\_TRIGGERBUTTON**

The **ITriggerButton** member is valid, or the effect supports a trigger button.

**DIEP\_TRIGGERREPEATINTERVAL**

The **ITriggerRepeatInterval** member is valid, or the effect supports trigger repeat.

**DIEP\_TYPESPECIFICPARAMS**

The effect has type-specific parameters that can be changed by the application. This value is returned by **GetStaticParams** for all standard effects and for hardware-specific effects with application-modifiable parameters. DirectX for Visual Basic supports hardware effects only if they do not have this flag set.

## CONST\_DIESFLAGS

# The **CONST\_DIESFLAGS** are used to control behavior of force-feedback effect playback.

```
Enum CONST_DIESFLAGS
```

```
    DIES_NODOWNLOAD = -2147483648 (&H80000000)
```

```
    DIES_SOLO = 1
```

```
End Enum
```

---

```
# IDH_CONST_DIESFLAGS_dinput_vb
```

For an explanation of the members, see **DirectInputEffect.Start**.

## CONST\_DIGFFSFLAGS

# Members of the **CONST\_DIGFFSFLAGS** enumeration describe the state of a force-feedback device. They are returned by the **DirectInputDevice.GetForceFeedbackState** method.

```
Enum CONST_DIGFFSFLAGS
    DIGFFS_ACTUATORSOFF =    32 (&H20)
    DIGFFS_ACTUATORSON =    16 (&H10)
    DIGFFS_DEVICELOST = -2147483648 (&H80000000)
    DIGFFS_EMPTY =          1
    DIGFFS_PAUSED =          4
    DIGFFS_POWEROFF =       128 (&H80)
    DIGFFS_POWERON =        64 (&H40)
    DIGFFS_SAFETYSWITCHOFF = 512 (&H200)
    DIGFFS_SAFETYSWITCHON = 256 (&H100)
    DIGFFS_STOPPED =         2
    DIGFFS_USERFFSWITCHOFF = 2048 (&H800)
    DIGFFS_USERFFSWITCHON = 1024 (&H400)
End Enum
```

### DIGFFS\_ACTUATORSOFF

The device's force-feedback actuators are disabled.

### DIGFFS\_ACTUATORSON

The device's force-feedback actuators are enabled.

### DIGFFS\_DEVICELOST

The device suffered an unexpected failure and is in an indeterminate state. It must be reset either by unacquiring and reacquiring the device, or by sending a **DISFFC\_RESET** command.

### DIGFFS\_EMPTY

The device has no downloaded effects.

### DIGFFS\_PAUSED

Playback of all active effects has been paused.

### DIGFFS\_POWEROFF

The force-feedback system is not currently available. If the device cannot report the power state, then neither **DIGFFS\_POWERON** nor **DIGFFS\_POWEROFF** will be returned.

### DIGFFS\_POWERON

Power to the force-feedback system is currently available. If the device cannot report the power state, then neither **DIGFFS\_POWERON** nor **DIGFFS\_POWEROFF** will be returned.

### DIGFFS\_SAFETYSWITCHOFF

---

# IDH\_CONST\_DIGFFSFLAGS\_dinput\_vb

The safety switch is currently off, meaning that the device cannot operate. If the device cannot report the state of the safety switch, then neither DIGFFS\_SAFETYSWITCHON nor DIGFFS\_SAFETYSWITCHOFF will be returned.

#### DIGFFS\_SAFETYSWITCHON

The safety switch is currently on, meaning that the device can operate. If the device cannot report the state of the safety switch, then neither DIGFFS\_SAFETYSWITCHON nor DIGFFS\_SAFETYSWITCHOFF will be returned.

#### DIGFFS\_STOPPED

No effects are playing and the device is not paused.

#### DIGFFS\_USERFFSWITCHOFF

The user force-feedback switch is currently off, meaning that the device cannot operate. If the device cannot report the state of the user force-feedback switch, then neither DIGFFS\_USERFFSWITCHON nor DIGFFS\_USERFFSWITCHOFF will be returned.

#### DIGFFS\_USERFFSWITCHON

The user force-feedback switch is currently on, meaning that the device can operate. If the device cannot report the state of the user force-feedback switch, then neither DIGFFS\_USERFFSWITCHON nor DIGFFS\_USERFFSWITCHOFF will be returned.

## CONST\_DIJOYSTICKOFS

# The members of the **CONST\_DIJOYSTICKOFS** enumeration represent the offset of the data for the various joystick device objects within the data format.

#### Enum CONST\_DIJOYSTICKOFS

```

DIJOFS_BUTTON0 = 48 (&H30)
DIJOFS_BUTTON1 = 49 (&H31)
DIJOFS_BUTTON10 = 58 (&H3A)
DIJOFS_BUTTON11 = 59 (&H3B)
DIJOFS_BUTTON12 = 60 (&H3C)
DIJOFS_BUTTON13 = 61 (&H3D)
DIJOFS_BUTTON14 = 62 (&H3E)
DIJOFS_BUTTON15 = 63 (&H3F)
DIJOFS_BUTTON16 = 64 (&H40)
DIJOFS_BUTTON17 = 65 (&H41)
DIJOFS_BUTTON18 = 66 (&H42)
DIJOFS_BUTTON19 = 67 (&H43)
DIJOFS_BUTTON2 = 50 (&H32)
DIJOFS_BUTTON20 = 68 (&H44)
DIJOFS_BUTTON21 = 69 (&H45)
DIJOFS_BUTTON22 = 70 (&H46)

```

# IDH\_CONST\_DIJOYSTICKOFS\_dinput\_vb



---

```

DIJOFS_BUTTON23 = 71 (&H47)
DIJOFS_BUTTON24 = 72 (&H48)
DIJOFS_BUTTON25 = 73 (&H49)
DIJOFS_BUTTON26 = 74 (&H4A)
DIJOFS_BUTTON27 = 75 (&H4B)
DIJOFS_BUTTON28 = 76 (&H4C)
DIJOFS_BUTTON29 = 77 (&H4D)
DIJOFS_BUTTON3 = 51 (&H33)
DIJOFS_BUTTON30 = 78 (&H4E)
DIJOFS_BUTTON31 = 79 (&H4F)
DIJOFS_BUTTON4 = 52 (&H34)
DIJOFS_BUTTON5 = 53 (&H35)
DIJOFS_BUTTON6 = 54 (&H36)
DIJOFS_BUTTON7 = 55 (&H37)
DIJOFS_BUTTON8 = 56 (&H38)
DIJOFS_BUTTON9 = 57 (&H39)
DIJOFS_POV0 = 32 (&H20)
DIJOFS_POV1 = 36 (&H24)
DIJOFS_POV2 = 40 (&H28)
DIJOFS_POV3 = 44 (&H2C)
DIJOFS_RX = 12
DIJOFS_RY = 16 (&H10)
DIJOFS_RZ = 20 (&H14)
DIJOFS_SLIDER0 = 24 (&H18)
DIJOFS_SLIDER1 = 28 (&H1C)
DIJOFS_X = 0
DIJOFS_Y = 4
DIJOFS_Z = 8
End Enum

```

DIJOFS\_BUTTON(*n*)

Offset of the data for button *n*.

DIJOFS\_POV(*n*)

Offset of the data for point-of-view controller *n*.

DIJOFS\_RX, DIJOFS\_RY, DIJOFS\_RZ

Offset of the data for the axis rotation.

DIJOFS\_SLIDER0, DIJOFS\_SLIDER1

Offset of the data for the slider.

DIJOFS\_X, DIJOFS\_Y, DIJOFS\_Z

Offset of the data for the axis.

## CONST\_DIKEYFLAGS

# The **CONST\_DIKEYFLAGS** enumeration groups the Keyboard Device Constants.

## CONST\_DIMOUSEOFS

# The members of the **CONST\_DIMOUSEOFS** enumeration represent the offset of the data for the various mouse device objects within the data format.

```
Enum CONST_DIMOUSEOFS
```

```
    DIMOFS_BUTTON0 = 12
```

```
    DIMOFS_BUTTON1 = 13
```

```
    DIMOFS_BUTTON2 = 14
```

```
    DIMOFS_BUTTON3 = 15
```

```
    DIMOFS_X = 0
```

```
    DIMOFS_Y = 4
```

```
    DIMOFS_Z = 8
```

```
End Enum
```

```
DIMOFS_BUTTON(n)
```

Offset of the data for button *n*.

```
DIMOFS_X, DIMOFS_Y, DIMOFS_Z
```

Offset of the data for the axis.

## CONST\_DINPUT

# The **CONST\_DINPUT** enumeration contains various constants that are used throughout DirectInput.

```
Enum CONST_DINPUT
```

```
    DIPROPAXISMODE_ABS = 0
```

```
    DIPROPAXISMODE_REL = 1
```

```
    DIPROPCALIBRATIONMODE_COOKED = 0
```

```
    DIPROPCALIBRATIONMODE_RAW = 1
```

```
    DIPROP RANGE_NOMAX = 2147483647 (&H7FFFFFFF)
```

```
    DIPROP RANGE_NOMIN = -2147483648 (&H80000000)
```

```
End Enum
```

```
DIPROPAXISMODE_ABS
```

Used in **DirectInputDevice.GetProperty** and **DirectInputDevice.SetProperty** to represent absolute axis mode.

```
DIPROPAXISMODE_REL
```

Used in **DirectInputDevice.GetProperty** and **DirectInputDevice.SetProperty** to represent relative axis mode.

---

```
# IDH_CONST_DIKEYFLAGS_dinput_vb
```

```
# IDH_CONST_DIMOUSEOFS_dinput_vb
```

```
# IDH_CONST_DINPUT_dinput_vb
```

**DIPROPCALIBRATIONMODE\_COOKED**

Used in setting the **DIPROP\_CALIBRATIONMODE** property to indicate that **DirectInput** should return axis data after applying calibration information.

**DIPROPCALIBRATIONMODE\_RAW**

Used in setting the **DIPROP\_CALIBRATIONMODE** property to indicate that **DirectInput** should return raw, uncalibrated data. This mode is typically used only by Control Panel-type applications.

**DIPROPRANGE\_NOMAX**

Returned from **DirectInputDevice.GetProperty** if the axis has no upper limit on its range.

**DIPROPRANGE\_NOMIN**

Returned from **DirectInputDevice.GetProperty** if the axis has no lower limit on its range.

## CONST\_DINPUTERR

# The **CONST\_DINPUTERR** enumeration contains the error codes for **DirectInput**. All the error codes and definitions can be found in the Error Codes topic.

## CONST\_DIPHFLAGS

# Members of the **CONST\_DIPHFLAGS** enumeration are used to specify how a device object is identified. They are used in the **DirectInputDevice.GetObjectInfo** method as well as in the **DIPROPLONG** and **DIPROPRANGE** types.

Enum **CONST\_DIPHFLAGS**

**DIPH\_DEVICE** = 0

**DIPH\_BYID** = 2

**DIPH\_BYOFFSET** = 1

End Enum

**DIPH\_DEVICE**

The property applies to the entire device, not to a particular object.

**DIPH\_BYOFFSET**

The device object is identified by the offset into the current data format of the object whose information is being accessed.

**DIPH\_BYID**

The device object is identified by the instance identifier obtained from the return value of the **DirectInputDeviceObjectInstance.GetType** method.

---

# **IDH\_CONST\_DINPUTERR\_dinput\_vb**

# **IDH\_CONST\_DIPHFLAGS\_dinput\_vb**

## CONST\_DISCLFLAGS

# The **CONST\_DISCLFLAGS** enumeration is used in the *flags* parameter of the **DirectInputDevice.SetCooperativeLevel** method to determine how the device interacts with other instances of the device and the rest of the system.

```
Enum CONST_DISCLFLAGS
    DISCL_BACKGROUND = 8
    DISCL_EXCLUSIVE = 1
    DISCL_FOREGROUND = 4
    DISCL_NONEXCLUSIVE = 2
End Enum
```

### DISCL\_BACKGROUND

The application requires background access. If background access is granted, the device may be acquired at any time, even when the associated window is not the active window.

### DISCL\_EXCLUSIVE

The application requires exclusive access. If exclusive access is granted, no other instance of the device may obtain exclusive access to the device while it is acquired. Note, however, non-exclusive access to the device is always permitted, even if another application has obtained exclusive access.

If an application acquires the mouse or keyboard device in exclusive mode, the user will not be able to use the window menu or move and resize the window.

### DISCL\_FOREGROUND

The application requires foreground access. If foreground access is granted, the device is automatically unacquired when the associated window moves to the background.

### DISCL\_NONEXCLUSIVE

The application requires non-exclusive access. Access to the device will not interfere with other applications that are accessing the same device.

Applications must specify either **DISCL\_FOREGROUND** or **DISCL\_BACKGROUND**; it is an error to specify both or neither. Similarly, applications must specify either **DISCL\_EXCLUSIVE** or **DISCL\_NONEXCLUSIVE**.

## CONST\_DISDDFLAGS

# Members of the **CONST\_DISDDFLAGS** enumeration are passed to the **DirectInputDevice.SendDeviceData** method.

```
Enum CONST_DISDDCFLAGS
    DISDD_CONTINUE = 1
    DISDD_DEFAULT = 0
```

# IDH\_CONST\_DISCLFLAGS\_dinput\_vb

# IDH\_CONST\_DISDDFLAGS\_dinput\_vb

---

End Enum

#### DISDD\_CONTINUE

Data will overlaid on existing data. For more information, see the Remarks for **DirectInputDevice.SendDeviceData**.

#### DISDD\_DEFAULT

Data will not be overlaid on existing data.

## CONST\_DISFFCFLAGS

# Members of the **CONST\_DISFFCFLAGS** enumeration are used to specify the command sent by using the **DirectInputDevice.SendForceFeedbackCommand** method.

#### Enum CONST\_DISFFCFLAGS

```
DISFFC_CONTINUE =      8
DISFFC_PAUSE =        4
DISFFC_RESET =         1
DISFFC_SETACTUATORSOFF = 32 (&H20)
DISFFC_SETACTUATORSON = 16 (&H10)
DISFFC_STOPALL =       2
```

End Enum

#### DISFFC\_CONTINUE

Paused playback of all active effects is to be continued. It is an error to send this command when the device is not in a paused state.

#### DISFFC\_PAUSE

Playback of all active effects is to be paused. This command also stops the clock on effects, so that they continue playing to their full duration when restarted.

While the device is paused, new effects may not be started and existing ones may not be modified. Doing so may result in the subsequent **DISFFC\_CONTINUE** command failing to perform properly.

To abandon a pause and stop all effects, use the **DISFFC\_STOPALL** or **DISFCC\_RESET** commands.

#### DISFFC\_RESET

The device's force-feedback system is to be put in its startup state. All effects are removed from the device, are no longer valid, and must be recreated if they are to be used again. The device's actuators are disabled.

#### DISFFC\_SETACTUATORSOFF

The device's force-feedback actuators are to be disabled. While the actuators are off, effects continue to play but are ignored by the device. Using the analogy of a sound playback device, they are muted rather than paused.

---

# IDH\_CONST\_DISFFCFLAGS\_dinput\_vb

**DISFFC\_SETACTUATORSON**

The device's force-feedback actuators are to be enabled.

**DISFFC\_STOPALL**

Playback of any active effects is to be stopped. All active effects will be reset, but are still being maintained by the device and are still valid. If the device is in a paused state, that state is lost.

This command is equivalent to calling the **DirectInputEffect.Stop** method for each effect playing.

## Keyboard Keys

This section contains information on the following topics:

- Keyboard Device Constants
- DirectInput and Japanese Keyboards

## Keyboard Device Constants

# Keyboard device constants, which are members of the **CONST\_DIKEYFLAGS** enumeration, represent offsets within a keyboard device's data packet, a 256-byte array. The data at a given offset is associated with a keyboard key.

The standard keyboard device constants are the following (in ascending order):

Constant	Note
DIK_ESCAPE	
DIK_1	On main keyboard
DIK_2	On main keyboard
DIK_3	On main keyboard
DIK_4	On main keyboard
DIK_5	On main keyboard
DIK_6	On main keyboard
DIK_7	On main keyboard
DIK_8	On main keyboard
DIK_9	On main keyboard
DIK_0	On main keyboard
DIK_MINUS	On main keyboard
DIK_EQUALS	On main keyboard
DIK_BACK	BACKSPACE
DIK_TAB	

---

# IDH\_Keyboard\_Device\_Constants\_dinput\_vb

DIK_Q	
DIK_W	
DIK_E	
DIK_R	
DIK_T	
DIK_Y	
DIK_U	
DIK_I	
DIK_O	
DIK_P	
DIK_LBRACKET	[
DIK_RBRACKET	]
DIK_RETURN	ENTER on main keyboard
DIK_LCONTROL	Left CTRL
DIK_A	
DIK_S	
DIK_D	
DIK_F	
DIK_G	
DIK_H	
DIK_J	
DIK_K	
DIK_L	
DIK_SEMICOLON	
DIK_APOSTROPHE	
DIK_GRAVE	Grave accent (`)
DIK_LSHIFT	Left SHIFT
DIK_BACKSLASH	
DIK_Z	
DIK_X	
DIK_C	
DIK_V	
DIK_B	
DIK_N	
DIK_M	
DIK_COMMA	
DIK_PERIOD	On main keyboard
DIK_SLASH	Forward slash (/)on main keyboard

---

DIK_RSHIFT	Right SHIFT
DIK_MULTIPLY	Asterisk on numeric keypad
DIK_LMENU	Left ALT
DIK_SPACE	Spacebar
DIK_CAPITAL	CAPS LOCK
DIK_F1	
DIK_F2	
DIK_F3	
DIK_F4	
DIK_F5	
DIK_F6	
DIK_F7	
DIK_F8	
DIK_F9	
DIK_F10	
DIK_NUMLOCK	
DIK_SCROLL	SCROLL LOCK
DIK_NUMPAD7	
DIK_NUMPAD8	
DIK_NUMPAD9	
DIK_SUBTRACT	Hyphen (minus sign) on numeric keypad
DIK_NUMPAD4	
DIK_NUMPAD5	
DIK_NUMPAD6	
DIK_ADD	Plus sign on numeric keypad
DIK_NUMPAD1	
DIK_NUMPAD2	
DIK_NUMPAD3	
DIK_NUMPAD0	
DIK_DECIMAL	Period (decimal point) on numeric keypad
DIK_F11	
DIK_F12	
DIK_F13	
DIK_F14	
DIK_F15	
DIK_KANA	On Japanese keyboard
DIK_CONVERT	On Japanese keyboard
DIK_NOCONVERT	On Japanese keyboard



---

DIK_YEN	On Japanese keyboard
DIK_NUMPADEQUALS	On numeric keypad (NEC PC98)
DIK_CIRCUMFLEX	On Japanese keyboard
DIK_AT	On Japanese keyboard
DIK_COLON	On Japanese keyboard
DIK_UNDERLINE	On Japanese keyboard
DIK_KANJI	On Japanese keyboard
DIK_STOP	On Japanese keyboard
DIK_AX	On Japanese keyboard
DIK_UNLABELED	On Japanese keyboard
DIK_NUMPADENTER	
DIK_RCONTROL	Right CTRL key
DIK_NUMPADCOMMA	Comma on NEC PC98 numeric keypad
DIK_DIVIDE	Forward slash (/)on numeric keypad
DIK_SYSRQ	
DIK_RMENU	Right ALT
DIK_HOME	
DIK_UP	Up arrow
DIK_PRIOR	PAGE UP
DIK_LEFT	Left arrow
DIK_RIGHT	Right arrow
DIK_END	
DIK_DOWN	Down arrow
DIK_NEXT	PAGE DOWN
DIK_INSERT	
DIK_DELETE	
DIK_LWIN	Left Windows key
DIK_RWIN	Right Windows key
DIK_APPS	Application key
DIK_PAUSE	

For information on Japanese keyboards, see DirectInput and Japanese Keyboards.

## DirectInput and Japanese Keyboards

# There are substantial differences between Japanese and U.S. keyboards. The following chart lists the additional keys that are available on each type of Japanese keyboard. It also lists the keys that are available on U.S. keyboards but are missing on the various Japanese keyboards.

Also note that on some NEC PC-98 keyboards, the DIK\_CAPSLOCK and DIK\_KANA keys are toggle buttons and not push buttons. These generate a down event when first pressed, then generate an up event when pressed a second time.

Keyboard	Additional Keys	Missing Keys
DOS/V 106 Keyboard, NEC PC-98 106 Keyboard	DIK_AT, DIK_CIRCUMFLEX, DIK_COLON, DIK_CONVERT, DIK_KANA, DIK_KANJI, DIK_NOCONVERT, DIK_YEN	DIK_APOSTROPHE, DIK_EQUALS, DIK_GRAVE
NEC PC-98 Standard Keyboard, NEC PC-98 Laptop Keyboard	DIK_AT, DIK_CIRCUMFLEX, DIK_COLON, DIK_F13, DIK_F14, DIK_F15, DIK_KANA, DIK_KANJI, DIK_NOCONVERT, DIK_NUMPADCOMMA, DIK_NUMPADEQUALS, DIK_STOP, DIK_UNDERLINE, DIK_YEN	DIK_APOSTROPHE, DIK_BACKSLASH, DIK_EQUALS, DIK_GRAVE, DIK_NUMLOCK, DIK_NUMPADENTER, DIK_RCONTROL, DIK_RMENU, DIK_RSHIFT, DIK_SCROLL
AX Keyboard	DIK_AX, DIK_CONVERT, DIK_KANJI, DIK_NOCONVERT, DIK_YEN	DIK_RCONTROL, DIK_RMENU
J-3100 Keyboard	DIK_KANA, DIK_KANJI, DIK_NOLABEL, DIK_YEN	DIK_RCONTROL, DIK_RMENU

---

## Error Codes

This table lists the error codes that can be returned by DirectInput methods and functions. Errors are represented by negative values and cannot be combined.

For a list of the errors each method or function can raise, see the individual descriptions. Lists of error codes in the documentation are necessarily incomplete. For example, any DirectInput method can return DIERR\_OUTOFMEMORY even though the error code is not explicitly listed as a possible return value in the documentation for that method.

### DI\_BUFFEROVERFLOW

The input buffer overflowed and data was lost.

### DIERR\_ACQUIRED

The operation cannot be performed while the device is acquired.

### DIERR\_ALREADYINITIALIZED

This object is already initialized

### DIERR\_BADDRIVERVER

The object could not be created due to an incompatible driver version or mismatched or incomplete driver components.

### DIERR\_BETADIRECTINPUTVERSION

The application was written for an unsupported prerelease version of DirectInput.

### DIERR\_DEVICEFULL

The device is full.

### DIERR\_DEVICENOTREG

The device or device instance is not registered with DirectInput. This value is equal to the REGDB\_E\_CLASSNOTREG standard COM return value.

### DIERR\_EFFECTPLAYING

The parameters were updated in memory but were not downloaded to the device because the device does not support updating an effect while it is still playing.

### DIERR\_HASEFFECTS

The device cannot be reinitialized because there are still effects attached to it.

### DIERR\_GENERIC

An undetermined error occurred inside the DirectInput subsystem. This value is equal to the E\_FAIL standard COM return value.

### DIERR\_HANDLEEXISTS

The device already has an event notification associated with it. This value is equal to the E\_ACCESSDENIED standard COM return value.

### DIERR\_INCOMPLETEEFFECT

The effect could not be downloaded because essential information is missing. For example, no axes have been associated with the effect, or no type-specific information has been supplied.

### DIERR\_INPUTLOST

Access to the input device has been lost. It must be reacquired.

### DIERR\_INVALIDHANDLE

An invalid window handle was passed to the method.

**DIERR\_INVALIDPARAM**

An invalid parameter was passed to the returning function, or the object was not in a state that permitted the function to be called. This value is equal to the E\_INVALIDARG standard COM return value.

**DIERR\_MOREDATA**

Not all the requested information fitted into the buffer.

**DIERR\_NOAGGREGATION**

This object does not support aggregation.

**DIERR\_NOINTERFACE**

The specified interface is not supported by the object. This value is equal to the E\_NOINTERFACE standard COM return value.

**DIERR\_NOTACQUIRED**

The operation cannot be performed unless the device is acquired.

**DIERR\_NOTBUFFERED**

The device is not buffered. Set the DIPROP\_BUFFERSIZE property to enable buffering.

**DIERR\_NOTDOWNLOADED**

The effect is not downloaded.

**DIERR\_NOTEXCLUSIVEACQUIRED**

The operation cannot be performed unless the device is acquired in DISCL\_EXCLUSIVE mode.

**DIERR\_NOTINITIALIZED**

The object has not been initialized.

**DIERR\_NOTFOUND**

The requested object does not exist.

**DIERR\_OBJECTNOTFOUND**

The requested object does not exist.

**DIERR\_OLDDIRECTINPUTVERSION**

The application requires a newer version of DirectInput.

**DIERR\_OTHERAPPHASPRIO**

Another application has a higher priority level, preventing this call from succeeding. This value is equal to the E\_ACCESSDENIED standard COM return value. This error can be returned when an application has only foreground access to a device but is attempting to acquire the device while in the background.

**DIERR\_OUTOFMEMORY**

The DirectInput subsystem couldn't allocate sufficient memory to complete the call. This value is equal to the E\_OUTOFMEMORY standard COM return value.

**DIERR\_READONLY**

The specified property cannot be changed. This value is equal to the E\_ACCESSDENIED standard COM return value.

**DIERR\_REPORTFULL**

More information was requested to be sent than can be sent to the device.

**DIERR\_UNPLUGGED**

The operation could not be completed because the device is not plugged in.

**DIERR\_UNSUPPORTED**

The function called is not supported at this time. This value is equal to the E\_NOTIMPL standard COM return value.

**E\_PENDING**

Data is not yet available.

## DirectInput Tutorials

The following sections contain tutorials providing step-by-step instructions for implementing basic Microsoft® DirectInput® functionality:

- DirectInput C/C++ Tutorials
- DirectInput Visual Basic Tutorials

## DirectInput C/C++ Tutorials

---

**[Visual Basic]**

The information in this topic pertains only to applications written in C++. See DirectInput Visual Basic Tutorials.

---

**[C++]**

This section contains four tutorials, each providing step-by-step instructions for implementing basic DirectInput functionality in a C or C++ application.

- Tutorial 1: Using the Keyboard  
The first tutorial shows how to add DirectInput keyboard support to an existing application.
- Tutorial 2: Using the Mouse  
The next tutorial takes you through the steps of providing DirectInput mouse support in an application. The tutorial is based on the Scrawl sample, and focuses on buffered data.
- Tutorial 3: Using the Joystick  
This tutorial shows how to enumerate all the joysticks connected to a system, how to create and initialize DirectInputDevice objects for each of them in a callback function, and how to retrieve immediate data. Sample code is based on the Space Donuts sample.
- Tutorial 4: Using Force Feedback

The final tutorial illustrates the creation and manipulation of a simple effect on a force-feedback joystick.

---

## Tutorial 1: Using the Keyboard

---

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

The information in this topic pertains only to applications written in C++. See [DirectInput Visual Basic Tutorials](#).

---

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

To prepare for keyboard input, you first create an instance of a `DirectInput` object. Then you use the **`IDirectInput7::CreateDeviceEx`** method to create an instance of an **`IDirectInputDevice7`** interface. The **`IDirectInputDevice7`** interface methods are used to manipulate the device, set its behavior, and retrieve data.

The tutorial breaks down the required tasks into the following steps:

- Step 1: Create the `DirectInput` Object
- Step 2: Create the `DirectInput` Keyboard Device
- Step 3: Set the Keyboard Data Format
- Step 4: Set the Keyboard Behavior
- Step 5: Gain Access to the Keyboard
- Step 6: Retrieve Data from the Keyboard
- Step 7: Close Down the `DirectInput` System

Adding `DirectInput` keyboard support to an application is relatively simple, so this tutorial is not accompanied by a complete sample application. All of the tutorial steps are illustrated by code within the text. The related steps for initializing the system are gathered in Sample Function 1: `DI_Init`. Another function, Sample Function 2: `DI_Term`, is called whenever the system needs to be closed down.

---

## Step 1: Create the `DirectInput` Object

---

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

The information in this topic pertains only to applications written in C++. See [DirectInput Visual Basic Tutorials](#).

---

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

The first step in setting up the DirectInput system is to create a single DirectInput object as overall manager. This is done with a call to the **DirectInputCreateEx** function.

```
// HINSTANCE hinst; // initialized earlier
HRESULT      hr;
LPDIRECTINPUT7 g_lpdi;

hr = DirectInputCreateEx(hinst, DIRECTINPUT_VERSION,
    IID_IDirectInput7, (void**) &g_lpdi, NULL);
if FAILED(hr)
{
    // DirectInput not available; take appropriate action
}
```

The first parameter for **DirectInputCreateEx** is the instance handle to the application or DLL that is creating the object.

The second parameter tells the DirectInput object which version of the DirectInput system should be used. You can design your application to be compatible with earlier versions of DirectInput. For more information, see *Designing for Previous Versions of DirectInput*.

The third parameter determines which interface is returned. Most applications will obtain the most recent version, by passing IID\_IDirectInput7.

The fourth parameter is the address of a variable that will be initialized with a valid interface pointer if the call succeeds.

The last parameter specifies the address of the controlling object's **IUnknown** interface for use in COM aggregation. Most applications will not be using aggregation and so will pass NULL.

Next: Step 2: Create the DirectInput Keyboard Device

---

## Step 2: Create the DirectInput Keyboard Device

---

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

The information in this topic pertains only to applications written in C++. See *DirectInput Visual Basic Tutorials*.

---

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

After creating the DirectInput object, your application must create the keyboard object—the device—and retrieve a pointer to an **IDirectInputDevice7** interface. The device will perform most of the keyboard-related tasks, using the methods of the interface.

To do this your application must call the **IDirectInput7::CreateDeviceEx** method, as shown in Sample Function 1: **DI\_Init**. **CreateDeviceEx** accepts four parameters.

The first parameter is the GUID for the device being created. Since the system keyboard will be used, your application should pass the *GUID\_SysKeyboard* predefined global variable.

The second parameter is the GUID for the desired interface. Most applications will obtain the most recent version, by passing **IID\_IDirectInputDevice7**.

The third parameter is the address of a variable that will be initialized with the interface pointer if the call succeeds.

The fourth parameter specifies the address of the controlling object's **IUnknown** interface for use in COM aggregation. Your application will likely not use aggregation, in which case the parameter is **NULL**.

The following example attempts to retrieve a pointer to an **IDirectInputDevice7** interface. If this fails, it calls the **DI\_Term** application-defined sample function to deallocate existing DirectInput objects, if any.

#### Note

In all the examples, *g\_lpdi* is the initialized pointer to the DirectInput object. The method calls are in the C++ form.

```
HRESULT          hr;
LPDIRECTINPUTDEVICE7 g_lpDIDEVICE

hr = g_lpDI->CreateDeviceEx(GUID_SysKeyboard, IID_IDirectInputDevice7,
    (void**)&g_lpDIDevice, NULL);
if FAILED(hr)
{
    DI_Term();
    return FALSE;
}
```

Next: Step 3: Set the Keyboard Data Format

---

### Step 3: Set the Keyboard Data Format

---

[\[Visual Basic\]](#)

The information in this topic pertains only to applications written in C++. See DirectInput Visual Basic Tutorials.

---

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



After retrieving an **IDirectInputDevice7** pointer, your application must set the device's data format, as shown in Sample Function 1: `DI_Init`. For keyboards, this is a very simple task. Call the **IDirectInputDevice7::SetDataFormat** method, specifying the data format provided for your convenience by `DirectInput` in the *`c_dfDIKeyboard`* global variable.

The following example attempts to set the data format. If this fails, it calls the `DI_Term` sample function to deallocate existing `DirectInput` objects, if any.

```
hr = g_lpDIDevice->SetDataFormat(&c_dfDIKeyboard);

if FAILED(hr){
    DI_Term();
    return FALSE;
}
```

Next: Step 4: Set the Keyboard Behavior

---

## Step 4: Set the Keyboard Behavior

---

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

The information in this topic pertains only to applications written in C++. See `DirectInput Visual Basic Tutorials`.

---

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

Before your application can gain access to the keyboard, it must set the device's behavior using the **IDirectInputDevice7::SetCooperativeLevel** method, as shown in Sample Function 1: `DI_Init`. This method accepts the handle to the window to be associated with the device and a combination of flags that determine the cooperative level. In this case the application is requesting foreground, nonexclusive access.

The following example attempts to set the device's cooperative level. If this fails, it calls the `DI_Term` application-defined sample function to deallocate existing `DirectInput` objects, if any.

```
// Set the cooperative level
hr = g_lpDIDevice->SetCooperativeLevel(g_hwndMain,
    DISCL_FOREGROUND | DISCL_NONEXCLUSIVE);

if FAILED(hr){
    DI_Term();
    return FALSE;
}
```

Next: Step 5: Gain Access to the Keyboard

---

## Step 5: Gain Access to the Keyboard

---

### [Visual Basic]

The information in this topic pertains only to applications written in C++. See DirectInput Visual Basic Tutorials.

---

### [C++]

After your application sets the keyboard's behavior, it can acquire access to the device by calling the **IDirectInputDevice7::Acquire** method. The application must acquire the device before retrieving data from it. The **Acquire** method accepts no parameters.

The following line of code acquires the keyboard device that was created in Step 2: Create the DirectInput Keyboard Device:

```
if (g_lpDIDevice) g_lpDIDevice->Acquire();
```

Next: Step 6: Retrieve Data from the Keyboard

---

---

## Step 6: Retrieve Data from the Keyboard

---

### [Visual Basic]

The information in this topic pertains only to applications written in C++. See DirectInput Visual Basic Tutorials.

---

### [C++]

Once a device is acquired, your application can start retrieving data from it. The simplest way to do this is to call the **IDirectInputDevice7::GetDeviceState** method, which takes a snapshot of the device's state at the time of the call.

The **GetDeviceState** method accepts two parameters: the size of a buffer to be filled with device state data, and a pointer to that buffer. For keyboards, always declare a buffer of 256 unsigned bytes.

The following sample attempts to retrieve the state of the keyboard. If this fails, it calls an application-defined sample function to deallocate existing DirectInput objects, if any. (See Sample Function 2: `DI_Term`.)

After retrieving the keyboard's current state, your application may respond to specific keys that were down at the time of the call. Each element in the buffer represents a key. If an element's high bit is on, the key was down at the moment of the call; otherwise, the key was up. To check the state of a given key, use the DirectInput Keyboard Device Constants to index the buffer for a given key.

The following skeleton function, called from the main loop of a hypothetical spaceship game, uses the **IDirectInputDevice7::GetDeviceState** method to poll the keyboard. It then checks to see if the LEFT ARROW, RIGHT ARROW, UP ARROW or DOWN ARROW keys were pressed when the device state was retrieved. This is accomplished with the **KEYDOWN** macro defined in the body of the function. The macro accepts a buffer's variable name and an index value, then checks the byte at the specified index to see if the high bit is set and returns TRUE if it is.

```
void WINAPI ProcessKBInput()
{
    #define KEYDOWN(name,key) (name[key] & 0x80)

    char    buffer[256];
    HRESULT hr;

    hr = g_lpDIDevice->GetDeviceState(sizeof(buffer),(LPVOID)&buffer);
    if FAILED(hr)
    {
        // If it failed, the device has probably been lost.
        // We should check for (hr == DIERR_INPUTLOST)
        // and attempt to reacquire it here.
        return;
    }

    // Turn the ship right or left
    if (KEYDOWN(buffer, DIK_RIGHT));
        // Turn right.
    else if (KEYDOWN(buffer, DIK_LEFT));
        // Turn left.

    // Thrust or stop the ship
    if (KEYDOWN(buffer, DIK_UP)) ;
        // Move the ship forward.
    else if (KEYDOWN(buffer, DIK_DOWN));
        // Stop the ship.
}
```

Next: Step 7: Close Down the DirectInput System

---

## Step 7: Close Down the DirectInput System

---

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

The information in this topic pertains only to applications written in C++. See DirectInput Visual Basic Tutorials.

**[C++]**

When an application is about to close, it should destroy all DirectInput objects. This is a three-step process:

- Unacquire all DirectInput devices (**IDirectInputDevice7::Unacquire**)
- Release all DirectInput devices (**IDirectInputDevice7::Release**)
- Release the DirectInput object (**IDirectInput7::Release**)

For a sample function that closes down the DirectInput system, see Sample Function 2: DI\_Term.

---

## Sample Function 1: DI\_Init

---

**[Visual Basic]**

The information in this topic pertains only to applications written in C++. See DirectInput Visual Basic Tutorials.

---

**[C++]**

This application-defined sample function creates a DirectInput object, initializes it, and retrieves the necessary interface pointers, assigning them to global variables. When initialization is complete, it acquires the device.

If any part of the initialization fails, this function calls the DI\_Term application-defined sample function to deallocate DirectInput objects and interface pointers in preparation for terminating the program. (See Sample Function 2: DI\_Term.)

Besides creating the DirectInput object, the DI\_Init function performs the tasks discussed in the following tutorial steps:

- Step 2: Create the DirectInput Keyboard Device
- Step 3: Set the Keyboard Data Format
- Step 4: Set the Keyboard Behavior
- Step 5: Gain Access to the Keyboard

Here is the DI\_Init function:

```
/* The following variables are presumed initialized:
HINSTANCE      g_hinst; // application instance
HWND           g_hwndMain; // application window
*/
LPDIRECTINPUT7  g_lpDI;
LPDIRECTINPUTDEVICE7 g_lpDIDevice;
```

```
BOOL WINAPI DI_Init()
{
    HRESULT hr;

    // Create the DirectInput object.
    hr = DirectInputCreateEx(g_hinst, DIRECTINPUT_VERSION,
        IID_IDirectInput7, (void**)&g_lpDI, NULL);
    if FAILED(hr) return FALSE;

    // Retrieve a pointer to an IDirectInputDevice7 interface
    hr = g_lpDI->CreateDeviceEx(GUID_SysKeyboard,
        IID_IDirectInputDevice7, (void**)&g_lpDIDevice, NULL);
    if FAILED(hr)
    {
        DI_Term();
        return FALSE;
    }

    // Now that you have an IDirectInputDevice7 interface, get
    // it ready to use.

    // Set the data format using the predefined keyboard data
    // format provided by the DirectInput object for keyboards.
    hr = g_lpDIDevice->SetDataFormat(&c_dfDIKeyboard);
    if FAILED(hr)
    {
        DI_Term();
        return FALSE;
    }

    // Set the cooperative level
    hr = g_lpDIDevice->SetCooperativeLevel(g_hwndMain,
        DISCL_FOREGROUND | DISCL_NONEXCLUSIVE);
    if FAILED(hr)
    {
        DI_Term();
        return FALSE;
    }

    // Get access to the input device.
    hr = g_lpDIDevice->Acquire();
    if FAILED(hr)
    {
        DI_Term();
        return FALSE;
    }
}
```

```
    return TRUE;
}
```

---

## Sample Function 2: DI\_Term

---

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

The information in this topic pertains only to applications written in C++. See DirectInput Visual Basic Tutorials.

---

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

This application-defined sample function deallocates existing DirectInput interface pointers in preparation for program shutdown or in the event of a failure to properly initialize a device.

```
/* The following variables are presumed initialized:
LPDIRECTINPUT7      g_lpDI;
LPDIRECTINPUTDEVICE7 g_lpDIDevice;
*/

void WINAPI DI_Term()
{
    if (g_lpDI)
    {
        if (g_lpDIDevice)
        {
            /*
Always unacquire device before calling Release().      */
            g_lpDIDevice->Unacquire();
            g_lpDIDevice->Release();
            g_lpDIDevice = NULL;
        }
        g_lpDI->Release();
        g_lpDI = NULL;
    }
}
```

---

## Tutorial 2: Using the Mouse

---

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

The information in this topic pertains only to applications written in C++. See [DirectInput Visual Basic Tutorials](#).

---

#### [C++]

This tutorial guides you through the process of setting up a mouse device and retrieving buffered input data. The examples are based on the Scrawl sample.

To prepare for mouse input, you first create an instance of a DirectInput object. Then you use the **IDirectInput7::CreateDeviceEx** method to create an instance of an **IDirectInputDevice7** interface. The **IDirectInputDevice7** interface methods are used to manipulate the device, set its behavior, and retrieve data.

The preliminary step of setting up the DirectInput system and the final step of closing it down are the same for any application and are covered in Tutorial 1: Using the Keyboard.

This tutorial breaks down the required tasks into the following steps:

- Step 1: Create the DirectInput Mouse Device
- Step 2: Set the Mouse Data Format
- Step 3: Set the Mouse Behavior
- Step 4: Prepare for Buffered Input from the Mouse
- Step 5: Manage Access to the Mouse
- Step 6: Retrieve Buffered Data from the Mouse

#### Note

When an application acquires the mouse at the exclusive cooperative level, Windows does not show a mouse pointer on the screen. For this, your application needs a simple sprite engine. The Scrawl sample application uses the Win32 function **DrawIcon** to display a crosshair cursor.

---

## Step 1: Create the DirectInput Mouse Device

---

#### [Visual Basic]

The information in this topic pertains only to applications written in C++. See [DirectInput Visual Basic Tutorials](#).

---

#### [C++]

After creating the DirectInput object, your application should retrieve a pointer to an **IDirectInputDevice7** interface, which will be used to perform most mouse-related tasks. To do this, call the **IDirectInput7::CreateDeviceEx** method.

The first parameter of **CreateDeviceEx** is the globally unique identifier (GUID) for the device your application is creating. In this case, since the system mouse will be used, your application should pass the predefined global variable *GUID\_SysMouse*.

The second parameter is the GUID of the desired **DirectInputDevice** interface. Most applications will want the latest interface and so will pass **IID\_IDirectInputDevice7**.

The third parameter is the address of a variable that will be initialized with a valid **IDirectInputDevice7** interface pointer if the call succeeds.

The fourth parameter specifies the address of the controlling object's **IUnknown** interface for use in COM aggregation. Your application probably won't be using aggregation, in which case the parameter will be **NULL**.

The following sample code attempts to retrieve a pointer to an **IDirectInputDevice7** interface. If the call fails, **FALSE** is returned. It is assumed that *g\_pdi* is a valid pointer to **IDirectInput7**.

```
LPDIRECTINPUTDEVICE g_pMouse;
HRESULT             hr;

hr = g_pdi->CreateDeviceEx(GUID_SysMouse, IID_IDirectInputDevice7,
                          (void**)&g_pMouse, NULL);

if (FAILED(hr)) {
    return FALSE;
}
```

Next: Step 2: Set the Mouse Data Format

---

## Step 2: Set the Mouse Data Format

---

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

The information in this topic pertains only to applications written in C++. See **DirectInput Visual Basic Tutorials**.

---

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

After retrieving an **IDirectInputDevice7** pointer, your application must set the device's data format. For mouse devices, this is a very simple task. Call the **IDirectInputDevice7::SetDataFormat** method, specifying the data format provided for your convenience by **DirectInput** in the *c\_dfDIMouse* global variable.

The following code attempts to set the device's data format. If the call fails, **FALSE** is returned.

```
hr = g_pMouse->SetDataFormat(&c_dfDIMouse);
```



```
if (FAILED(hr)) {  
    return FALSE;  
}
```

Next: Step 3: Set the Mouse Behavior

---

### Step 3: Set the Mouse Behavior

---

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

The information in this topic pertains only to applications written in C++. See [DirectInput Visual Basic Tutorials](#).

---

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

Before it can gain access to the mouse, your application must set the mouse device's behavior using the **IDirectInputDevice7::SetCooperativeLevel** method. This method accepts the handle to the window to be associated with the device. In Scrawl, the DISCL\_EXCLUSIVE flag is included to ensure that this application is the only one that can have exclusive access to the device. This flag is combined with DISCL\_FOREGROUND because Scrawl is not interested in what the mouse is doing when another application is in the foreground.

The following code attempts to set the device's cooperative level. If this attempt fails, FALSE is returned.

```
hr = g_pMouse->SetCooperativeLevel(hwnd,  
    DISCL_EXCLUSIVE | DISCL_FOREGROUND);  
  
if (FAILED(hr)) {  
    return FALSE;  
}
```

Next: Step 4: Prepare for Buffered Input from the Mouse

---

### Step 4: Prepare for Buffered Input from the Mouse

---

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

The information in this topic pertains only to applications written in C++. See [DirectInput Visual Basic Tutorials](#).

---

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

The Scrawl application demonstrates how to use event notification to find out about mouse activity, and how to read buffered input from the mouse. Both these techniques require some setup. You can perform these steps at any time after creating the mouse device and before acquiring it.

First, create an event and associate it with the mouse device. You are instructing DirectInput to notify the mouse device object whenever a hardware interrupt indicates that new data is available.

This is how it's done in Scrawl, where *g\_hevtMouse* is a global **HANDLE**.

```
g_hevtMouse = CreateEvent(0, 0, 0, 0);

if (g_hevtMouse == NULL) {
    return FALSE;
}

hr = g_pMouse->SetEventNotification(g_hevtMouse);

if (FAILED(hr)) {
    return FALSE;
}
```

Now you need to set the buffer size so that DirectInput can store any input data until you're ready to look at it. Remember, by default the buffer size is zero, so this step is essential if you want to use buffered data.

It's not necessary to use buffered data with event notification; if you prefer, you can retrieve immediate data when an event is signaled.

To set the buffer size you need to initialize a **DIPROPDWORD** structure with information about itself and about the property you wish to set. Most of the values are boilerplate; the key value is the last one, **dwData**, which is initialized with the number of items you want the buffer to hold.

```
#define DINPUT_BUFFERSIZE 16

DIPROPDWORD dipdw =
{
    // the header
    {
        sizeof(DIPROPDWORD),    // diph.dwSize
        sizeof(DIPROPHEADER),   // diph.dwHeaderSize
        0,                      // diph.dwObj
        DIPH_DEVICE,            // diph.dwHow
    },
    // the data
    DINPUT_BUFFERSIZE,         // dwData
};
```

You then pass the address of the header (the **DIPROPHEADER** structure within the **DIPROPDWORD** structure), along with the identifier of the property you want to change, to the **IDirectInputDevice7::SetProperty** method, as follows:

```
hr = g_pMouse->SetProperty(DIPROP_BUFFERSIZE, &dipdw.diph);

if (FAILED(hr)) {
    return FALSE;
}
```

The setup is now complete, and you're ready to acquire the mouse and start collecting data.

Next: Step 5: Manage Access to the Mouse

---

## Step 5: Manage Access to the Mouse

---

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

The information in this topic pertains only to applications written in C++. See [DirectInput Visual Basic Tutorials](#).

---

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

DirectInput provides the **IDirectInputDevice7::Acquire** and **IDirectInputDevice7::Unacquire** methods to manage device access. Your application must call the **Acquire** method to gain access to the device before requesting mouse information with the **IDirectInputDevice7::GetDeviceState** and **IDirectInputDevice7::GetDeviceData** methods.

Most of the time your application will have the device acquired. However, if you have only foreground access the mouse will automatically be unacquired whenever your application moves to the background. You are responsible for reacquiring it when you get the focus back again. This can be done in response to a **WM\_ACTIVATE** message.

Scrawl handles this message by setting a global variable, *g\_fActive*, according to whether the application is gaining or losing the focus. It then calls a helper function, *Scrawl\_SyncAcquire*, which acquires the mouse if *g\_fActive* is **TRUE** and unacquires it otherwise.

```
case WM_ACTIVATE:
    g_fActive = wParam == WA_ACTIVE || wParam == WA_CLICKACTIVE;
    Scrawl_SyncAcquire(hwnd);
    break;
```

If you have exclusive access, your application may need to let go of the mouse to let the user interact with Windows—for example, to access a menu or a dialog box. In Scrawl this can happen when the user opens the system menu with ALT+SPACEBAR.

The Scrawl window procedure has a handler for WM\_ENTERMENULOOP that responds by setting the global variable *g\_fActive* to FALSE and calling the Scrawl\_SyncAcquire function. This handler allows Windows to have the mouse and display its own cursor.

When the user is done using a menu, Windows sends the application a WM\_EXITMENULOOP message. In this case, the Scrawl window process posts an application-defined message, WM\_SYNCACQUIRE, to its own message queue. This allows other pending messages to be processed before the mouse is reacquired with the Scrawl\_SyncAcquire function.

Scrawl also unacquires the mouse in response to a right button click, which opens up a context menu. Although the mouse would get unacquired later, in the WM\_ENTERMENULOOP handler, it's done here first so that the position of the Windows cursor can be set before the menu appears.

Finally, Scrawl tries to reacquire the mouse if it receives a DIERR\_INPUTLOST error after an attempt to retrieve data. This is just in case the device has been unacquired by some mechanism not covered elsewhere; for instance, if the user has pressed CTRL+ALT+DEL.

In summary, your application needs to acquire the mouse before it can get data from it. This needs to be done only once, as long as nothing happens to force your application to give up access to it. In exclusive mode, you are responsible for giving up control of the mouse when Windows needs it. You are also responsible for reacquiring the mouse whenever your program needs access to it after losing it to Windows or another application.

Next: Step 6: Retrieve Buffered Data from the Mouse

---

## Step 6: Retrieve Buffered Data from the Mouse

---

### [Visual Basic]

The information in this topic pertains only to applications written in C++. See DirectInput Visual Basic Tutorials.

---

### [C++]

Once the mouse is acquired, your application can begin to retrieve data from it.

In the Scrawl sample, retrieval is triggered by a signaled event. In the **WinMain** function, the application sleeps until **MsgWaitForMultipleObjects** indicates that there is either a signal or a message. If there's a signal associated with the mouse, the

Scrawl\_OnMouseInput function is called. This function is a good illustration of how buffered input is handled, so we'll look at it in detail.

First the function makes sure the old cursor position will be cleaned up. Remember, Scrawl is maintaining its own cursor and is wholly responsible for drawing and erasing it.

```
void Scrawl_OnMouseInput(HWND hwnd)
{
    /* Invalidate the old cursor so it will be erased */
    InvalidateCursorRect(hwnd);
```

Now the function enters a loop to read and respond to the entire contents of the buffer. Because it retrieves just one item at a time, it needs only a single **DIDeviceObjectData** structure to hold the data.

Another way to go about handling input would be to read the entire buffer at once and then loop through the retrieved items, responding to each one in turn. In that case, *dwElements* would be the size of the buffer, and *od* would be an array with the same number of elements.

```
while (!fDone) {
    DIDeviceObjectData od;
    DWORD dwElements = 1; // number of items to be retrieved
```

The application calls the **IDirectInputDevice7::GetDeviceData** method in order to fetch the data. The second parameter tells DirectInput where to put the data, and the third tells it how many items are wanted. The final parameter would be **DIGDD\_PEEK** if the data was to be left in the buffer, but in this case the data is not going to be needed again, so it is removed.

```
HRESULT hr = g_pMouse->GetDeviceData(
    sizeof(DIDeviceObjectData),
    &od,
    &dwElements, 0);
```

Now the application checks to see if access to the device has been lost and, if so, tells itself to try to reacquire the mouse at the first opportunity. This step was discussed in Step 5: Manage Access to the Mouse.

```
if (hr == DIERR_INPUTLOST) {
    PostMessage(hwnd, WM_SYNCACQUIRE, 0, 0L);
    break;
}
```

Next the application makes sure the call to the **GetDeviceData** method succeeded and that there was actually data to be retrieved. Remember, after the call to **GetDeviceData** the *dwElements* variable shows how many items were actually retrieved.

```

/* Unable to read data or no data available */
if (FAILED(hr) || dwElements == 0) {
    break;
}

```

If execution has proceeded to this point, everything is fine: the call succeeded and there is an item of data in the buffer. Now the application looks at the **dwOfs** member of the **DIDEVICEOBJECTDATA** structure to determine which object on the device reported a change of state, and calls helper functions to respond appropriately. The value of the **dwData** member, which gives information about what happened, is passed to these functions.

```

/* Look at the element to see what happened */

switch (od.dwOfs) {

/* DIMOFS_X: Mouse horizontal motion */
case DIMOFS_X: UpdateCursorPosition(od.dwData, 0); break;

/* DIMOFS_Y: Mouse vertical motion */
case DIMOFS_Y: UpdateCursorPosition(0, od.dwData); break;

/* DIMOFS_BUTTON0: Button 0 pressed or released */
case DIMOFS_BUTTON0:

    if (od.dwData & 0x80) { /* Button pressed */
        fDone = 1;
        Scrawl_OnButton0Down(hwnd); /* Go into button-down
                                     mode */
    }
    break;

/* DIMOFS_BUTTON1: Button 1 pressed or released */
case DIMOFS_BUTTON1:

    if (!(od.dwData & 0x80)) { /* Button released */
        fDone = 1;
        Scrawl_OnButton1Up(hwnd); /* Context menu time */
    }
}

}

```

Finally, the `Scrawl_OnMouseInput` sample function invalidates the screen rectangle occupied by the cursor, in case the cursor has been moved by one of the helper functions.

```
/* Invalidate the new cursor so it will be drawn */  
InvalidateCursorRect(hwnd);  
}
```

Scrawl also collects mouse data in the `Scrawl_OnButton0Down` function. This is where the application keeps track of mouse movements while the primary button is being held down—that is, while the user is drawing. This function does not rely on event notification, but repeatedly polls the `DirectInput` buffer until the button is released.

A key point to note in the `Scrawl_OnButton0Down` function is that no actual drawing is done until all pending data has been read. The reason is that each horizontal or vertical movement of the mouse is reported as a separate event. (Both events are, however, placed in the buffer at the same time.) If a line were immediately drawn in response to each separate axis movement, a diagonal movement of the mouse would produce two lines at right angles.

Another way you can be sure that the movement in both axes is taken into account before responding in your application is to check the sequence numbers of the x-axis item and the y-axis item. If the numbers are the same, the two events took place simultaneously. For more information, see [Time Stamps and Sequence Numbers](#).

---

## Tutorial 3: Using the Joystick

---

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

The information in this topic pertains only to applications written in C++. See [DirectInput Visual Basic Tutorials](#).

---

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

This tutorial shows you how to enumerate joysticks on a system and set up two or more joysticks for input. Code samples are based on the `Space Donuts` sample. The method calls are in the C form.

The preliminary step of setting up the `DirectInput` system and the final step of closing it down are the same for any application and are covered in [Tutorial 1: Using the Keyboard](#).

The first step in the tutorial is to enumerate devices; that is, to see what joysticks are available. As part of this process you initialize each joystick device and set its desired characteristics. You then use the **IDirectInputDevice7** interface methods to retrieve data from each joystick.

The tutorial breaks down the required tasks into the following steps:

- Step 1: Enumerate the Joysticks
- Step 2: Create the `DirectInput Joystick Device`

- Step 3: Set the Joystick Data Format
  - Step 4: Set the Joystick Behavior
  - Step 5: Gain Access to the Joystick
  - Step 6: Retrieve Data from the Joystick
- 

## Step 1: Enumerate the Joysticks

---

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

The information in this topic pertains only to applications written in C++. See [DirectInput Visual Basic Tutorials](#).

---

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

After creating the DirectInput system, call the **IDirectInput7::EnumDevices** method to enumerate the joysticks. The following code accomplishes this.

```
// pdi is an initialized pointer to IDirectInput7

pdi->lpVtbl->EnumDevices(pdi, DIDEVTYPE_JOYSTICK,
    InitJoystickInput, pdi, DIEDFL_ATTACHEDONLY);
```

The method call is in the C form. Note that you could use the **IDirectInput7\_EnumDevices** macro to simplify the call. All DirectInput methods have corresponding macros defined in `Dinput.h` that expand to the appropriate C or C++ syntax.

The `DIDEVTYPE_JOYSTICK` constant, passed as the second parameter, specifies the type of device to be enumerated.

*InitJoystickInput* is the address of a callback function to be called each time a joystick is found. This is where the individual devices will be initialized in the following steps of the tutorial.

The fourth parameter can be any 32-bit value that you want to make available to the callback function. In this case it's a pointer to the DirectInput interface, which the callback function needs to know so it can call the **IDirectInput7::CreateDeviceEx** method.

The last parameter, `DIEDFL_ATTACHEDONLY`, is a flag that restricts enumeration to devices that are attached to the computer.

Next: Step 2: Create the DirectInput Joystick Device

---



---

## Step 2: Create the DirectInput Joystick Device

---

### [Visual Basic]

The information in this topic pertains only to applications written in C++. See DirectInput Visual Basic Tutorials.

---

### [C++]

After creating the DirectInput object, the application must retrieve a pointer to an **IDirectInputDevice7** interface, which will be used to perform most joystick-related tasks. In the Space Donuts sample, this is done in the callback function `InitJoystickInput`, which is called each time a joystick is enumerated.

Here is the first part of the callback function:

```

BOOL FAR PASCAL InitJoystickInput(LPCDIDEVICEINSTANCE pdinst,
                                  LPVOID pvRef)
{
    LPDIRECTINPUT7 pdi = pvRef;
    LPDIRECTINPUTDEVICE7 pdev;

    // Create the DirectInput joystick device.
    if (pdi->lpVtbl->CreateDeviceEx(pdi, &pdinst->guidInstance,
                                   IID_IDirectInputDevice7, (void**)&pdev, NULL) != DI_OK)
    {
        OutputDebugString("IDirectInput7::CreateDeviceEx FAILED\n");
        return DIENUM_CONTINUE;
    }
}

```

The parameters to the callback function `InitJoystickInput` are:

- A pointer to the device instance, supplied by the DirectInput system when the device is enumerated.
- A pointer to the DirectInput interface, which you supplied as an parameter to **IDirectInput7::EnumDevices**. This parameter could have been any 32-bit value but in this case you want the DirectInput interface so that you can call the **IDirectInput7::CreateDeviceEx** method.

The `InitJoystickInput` sample function declares a local pointer to the DirectInput object, *pdi*, and assigns it the value passed into the callback. It also declares a local pointer to a DirectInput device, *pdev*, which is initialized when the device is created.

The first task of the callback function, then, is to create the device. The **IDirectInput7::CreateDeviceEx** method accepts four parameters.

The first, unnecessary in C++, is a *this* pointer to the calling DirectInput interface.

The second parameter is a reference to the globally unique identifier (GUID) for the instance of the device. In this case, the GUID is taken from the

**DIDEVICEINSTANCE** structure supplied by DirectInput when it enumerated the device.

The third parameter is the address of the variable that will be initialized with a valid interface pointer if the call succeeds.

The fourth parameter specifies the address of the controlling object's **IUnknown** interface for use in COM aggregation. Space Donuts doesn't use aggregation, so the parameter is NULL.

Note that if for some reason the device interface cannot be created, DIENUM\_CONTINUE is returned from the callback function. This flag instructs DirectInput to keep enumerating as long as there are devices to be enumerated.

Next: Step 3: Set the Joystick Data Format

---

### Step 3: Set the Joystick Data Format

---

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

The information in this topic pertains only to applications written in C++. See DirectInput Visual Basic Tutorials.

---

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

Now that the application has a pointer to a DirectInput device, it can call the **IDirectInputDevice7** methods to manipulate that device. The first step, which is an essential one, is to set the data format for the joystick. This step tells DirectInput how to format the input data.

The Space Donuts sample performs this action inside the callback function introduced in the previous step.

```
if (pdev->lpVtbl->SetDataFormat(pdev, &c_dfDIJoystick) != DI_OK)
{
    OutputDebugString("IDirectInputDevice7::SetDataFormat FAILED\n");
    pdev->lpVtbl->Release(pdev);
    return DIENUM_CONTINUE;
}
```

The *pdev* variable is a pointer to the device interface created by **IDirectInput7::CreateDeviceEx**.

The **IDirectInputDevice7::SetDataFormat** method takes two parameters. The first is a *this* pointer to the calling instance of the interface and is unnecessary in C++. The second is a pointer to a **DIDATAFORMAT** structure containing information about how the data for the device is to be formatted. For the joystick, the predefined global variable *c\_dfDIJoystick* can be used here.

As in the previous step, the callback function returns `DIENUM_CONTINUE` if it fails to initialize the device. This flag instructs DirectInput to keep enumerating as long as there are devices to be enumerated.

Next: Step 4: Set the Joystick Behavior

---

## Step 4: Set the Joystick Behavior

---

### [Visual Basic]

The information in this topic pertains only to applications written in C++. See DirectInput Visual Basic Tutorials.

---

### [C++]

The joystick device has been created, and its data format has been set. The next step is to set its cooperative level. In the Space Donuts sample, this is done in the callback function called when the device is enumerated. As in the previous step, *pdev* is a pointer to the device interface.

```
if(pdev->lpVtbl->SetCooperativeLevel(pdev, hWndMain,
    DISCL_NONEXCLUSIVE | DISCL_FOREGROUND) != DI_OK)
{
    OutputDebugString("IDirectInputDevice7::SetCooperativeLevel
        FAILED\n");
    pdev->lpVtbl->Release(pdev);
    return DIENUM_CONTINUE;
}
```

Once again, the first parameter to **IDirectInputDevice7::SetCooperativeLevel** is a *this* pointer.

The second parameter is a window handle. In this case the handle to the main program window is passed in.

The final parameter is a combination of flags describing the desired cooperative level. Space Donuts requires input from the joystick only when it is the foreground application, and does not care whether another program is using the joystick in exclusive mode, so the flags are set to `DISCL_NONEXCLUSIVE | DISCL_FOREGROUND`. (See Cooperative Levels for a full explanation of these flags.)

The final step carried out for each joystick enumerated in the callback function is to set the properties of the device. In the sample, the properties changed include the range and the dead zone for both the x-axis and y-axis.

By setting the range, you are telling DirectInput what maximum and minimum values you want returned for an axis. If you set a range of -1,000 to +1,000 for the x-axis, as

in the example, you are asking that a value of -1,000 be returned when the stick is at the extreme left, +1,000 when it is at the extreme right, and zero when it is in the middle.

The dead zone is a region of tolerance in the middle of the axis, measured in ten-thousandths of the physical range of axis travel. If you set a dead zone of 1,000 for the x-axis, you are saying that the stick can travel one-tenth of its range to the left or right of center before a non-center value will be returned. For more information on the dead zone, see *Interpreting Joystick Axis Data*.

Here's the code to set the range of the x-axis:

```
DIPROPRANGE diprg;

diprg.diph.dwSize      = sizeof(diprg);
diprg.diph.dwHeaderSize = sizeof(diprg.diph);
diprg.diph.dwObj       = DIJOFS_X;
diprg.diph.dwHow       = DIPH_BYOFFSET;
diprg.lMin             = -1000;
diprg.lMax             = +1000;

if FAILED(pdev->lpVtbl->SetProperty(pdev,
    DIPROP_RANGE, &diprg.diph))
{
    OutputDebugString("IDirectInputDevice7::SetProperty(DIPH_RANGE)
        FAILED\n");
    pdev->lpVtbl->Release(pdev);
    return FALSE;
}
```

The first task here is to set up the **DIPROPRANGE** structure *diprg*, whose address will be passed into the **IDirectInputDevice7::SetProperty** method. Actually, it's not the address of the structure itself that is passed but rather the address of its first member, which is a **DIPROPHEADER** structure. For more information, see *Device Properties*.

The property header is initialized with the following values:

- The size of the property structure
- The size of the header structure
- The value returned by the **DIJOFS\_X** joystick device constant macro, which points to the object whose property is being changed
- A flag to indicate how the third parameter is to be interpreted

The **lmin** and **lmax** members of the **DIPROPRANGE** structure are assigned the desired range values.

The application now calls the **IDirectInputDevice7::SetProperty** method. As usual, the first parameter is a *this* pointer. The second parameter is a flag indicating which

property is being changed. The third parameter is the address of the **DIPROPHEADER** member of the property structure.

Setting the dead zone of the x-axis requires a similar procedure. The Space Donuts sample uses a helper function, `SetDIDwordProperty`, to initialize a **DIPROPDWORD** property structure. Unlike **DIPROP RANGE**, this structure contains only one data member, which in the example is set to 5,000, indicating that the stick must move half of its range from the center before the axis is reported to be off-center.

```
// set X axis dead zone to 50% (to avoid accidental turning)
if FAILED(SetDIDwordProperty(pdev, DIPROP_DEADZONE, DIJOFS_X,
                             DIPH_BYOFFSET, 5000))
{
    OutputDebugString("IDirectInputDevice7::
                     SetProperty(DIPH_DEADZONE) FAILED\n");
    pdev->lpVtbl->Release(pdev);
    return FALSE;
}
```

Next: Step 5: Gain Access to the Joystick

---

## Step 5: Gain Access to the Joystick

---

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

The information in this topic pertains only to applications written in C++. See [DirectInput Visual Basic Tutorials](#).

---

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

After your application sets a joystick's behavior, it can acquire access to the device by calling the **IDirectInputDevice7::Acquire** method. The application must acquire the device before retrieving data from it. The **Acquire** method accepts no parameters.

The Space Donuts application takes care of acquisition in the `ReacquireInput` function. This function does double duty, serving both to acquire the device on startup and to reacquire it if for some reason a `DIERR_INPUTLOST` error is returned when the application tries to get input data.

In the following code, *g\_pdevCurrent* is a global pointer to whatever DirectInput device is currently in use.

```
BOOL ReacquireInput(void)
{
    HRESULT hRes;

    // if we have a current device
```

```
if (g_pdevCurrent)
{
    // acquire the device
    hRes = IDirectInputDevice7_Acquire(g_pdevCurrent);
    // The call above is a macro that expands to:
    // g_pdevCurrent->lpVtbl->Acquire(g_pdevCurrent);

    if (SUCCEEDED(hRes))
    {
        // acquisition successful
        return TRUE;
    }
    else
    {
        // acquisition failed
        return FALSE;
    }
}
else
{
    // we don't have a current device
    return FALSE;
}
}
```

In this example, acquisition is effected by a call to **IDirectInputDevice7\_Acquire**, a macro defined in `Dinput.h` that simplifies the C call to the **IDirectInputDevice7::Acquire** method.

Next: Step 6: Retrieve Data from the Joystick

---

## Step 6: Retrieve Data from the Joystick

---

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

The information in this topic pertains only to applications written in C++. See [DirectInput Visual Basic Tutorials](#).

---

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

Since your application is more likely concerned with the position of the joystick axes than with their movement, you will probably want to retrieve immediate rather than buffered data from the device. You can do this by polling with **IDirectInputDevice7::GetDeviceState**. Remember, not all device drivers will notify

DirectInput when the state of the device changes, so it's always good policy to call the **IDirectInputDevice7::Poll** method before checking the device state.

The Space Donuts application calls the following function on each pass through the rendering loop, provided the joystick is the active input device.

```

DWORD ReadJoystickInput(void)
{
    DWORD          dwKeyState;
    HRESULT         hRes;
    DIJOYSTATE      js;

    // poll the joystick to read the current state
    hRes = IDirectInputDevice7_Poll(g_pdevCurrent);

    // get data from the joystick
    hRes = IDirectInputDevice7_GetDeviceState(g_pdevCurrent,
                                              sizeof(DIJOYSTATE), &js);

    if (hRes != DI_OK)
    {
        // did the read fail because we lost input for some reason?
        // if so, then attempt to reacquire. If the second acquire
        // fails, then the error from GetData will be
        // DIERR_NOTACQUIRED, so we won't get stuck an infinite loop.
        if(hRes == DIERR_INPUTLOST)
            ReacquireInput();

        // return the fact that we did not read any data
        return 0;
    }

    // Now study the position of the stick and the buttons.

    dwKeyState = 0;
    if (js.IX < 0) {
        dwKeyState |= KEY_LEFT;
    } else if (js.IX > 0) {
        dwKeyState |= KEY_RIGHT;
    }

    if (js.IY < 0) {
        dwKeyState |= KEY_UP;
    } else if (js.IY > 0) {
        dwKeyState |= KEY_DOWN;
    }
}

```

```
if (js.rgbButtons[0] & 0x80) {  
    dwKeyState |= KEY_FIRE;  
}  
  
if (js.rgbButtons[1] & 0x80) {  
    dwKeyState |= KEY_SHIELD;  
}  
  
if (js.rgbButtons[2] & 0x80) {  
    dwKeyState |= KEY_STOP;  
}  
  
return dwKeyState;  
}
```

Note the calls to **IDirectInputDevice7\_Poll** and **IDirectInputDevice7\_GetDeviceState**. These are macros that expand to C calls to the corresponding methods, similar to the macro in the previous step of this tutorial. The parameters to the macro are the same as those you would pass to the method. Here is what the call to **GetDeviceState** looks like:

```
hRes = IDirectInputDevice7_GetDeviceState(g_pdevCurrent,  
                                          sizeof(DIJOYSTATE), &js);
```

The first parameter is the *this* pointer; that is, a pointer to the calling object. The second parameter is the size of the structure in which the data will be returned, and the last parameter is the address of this structure, which is of type **DIJOYSTATE**. This structure holds data for up to six axes, 32 buttons, and a point-of-view hat. The sample program looks at the state of two axes and three buttons.

If the position of an axis is reported as nonzero, that axis is outside the dead zone, and the function responds by setting the *dwKeyState* variable appropriately. This variable holds the current set of user commands as entered with either the keyboard or the joystick. For example, if the x-axis of the stick is greater than zero, that is considered the same as the RIGHT ARROW key being down.

Joystick buttons work just like keys or mouse buttons: if the high bit of the returned byte is set, the button is down.

---

## Tutorial 4: Using Force Feedback

---

[\[Visual Basic\]](#)

The information in this topic pertains only to applications written in C++. See DirectInput Visual Basic Tutorials.

---



### [C++]

This tutorial takes you through the process of creating, playing, and modifying a simple effect on a force-feedback joystick. The effect is something like a balky chain saw that you're trying to get started. The sample code uses C++ syntax.

The preliminary step of setting up the DirectInput system and the final step of closing it down are essentially the same for any application and are covered in Tutorial 1: Using the Keyboard. However, when closing down the DirectInput force-feedback system you must take the additional step of releasing any effects you have created.

The tutorial breaks down the required tasks into the following steps:

- Step 1: Enumerate Force-Feedback Devices
  - Step 2: Create the DirectInput Force-Feedback Device
  - Step 3: Enumerate Supported Effects
  - Step 4: Create an Effect
  - Step 5: Play an Effect
  - Step 6: Change an Effect
- 

## Step 1: Enumerate Force-Feedback Devices

---

### [Visual Basic]

The information in this topic pertains only to applications written in C++. See DirectInput Visual Basic Tutorials.

---

### [C++]

The first step is to ensure that a force-feedback device is available on the system. You do this by calling the **IDirectInput7::EnumDevices** method. In the following example, the global pointer to the game device interface is initialized only if the enumeration has succeeded in finding at least one suitable device:

```
LPDIRECTINPUTDEVICE7 g_lpdid7 = NULL;

lpdi->EnumDevices(DIDEVTYPE_JOYSTICK,
                  DIEnumDevicesProc,
                  NULL,
                  DIEDFL_FORCEFEEDBACK | DIEDFL_ATTACHEDONLY);
if (g_lpdid7 == NULL)
{
    // No force-feedback joystick available; take appropriate action.
}
```

In the example, *lpdi* is an initialized pointer to the **IDirectInput7** interface. The first parameter to **EnumDevices** restricts the enumeration to joystick-type devices. The second parameter is the callback function that's going to be called each time DirectInput identifies a device that qualifies for enumeration. The third parameter is for user-defined data to be passed in or out of the callback function; in this case it's not used. Finally, the flags restrict the enumeration further to devices actually attached to the system that support force feedback.

The callback function is a convenient place to initialize the device as soon as it has been found. (It's assumed that the first device found is the one you want to use.) You'll do this next in Step 2: Create the DirectInput Force-Feedback Device.

---

## Step 2: Create the DirectInput Force-Feedback Device

---

### [Visual Basic]

The information in this topic pertains only to applications written in C++. See DirectInput Visual Basic Tutorials.

---

### [C++]

In order to have DirectInput enumerate devices, you must create a callback function of the same type as **DIDEnumDevicesCallback**. In Step 1 you passed the address of this function to the **IDirectInput7::EnumDevices** method.

DirectInput passes into the callback, as the first parameter, a pointer to a **DIDeviceInstance** structure that tells you what you need to know about the device. The structure member of chief interest in the example is **guidInstance**, the unique identifier for the particular piece of hardware on the user's system. You will need to pass this GUID to the **IDirectInput7::CreateDeviceEx** method.

Here's the first part of the callback, which extracts the GUID and creates the device object:

```

BOOL CALLBACK DIEnumDevicesProc(LPCDIDeviceInstance lpddi,
                                LPVOID pvRef)
{
    HRESULT hr;
    GUID DeviceGuid = lpddi->guidInstance;

    // Create game device.

    hr = lpdi->CreateDeviceEx(DeviceGuid, IID_IDirectInputDevice7,
                            (void**)&g_lpdi7, NULL);

    if (FAILED(hr))
    {

```

```

OutputDebugString("Failed to create device.\n");
return DIENUM_STOP;
}

```

The next steps, still within the callback function, are similar to those for setting up any input device. Note that you need the exclusive cooperative level for any force-feedback device. You also need to set the data format.

```

// Set cooperative level.
if (FAILED(g_lpdd7->SetCooperativeLevel(hMainWindow,
    DISCL_EXCLUSIVE | DISCL_FOREGROUND)))
{
    OutputDebugString(
        "Failed to set cooperative level.\n");
    g_lpdd7->Release();
    g_lpdd7 = NULL;
    return DIENUM_STOP;
}

// set game data format
if (FAILED(g_lpdd7->SetDataFormat(&c_dfDIJoystick)))
{
    OutputDebugString("Failed to set game device data format.\n");
    g_lpdd7->Release();
    g_lpdd7 = NULL;
    return DIENUM_STOP;
}

```

Finally, you may want to turn off the device's autocenter feature. Autocenter is essentially a condition effect that uses the motors to simulate the springs in a standard joystick. It's a good idea to turn it off so that it doesn't interfere with other effects.

```

DIPROPDWORD DIPropAutoCenter;

DIPropAutoCenter.diph.dwSize = sizeof(DIPropAutoCenter);
DIPropAutoCenter.diph.dwHeaderSize = sizeof(DIPROPHEADER);
DIPropAutoCenter.diph.dwObj = 0;
DIPropAutoCenter.diph.dwHow = DIPH_DEVICE;
DIPropAutoCenter.dwData = 0;

if (FAILED(lpdd7->SetProperty(DIPROP_AUTOCENTER,
    &DIPropAutoCenter.diph)))
{
    OutputDebugString("Failed to change device property.\n");
}

return DIENUM_STOP; // One is enough.

```

---

```
} // end DIEnumDevicesProc
```

Before using the device, you must acquire it. See Step 5: Gain Access to the Joystick in the previous tutorial for an example of how to handle acquisition.

Next: Step 3: Enumerate Supported Effects

---

## Step 3: Enumerate Supported Effects

---

### [Visual Basic]

The information in this topic pertains only to applications written in C++. See DirectInput Visual Basic Tutorials.

---

### [C++]

Now that you've successfully enumerated and created a force-feedback device, you can enumerate the effect types it supports.

Effect enumeration is not strictly necessary if you want to create only standard effects that will be available on any device, such as constant forces. When creating the effect object, you can identify the desired effect type simply by using one of the predefined GUIDs, such as GUID\_ConstantForce. (For a complete list of these identifiers, see **IDirectInputDevice7::CreateEffect**.)

Another, more flexible approach is to enumerate supported effects of a particular type, and obtain the GUID for the effect from the callback function. This is the approach taken in the FFDonuts sample, and you'll adopt it here as well. You could, of course, use the callback to obtain more information about the device's support for the effect—for example, whether it supports an envelope—but in this tutorial you'll get only the effect GUID.

First, create the callback function that will be called by DirectInput for each effect enumerated. For information on this standard callback, see **DIEnumEffectsCallback**. You can give the function any name you like.

```
BOOL EffectFound = FALSE; // global flag

BOOL CALLBACK DIEnumEffectsProc(LPCDIEFFECTINFO pei, LPVOID pv)
{
    *((GUID *)pv) = pei->guid;
    EffectFound = TRUE;
    return DIENUM_STOP; // one is enough
}
```

The GUID variable pointed to by the application-defined value *pv* is assigned the value passed in the **DIEFFECTINFO** structure created by DirectInput for the effect.

In order to obtain the effect GUID, you set the callback in motion by calling the **IDirectInputDevice7::EnumEffects** method, as follows:

```
HRESULT hr;
GUID    guidEffect;

hr = g_lpdi7->EnumEffects(
    (LPDIENUMEFFECTSCALLBACK) DIEnumEffectsProc,
    &guidEffect,
    DIEFT_PERIODIC);
if (FAILED(hr))
{
    OutputDebugString("Effect enumeration failed\n");
    // Note: success doesn't mean any effects were found,
    // only that the process went smoothly.
}
```

Note that you pass the address of a GUID variable, *guidEffect*, to the **EnumEffects** method. This address is passed in turn to the callback as the *pv* parameter. You also restrict the enumeration to periodic effects by setting the flag **DIEFT\_PERIODIC**.

Next: Step 4: Create an Effect

---

## Step 4: Create an Effect

---

### [Visual Basic]

The information in this topic pertains only to applications written in C++. See DirectInput Visual Basic Tutorials.

---

### [C++]

If the *EffectFound* value is no longer **FALSE** after effect enumeration, you can safely assume that DirectInput has found support for at least one effect of the type you requested. (Of course, in real life you would probably not be content with finding just any periodic effect; you would want to use a particular kind such as a sine or sawtooth.) Armed with the effect GUID, you can now create the effect object.

Before calling the **IDirectInputDevice7::CreateEffect** method, you need to set up the following arrays and structures:

- An array of axes that will be involved in the effect. For a joystick this array will normally consist of the identifiers for the x-axis and the y-axis.
- An array of values for setting the direction. The values will differ according to the number of axes, and according to whether you want to use polar, spherical, or Cartesian coordinates. For a full explanation of this rather complicated business, see Effect Direction.

- A structure of type-specific parameters. In the example, since you are creating a periodic effect, this will be of type **DIPERIODIC**.
- A **DIENVELOPE** structure for defining the envelope to be applied to the effect.
- Finally, a **DIEFFECT** structure to contain the basic parameters for the effect.

First, declare the arrays and structures. You can initialize the arrays at the same time:

```
DWORD    dwAxes[2] = { DIJOFS_X, DIJOFS_Y };
LONG     lDirection[2] = { 0, 0 };

DIPERIODIC diPeriodic;    // type-specific parameters
DIENVELOPE diEnvelope;    // envelope
DIEFFECT   diEffect;      // general parameters
```

Now initialize the type-specific parameters. If you use the values in the example, you will create a full-force periodic effect with a period of one-twentieth of a second.

```
diPeriodic.dwMagnitude = DI_FFNOMINALMAX;
diPeriodic.lOffset = 0;
diPeriodic.dwPhase = 0;
diPeriodic.dwPeriod = (DWORD) (0.05 * DI_SECONDS);
```

To get the effect of the chain-saw motor trying to start, briefly coughing into life, and then slowly dying, you will set an envelope with an attack time of half a second and a fade time of one second. You'll get to the sustain value in a moment.

```
diEnvelope.dwSize = sizeof(DIENVELOPE);
diEnvelope.dwAttackLevel = 0;
diEnvelope.dwAttackTime = (DWORD) (0.5 * DI_SECONDS);
diEnvelope.dwFadeLevel = 0;
diEnvelope.dwFadeTime = (DWORD) (1.0 * DI_SECONDS);
```

Now you set up the basic effect parameters. These include flags to determine how the directions and device objects (buttons and axes) are identified, the sample period and gain for the effect, and pointers to the other data that you have just prepared. You also associate the effect with the fire button of the joystick, so that it will automatically be played whenever that button is pressed.

```
diEffect.dwSize = sizeof(DIEFFECT);
diEffect.dwFlags = DIEFF_POLAR | DIEFF_OBJECTOFFSETS;
diEffect.dwDuration = (DWORD) (2 * DI_SECONDS);

diEffect.dwSamplePeriod = 0;           // = default
diEffect.dwGain = DI_FFNOMINALMAX;    // no scaling
diEffect.dwTriggerButton = DIJOFS_BUTTON0;
diEffect.dwTriggerRepeatInterval = 0;
diEffect.cAxes = 2;
diEffect.rgdwAxes = dwAxes;
```

```
diEffect.rglDirection = &IDirection[0];
diEffect.lpEnvelope = &diEnvelope;
diEffect.cbTypeSpecificParams = sizeof(diPeriodic);
diEffect.lpvTypeSpecificParams = &diPeriodic;
```

So much for the setup. At last you can create the effect:

```
LPDIEFFECT g_lpdiEffect; // global effect object

HRESULT hr = g_lpdiEffect->CreateEffect(
    guidEffect,    // GUID from enumeration
    &diEffect,     // where the data is
    &g_lpdiEffect, // where to put interface pointer
    NULL);        // no aggregation
if (FAILED(hr))
{
    OutputDebugString("Failed to create periodic effect");
}
```

Remember that, by default, the effect is downloaded to the device as soon as it has been created, provided that the device is in an acquired state at the exclusive cooperative level. So if everything has gone according to plan, you should be able to compile, run, press the "fire" button, and feel the sputtering of a chain saw that's out of gas.

Next: Step 5: Play an Effect

---

## Step 5: Play an Effect

---

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

The information in this topic pertains only to applications written in C++. See [DirectInput Visual Basic Tutorials](#).

---

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

The effect created in the previous step starts in response to the press of a button. In order to create an effect that is to be played in response to an explicit call, you need to go back to Step 4 and modify the **dwTriggerButton** member of the **DIEFFECT** structure, as follows:

```
diEffect.dwTriggerButton = DIEB_NOTRIGGER;
```

Now, suppose you want to make a chain saw that actually starts and keeps going. This is simply a matter of changing the **dwDuration** member as follows:

```
diEffect.dwDuration = INFINITE;
```

Starting the effect is very simple:

```
g_lpdiEffect->Start(1, 0);
```

The effect will keep running until you stop it:

```
g_lpdiEffect->Stop();
```

Note that you don't need to change the envelope you created in the previous step. The attack is played as the effect starts, but the fade value is ignored.

Next: Step 6: Change an Effect

---

## Step 6: Change an Effect

---

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

The information in this topic pertains only to applications written in C++. See [DirectInput Visual Basic Tutorials](#).

---

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

Your chain saw is merrily rattling away, and now you want to modify the effect to simulate the slowing down of the engine as the saw bites into wood. Fortunately, DirectInput lets you modify the parameters of an effect while it is playing.

To change the effect, you need to set up a **DIEFFECT** structure or have access to the one you used to create the effect. If you are setting up a new structure with local scope, you need to initialize only the **dwSize** member and any members that contain or point to data that is to be changed.

In this case you want to change a type-specific parameter—the period of the effect—so you need to have access to the **DIPERIODIC** structure you used when creating the effect, or else create a local copy with all members initialized. Make sure that the address of the **DIPERIODIC** structure is in the **lpvTypeSpecificParams** member of the **DIEFFECT** structure.

Now set the new period of the effect:

```
diPeriodic.dwPeriod = (DWORD) (0.08 * DI_SECONDS);
```

Then call the method that actually makes the changes:

```
HRESULT hr = g_lpdiEffect->SetParameters(&diEffect,  
                                           DIEP_TYPESPECIFICPARAMS)
```

Note the flag that restricts the changes to a single member of the **DIEFFECT** structure.



You can control the way changes are handled by using other flags. For example, by using the `DIEP_NODOWNLOAD` flag you could change the parameters immediately after starting the effect but delay the implementation until the user actually started cutting wood. Then you would call the **IDirectInputEffect::Download** method. For more information on how to use the various control flags, see **IDirectInputEffect::SetParameters**.

---

## DirectInput Visual Basic Tutorials

---

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

This section pertains only to application development in Visual Basic. See [DirectInput C/C++ Tutorials](#).

---

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

This section contains the following tutorials, each providing step-by-step instructions for implementing DirectInput in a Visual Basic application:

- **Tutorial 1: Using the Keyboard**  
The first tutorial shows how to add DirectInput keyboard support to an existing application, using event polling.
  - **Tutorial 2: Using the Mouse**  
The next tutorial takes you through the steps of providing DirectInput mouse support in an application, using the exclusive cooperative level and event notification. The tutorial is based on the `ScrawlB` sample, and focuses on buffered data.
  - **Tutorial 3: Using the Joystick**  
This tutorial shows how to enumerate all the joysticks connected to a system, how to create and initialize `DirectInputDevice` objects for each of them in a callback function, and how to retrieve immediate data.
  - **Tutorial 4: Using Force Feedback**  
The final tutorial shows how to implement simple force-feedback effects in an application.
- 

## Tutorial 1: Using the Keyboard

---

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

This topic pertains only to application development in Visual Basic. See [DirectInput C/C++ Tutorials](#).

#### [Visual Basic]

To prepare for keyboard input, you first create a **DirectInput** object. Then you create a **DirectInputDevice** object representing the keyboard. The **DirectInputDevice** class methods are used to set the behavior of the device and retrieve data.

The tutorial breaks down the required tasks into the following steps:

- Step 1: Create DirectInput and the Keyboard Device
  - Step 2: Set the Keyboard Parameters
  - Step 3: Gain Access to the Keyboard
  - Step 4: Retrieve Immediate Data from the Keyboard
- 

## Step 1: Create DirectInput and the Keyboard Device

---

#### [C++]

This topic pertains only to application development in Visual Basic. See **DirectInput C/C++ Tutorials**.

---

#### [Visual Basic]

The first step in setting up the DirectInput system is to create a single **DirectInput** object as overall manager. This is done with a call to the **DirectX7.DirectInputCreate** method, typically in the **Load** event handler for the main form or in **Sub Main**:

```
Dim dx As New DirectX7
Dim di As DirectInput
```

```
Set di = dx.DirectInputCreate()
```

The keyboard is then created as a standard device by passing the keyboard GUID alias to **DirectInput.CreateDevice**:

```
Dim didev As DirectInputDevice
Set didev = di.CreateDevice("GUID_SysKeyboard")
```

Next: Step 2: Set the Keyboard Parameters

---

## Step 2: Set the Keyboard Parameters

---

#### [C++]

This topic pertains only to application development in Visual Basic. See DirectInput C/C++ Tutorials.

---

#### [Visual Basic]

After creating a **DirectInputDevice**, your application must set the device's data format. For keyboards, as with other standard devices, this is a very simple task. Call the **DirectInputDevice.SetCommonDataFormat** method, specifying the data format provided for your convenience by DirectInput when you pass **DIFORMAT\_KEYBOARD** as the parameter.

```
Call didev.SetCommonDataFormat(DIFORMAT_KEYBOARD)
```

Remember, you have to set the data format even if you intend to retrieve buffered data. DirectInput identifies the device objects by their offset within the data format. In the case of the keyboard, keys are identified by their offsets within the **DIKEYBOARDSTATE** type.

Before your application can gain access to the keyboard, it must set the device's behavior using the **DirectInputDevice.SetCooperativeLevel** method, as follows:

```
didev.SetCooperativeLevel Me.hWnd, _  
DISCL_NONEXCLUSIVE Or DISCL_BACKGROUND
```

This method accepts the handle to the window to be associated with the device, and exactly two flags (one of **DISCL\_EXCLUSIVE** and **DISCL\_NONEXCLUSIVE**, and one of **DISCL\_FOREGROUND** and **DISCL\_BACKGROUND**), indicating the desired cooperative level. DirectInput does not support exclusive access to keyboard devices, so the **DISCL\_NONEXCLUSIVE** flag must be included in this case.

The example also sets the background cooperative level, so input will be available regardless of whether the form is in the foreground. Note also that keystrokes continue to be passed through to whatever application has the focus. Most applications don't need input when they're in the background, and in such cases the **DISCL\_FOREGROUND** flag should be set instead.

Next: Step 3: Gain Access to the Keyboard

---

### Step 3: Gain Access to the Keyboard

---

#### [C++]

This topic pertains only to application development in Visual Basic. See DirectInput C/C++ Tutorials.

---

#### [Visual Basic]

After your application sets the keyboard's behavior, it can acquire access to the device by calling the **DirectInputDevice.Acquire** method. The application must acquire the device before retrieving data from it. The **Acquire** method accepts no parameters.

Call `didev.Acquire`

In the sample, the application is unlikely to fail to acquire the keyboard, or to lose it later, because it is using the background, nonexclusive cooperative level. However, in general it is good practice to handle errors on calls to **Acquire** as well as when attempting to retrieve data.

Next: Step 4: Retrieve Immediate Data from the Keyboard

---

## Step 4: Retrieve Immediate Data from the Keyboard

---

### [C++]

This topic pertains only to application development in Visual Basic. See DirectInput C/C++ Tutorials.

---

### [Visual Basic]

Once a device is acquired, your application can start retrieving data from it. The simplest way to do this is to call the **DirectInputDevice.GetDeviceStateKeyboard** method, which takes a snapshot of the device's state at the time of the call.

The **GetDeviceStateKeyboard** method accepts as its single parameter a **DIKEYBOARDSTATE** type, which simply contains an array of 256 bytes.

The following sample retrieves the state of the keyboard:

```
Dim state As DIKEYBOARDSTATE
Call didev.GetDeviceStateKeyboard(state)
```

After retrieving the keyboard's current state, your application may respond to specific keys that were down at the time of the call. Each element in the buffer represents a key. If an element's high bit is on, the key was down at the moment of the call; otherwise, the key was up. To check the state of a given key, use the constants of the **CONST\_DIKEYFLAGS** enumeration to index the buffer for a given key.

The following sample code shows how an application might move a vehicle around in response to the arrow keys:

```
If state.Key(DIK_UP) And &H80 Then
    ' Move the vehicle up
End If
If state.Key(DIK_DOWN) And &H80 Then
    ' Move the vehicle down
End If
```

' And so on.

The following works just as well:

```
If state.Key(DIK_UP)
  ' Move the vehicle up
End If
```

The value of the key state is 0 if the key is up, as is the case with all buttons in the current version of DirectInput. However, future versions might support "analog" keys with more than two states and hence more than a single nonzero value, so this shorter version of the code should be used with caution, especially for devices other than the keyboard.

Remember also that DIK\_UP is a single key, the dedicated up arrow key. DirectInput treats the 8 key on the numerical keypad as a distinct key, and gives it the same identifier regardless of whether NUM LOCK is on. In order to allow input from either of the arrow keys, you would have to write code like this:

```
If state.Key(DIK_UP) Or state.Key(DIK_NUMPAD8) Then
  ' Move the vehicle up
End If
```

---

## Tutorial 2: Using the Mouse

---

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

This topic pertains only to application development in Visual Basic. See DirectInput C/C++ Tutorials.

---

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

This tutorial focuses on using the mouse at the exclusive cooperative level and shows how to retrieve buffered data in response to notifications. The sample code is based on the ScrawlB sample.

The tutorial is divided into the following steps:

- Step 1: Set Up the Mouse
  - Step 2: Set Up Notifications
  - Step 3: Manage Exclusive Access to the Mouse
  - Step 4: Retrieve Buffered Data from the Mouse
-

---

## Step 1: Set Up the Mouse

---

### [C++]

This topic pertains only to application development in Visual Basic. See DirectInput C/C++ Tutorials.

---

### [Visual Basic]

First steps in setting up the mouse for use under DirectInput are similar to those taken in Tutorial 1: Using the Keyboard. In the ScrawlB sample, initialization takes place in **Sub Main** after some global declarations:

```
Public objDX As New DirectX7
Public objDI As DirectInput
.
.
.

Set objDIDev = objDI.CreateDevice("guid_SysMouse")
Call objDIDev.SetCommonDataFormat(DIFORMAT_MOUSE)
Call objDIDev.SetCooperativeLevel(frmCanvas.hwnd, _
    DISCL_FOREGROUND Or DISCL_EXCLUSIVE)
```

This time the device takes exclusive control of the device. The result is that as long as the application has the mouse in the acquired state, Windows does not generate mouse messages or display the system cursor. Note that **DISCL\_EXCLUSIVE** must be combined with **DISCL\_FOREGROUND**; it is not possible for an application to have exclusive access to the mouse and also receive input when it loses the focus.

Because the ScrawlB sample application is taking over full responsibility for the mouse, it must also track the position of its private cursor and also must scale movement of the cursor to movements of the mouse. The following global variables are used to store the cursor coordinates (in pixels relative to the upper left corner of the main form) and movement scaling:

```
Public g_cursorex As Long
Public g_cursory As Long
Public g_Sensitivity
```

Back in **Sub Main**, the application sets the buffer size so that it can receive buffered data, using the **DirectInputDevice.SetProperty** method:

```
Dim diProp As DIPROPLONG
diProp.lHow = DIPH_DEVICE
diProp.lObj = 0
diProp.lData = BufferSize ' BufferSize is a constant
diProp.lSize = Len(diProp)
Call objDIDev.SetProperty("DIPROP_BUFFERSIZE", diProp)
```

Next: Step 2: Set Up Notifications

---

## Step 2: Set Up Notifications

---

### [C++]

This topic pertains only to application development in Visual Basic. See DirectInput C/C++ Tutorials.

---

### [Visual Basic]

Rather than polling for mouse input in **Sub Main**, the ScrawlB sample application relies on DirectInput to notify it whenever a mouse event takes place. As part of initialization, the application gets an event handle and passes it to

**DirectInputDevice.SetEventNotification:**

```
Dim EventHandle As Long
EventHandle = objDX.CreateEvent(frmCanvas)
Call objDIDev.SetEventNotification(EventHandle)
```

DirectInput will now notify any object that implements the **DirectXEvent** class. In the case of ScrawlB, the implementing object is the *frmCanvas* form, whose declarations section contains the following line:

```
Implements DirectXEvent
```

This line causes *frmCanvas* to inherit all the methods of the **DirectXEvent** class. As it happens, the only visible method of that class is **DirectXEvent.DXCallback**, and the inheriting class must implement this method. DirectInput will call this method whenever an input event is signaled.

The implementation of **DXCallback** is covered under Step 4: Retrieve Buffered Data from the Mouse.

Next: Step 3: Manage Exclusive Access to the Mouse

---

## Step 3: Manage Exclusive Access to the Mouse

---

### [C++]

This topic pertains only to application development in Visual Basic. See DirectInput C/C++ Tutorials.

---

### [Visual Basic]

The ScrawlB sample demonstrates using the mouse with the exclusive foreground cooperative level. At this level, Windows does not track the mouse while it is acquired by the application. There are three major consequences for the application:

- It is entirely responsible for displaying a cursor if one is needed, and for moving it at an appropriate speed in response to `DirectInput` data.
- It loses acquisition when the user switches to another application by using the keyboard, and must reacquire the mouse when the user switches back.
- It must provide a means for Windows to get the mouse back whenever the user needs to use the system cursor—for example, to navigate a menu within the application.

ScrawlB handles the first responsibility by keeping a private record of mouse movements, adjusted by a user-defined sensitivity value, and displaying an icon that serves as a cursor for drawing.

ScrawlB loses acquisition of the mouse involuntarily whenever the application window moves to the background. When it comes to the foreground again, the application automatically reacquires the mouse by calling

**`DirectInputDevice.Acquire`** in the `MouseMove` event handler, which is called whenever Windows sends a mouse message to the application. This happens as soon as the system cursor moves over the client window, or the client window gains the focus under the system cursor. Once ScrawlB reacquires the mouse, of course, no more Windows mouse messages are sent, so the **`Form_MouseMove`** method is not called in response to subsequent mouse events.

In order to let the user navigate the context menu, or to display the system cursor for some other purpose such as resizing the window, ScrawlB calls the **`DirectInputDevice.Unacquire`** method whenever the user opens the menu. When the menu is closed, **`Form_MouseMove`** is called and the mouse is reacquired, unless the user has chosen **`Suspend`** from the menu, in which case a flag is set. This flag prevents **`Form_MouseMove`** from reacquiring, so that the user can continue using the system cursor.

The sample application demonstrates one more technique for releasing the mouse. If the user opens the system menu by pressing ALT+SPACE, Windows sends a `WM_ENTERMENULOOP` message. This message is intercepted by a subclassed window procedure, which then unacquires the mouse. As long as Windows is using the system cursor for navigating the menu or allowing the user to move or resize the window, it sends no mouse messages to *frmCanvas*, so the application doesn't attempt to reacquire the mouse in **`Form_MouseMove`**. A similar technique could be used for intercepting other Windows messages such as `WM_ACTIVATE` or `WM_ACTIVATEAPP`, so that the application could fully control acquisition and unacquisition of the mouse in response to gaining and losing the focus.

Next: Step 4: Retrieve Buffered Data from the Mouse

---



---

## Step 4: Retrieve Buffered Data from the Mouse

---

### [C++]

This topic pertains only to application development in Visual Basic. See *DirectInput C/C++ Tutorials*.

---

### [Visual Basic]

The ScrawlB sample retrieves buffered mouse data inside the **DirectXEvent.DXCallback** method implemented by *frmCanvas*. This method is called each time an input event is signaled.

The method declares a buffer of the same size as the buffer *DirectInput* is using to store the data. This is the size that was set previously by a call to **DirectInputDevice.SetProperty**.

```
Dim diDeviceData(1 To BufferSize) As DIDEVICEOBJECTDATA
```

Also required are a variable to receive the number of items actually retrieved, a loop counter, and a variable to track the previous sequence number of an event:

```
Dim NumItems As Integer
Dim i As Integer
Static OldSequence As Long
```

The application now retrieves all the data available, in a single call to **DirectInputDevice.GetDeviceData**:

```
On Error GoTo INPUTLOST
NumItems = objDIDev.GetDeviceData(diDeviceData, 0)
On Error GoTo 0
```

Note the error trap. One of the events that *DirectInput* will signal is loss of acquisition. If the user switches to another application, for instance, *ScrawlB* will no longer have the mouse in the acquired state. An event will be signaled and this method will be called, but **GetDeviceData** will fail, because data can only be retrieved from an acquired device.

The application now iterates through the retrieved items and examines the data in the **DIDEVICEOBJECTDATA** type, comparing the **IOfs** member with the constants for the various buttons and axes that are of interest. For the x-axis, for instance, the application extracts the change in axis position from the **IData** member and uses this to adjust the cursor position, taking into account the user-defined sensitivity:

```
For i = 1 To NumItems
    Select Case diDeviceData(i).IOfs
        Case DIMOFS_X
            g_cursorx = g_cursorx + diDeviceData(i).IData * _
                g_Sensitivity
```

The application also examines the sequence number and compares it with the previous one. If two axis events have the same sequence number, the mouse has been moved diagonally, and it's not desirable to update the cursor position or draw a line until both movements have been taken into account.

```
If OldSequence <> diDeviceData(i).ISequence Then
    UpdateCursor ' Move cursor and maybe draw line
    OldSequence = diDeviceData(i).ISequence
Else
    OldSequence = 0
End If
```

For the buttons, the method determines the kind of event by checking the appropriate bit in **IData**. If the bit is set, the mouse button was pressed; otherwise, it was released. Remember, **GetDeviceData** does not return the current state of the device, so it is up to the application to keep a private record of whether a button is being held down. For the left button, ScrawlB keeps this information in the *Drawing* Boolean variable.

```
Case DIMOFS_BUTTON0
    If diDeviceData(i).IData And &H80 Then
        Drawing = True

        ' Keep record for Line function
        CurrentX = g_cursorex
        CurrentY = g_cursory

        ' Draw point in case button-up follows immediately
        PSet (g_cursorex, g_cursory)
    Else
        Drawing = False
    End If
End Select
Next i
```

---

## Tutorial 3: Using the Joystick

---

### [C++]

This topic pertains only to application development in Visual Basic. See DirectInput C/C++ Tutorials.

---

### [Visual Basic]

This tutorial shows how to create a joystick device, set its properties, and retrieve immediate data. The example code is based on the Joystick sample.

The tutorial breaks down the required tasks into the following steps:

- Step 1: Enumerate and Create the Joystick
- Step 2: Get Joystick Capabilities
- Step 3: Set Joystick Properties
- Step 4: Retrieve Immediate Data from the Joystick

---

## Step 1: Enumerate and Create the Joystick

---

### [C++]

This topic pertains only to application development in Visual Basic. See DirectInput C/C++ Tutorials.

---

### [Visual Basic]

Because there is no system joystick, in the sense that there can be a system keyboard or mouse, in order to create a **DirectInputDevice** object for a joystick you first need to obtain an instance GUID, or globally unique identifier. Generally this is done by enumerating the available joysticks, presenting the user with a choice, and then obtaining the information for the selected device.

The following function initializes DirectInput and enumerates attached joysticks:

```
Dim dx As New DirectX7
Dim di As DirectInput
Dim diDev As DirectInputDevice
Dim diDevEnum As DirectInputEnumDevices

Sub InitDirectInput()
    On Error GoTo Error_Out

    Set di = dx.DirectInputCreate()
    Set diDevEnum = di.GetDIEnumDevices( _
        DIDEVTYPE_JOYSTICK, DIEDFL_ATTACHEDONLY)
    If diDevEnum.GetCount = 0 Then
        MsgBox "No joystick attached."
        Unload Me
    End If

    'Add attached joysticks to the listbox
    Dim i As Integer
    For i = 1 To diDevEnum.GetCount
```

```
        Call lstJoySticks.AddItem( _  
            diDevEnum.GetItem(i).GetInstanceName)  
    Next  
  
    ' Get an event handle to associate with the device  
    EventHandle = dx.CreateEvent(Me)  
    Exit Sub  
  
Error_Out:  
    MsgBox "Error initializing DirectInput."  
    Unload Me  
  
End Sub
```

When the user selects a joystick from the list, the device is created and initialized in the `lstJoySticks_Click` procedure:

```
Set diDev = di.CreateDevice(diDevEnum.GetItem( _  
    lstJoySticks.ListIndex + 1).GetGuidInstance)  
diDev.SetCommonDataFormat DIFORMAT_JOYSTICK  
diDev.SetCooperativeLevel Me.hWnd, _  
    DISCL_FOREGROUND Or DISCL_NONEXCLUSIVE
```

The call to **DirectInput.CreateDevice** takes as its parameter the GUID for a **DirectInputDeviceInstance** object obtained from the enumeration. Note that like all enumerated collections in DirectX for Visual Basic, the device enumeration is 1-based, so the index is one greater than the index of the selected item in the list box. The list box must also be unsorted.

You must set the data format before attempting to enumerate objects on the device or manipulate its properties.

Next: Step 2: Get Joystick Capabilities

---

## Step 2: Get Joystick Capabilities

---

### [C++]

This topic pertains only to application development in Visual Basic. See **DirectInput C/C++ Tutorials**.

---

### [Visual Basic]

Getting basic information about the buttons, axes, and point-of-view controllers on the device requires a simple call to **DirectInputDevice.GetCapabilities**:

```
Dim joyCaps As DIDEVCAPS
```

---

Call diDev.GetCapabilities joyCaps

The Joystick sample is interested only in the number of buttons and point-of-view controllers on the device. Although the **DIDEVCAPS** type also reports the number of axes on the device, it does not reveal anything about what those axes are. For this information, the sample calls the IdentifyAxes procedure:

```
Sub IdentifyAxes(diDev As DirectInputDevice)

    Dim didoEnum As DirectInputEnumDeviceObjects
    Dim dido As DirectInputDeviceObjectInstance
    Dim i As Integer

    For i = 1 To 8
        AxisPresent(i) = False
    Next
```

After declaring a few variables and clearing out the *AxisPresent* array (which stores Boolean values for each possible axis), the procedure goes on to enumerate all device objects:

```
Set didoEnum = diDev.GetDeviceObjectsEnum(DIDFT_AXIS)
```

It then queries each **DirectInputDeviceObjectInstance** for its offset within the data format that was established earlier by the call to **DirectInputDevice.SetCommonDataFormat**. This offset identifies the conventional role or type of the axis; for instance, you know that a DIJOFS\_RZ axis likely corresponds to a twisting motion on the main stick. Keep in mind, though, that device drivers are free to assign any designation to an axis. It is always a good idea to allow users to change the mapping of the axes to actions within your application.

```
For i = 1 To didoEnum.GetCount
    Set dido = didoEnum.GetItem(i)
    Select Case dido.GetOfs
        Case DIJOFS_X
            AxisPresent(1) = True
        Case DIJOFS_Y
            AxisPresent(2) = True
    ' and so on
    .
    .
    .
    End Select
Next
End Sub
```

The application now passes the event handle created earlier to the device, so that notifications will be sent to the form when an input event takes place.

---

Call `diDev.SetEventNotification(EventHandle)`

Next: Step 3: Set Joystick Properties

---

## Step 3: Set Joystick Properties

---

### [C++]

This topic pertains only to application development in Visual Basic. See [DirectInput C/C++ Tutorials](#).

---

### [Visual Basic]

Property changes are made through the **DirectInputDevice.SetProperty** method, and must be made after the data format of the device is established, but before it is acquired.

Because different devices might return different ranges of axis values, it is a good idea to set a range that will apply to all devices. The Joystick sample requests that all axis values be within the range 0 to 10,000.

```
Dim DiProp_Range As DIPROPRANGE
```

```
With DiProp_Range
```

```
    .IHow = DIPH_DEVICE ' Set for all axes
```

```
    .ISize = Len(DiProp_Range)
```

```
    .IMin = 0
```

```
    .IMax = 10000
```

```
End With
```

```
diDev.SetProperty "DIPROP_RANGE", DiProp_Range
```

Note that you must specify the length of the **DIPROPRANGE** type within its **ISize** member, because the method can handle parameters of different types, and must know how much memory to allocate. If you're used to programming in C++, you should also note that **Len(DIPROPRANGE)** is not valid, because it returns zero.

The Joystick sample also sets the dead zone and saturation zones for the stick. In this case, it's not desirable to set these values for other axes such as a throttle or rudder, so the **IHow** member of the property type is set to indicate that the property change applies to a device object identified by its offset within the data format, and the **IObj** member is set to that offset value.

```
Dim DiProp_Dead As DIPROPLONG
```

```
With DiProp_Dead
```

```
    .IData = 1000
```

```
    .IObj = DIJOFS_X
```

```
.ISize = Len(DiProp_Dead)
.IHow = DIPH_BYOFFSET
' Set for x-axis
.IObj = DIJOFS_X
diDev.SetProperty "DIPROP_DEADZONE", DiProp_Dead
' Set for y-axis
.IObj = DIJOFS_Y
diDev.SetProperty "DIPROP_DEADZONE", DiProp_Dead
End With
```

The value in **IData** is the proportion of the range of travel, in units of 10,000, that is set up as a dead zone. The value of 1000 used in the example specifies that the middle 10 percent of the range of travel on the x-axis and y-axis will be reported as 5000, which happens to be the center of the range set in the previous step.

#### Note

Dead zone and saturation values are always in units of 10,000, regardless of the range of values reported by the device.

The saturation zones for the device are set similarly. In this case **IData** is 9500, because it represents the proportion of the range of travel that lies outside the saturation zones at the extremities. In other words, the bottom and top 5 percent of the raw values returned by the stick will be reported as the minimum or maximum range value set for the device (in this case, 0 or 10,000).

Next: Step 4: Retrieve Immediate Data from the Joystick

---

## Step 4: Retrieve Immediate Data from the Joystick

---

### [C++]

This topic pertains only to application development in Visual Basic. See DirectInput C/C++ Tutorials.

---

### [Visual Basic]

Now that the necessary properties have been set, the device can be acquired and data can be collected from the joystick. In the Joystick sample, the device is polled in a loop after its properties have been set. In a real-world multimedia application, polling would take place in the main game loop or rendering loop.

```
diDev.Acquire

While DoEvents
    diDev.Poll
Wend
```

The sample application does not actually retrieve immediate data each time the device is polled. To avoid unnecessary screen updates, it relies on notification, and the call to **Poll** is only to ensure that notifications are issued by devices that do not generate interrupts. Polling is not necessary for some game devices (such as HIDs), but it is just as efficient to make a redundant call to **DirectInputDevice.Poll** as it would be to check a flag for the device before calling **Poll**.

Notifications are handled, as usual, in the implementation of the **DirectXEvent.DXCallback** method. Each time this procedure is called, the application knows that some joystick event has occurred. At this point it could call **DirectInputDevice.GetDeviceData** to retrieve any pending axis changes or button events. However, with a game controller it would be more usual to retrieve the state of the entire device, so the sample uses **DirectInputDevice.GetDeviceStateJoystick**.

Dim js As DIJOYSTATE

diDev.GetDeviceStateJoystick js

The values in the **DIJOYSTATE** type now tell the application everything it needs to know about the state of the device objects. Since it knows from Step 2: Get Joystick Capabilities what buttons, axes, and point-of-view controllers are actually present, it also knows which members of the type to ignore.

---

## Tutorial 4: Using Force Feedback

---

### [C++]

This topic pertains only to application development in Visual Basic. See DirectInput C/C++ Tutorials.

---

### [Visual Basic]

This tutorial shows how to test for the presence of a force-feedback device, how to create and play an effect, and how to change the parameters of an effect as it is playing.

The tutorial breaks down the tasks into the following steps:

- Step 1: Initialize the Force-Feedback Device
  - Step 2: Set Device Properties
  - Step 3: Create an Effect
  - Step 4: Modify an Effect
-



---

## Step 1: Initialize the Force-Feedback Device

---

### [C++]

This topic pertains only to application development in Visual Basic. See DirectInput C/C++ Tutorials.

---

### [Visual Basic]

In order to implement force feedback, you must first determine whether an appropriate device is available, and if so, create a **DirectInputDevice** object for it. Let's presume that your application requires a device that can play effects on the x-axis and y-axis.

The following example starts by creating a **DirectInput** object, then using that to enumerate the available devices in a **DirectInputEnumDevices** object. The enumeration is restricted to attached devices of type DIDEVTYPE\_JOYSTICK, since all force-feedback devices are of this type.

```
' dx is the global DirectX object, already initialized
' di is the global DirectInput object

Dim diDevInst As DirectInputDeviceInstance
Dim diDevObjEnum As DirectInputEnumDeviceObjects
Dim devobj As DirectInputDeviceObjectInstance

Dim devcaps As DIDEVCAPS
Dim ForceX As Boolean, ForceY As Boolean
Dim FoundForce As Boolean
Dim eftype As Long
Dim strObjGuid As String

Dim i As Integer, iAxes As Integer

Set di = dx.DirectInputCreate
Set diEnumDev = di.GetDIEnumDevices(DIDEVTYPE_JOYSTICK, _
    DIEDFL_ATTACHEDONLY)
```

The example now iterates through the available devices till it finds one that has force-feedback capabilities.

```
For i = 1 To diEnumDev.GetCount
    Set diDevInst = diEnumDev.GetItem(i)
    Set didev = di.CreateDevice(diDevInst.GetGuidInstance)
    Call didev.GetCapabilities(devcaps)
    If devcaps.IFlags And DIDC_FORCEFEEDBACK Then
```

Now the example enumerates the axes on the device in a **DirectInputEnumDeviceObjects** collection:

```
Set diDevObjEnum = didev.GetDeviceObjectsEnum(DIDFT_AXIS)
ForceX = False
ForceY = False
```

It then goes through the axes looking for the x-axis and y-axis, and checks to see whether they support force-feedback effects:

```
For iAxes = 1 To diDevObjEnum.GetCount
    Set devobj = diDevObjEnum.GetItem(iAxes)
    strObjGuid = devobj.GetGuidType
    If strObjGuid = "GUID_XAxis" Then
        If devobj.GetFlags And DIDOI_FFACTUATOR Then
            ForceX = True
        End If
    ElseIf strObjGuid = "GUID_YAxis" Then
        If devobj.GetFlags And DIDOI_FFACTUATOR Then
            ForceY = True
        End If
    End If
Next iAxes
FoundForce = ForceX And ForceY
If FoundForce Then Exit For
End If
Next i

If Not FoundForce Then
    MsgBox "Two force feedback axes required."
End
```

Next: Step 2: Set Device Properties

---

## Step 2: Set Device Properties

---

### [C++]

This topic pertains only to application development in Visual Basic. See DirectInput C/C++ Tutorials.

---

### [Visual Basic]

Presuming you have found a suitable force-feedback device, the **DirectInputDevice** object created in the last step in order to check capabilities can be left in place. It requires some further initialization, as covered in Tutorial 3: Using the Joystick:

setting the data format and cooperative level, and setting the range properties for input. Note that the cooperative level must include DISCL\_EXCLUSIVE. Although you probably won't need background access to the device, the DISCL\_BACKGROUND flag makes debugging easier, because you won't keep losing acquisition when you switch to the code window.

```
Call didev.SetCommonDataFormat(DIFORMAT_JOYSTICK2)
Call didev.SetCooperativeLevel(Me.hWnd, _
    DISCL_BACKGROUND Or DISCL_EXCLUSIVE)
' Set range properties...
.
.
.
```

For force feedback, you might also want to disable the autocenter property of the device, so that the default spring action will not interfere with effects you create:

```
Dim prop As DIPROPLONG
prop.lData = 0
prop.lHow = DIPH_DEVICE
prop.lObj = 0
prop.lSize = Len(prop)
Call didev.SetProperty("DIPROP_AUTOCENTER", prop)

didev.Acquire
```

Next: Step 3: Create an Effect

---

## Step 3: Create an Effect

---

### [C++]

This topic pertains only to application development in Visual Basic. See DirectInput C/C++ Tutorials.

---

### [Visual Basic]

Let's suppose that in your application you wish to simulate the pull of a stationary object (say a magnet) on a movable object (say an iron ball) controlled by the joystick. At the beginning of the simulation, the iron ball is due south of the magnet and experiencing one-half the maximum possible pull.

You will simulate the pull of the magnet by a constant force. First you set up the **DIEFFECT** type that describes the force:

```
Dim EffectInfo as DIEFFECT
With EffectInfo
```

```
.constantForce.IMagnitude = 5000  
.IDuration = -1      ' Infinite  
.x = 18000  
.IGain = 10000      ' Play at full magnitude  
.ITriggerButton = -1 ' No trigger button  
End With
```

All the other members of **DIEFFECT** are either not relevant to a constant force or are valid as 0.

Now create the effect:

```
Dim di_effect As DirectInputEffect  
Set di_effect = didev.CreateEffect("GUID_ConstantForce", EffectInfo)
```

If the device is in an acquired state at this time, the effect is automatically downloaded. If not, the effect will be downloaded when it is started.

```
di_effect.Start(1, 0)
```

Only one iteration of the effect is needed, since it has infinite duration, and no flags are required in this case.

Next: Step 4: Modify an Effect

---

## Step 4: Modify an Effect

---

### [C++]

This topic pertains only to application development in Visual Basic. See DirectInput C/C++ Tutorials.

---

### [Visual Basic]

As the user moves the iron ball around, the application will adjust the direction and magnitude of the effect. First of all, though, you may want to check whether the device allows you to change the direction of the effect without stopping and restarting it. DirectInput will stop and restart the effect automatically if necessary, but you might want to modify the behavior of your application to avoid breaks in continuity.

In order to examine the capabilities for any effect, you need to enumerate supported effects and then look for the particular one you are interested in.

```
Dim diEnumEffects As DirectInputEnumEffects  
Dim params As Long  
Dim i As Long
```

```
' Enumerate constant forces
```

```
Set diEnumEffects = didev.GetEffectsEnum(DIEFT_CONSTANTFORCE)
```

```
For i = 1 To diEnumEffects.GetCount
```

```
    ' Look for the standard constant force. There could be others.
```

```
    If diEnumEffects.GetEffectGuid(i) = "GUID_ConstantForce" Then
```

```
        params = diEnumEffects.GetDynamicParams(i)
```

```
        Exit For
```

```
    End If
```

```
Next i
```

```
If Not (params And DIEP_DIRECTION) Then
```

```
    ' Cannot change direction dynamically. Take remedial action such
```

```
    ' as limiting points at which modifications will be made.
```

```
End If
```

Similarly, to learn whether you can change the magnitude of the effect dynamically, perform the following check:

```
if Not (params And DIEP_TYPESPECIFICPARAMS)
```

```
    ' Can't change type-specific parameters dynamically.
```

```
End If
```

Now you wish to modify the effect each time the device has been polled for input and the iron ball has moved. Let's presume that you have calculated the new magnitude in *CurrentMag* and the new direction (in degrees) in *CurrentBearing*. In order to make the necessary changes, you need to initialize only those members of **DIEFFECT** that are relevant.

```
EffectInfo.constantForce.IMagnitude = CurrentMag
```

```
EffectInfo.x = CurrentBearing * 100
```

You now pass the **DIEFFECT** type to **DirectInputEffect.SetParameters** along with flags indicating which members contain valid data:

```
di_effect.SetParameters(EffectInfo, _  
    DIEP_DIRECTION Or DIEP_TYPESPECIFICPARAMS)
```

The **DIEP\_TYPESPECIFICPARAMS** flag indicates that any member containing information particular to that type of effect is valid. In the case of a constant force, this means the **constantForce** member.

---

# DirectInput Tools and Samples

This section describes tools and sample applications included with the DirectX SDK that pertain to DirectInput. Descriptions are organized as follows:

- DirectInput Tools
- DirectInput C/C++ Samples
- DirectInput Visual Basic Samples

## DirectInput Tools

The following tools are provided for use with DirectInput:

- DirectInput QuickTest
- Force Editor

## DirectInput QuickTest

### Description

The DirectInput QuickTest application allows you to see the properties of devices and device objects, and to see the input data from a device.

### Path

Executable: (*SDK root*)\Samples\Multimedia\DInput\Bin\Diquick.exe

### User's Guide

On the opening screen, select a device and click the **Create** button. If the device is successfully created, you see a property sheet with four different tabbed pages.

On the **Mode** tabbed page, choose a cooperative level, a data mode, and an axis mode. For a joystick, you will have to choose the polled data mode. You also need to confirm the device type in the "Form of" box. If you want to see buffered data, you must set the buffer size to greater than zero on this tabbed page.

The **Caps** tabbed page gives you miscellaneous information about the device.

On the **Objects** tabbed page you can enumerate selected buttons and axes on the device. The **Properties** button brings up a summary of the data represented by the **DIDeviceObjectInstance** structure (C++) or **DirectInputDeviceObjectInstance** object (Visual Basic) for the selected object. You can also set the range, deadzone, and saturation for an axis on this property sheet.

Finally, on the **Data** tabbed page you see input data from the device: immediate data on the left, and buffered data (if you've set a buffer size) on the right.

## Force Editor

### Description

The Force Editor application lets you design force-feedback effects and test them singly or in combination. Effects can be saved to file and then loaded into DirectInput applications.

### Path

Executable: (*SDK root*)\Bin\Dxutils\Fedit.exe

### User's Guide

Press F1 for online help.

## DirectInput C/C++ Samples

---

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

This section pertains only to application development in C and C++. See DirectInput Visual Basic Samples.

---

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

The following sample programs demonstrate the use and capabilities of DirectInput:

- DIGame Sample
- FFDonuts Sample
- FFFileRead Sample
- JoyFFeed Sample
- JoystImm Sample
- KeybdBuf Sample
- KeybdExc Sample
- KeybdImm Sample
- MouseExc Sample
- MouseNon Sample
- Scrawl Sample

In addition, amply commented source code illustrating basic DirectInput functionality is available at the following location:

(*SDK root*)\Samples\Multimedia\DInput\Src\Tutorials

Although DirectX samples include Microsoft® Visual C++® project workspace files, you might need to verify other settings in your development environment to ensure that the samples compile properly. For more information, see *Compiling DirectX Samples and Other DirectX Applications*.

---

## DIGame Sample

---

[\[Visual Basic\]](#)

This topic pertains only to application development in C and C++. See *DirectInput Visual Basic Samples*.

---

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

### Description

The DIGame sample application is a simple shooting game that can be played with keyboard, mouse, or joystick.

### Path

Source: (*SDK root*)\Samples\Multimedia\DInput\Src\DIGame

Executable: (*SDK root*)\Samples\Multimedia\DInput\Bin

### User's Guide

The default input device is the keyboard. Change the device by choosing **Select Input Device** from the **File** menu.

Move the ship up and down with the arrow keys, the mouse, or the joystick. Change directions by moving left or right. Fire by pressing the spacebar, the left mouse button, or the primary button on the joystick.

To start again, select **New Game** from the **File** menu.

### Programming Notes

The input code is all in Digame.cpp.

---

## FFDonuts Sample

---

[\[Visual Basic\]](#)



This topic pertains only to application development in C and C++. See DirectInput Visual Basic Samples.

---

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

## Description

This is a variation on the Space Donuts sample program that adds force-feedback output if a force-feedback joystick is installed in your computer.

## Path

Source: (*SDK root*)\Samples\Multimedia\DInput\Src\FFdonuts

Executable: (*SDK root*)\Samples\Multimedia\DInput\Bin

## User's Guide

When the program is started, you see a dialog box that lets you set the magnitude of the force-feedback effects.

When your ship appears, move the joystick forward to accelerate forward and pull it back to decelerate or move backward. Moving the joystick left or right rotates the ship. Press the trigger button to fire. Press the second button to activate the shields. Observe the force-feedback effects as you fire, collide with objects, and bounce off the edge of the screen.

## Programming Notes

The force-feedback routines are in the Input.c file. The program illustrates the use of constant and periodic effects, envelopes, and gain. Note that the fire-button effect is played in response to an ordinary input event rather than being associated with a trigger button in the **DIEFFECT** structure.

## See Also

Space Donuts (DirectDraw sample)

---

## FFFileRead Sample

---

[\[Visual Basic\]](#)

This topic pertains only to application development in C and C++. See DirectInput Visual Basic Samples.

---

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

## Description

This sample shows how to read and create effects from a file created by Force Editor.

## Path

Source: *(SDK root)\Samples\Multimedia\DInput\Src\FFFileRead*

Executable: *(SDK root)\Samples\Multimedia\DInput\Bin\FFeedFileRead.exe*

Media: *(SDK root)\Samples\Multimedia\DInput\Media*

## User's Guide

Click **Read File** and load a supplied .ffe file from the Media folder or one of your own files. Then click **Play Effects**.

## Programming Notes

For each file effect enumerated, the EnumAndCreateEffectsCallback function initializes an **IDirectInputEffect** pointer and adds it to a linked list. The OnPlayEffects function traverses this list and plays all effects.

---

## JoyFFeed Sample

---

[\[Visual Basic\]](#)

This topic pertains only to application development in C and C++. See DirectInput Visual Basic Samples.

---

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

## Description

This application applies raw forces to a force-feedback joystick, illustrating how a simulator-type application can use force feedback to generate forces computed by a physics engine.

You must have a force-feedback device connected to your system in order to run the application.

## Path

Source: *(SDK root)\Samples\Multimedia\DInput\Src\JoyFFeed*

Executable: (*SDK root*)\Samples\Multimedia\DInput\Bin

## User's Guide

When you run the application, it displays a window with a crosshair and a black spot in it. Click the mouse anywhere within the window's client area to move the black spot. (Note that moving the joystick handle does not do anything.) JoyFFeed exerts a constant force on the joystick handle from the direction of the spot, in proportion to the distance from the crosshair. You can also hold down the mouse button and move the spot continuously.

## Programming Notes

This sample program enumerates the input devices and acquires the first force-feedback joystick that it finds. If none are detected, it displays a message and terminates.

When the user moves the black spot, the joySetForcesXY function converts the cursor coordinates to a force direction and magnitude. This data is used to modify the parameters of the constant force effect.

---

## JoystImm Sample

---

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

This topic pertains only to application development in C and C++. See DirectInput Visual Basic Samples.

---

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

## Description

The JoystImm sample obtains and displays joystick data.

## Path

Source: (*SDK root*)\Samples\Multimedia\DInput\Src\JoystImm

Executable: (*SDK root*)\Samples\Multimedia\DInput\Bin

## User's Guide

Observe how the displayed data changes when you move and twist the stick, rotate the throttle wheel, and press buttons in various combinations.

## Programming Notes

The application polls the joystick for immediate data in response to a timer set inside the dialog procedure.

---

## KeybdBuf Sample

---

[\[Visual Basic\]](#)

This topic pertains only to application development in C and C++. See DirectInput Visual Basic Samples.

---

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

### Description

The KeybdBuf program obtains and displays keyboard data.

### Path

Source: (*SDK root*)\Samples\Multimedia\DInput\src\KeybdBuf

Executable: (*SDK root*)\Samples\Multimedia\DInput\bin

### User's Guide

Each time you press or release a key, the event is recorded on the screen. Keys are identified by their index values (see Keyboard Device Constants).

## Programming Notes

This sample illustrates how an application can use DirectInput to obtain buffered keyboard data.

---

## KeybdExc Sample

---

[\[Visual Basic\]](#)

This topic pertains only to application development in C and C++. See DirectInput Visual Basic Samples.

---

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

## Description

The KeybdExc sample shows how to use the keyboard at the exclusive cooperative level.

## Path

Source: *(SDK root)\Samples\Multimedia\DInput\Src\KeybdExc*

Executable: *(SDK root)\Samples\Multimedia\DInput\Bin\KeybdExc.exe*

## User's Guide

Hold down one or more keys and the index value of each key (see Keyboard Device Constants) is shown. Note that the Windows key does not activate the system Start menu, as it does in the KeybdImm Sample.

## Programming Notes

This sample illustrates how an application can use DirectInput to obtain immediate keyboard data at the exclusive foreground cooperative level. The application repeatedly calls **IDirectInputDevice7::GetDeviceState** and displays a string containing the values of all the keys that are down.

---

## KeybdImm Sample

---

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

This topic pertains only to application development in C and C++. See DirectInput Visual Basic Samples.

---

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

## Description

The KeybdImm program obtains and displays keyboard data.

## Path

Source: *(SDK root)\Samples\Multimedia\DInput\Src\KeybdImm*

Executable: *(SDK root)\Samples\Multimedia\DInput\Bin*

## User's Guide

Hold down one or more keys and the index value of each key (see Keyboard Device Constants) is shown.

## Programming Notes

This sample illustrates how an application can use DirectInput to obtain immediate keyboard data at the nonexclusive foreground cooperative level. The application repeatedly calls **IDirectInputDevice7::GetDeviceState** and displays a string containing the values of all the keys that are down.

---

## MouseExc Sample

---

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

This topic pertains only to application development in C and C++. See DirectInput Visual Basic Samples.

---

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

## Description

The MouseExc program demonstrates how to initialize and get immediate data from a DirectInput device.

## Path

Source: *(SDK root)\Samples\Multimedia\DInput\Src\MouseExc*

Executable: *(SDK root)\Samples\Multimedia\DInput\Bin*

## User's Guide

Move the mouse around and observe how the change in coordinates is displayed. Hold down a mouse button and its number is shown. Note that the system cursor is not present.

## Programming Notes

This sample illustrates how an application can use DirectInput to obtain relative mouse data in exclusive foreground mode. Approximately 30 times per second the program displays the change in mouse coordinates since the last call to **IDirectInputDevice7::GetDeviceState**.

---

## MouseNon Sample

---

[\[Visual Basic\]](#)

This topic pertains only to application development in C and C++. See DirectInput Visual Basic Samples.

---

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

### Description

The MouseNon program demonstrates how to initialize and get immediate data from a DirectInput device.

### Path

Source: *(SDK root)\Samples\Multimedia\DInput\Src\MouseNon*

Executable: *(SDK root)\Samples\Multimedia\DInput\Bin*

### User's Guide

Move the mouse around and observe how the change in coordinates is displayed. Hold down a mouse button and its number is shown. Note that the cursor doesn't have to be in the application window, but the application does have to be in the foreground.

### Programming Notes

This sample illustrates how an application can use DirectInput to obtain relative mouse data in non-exclusive foreground mode. Approximately 30 times per second the program displays the change in mouse coordinates since the last call to **IDirectInputDevice7::GetDeviceState**.

---

## Scrawl Sample

---

[\[Visual Basic\]](#)

This topic pertains only to application development in C and C++. See DirectInput Visual Basic Samples.

---

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

## Description

The Scrawl application demonstrates use of the mouse in exclusive mode in a windowed application.

## Path

Source: *(SDK root)\Samples\Multimedia\DInput\Src\Scrawl*

Executable: *(SDK root)\Samples\Multimedia\DInput\Bin*

## User's Guide

The main mouse button is always the left button, and the secondary button is always the right button, regardless of any settings the user may have made in Control Panel.

To scrawl, hold down the left button and move the mouse. Click the right mouse button to invoke a pop-up menu. From the pop-up menu you can clear the client window, set the mouse sensitivity, or close the application.

## Programming Notes

The Scrawl application demonstrates many aspects of DirectInput programming, including the following:

- Using the mouse in exclusive mode in a windowed application.
  - Releasing the mouse when Windows needs to use it for menu access.
  - Reacquiring the mouse when Windows no longer needs it.
  - Reading buffered device data.
  - Deferring screen updates till movement on both axes has been fully processed.
  - Event notifications of device activity.
  - Restricting the cursor to an arbitrary region.
  - Scaling raw mouse coordinates before using them.
  - Using relative axis mode.
-