DirectInput

This section provides information about the DirectInput® component of DirectX® The information is divided into the following topics:

- About DirectInput
- DirectInput Architecture
- DirectInput Essentials
- DirectInput Tutorials
- DirectInput Reference

About DirectInput

Microsoft® DirectInput® provides support for input devices including the mouse, keyboard, and joystick, as well as for force-feedback (input/output) devices. Like other DirectX® components, DirectInput is based on the Component Object Model (COM).

This release of DirectInput is the first to provide COM-based services for the joystick and other devices such as game pads, flight yokes, and virtual-reality headgear. These new services supersede the Win32® Application Programming Interface (API) functions centered on **joyGetPosEx**, which were previously documented as part of DirectInput.

Also new are the services for the force-feedback devices that are starting to appear in the game market.

Why Use DirectInput?

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

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. It also introduces the way DirectInput handles device access, input data, and force feedback effects.

- Architectural Overview of DirectInput
- The DirectInput Object
- The DirectInputDevice Object
- DirectInput Device Object Instances
- The DirectInputEffect Object
- Integration with Windows

Architectural Overview of DirectInput

The basic architecture of DirectInput consists of the DirectInput object, which supports a COM interface, and an object for each input device that provides data. Each device in turn has "object instances," which are individual controls or switches such as keys, buttons, or axes. Individual force feedback effects are also represented by objects.

Note

The word "object" is used to describe an entity created by the DirectInput system to support the methods of a COM interface, even when these methods are not being called through an object-oriented programming language such as C++. Somewhat confusingly, "object" may also mean one of the individual controls on an input device.

In the interests of speed and responsiveness, DirectInput works directly with device drivers, bypassing the Windows message system.

DirectInput enables an application to gain access to input devices even when the application is in the background.

The DirectInput Object

The DirectInput object in an application represents the DirectInput subsystem. It is used to enumerate and manage input devices.

You create the DirectInput object by calling the **DirectInputCreate** function, which returns a pointer to an **IDirectInput** COM interface. There are different versions of this interface for the ANSI and Unicode character sets.

Having created the DirectInput object, you can use the methods of the interface to enumerate individual devices available to the system and to create a DirectInputDevice object for each device you wish to use in your application.

The DirectInputDevice Object

Each DirectInputDevice object represents one input device such as a mouse, keyboard, or joystick. (In this documentation, the term "joystick" includes other game devices that behave similarly, such as steering wheels and game pads.)

You create a DirectInputDevice instance by calling the **IDirectInput::CreateDevice** method, which returns a pointer to the **IDirectInputDevice** interface. The **IDirectInputDevice** methods are used to get information about the device, set its properties, and get data from it.

Note

A physical device that is really a combination of different types of input devices, such as a keyboard with a trackball, may 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.

DirectInput Device Object Instances

An *object instance*, sometimes called simply an object, is one of the various switches and other controls available on an input device. For example, object instances on a joystick might include the x-axis and y-axis of the stick, several buttons, and a throttle slider. Mouse objects might include two or three buttons, the x-axis and y-axis, and a wheel. For a keyboard, each key is an object.

The application ascertains the number and type of objects available on a device through the **IDirectInputDevice::EnumObjects** method. Individual device objects are not encapsulated as code objects but are described in **DIDEVICEOBJECTINSTANCE** structures.

The DirectInputEffect Object

A DirectInputEffect object represents a force feedback effect that you have defined. It is used to manipulate the effect on the input/output device.

You create an DirectInputEffect object by calling the **IDirectInputDevice2::CreateEffect** function, which returns a pointer to an **IDirectInputEffect** COM interface.

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; as a result, 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 will be 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 will report a click of the middle button as two clicks of the primary button.

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

DirectInput Essentials

This section introduces the concepts and components of DirectInput, and provides enough information for you to get started in implementing the DirectInput system in your application.

The following topics are discussed:

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

DirectInput Device Enumeration

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

If your application is using only the standard keyboard or mouse, or both, you don't need to enumerate the available input devices. As explained under Creating the DirectInput Device, you can simply use predefined global variables when calling the **IDirectInput::CreateDevice** method.

For all other input devices, and for systems with multiple keyboards or mice, you need to call **IDirectInput::EnumDevices** in order to obtain at least the instance GUIDs (globally unique identifiers) so that device objects can be created.

Here's a sample implementation of the **IDirectInput::EnumDevices** method

```
GUID            KeyboardGUID = GUID_SysKeyboard;
// LPDIRECTINPUT  lpdi;   //This has been initialized with
                          //DirectInputCreate and points to
                          //the DirectInput object

lpdi->EnumDevices(DIDEVTYPE_KEYBOARD,
                  DIEnumDevicesProc,
                  &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 the reference for **DIDEVICEINSTANCE**.

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

The third parameter to the **EnumDevices** method is any 32-bit value that you want to pass into the callback function. In the examples above, it's 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.) The callback function is where you obtain the instance GUID of the device. You can also perform other processing here, such as looking for particular subtypes of devices.

Here is a sample callback function that 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 **EnumDevices** call above), which can then be used in a call to **IDirectInput::CreateDevice**.

```
BOOL        hasEnhanced;

BOOL CALLBACK DIEnumKbdProc(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 will be given a new value each time a device is enumerated. It is not actually important what instance GUID you use for a single keyboard, but the code does illustrate 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 will automatically stop as soon as all devices have been enumerated.

DirectInput Devices

This section contains information about the code objects that represent devices such as mice, 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

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

Device Setup

Your application must obtain a COM interface 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 may also wish 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 may only be necessary if you need further information about devices or need to change default values.

- Create the DirectInput device (required). See Creating a DirectInput Device.
- Get the device capabilities (optional).
- Enumerate the keys, buttons, and axes on the device (optional). See Device Object Enumeration.
- Set the cooperative level (recommended).
- Set the data format (required).
- Set the device properties (optional).
- When ready to read data, acquire the device (required). See Acquiring Devices.

Creating a DirectInput Device

The **IDirectInput::CreateDevice** method is used to obtain a pointer to the **IDirectInputDevice** interface. Methods of this interface are then used to manipulate the device and obtain data.

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

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

The first parameter in **IDirectInput::CreateDevice** 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, and you can pass these identifiers into the **CreateDevice** function. The global variable *GUID_Joystick* should not be used as a parameter for **CreateDevice**, because it is a product GUID, not an instance GUID.

Note

If the workstation 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 returned by **IDirectInput::EnumDevices**. The instance GUID for a device will always be 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.

If you want to use the **IDirectInputDevice2** interface methods for force feedback devices, you must obtain a pointer to that interface instead of **IDirectInputDevice**. The following function is a wrapper for the **CreateDevice** method that attempts to

obtain the **IDirectInputDevice2** interface. Note the use of macros to call the **Release** and **CreateDevice** methods according to either the C or C++ syntax.

```
HRESULT IDirectInput_CreateDevice2(LPDIRECTINPUT pdi,
                           REFGUID rguid,
                           LPDIRECTINPUTDEVICE2 *ppdev2,
                           LPUNKNOWN punkOuter)
{
   LPDIRECTINPUTDEVICE *pdev;
   HRESULT hres;

   hres = IDirectInput_CreateDevice(pdi, rguid, &pdev, punkOuter);

   if (SUCCEEDED(hres)) {
#ifdef __cplusplus
   hres = pdev->QueryInterface(IID_IDirectInputDevice2,
                               (LPVOID *)ppdev2);
#else
   hres = pdev->lpVtbl->QueryInterface(pdev,
                     &IID_IDirectInputDevice2,
                          (LPVOID *)ppdev2);
#endif
   IDirectInputDevice_Release(pdev);
   } else {
   *ppdev2 = 0;
   }
   return hres;
}
```

Device Capabilities

Before you begin asking for input from a device, you may 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 machine? Such questions are answered with a call to the **IDirectInputDevice::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 function.

Note

To optimize speed or memory usage, you can use the smaller **DIDEVCAPS_DX3** structure instead.

Here's an example that checks whether the mouse is attached and whether it has a third axis (presumably a wheel):

```
//  LPDIRECTINPUTDEVICE  lpdiMouse;  // initialized previously
```

```
DIDEVCAPS  DIMouseCaps;
HRESULT    hr;
BOOLEAN    HasWheel;

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

Another way to check for a certain button or axis is to call **IDirectInputDevice::GetObjectInfo** for that object. If the call returns DIERR_OBJECTNOTFOUND, the object is not present. The following code 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);
HasWheel = SUCCEEDED(hr);
```

Cooperative Levels

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 **IDirectInputDevice::SetCooperativeLevel** method, as in this example:

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

The parameters are the handle of the top-level window associated with the device (generally the application window) and one or more flags. 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
DISCL_EXCLUSIVE | DISC_BACKGROUND	Not valid for keyboard or mouse

Note

Although DirectInput provides a default setting, you should still explicitly set the cooperative level, because doing so is the only way to give DirectInput the

window handle. Without this handle, DirectInput will not be able to react to situations that involve window messages, such as joystick recalibration.

The cooperative level has two components: whether the device is being used in the foreground or the background, and whether it is being used exclusively or nonexclusively . Both these components require some explanation.

Foreground and 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 have a conditional define that sets the background cooperative level during debugging. This will prevent your application from losing access to the device every time it moves to the background as you switch to a debugging window.

Exclusive and 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.

Why does it matter? 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. Now suppose you run a similar application that plays movies, again in response to signals from the 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.

In order to use force feedback effects, an application must have exclusive access to the device.

Windows itself requires exclusive access to the mouse and keyboard. The reason is that mouse and keyboard events such as a click on an inactive window or ALT+TAB could force an application to unacquire the device, with potentially harmful results such as a loss of data from the input buffer.

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.

DirectInput does not allow any application to have exclusive access to the keyboard. If it did, Windows would not have access to the keyboard and the user would not even be able to use CTRL+ALT+DELETE to restart the system.

Device Object Enumeration

It may 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 you enumerate devices.

To some extent **IDirectInputDevice::EnumObjects** overlaps the functionality of **IDirectInputDevice::GetCapabilities**. Either method may be used to determine how many buttons or axes are available. However, **EnumObjects** is really intended for cataloguing all the available objects rather than checking for a particular one. The DIQuick application in the DirectX code samples in the Platform SDK References, for example, uses **EnumObjects** to populate the list on the Objects page for the selected device.

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

Here's a callback function that simply 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 **DIEnumDeviceObjectsProc**, but you can give it any name you like. Remember, this function is called once for each object enumerated.

```
char  szName[MAX_PATH];

BOOL CALLBACK DIEnumDeviceObjectsProc(
                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 still objects to be enumerated.

Now here's the call to the **EnumObjects** method, which puts the callback function to work.

```
lpdiMouse->EnumObjects(DIEnumDeviceObjectsProc,
                NULL, DIDFT_ALL);
```

The first parameter is the address of the callback function.

The second parameter can be a pointer to any data you want to use or modify in the callback. The example does not use this parameter and so 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 the reference for **IDirectInput::EnumDevices**.

Note

Some of the DIDFT_* flags are combinations of others; for example, DIDFT_AXIS is equivalent to DIDFT_ABSAXIS | DIDFT_RELAXIS.

Device Data Formats

Setting the data format for a device is an essential step before you can acquire and begin using the device. The **IDirectInputDevice::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** structure and **DIOBJECTDATAFORMAT** structure will give you an idea of how to set up custom data formats for nonstandard devices. Fortunately, this step is not necessary for the joystick, keyboard, and mouse. DirectInput provides four global variables, *c_dfDIJoystick*, *c_dfdiJoystick2*, *c_dfDIKeyboard*, and *c_dfDIMouse*, which can be passed into **SetDataFormat** to create a standard data format for these devices.

Here is an example, where *lpdiMouse* is an initialized pointer to the mouse device object:

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

Note

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

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 may have other properties as well.

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

Before calling the **IDirectInputDevice::SetProperty** or **IDirectInputDevice::GetProperty** methods you need to 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 two 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.

For **SetProperty**, the data members of the property structure are the values 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 "how" code indicating the way the object identifier should be interpreted

When getting or setting properties for a whole device, the object identifier *dwObj* is zero and the "how" code *dwHow* 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 the reference for the **DIPROPHEADER** structure.

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

The following values are used to identify the property passed to **SetProperty** and **GetProperty.** See the reference for **IDirectInputDevice::GetProperty** for more information.

- 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 example sets the buffer size for a device so that it will 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);
```

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 may manipulate its characteristics but not obtain any data.

Acquisition is not permanent; your application may acquire and unacquire a device many times.

In certain cases, depending on the cooperative level, a device may be unacquired automatically whenever your 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 need to unacquire a device before changing its properties. The only exception is that you may change the gain for a force feedback device while it is in an acquired state.

Why is the acquisition mechanism needed? There are two main reasons.

First, DirectInput has to 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 may have changed. Or 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 you wanted to get data with the **IDirectInputDevice::GetDeviceState** or **IDirectInputDevice::GetDeviceData** methods. Obviously this would be very inefficient. Even worse, potentially disastrous things could happen like a hardware interrupt accessing a data buffer just as you were changing the buffer size. So DirectInput requires your application to unacquire the device before changing properties. When you reacquire it, DirectInput looks at the properties and decides on the optimal way of transferring data from the device to the application. This is done only once, thereby making **GetDeviceState** and **GetDeviceData** very fast.

Since the most common cause of losing a device is that your application moves to the background, you may want to reacquire devices whenever your application is activated. However, this mechanism is not going to cover all cases where a device is unacquired, especially for devices other than the standard mouse or keyboard. Because your application may unacquire a device unexpectedly, you need to have 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, where a DIERR_INPUTLOST error triggers a message to reacquire the mouse. (See also Tutorial 2: Using the Mouse.)

There's no harm in attempting to reacquire a device that is already acquired. Redundant calls to **IDirectInputDevice::Acquire** are ignored, and the device can always be unacquired with a single call to **IDirectInputDevice::Unacquire**.

Remember, Windows doesn't 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 let it go. There's an example in Scrawl, which 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.

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 data are given in the following sections:

- Mouse Data
- Keyboard Data
- Joystick Data

Examples of retrieving data from input devices are found in the following tutorials:

- Tutorial 1: Using the Keyboard
- Tutorial 2: Using the Mouse
- Tutorial 3: Using the Joystick

Buffered and Immediate Data

DirectInput supplies two types of data: buffered and immediate. Buffered data is a record of events that is stored until an application retrieves it. 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 a pressed "fire" button. 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 get immediate data with the **IDirectInputDevice::GetDeviceState** method. As the name implies, this method simply 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 you can derive by comparing the current state with the last one. If the user manages to press and release a button between two calls to **GetDeviceState**, your application won't know anything about it. On the other hand, if the user is holding a button down, **GetDeviceState** will continue reporting "button down" until the user releases it.

This way of reporting the device state is different from the way Windows reports events with one-time messages like WM_LBUTTONDOWN; it is more akin to 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 doesn't keep responding to a button-down or key-down state when it's not appropriate to do so.

With buffered data, events are stored until you're 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 **IDirectInputDevice::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. Peeking could be used, for example, in an application that treats a double-click differently from a single click. Before responding to the first click, the application would peek at the buffer to see if another button-down event took place within a certain time interval after the first.

In order to get buffered data you must first set the buffer size with the **IDirectInputDevice::SetProperty** method. (See the example under Device Properties.) You set the buffer size before acquiring the device for the first time. For reasons of efficiency, the default size of the buffer is zero. You will not be able to obtain buffered data unless you change this value.

Note

The size of the buffer is measured in items of data for that type of device, not in bytes or words.

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

The DIQuick application supplied with the DirectX code samples in the Platform SDK Reference 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 page. Now, on the Data page, you see immediate data on the left and buffered data on the right.

Note

For devices that do not generate interrupts, such as analog joysticks, DirectInput does not obtain any data until you call the **IDirectInputDevice2::Poll** method. For more information, see Polling and Events.

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

See also:

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

Time Stamps and Sequence Numbers

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 the time the event took place. This is equivalent to the value that would have been returned by the Win32 **GetTickCount** function.

The **dwSequence** member contains a sequence number assigned by DirectInput. The DirectInput system keeps a single sequence counter, which is incremented by each non-simultaneous buffered event from any device. You can use this number to compare events from different devices and see which came first. The **DISEQUENCE_COMPARE** macro takes wraparound into account.

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 don't need to check the sequence numbers just to sort the events from a single device.

Polling and Events

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

Polling a device means regularly getting the current state of the device objects with **IDirectInputDevice::GetDeviceState** or retrieving the contents of the buffer with **IDirectInputDevice::GetDeviceData**. Polling is typically used by real-time games that are never idle but are constantly updating and rendering the game world.

Event notification is suitable for applications like the Scrawl sample that wait for input before doing anything. To use event notification, you 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 **IDirectInputDevice::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 sample code, see the Scrawl sample and the reference for **IDirectInputDevice::SetEventNotification**.

Some joysticks and other game devices, or particular objects on them, do not generate hardware interrupts and will not return any data or signal any events until you call the **IDirectInputDevice2::Poll** method. To find out whether this is necessary, first set the data format for the device, then call the **IDirectInputDevice::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 will be 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 **IDirectInputDevice::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, HID keyboards will be marked as DIDC_POLLEDDEVICE because the LEDs that indicate status, such as CAPS LOCK, require polling. However, the standard keyboard data format does not read the LEDs, so DIDC_POLLEDDATAFORMAT will not be set. Polling the device under these conditions is pointless, because the device objects that require polling (the LEDs) are inaccessible from the data format anyway.

It doesn't hurt to call the **IDirectInputDevice2::Poll** method for any input device. If the call is unnecessary, it will have no effect and will be very fast.

Relative and Absolute Axis Coordinates

Axis coordinates may be returned as relative values; that is, the amount by which they have changed since the last call to the **IDirectInputDevice::GetDeviceState** method 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 use the **IDirectInputDevice::SetProperty** method to change the default behavior of any axis or all the axes of a device.

Mouse Data

To set up the mouse device for data retrieval, first call the **IDirectInputDevice::SetDataFormat** method with the *c_dfDIMouse* global variable as the parameter.

For maximum performance in a full-screen application, set the cooperative level to DISCL_EXCLUSIVE | DISCL_FOREGROUND. Note that the exclusive setting will cause the Windows cursor to disappear. Remember too that the DISCL_FOREGROUND setting will cause the application to lose access to the mouse when you switch to a debugging window. Changing to DISCL_BACKGROUND will allow 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.

See also:

- Device Data Formats
- Cooperative Levels

Immediate Mouse Data

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

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

The **rgbButtons** member is an array of bytes, one for each of four buttons. Generally the first element in the array is 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 zero, so this step is essential. You then declare an array of **DIDEVICEOBJECTDATA** structures with the same number of elements as the buffer size.

After acquiring the device, you can examine and flush the buffer anytime by using the **IDirectInputDevice::GetDeviceData**method. (See Buffered and Immediate Data.)

Each element in the **DIDEVICEOBJECTDATA** array represents a change in state for a single object on the mouse. For instance, a typical mouse contains four objects or input sources: x-axis, y-axis, button 0 and button 1. If the user presses button 0 and moves the mouse diagonally, the array passed to **IDirectInputDevice::GetDeviceData** will have 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.

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:

- 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** 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 non-zero.

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

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, sometimes called mickeys, are based on the actual values returned by the mouse hardware. 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.

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 **IDirectInputDevice::SetProperty** with the DIPROP_AXISMODE value in the *rguid* parameter and with DIPROPAXISMODE_ABS in the **dwData** member of the **DIPROPDWORD** structure.

When the axis mode for the mouse is set to relative, the axis coordinate represents the number of mickeys 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 back. 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 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 may be larger.

Checking for Lost Mouse Input

Because Windows may 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 **IDirectInputDevice::GetDeviceData** or **IDirectInputDevice::GetDeviceState** methods, 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 are likely to get caught in an infinite loop: acquisition will fail, you will 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, don't use DirectInput methods; it is far easier to retrieve the data from the normal Windows messages, where 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 may require special translation of key presses.

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

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

Immediate Keyboard Data

To retrieve the current state of the keyboard, call the **IDirectInputDevice::GetDeviceState** method with a pointer to an array of 256 bytes that will 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.)

Here is an example that does something in response to the ESC key being down:

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

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

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 zero. You then declare an array of **DIDEVICEOBJECTDATA** structures with the same number of elements as the buffer size.

After acquiring the keyboard device, you can examine and flush the buffer anytime by using the **IDirectInputDevice::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 only counted 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 non-zero.

Interpreting Keyboard Data

This section covers the identification of keys for which data is reported by the **IDirectInputDevice:GetDeviceState** and **IDirectInputDevice: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. (For a list of the DIK_* values, see Keyboard Device Constants.)

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.

There is no key code for the PAUSE key. The PC enhanced keyboard does not generate a separate scan code for this key; rather, it synthesizes a "pause" from the DIK_LCONTROL and DIK_NUMLOCK codes.

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, then 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

Because Windows may 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 **IDirectInputDevice::GetDeviceData** or **IDirectInputDevice::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 are likely to get caught in an

infinite loop: acquisition will fail, you will get another DIERR_NOTACQUIRED error, and so on.

Joystick Data

The following sections cover getting and interpreting data from a joystick or other similar input device such as a game pad or flight yoke.

To set up the joystick device for data retrieval, first call the **IDirectInputDevice::SetDataFormat** method with the *c_dfDIJoystick* or *c_dfDIJoystick2* global variable as the parameter. (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 **IDirectInputDevice2::Poll** method before attempting to retrieve data from the joystick. For more information, see Polling and Events.

Immediate Joystick Data

To retrieve the current state of the joystick, call the **IDirectInputDevice::GetDeviceState** method with a pointer to a **DIJOYSTATE** or **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, **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 degrees clockwise from north (or forward). The center position is reported as -1. For controllers that have only five positions, **dwPOV** will be one of the following values:

-1

0

90 * DI_DEGREES

180 * DI_DEGREES

270 * DI_DEGREES.

Note

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.

See also Buffered and Immediate Data.

Buffered Joystick Data

To retrieve buffered data from the joystick, you must first set the buffer size (see Device Properties) and declare an array of **DIDEVICEOBJECTDATA** structures. The number of elements required in this array is the same as the buffer size. After acquiring the device, you can examine and flush the buffer anytime with the **IDirectInputDevice: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, a simple joystick contains four objects or input sources: x-axis, y-axis, button 0 and button 1. If the user presses button 0 and moves the stick diagonally, the array passed to **IDirectInputDevice::GetDeviceData** will have 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.

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
}
```

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.

Interpreting Joystick Axis Data

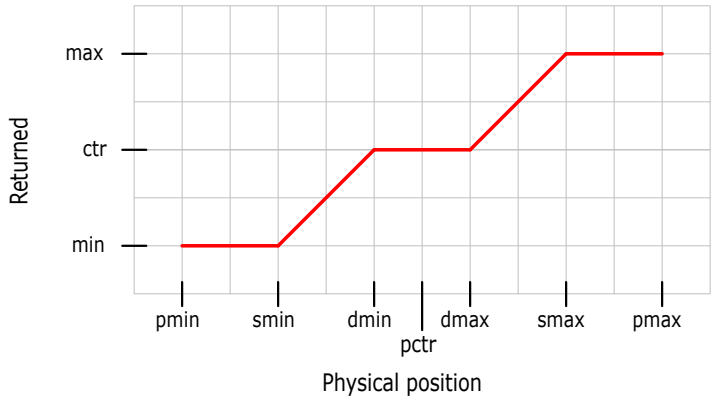
Axis values for the joystick are like those for the mouse: the value returned for the x-axis is greater as the stick moves to the right, and the value for the y-axis increases as the stick moves toward the user.

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 0 to 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 may be greater than 1, in which case values will be rounded off; for example, if the granularity is 10 then values will be 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 diagram 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*.



For more information on joystick properties, see the following:

- **IDirectInputDevice::GetProperty**
- **IDirectInputDevice::SetProperty**
- **DIPROPRANGE**

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.

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 **IDirectInputDevice::SetProperty** method with the DIPROP_AXISMODE value in the *rguid* parameter and with DIPROPAXISMODE_REL in the **dwData** member of the **DIPROPDWORD** structure.

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.

Checking for Lost Joystick Input

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

You can check for the DIERR_INPUTLOST return value from the **IDirectInputDevice::GetDeviceData** or **IDirectInputDevice::GetDeviceState** methods, 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 are likely to get caught in an infinite loop: acquisition will fail, you will get another DIERR_NOTACQUIRED error, and so on.

Because access to the joystick is not going to be 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 to the above method is to reacquire the device in response to a WM_ACTIVATE message.

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
- Information About a Supported Effect
- Creating an Effect
- Effect Direction
- Examples of Setting Effect Direction
- Envelopes and Offsets
- Effect Playback
- Downloading and Unloading Effects
- Changing an Effect
- Gain
- Force Feedback State
- Effect Object Enumeration
- Constant Forces
- Ramp Forces
- Periodic Effects
- Conditions
- Custom Forces
- Device-Specific Effects

To enumerate, create, and manipulate effects, you must first obtain a pointer to the **IDirectInputDevice2** interface for the force feedback device. For an example of how to do this, see Creating the DirectInput Device.

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.

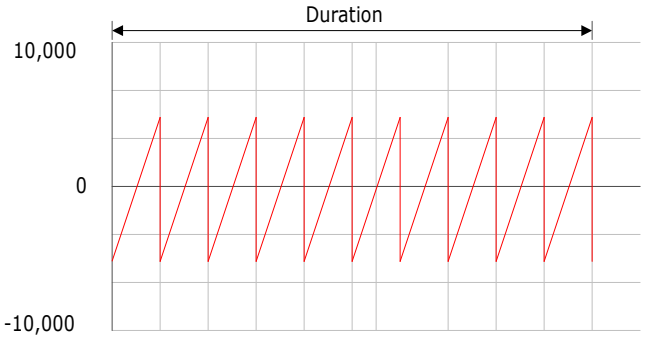
The strength of the force is called its *magnitude*. Magnitude is measured in units ranging from zero (no force) to 10,000 (maximum force for the device, defined in Dinput.h as DI_FFNOMINALMAX). A negative value indicates force in the opposite direction. Magnitudes are linear: a force of 10,000 is twice 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; just as a north wind comes from the north, a positive force on an given axis pushes from the positive toward the negative.

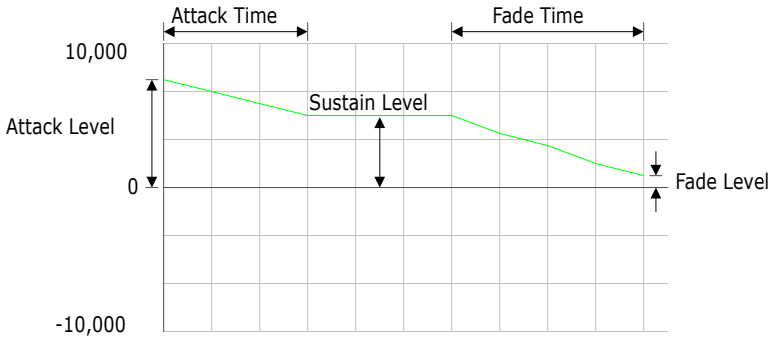
Effects also have *duration*, measured in microseconds. Periodic effects have a *period*, or the duration of one cycle, also measured in microseconds. The *phase* of a periodic effect is the point along the wave where playback begins.

The following diagram 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.

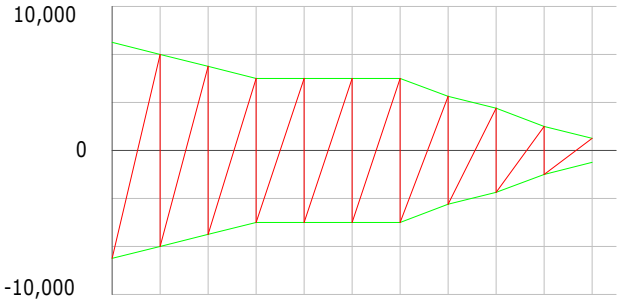


A force may be further shaped by an *envelope*. An envelope defines an *attack value* and a *fade value*, which modify the beginning and ending magnitude of the effect. Attack and fade also have duration, which determines how long the magnitude takes to reach or fall away from the *sustain value*, the magnitude in the middle portion of the effect.

The following diagram represents an envelope. The attack level is set to 8,000 and the fade level to 1,000. The sustain level will be defined by the basic magnitude of the force to which the envelope is being applied; in the example it is 5,000. Note that 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 lesser than the sustain level.



The next diagram shows the result of the envelope being applied to the periodic effect in the first diagram. Note that 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 will be 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

The **IDirectInputDevice2::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 may 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 **IDirectInputDevice2::EnumEffects** method requires a callback function; this is documented with the placeholder name **DIEnumEffectsProc**, but you can use a different name if you wish. 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.

Here is a skeletal C++ example of the callback function, and the call to the **IDirectInputDevice2::EnumEffects** method that sets the enumeration in motion. Note that 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 DIEnumEffectsProc(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 into motion
hr = lpdid2->EnumEffects(&EnumEffectsProc,
                    lpdid2, DIEFT_ALL);
```

For more information on how to initialize an effect, see Creating an Effect.

Information About a Supported Effect

The **IDirectInputDevice2::GetEffectInfo** method can be used to retrieve information about the device's support for an effect. It retrieves the same information that is returned in the **DIEFFECTINFO** structure during enumeration. For more information, see Effect Enumeration.

The following C++ example fetches information about an enumerated effect whose GUID is stored in the *EffectGuid* variable:

```
DIEFFECTINFO diEffectInfo;
diEffectInfo.dwSize = sizeof(DIEFFECTINFO);
lpdid2->GetEffectInfo(&diEffectInfo, EffectGuid);
```

Creating an Effect

You create an effect object by using the **IDirectInputDevice2::CreateEffect** method, as in the following C++ example, where *pdev2* points to an instance of the interface. This example creates a very simple effect that will pull 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. Note that if you use a predefined GUID, the call will fail if the device doesn't 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 will be 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 will be identified by the **dwType** member of the **DIDEVICEOBJECTINSTANCE** structure returned for the object when it was enumerated with the **IDirectInputDevice::EnumObjects** method.

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

Effect Direction

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 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, spherical,* or *Cartesian* coordinates.

Polar coordinates are expressed as a single angle, in hundredths of degrees 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 in order to counteract it.

Spherical coordinates are also in hundredths of degrees but may contain two or more angles; for each angle, the direction is rotated in the positive direction of the next axis. For a three-dimensional device, the first angle would normally be rotated from the positive x-axis toward the positive y-axis (clockwise from east); 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 angel 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 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 simply 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" (zero 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 *lDirection* must be zero.

In the example under 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 that 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  lDirection[2] = { 18000, 0 };
```

For greater clarity, the assignment could also be expressed this way:

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

The reference for the **DIEFFECT** structure tells us that the first angle is measured in hundredths of degrees from the (1, 0) direction, rotated in the direction of (0, 1); the second angle is measured in hundredths of degrees towards (0, 0, 1). The elements of the vector notation again correspond to elements in the array pointed to by the **rgdwAxes** member. Suppose 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 *lDirection* is 0, there is no rotation on the third axis.

How do you accomplish the same thing with Cartesian coordinates? Presuming you have used the DIEFF_CARTESIAN flag in the **dwFlags** member, you would specify the direction like this:

```
LONG    lDirection[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 zero 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 sample code, see Examples of Setting Effect Direction.

Examples of Setting Effect Direction

Single-Axis Effects

Setting up the direction for a single-axis effect is extremely simple, because there is really nothing to specify. You 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 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. You set the DIEFF_POLAR flag in **dwFlags** and set **rglDirection** to point to an array of two **LONG**s. The first element in this array is the direction you want the effect to come from. The second element in the array must be zero.

The following 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, i.e. 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 bit trickier, but not by much. You set the DIEFF_CARTESIAN flag in **dwFlags** and again set **rglDirection** to point to an array of two **LONG**s. 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 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

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, you call the **IDirectInputDevice2::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 **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.

Effect Playback

There are two principal ways to start playback of an effect: manually by a call to the **IDirectInputEffect::Start** method, and 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 **IDirectInputDevice2::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. You should always check the return value from **Start** to make sure the effect played successfully.

To associate an effect with a button press, you set the **dwTriggerButton** member of the **DIEFFECT** structure. You 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. 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 will be the one triggered.

To dissociate an effect from its trigger button, you must 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. In order to make them active again, you must 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 will stop 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.

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 will place the parameters of the effect in hardware memory in order 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 will not have to communicate with the system at all in order 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. 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 stated above, this will be done automatically when you call the **IDirectInputEffect::Start** method, but you may 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 will not normally be called for such effects.

If your application gets the DIERR_DEVICEFULL error when downloading an effect, you have to 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 **IDirectInputDevice2::SendForceFeedbackCommand**.

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

Changing an Effect

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, then DirectInput will normally stop the effect, update the parameters, and restart the effect. You can override this default behavior by passing the DIEP_NORESTART flag.

The following C++ 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.lMagnitude = 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.

Gain

You may want to scale the force of your effects according to the actual 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, then you may 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.)

The actual 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 **IDirectInputDevice::SetProperty** method. You can also set the gain for individual effects when creating or modifying them. Put the new gain value in the **dwGain** member of the **DIEFFECT** structure. If modifying the effect with the **SetProperty** method, be sure to include DIEP_GAIN in the flags parameter, unless you are using DIEP_ALLPARAMS, which includes DIEP_GAIN.

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. By setting the device gain, your application won't need to adjust the gain of each individual effect to suit the user's preferences.

A gain value may be in the range 0 to 10,000 (or DI_FFNOMINALMAX), 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

The **IDirectInputDevice2::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 **IDirectInputDevice2::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

Whenever you need to examine or manipulate all the effects you have created, you can use the **IDirectInputDevice2::EnumCreatedEffectObjects** method. As no flags are currently defined for this method, you cannot restrict the enumeration to particular kinds of effects; all effects will be 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 placeholdner name **DIEnumCreatedEffectObjectsProc**, but you can use a different name. The function is called for each effect enumerated. Within the function you can perform any processing you want; however, it is not safe to create a new effect while enumeration is going on.

Here is a skeletal C++ example of the callback function, and the call to the **EnumCreatedEffectObjects** method. Note that 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 DIEnumCreatedEffectObjectsProc(
                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 into motion
hr = lpdid->EnumCreatedEffectObjects(&EnumCreatedEffectObjectsProc,
                                     &lpdid, 0);
```

Constant Forces

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

You can apply an envelope to a constant force in order 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.

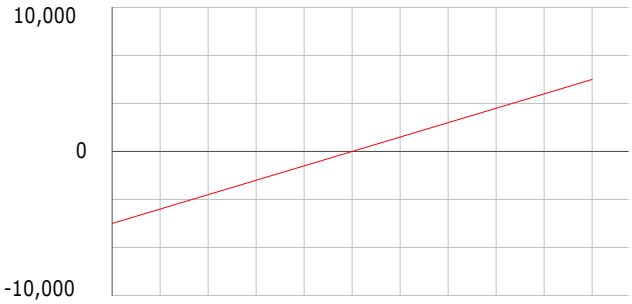
To create a constant force, pass GUID_ConstantForce to the **IDirectInputDevice2::CreateEffect** method. You can also pass any other GUID obtained by the **IDirectInputDevice2::EnumEffects** method, provided the low byte of the **dwEffType** member of the **DIEFFECTINFO** structure (**DIEFT_GETTYPE**(**dwEfftype**)) 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 of the force, while the duration is taken from the **DIEFFECT** structure. An envelope may be applied.

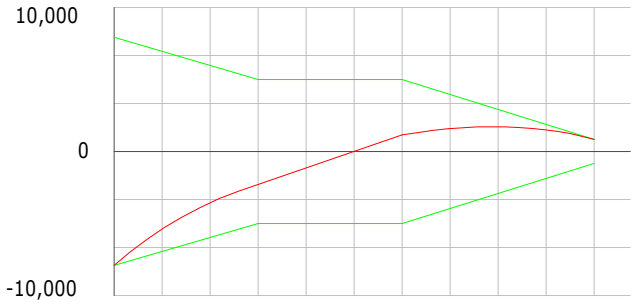
Ramp Forces

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

The following diagram 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 in order 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.



Note that 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 **IDirectInputDevice2::CreateEffect** method. You can also pass any other GUID obtained by the **IDirectInputDevice2::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, while the duration is taken from the **DIEFFECT** structure. Duration must never be set to INFINITE.

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 at 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, where 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 may round off a phase value to the nearest supported value. For example, for a sine effect some drivers support only values of 0 and 9,000 (to create a cosine); for other effects, only values of 0 and 18,000 are supported.

To create a periodic force, pass one of the following values in the *rguid* parameter of the **IDirectInputDevice2::CreateEffect** method:

- GUID_Square
- GUID_Sine
- GUID_Triangle
- GUID_SawtoothUp
- GUID_SawtoothDown

You can also pass any other GUID obtained by the **IDirectInputDevice2::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**. For more information on the **dwPhase** member of this structure, see the overview of Periodic Effects.

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. To determine whether a particular effect supports an envelope, check for the DIEP_ENVELOPE flag in the **dwStaticParams** member of the **DIEFFECTINFO** structure.

Conditions have the following type-specific parameters:

Offset.
The offset from the zero reading of the appropriate sensor value where the condition begins to be applied. For a spring effect, the neutral point—that is, the point along the axis where 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 zero. 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.
The saturation point is the point on an axis at which a force is considered to be at its maximum or minimum. Increases or decreases in magnitude beyond this point are ignored. For example, suppose a flight stick has a spring condition on the x-axis. The formula for the condition causes the force to keep increasing with the distance of the axis from the neutral point, but on some devices you could set the positive and negative saturation points so that the full force of the effect is reached when the stick has been moved halfway to the right or left, and does not increase beyond that point.

Deadband.
The deadband is a zone in an axis where a condition does not apply. Taking the spring example again, the deadband would define a zone within which the axis was considered to be at the neutral point, and no force would be applied.

Conditions can have duration, though in most cases you would probably want to set the duration to INFINITE and stop the effect only in response to some event in the application.

To create a condition, pass one of the following values in the *rguid* parameter of the **IDirectInputDevice2::CreateEffect** method:

- GUID_Spring
- GUID_Damper
- GUID_Inertia
- GUID_Friction

You can also pass any other GUID obtained by the **IDirectInputDevice2::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 may 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 need to set the **cbTypeSpecificParams** member of the **DIEFFECT** structure to the actual 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 **IDirectInputDevice2::GetEffectInfo** or **IDirectInputDevice2::EnumEffects** and examine the **dwEffectType** member of the **DIEFFECTINFO** structure in order to determine if the both a positive and negative coefficient and saturation for the effect are supported on the device. If the effect does not return the DIEFT_POSNEGCOEFFICIENTS flag, it will ignore the value in the **lNegativeCoefficient** member and the value in **lPositiveCoefficient** will be applied to the entire axis. Likewise, if the effect does not return the DIEFT_POSNEGSATURATION flag, it will ignore the value in the **dwNegativeSaturation** and the value in **dwPositiveSaturation** will be used as the negative saturation level. Finally, if the effect does not return the DIEFT_SATURATION flag, it will ignore both the **lPositiveSaturation** and **lNegativeSaturation** values and no saturation will be applied.

Note that 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 **dwEffectType** member for the DIEFT_DEADBAND flag, to see if deadband is supported for the condition. If it is not supported, the value in the **lDeadBand** member of the **DICONDITION** structure will be ignored.

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.

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

You should 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 will be played.

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

Device-Specific Effects

DirectInput provides a way to control device-specific effects. This is useful for hardware vendors who have extra effects that are not directly supported by DirectInput.

The hardware vendor must provide a GUID identifying the device-specific effect and may 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, just as with any other effect. You then call the **IDirectInputDevice2::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 which of the predefined DirectInput effect categories (constant force, ramp force, periodic, or condition) the effect falls into. If it does not fall into any of the predefined categories, then the value is DIEFT_HARDWARE.

If a device-specific effect falls into one of the predefined categories, then 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), then the values of the **lpvTypeSpecificParams** and **cbTypeSpecificParams** members depend on whether the effect requires custom type-specific parameters. If it does, then 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 **dwStaticParms** member of the **DIEFFECTINFO** structure for the hardware effect does not have the DIEP_TYPESPECIFICPARAMS flag – then **lpvTypeSpecificParams** must be NULL and **cbTypeSpecificParams** must be zero.

DirectInput passes the GUID and the **LPDIEFFECT** to the device driver for verification. If the GUID is unknown, the device will return

DIERR_DEVICENOTREG. If the GUID is known but the type-specific data is incorrect for that effect, the device will return DIERR_INVALIDPARAM.

Designing for Previous Versions of DirectInput

In several places, DirectInput requires you to pass a version number to a method. This parameter specifies which version of DirectInput 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, then the header file will generate structure definitions compatible with DirectInput 3.0.

If you do not define DIRECTINPUT_VERSION before including the Dinput.h header file, then the header file will generate structure definitions compatible with the current version of DirectInput. However, the DirectX 3–compatible structures will be available under the same names with "_DX3" appended. For example, the DirectX 3–compatible **DIDEVCAPS** structure is called **DIDEVCAPS_DX3**.

DirectInput Tutorials

This section contains four tutorials, each providing step-by-step instructions for implementing basic DirectInput functionality.

- 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 sample application Scrawl, included in the DirectX code samples under the Platform SDK References, 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 from the

Space Donuts application, included in the DirectX code samples in the Platform SDK.

- Tutorial 4: Using Force Feedback

  The final tutorial illustrates the creation and manipulation of a simple effect on a force feedback joystick.

  Additional sample code can be found in the Samples folder of the DirectX SDK. The Diex1 through Diex4 programs are simple examples of how to read mouse and keyboard data. The Diff1 program shows how to use force feedback in a very simple way.

Tutorial 1: Using the Keyboard

To prepare for keyboard input, you first create an instance of a DirectInput object. Then you use the **IDirectInput::CreateDevice** method to create an instance of an **IDirectInputDevice** interface. The **IDirectInputDevice** 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: Creating the DirectInput Object
- Step 2: Creating the DirectInput Keyboard Device
- Step 3: Setting the Keyboard Data Format
- Step 4: Setting the Keyboard Behavior
- Step 5: Gaining Access to the Keyboard
- Step 6: Retrieving Data from the Keyboard
- Step 7: Closing 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: Creating the DirectInput Object

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 **DirectInputCreate** function.

```
// HINSTANCE  hinst; // initialized earlier
HRESULT       hr;
LPDIRECTINPUT  g_lpdi;

hr = DirectInputCreate(hinst, DIRECTINPUT_VERSION, &g_lpdi, NULL);
if FAILED(hr)
```

```
{
  // DirectInput not available; take appropriate action
}
```

The first parameter for **DirectInputCreate** 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 is the address of a variable that will be initialized with a valid **IDirectInput** 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.

Step 2: Creating the DirectInput Keyboard Device

After creating the DirectInput object, your application must create the keyboard object–the device–and retrieve a pointer to an **IDirectInputDevice** 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 **IDirectInput::CreateDevice** method, as shown in Sample Function 1: DI_Init. **CreateDevice** accepts three 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* value.

The second parameter is the address of a variable that will be initialized with a valid **IDirectInputDevice** interface pointer if the call succeeds.

The third 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 **IDirectInputDevice** 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;
LPDIRECTINPUTDEVICE  g_lpDIDEVICE
```

```
hr = g_lpDI->CreateDevice(GUID_SysKeyboard, &g_lpDIDevice, NULL);
if FAILED(hr)
{
   DI_Term();
   return FALSE;
}
```

Step 3: Setting the Keyboard Data Format

After retrieving an **IDirectInputDevice** 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 **IDirectInputDevice::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;
}
```

Step 4: Setting the Keyboard Behavior

Before your application can gain access to the keyboard, it must set the device's behavior using the **IDirectInputDevice::SetCooperativeLevel** method, as shown in Sample Function 1: DI_Init. This method accepts the handle to the window to be associated with the device. DirectInput does not support exclusive access to keyboard devices, so the DISCL_NONEXCLUSIVE flag must be included in the *dwFlags* parameter.

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;
}
```

Step 5: Gaining Access to the Keyboard

After your application sets the keyboard's behavior, it can acquire access to the device by calling the **IDirectInputDevice::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: Creating the DirectInput Keyboard Device:

```
if (g_lpDIDevice) g_lpDIDevice->Acquire();
```

Step 6: Retrieving Data from the Keyboard

Once a device is acquired, your application can start retrieving data from it. The simplest way to do this is to call the **IDirectInputDevice::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 **IDirectInputDevice::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 == DI_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.
}
```

Step 7: Closing Down the DirectInput System

When an application is about to close, it should destroy all DirectInput objects. This is a three-step process:

- Unacquire all DirectInput devices (**IDirectInputDevice::Unacquire**)
- Release all DirectInput devices (**IDirectInputDevice::Release**)
- Release the DirectInput object (**IDirectInput::Release**)

For a sample function that closes down the DirectInput system, see Sample Function 2: DI_Term.

Sample Function 1: DI_Init

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: Creating the DirectInput Keyboard Device

- Step 3: Setting the Keyboard Data Format
- Step 4: Setting the Keyboard Behavior
- Step 5: Gaining Access to the Keyboard

Here is the **DI_Init** function:

```
// HINSTANCE           g_hinst;    //initialized application instance
// HWND                g_hwndMain; //initialized application window
LPDIRECTINPUT       g_lpDI;
LPDIRECTINPUTDEVICE  g_lpDIDevice;

BOOL WINAPI DI_Init()
{
   HRESULT hr;

   // Create the DirectInput object.
   hr = DirectInputCreate(g_hinst, DIRECTINPUT_VERSION,
                     &g_lpDI, NULL);
   if FAILED(hr) return FALSE;

   // Retrieve a pointer to an IDirectInputDevice interface
   hr = g_lpDI->CreateDevice(GUID_SysKeyboard, &g_lpDIDevice, NULL);
   if FAILED(hr)
   {
      DI_Term();
      return FALSE;
   }

// Now that you have an IDirectInputDevice 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

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.

```
// LPDIRECTINPUT       g_lpDI;
// LPDIRECTINPUTDEVICE  g_lpDIDevice;

void WINAPI DI_Term()
{
    if (g_lpDI)
    {
        if (g_lpDIDevice)
        {
            /*
             *  Always unacquire the device before calling Release().
             */
            g_lpDIDevice->Unacquire();
            g_lpDIDevice->Release();
            g_lpDIDevice = NULL;
        }
        g_lpDI->Release();
        g_lpDI = NULL;
    }
}
```

Tutorial 2: Using the Mouse

This tutorial guides you through the process of setting up a mouse device and retrieving buffered input data. The examples are taken from the Scrawl sample application included in the SDK\SAMPLES directory of the DirectX SDK.

To prepare for mouse input, you first create an instance of a DirectInput object. Then you use the **IDirectInput::CreateDevice** method to create an instance of an **IDirectInputDevice** interface. The **IDirectInputDevice** 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: Creating the DirectInput Mouse Device
- Step 2: Setting the Mouse Data Format
- Step 3: Setting the Mouse Behavior
- Step 4: Preparing for Buffered Input from the Mouse
- Step 5: Managing Access to the Mouse
- Step 6: Retrieving 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: Creating the DirectInput Mouse Device

After creating the DirectInput object, your application should retrieve a pointer to an **IDirectInputDevice** interface, which will be used to perform most mouse-related tasks. To do this, call the **IDirectInput::CreateDevice** method.

The **CreateDevice** method accepts three parameters.

The first parameter 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 address of a variable that will be initialized with a valid **IDirectInputDevice** interface pointer if the call succeeds.

The third 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 from Scrawl.cpp attempts to retrieve a pointer to an **IDirectInputDevice** interface. If the call fails, an error message is displayed and FALSE is returned.

```
// LPDIRECTINPUT    g_pdi;    // This has been initialized
LPDIRECTINPUTDEVICE g_pMouse;
HRESULT             hr;

hr = g_pdi->CreateDevice(GUID_SysMouse, &g_pMouse, NULL);

if (FAILED(hr)) {
   Complain(hwnd, hr, "CreateDevice(SysMouse)");
   return FALSE;
}
```

Step 2: Setting the Mouse Data Format

After retrieving an **IDirectInputDevice** pointer, your application must set the device's data format. For mouse devices, this is a very simple task. Call the **IDirectInputDevice::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, an error message is displayed and FALSE is returned.

```
hr = g_pMouse->SetDataFormat(&c_dfDIMouse);

if (FAILED(hr)) {
   Complain(hwnd, hr, "SetDataFormat(SysMouse, dfDIMouse)");
   return FALSE;
}
```

Step 3: Setting the Mouse Behavior

Before it can gain access to the mouse, your application must set the mouse device's behavior using the **IDirectInputDevice::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 from Scrawl.cpp attempts to set the device's cooperative level. If this attempt fails, an error message is displayed and FALSE is returned.

```
hr = g_pMouse->SetCooperativeLevel(hwnd,
            DISCL_EXCLUSIVE | DISCL_FOREGROUND);
```

```
if (FAILED(hr)) {
    Complain(hwnd, hr, "SetCooperativeLevel(SysMouse)");
    return FALSE;
}
```

Step 4: Preparing for Buffered Input from the Mouse

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. As usual, the **Complain** function informs the user of any errors.

```
// HANDLE  g_hevtMouse;   // This is global

g_hevtMouse = CreateEvent(0, 0, 0, 0);

if (g_hevtMouse == NULL) {
    Complain(hwnd, GetLastError(), "CreateEvent");
    return FALSE;
}

hr = g_pMouse->SetEventNotification(g_hevtMouse);

if (FAILED(hr)) {
    Complain(hwnd, hr, "SetEventNotification(SysMouse)");
    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 used 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 **IDirectInputDevice::SetProperty** method, as follows:

```
hr = g_pMouse->SetProperty(DIPROP_BUFFERSIZE, &dipdw.diph);

if (FAILED(hr)) {
   Complain(hwnd, hr, "Set buffer size(SysMouse)");
   return FALSE;
}
```

The setup is now complete, and you're ready to acquire the mouse and start collecting data.

Step 5: Managing Access to the Mouse

DirectInput provides the **IDirectInputDevice::Acquire** and **IDirectInputDevice::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 **IDirectInputDevice::GetDeviceState** and **IDirectInputDevice::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.

Step 6: Retrieving Buffered Data from the Mouse

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 **IDirectInputDevice::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: Managing 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** 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

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 application in the SDK\SAMPLES directory of the DirectX SDK. 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 **IDirectInputDevice** interface methods to retrieve data from each joystick.

The tutorial breaks down the required tasks into the following steps:

- Step 1: Enumerating the Joysticks
- Step 2: Creating the DirectInput Joystick Device
- Step 3: Setting the Joystick Data Format
- Step 4: Setting the Joystick Behavior
- Step 5: Gaining Access to the Joystick
- Step 6: Retrieving Data from the Joystick

Step 1: Enumerating the Joysticks

After creating the DirectInput system, call the **IDirectInput::EnumDevices** method to enumerate the joysticks. The following code from Input.c in the Space Donuts source directory accomplishes this.

```
// LPDIRECTINPUT  pdi;  // previously initialized

pdi->lpVtbl->EnumDevices(pdi, DIDEVTYPE_JOYSTICK,
                InitJoystickInput, pdi, DIEDFL_ATTACHEDONLY);
```

The method call is in the C form. Note that you could use the **IDirectInput_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 **IDirectInput::CreateDevice**.

The last parameter, DIEDFL_ATTACHEDONLY, is a flag that restricts enumeration to devices that are attached to the computer.

Step 2: Creating the DirectInput Joystick Device

After creating the DirectInput object, the application must retrieve a pointer to an **IDirectInputDevice** 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)
{
  LPDIRECTINPUT pdi = pvRef;
  LPDIRECTINPUTDEVICE pdev;

  // create the DirectInput joystick device
  if (pdi->lpVtbl->CreateDevice(pdi, &pdinst->guidInstance,
                        &pdev, NULL) != DI_OK)
```

```
{
  OutputDebugString("IDirectInput::CreateDevice FAILED\n");
  return DIENUM_CONTINUE;
}
```

The parameters to **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 **IDirectInput::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 **IDirectInput::CreateDevice** method.

The **InitJoystickInput** 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. This device starts life as an instance of the **IDirectInputDevice** interface, but when it is added to the application's list of input devices it is converted to an **IDirectInputDevice2** object so that it can use the **IDirectInputDevice2::Poll** method.

The first task of the callback function, then, is to create the device. The **IDirectInput::CreateDevice** 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 **IDirectInputDevice** 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.

Step 3: Setting the Joystick Data Format

Now that the application has a pointer to a DirectInput device, it can call the **IDirectInputDevice** 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("IDirectInputDevice::SetDataFormat FAILED\n");
  pdev->lpVtbl->Release(pdev);
  return DIENUM_CONTINUE;
}
```

The *pdev* variable is a pointer to the device interface created by **IDirectInput::CreateDevice**.

The **IDirectInputDevice::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.

Step 4: Setting the Joystick Behavior

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("IDirectInputDevice::SetCooperativeLevel
            FAILED\n");
  pdev->lpVtbl->Release(pdev);
  return DIENUM_CONTINUE;
}
```

Once again, the first parameter to **IDirectInputDevice::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(pdev->lpVtbl->SetProperty(pdev, DIPROP_RANGE, &diprg.diph)
                        != DI_OK)
{
  OutputDebugString("IDirectInputDevice::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 **IDirectInputDevice::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. See Device Properties for more information.

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** 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 **IDirectInputDevice::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 **DIPROPRANGE**, 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 (SetDIDwordProperty(pdev, DIPROP_DEADZONE, DIJOFS_X,
                    DIPH_BYOFFSET, 5000) != DI_OK)
{
  OutputDebugString("IDirectInputDevice::
                     SetProperty(DIPH_DEADZONE) FAILED\n");
  pdev->lpVtbl->Release(pdev);
  return FALSE;
}
```

Step 5: Gaining Access to the Joystick

After your application sets a joystick's behavior, it can acquire access to the device by calling the **IDirectInputDevice::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 = IDirectInputDevice_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 **IDirectInputDevice_Acquire**, a macro defined in Dinput.h that simplifies the C call to the **IDirectInputDevice::Acquire** method.

Step 6: Retrieving Data from the Joystick

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 **IDirectInputDevice::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 **IDirectInputDevice2::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 = IDirectInputDevice2_Poll(g_pdevCurrent);

  // get data from the joystick
  hRes = IDirectInputDevice_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 GetDeviceData 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.lX < 0) {
    dwKeyState |= KEY_LEFT;
  } else if (js.lX > 0) {
    dwKeyState |= KEY_RIGHT;
  }

  if (js.lY < 0) {
    dwKeyState |= KEY_UP;
  } else if (js.lY > 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 **IDirectInputDevice2_Poll** and **IDirectInputDevice_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 = IDirectInputDevice_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 non-zero, 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

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: Enumerating Force Feedback Devices
- Step 2: Creating the DirectInput Force Feedback Device
- Step 3: Enumerating Supported Effects
- Step 4: Creating an Effect
- Step 5: Playing an Effect
- Step 6: Changing an Effect

Step 1: Enumerating Force Feedback Devices

The first step is to ensure that a force feedback device is available on the system. You do this by calling the **IDirectInput::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:

```
LPDIRECTINPUTDEVICE2  g_lpdid2Game = NULL;

lpdi->EnumDevices(DIDEVTYPE_JOYSTICK,
            DIEnumDevicesProc,
            NULL,
            DIEDFL_FORCEFEEDBACK | DIEDFL_ATTACHEDONLY);
if (g_lpdid2Game == NULL)
 {
 // no force feedback joystick available; take appropriate action
 }
```

In the example, *lpdi* is an initialized pointer to the **IDirectInput** 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 in Step 2: Creating the DirectInput Force Feedback Device.

Step 2: Creating the DirectInput Force Feedback Device

In order to have DirectInput enumerate devices, you must create a callback function of the same type as **DIEnumDevicesProc**. In Step 1 you passed the address of this function to the **IDirectInput::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 **IDirectInput::CreateDevice** 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 hr1, hr2;
LPDIRECTINPUTDEVICE lpdidGame;
GUID DeviceGuid = lpddi->guidInstance;

// create game device

hr1 = lpdi->CreateDevice(DeviceGuid, &lpdidGame, NULL);
```

Note that the pointer to the **IDirectInputDevice** object, *lpdidGame*, is a local variable. You're not going to keep it, because in order to create force feedback effects you need to obtain a pointer to the **IDirectInputDevice2** interface, as follows:

```
if (SUCCEEDED(hr1))
 {
 hr2 = lpdidGame->QueryInterface(IID_IDirectInputDevice2,
                        (void **) &g_lpdid2Game);
 lpdidGame->Release();
 }
else
 {
 OutputDebugString("Failed to create device.\n");
 return DIENUM_STOP;
 }
if (FAILED(hr2))
 {
 OutputDebugString("Failed to obtain interface.\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. Since the joystick will be used for input as well as force feedback, you also need to set the data format.

```
// set cooperative level
 if (FAILED(g_lpdid2Game->SetCooperativeLevel(hMainWindow,
      DISCL_EXCLUSIVE | DISCL_FOREGROUND)))
 {
 OutputDebugString(
   "Failed to set cooperative level.\n");
 lpdid2Game->Release();
 lpdi2Game = NULL;
 return DIENUM_STOP;
 }

 // set game data format
```

```
if (FAILED(g_lpdid2Game->SetDataFormat(&c_dfDIJoystick)))
 {
 OutputDebugString("Failed to set game device data format.\n");
 lpdid2Game->Release();
 lpdid2Game = 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. Turning it off gives you more control over the device.

```
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(lpdid2Game->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: Gaining Access to the Joystick in the previous tutorial for an example of how to handle acquisition.

Step 3: Enumerating Supported Effects

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 **IDirectInputDevice2::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 application in the DirectX code samples in the Platform SDK, 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 **DIEnumEffectsProc**. 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 **IDirectInputDevice2::EnumEffects** method, as follows:

```
HRESULT  hr;
GUID     guidEffect;

hr = g_lpdid2Game->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.

Step 4: Creating an Effect

If the *EffectFound* flag 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 **IDirectInputDevice2::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[0];
diEffect.rglDirection = &lDirection[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_lpdid2Game->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.

Step 5: Playing an Effect

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.

Step 6: Changing an Effect

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 Reference

This section contains reference information for the API elements that DirectInput provides. Reference material 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:

- **IDirectInput**
- **IDirectInputDevice**
- **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 Dinput.h and are not documented separately.

IDirectInput

Applications use the methods of the **IDirectInput** 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 IDirectInput interface is obtained by using the **DirectInputCreate** function.

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

Device Management	**CreateDevice**
	EnumDevices
	GetDeviceStatus
Miscellaneous	**Initialize**
	RunControlPanel

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

AddRef

QueryInterface

Release

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

```
typedef struct IDirectInput    *LPDIRECTINPUT;
```

IDirectInput::CreateDevice

The **IDirectInput::CreateDevice** method creates and initializes an instance of a device based on a given GUID.

```
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 **IDirectInput::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 above 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 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 will pass NULL.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may 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 have pass-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 function with *punkOuter* = NULL is equivalent to creating the object via **CoCreateInstance**(*&CLSID_DirectInputDevice, NULL, CLSCTX_INPROC_SERVER, riid, lplpDirectInputDevice*) and then initializing it with **Initialize**.

Calling this function with *punkOuter* != NULL is equivalent to creating the object via **CoCreateInstance**(*&CLSID_DirectInputDevice, punkOuter, CLSCTX_INPROC_SERVER, &IID_IUnknown, lplpDirectInputDevice*). The aggregated object must be initialized manually.

IDirectInput::EnumDevices

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

```
HRESULT EnumDevices(
  DWORD dwDevType,
  LPDIENUMCALLBACK lpCallback,
  LPVOID pvRef,
  DWORD dwFlags
);
```

Parameters

dwDevType
: Device type filter. If this parameter is zero, 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 that will be called with a description of each DirectInput device.

pvRef

An application-defined 32-bit value that will 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 will be enumerated. This is the default behavior.

DIEDFL_ATTACHEDONLY

Only attached and installed devices.

DIEDFL_FORCEFEEDBACK

Only devices that support force feedback

Return Values

If the method succeeds, the return value is DI_OK.

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

DIERR_INVALIDPARAM

DIERR_NOTINITIALIZED

Remarks

Keep in mind that all installed devices can be enumerated, even if they are not present. For example, a flight stick may 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 will be 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 **DIEnumDevicesProc**. 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 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.

IDirectInput::GetDeviceStatus

The **IDirectInput::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 DirectInput, or DI_NOTATTACHED otherwise.

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

DIERR_GENERIC

DIERR_INVALIDPARAM

DIERR_NOTINITIALIZED

IDirectInput::Initialize

The **IDirectInput::Initialize** method initializes a DirectInput object. The **DirectInputCreate** function automatically initializes the DirectInput object device 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 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 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 may be necessary.

dwVersion

Version number of DirectInput for which the application is designed. This value will normally be DIRECTINPUT_VERSION. Passing the version number of a previous version will cause 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 may be one of the following error values:

DIERR_BETADIRECTINPUTVERSION

DIERR_OLDDIRECTINPUTVERSION

IDirectInput::RunControlPanel

The **IDirectInput::RunControlPanel** method runs the Windows Control Panel to allow the user to install a new input device or modify configurations.

```
HRESULT RunControlPanel(
  HWND hwndOwner,
  DWORD dwFlags
);
```

Parameters

hwndOwner

Handle to 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

This parameter is currently not used and must be set to zero.

Return Values

If the method succeeds, the return value is DI_OK.

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

DIERR_INVALIDPARAM

DIERR_NOTINITIALIZED

See Also

IDirectInputDevice::RunControlPanel

IDirectInputDevice

Applications use the methods of the **IDirectInputDevice** interface to gain and release access to DirectInput devices, manage device properties and information, set behavior, perform initialization, and invoke a device's control panel. The **IDirectInputDevice2** interface adds force feedback capabilities and support for polled devices.

The **IDirectInputDevice** interface is obtained by using the **IDirectInput::CreateDevice** method. The **IDirectInputDevice2** interface is obtained by calling the **IDirectInputDevice::QueryInterface** method; for an example, see Creating a DirectInput Device.

This section is a reference to the methods of these interfaces.

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

Accessing input devices	**Acquire**
	Unacquire
Device information	**GetCapabilities**
	GetDeviceData
	GetDeviceInfo
	GetDeviceState
	SetDataFormat
	SetEventNotification
Device objects	**EnumObjects**
	GetObjectInfo
Device properties	**GetProperty**
	SetProperty
Setting behavior	**SetCooperativeLevel**
Miscellaneous	**Initialize**
	RunControlPanel

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

AddRef

QueryInterface

Release

The **IDirectInputDevice2** interface supports all the above methods as well as the following additional methods:

CreateEffect

EnumCreatedEffectObjects

EnumEffects

Escape

GetEffectInfo

GetForceFeedbackState

Poll

SendForceFeedbackCommand

The **LPDIRECTINPUTDEVICE** and **LPDIRECTINPUTDEVICE2** types are defined as pointers to the **IDirectInput** interface:

```
typedef struct IDirectInputDevice    *LPDIRECTINPUTDEVICE;
typedef struct IDirectInputDevice2   *LPDIRECTINPUTDEVICE2;
```

IDirectInputDevice::Acquire

The **IDirectInputDevice::Acquire** method obtains access to the input device.

```
HRESULT Acquire();
```

Return Values

If the method succeeds, the return value is DI_OK or S_FALSE.

If the method fails, the return value may 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 **IDirectInputDevice::SetDataFormat** method.

Devices must be acquired before calling the **IDirectInputDevice::GetDeviceState** or **IDirectInputDevice::GetDeviceData** methods for that device.

Device acquisition does not use a reference count. Therefore, if an application calls the **IDirectInputDevice::Acquire** method twice, then calls the **IDirectInputDevice::Unacquire** method once, the device is unacquired.

IDirectInputDevice::EnumObjects

The **IDirectInputDevice::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 **DIEnumDeviceObjectsProc**.

pvRef

Reference data (context) for callback.

dwFlags

Flags specifying the types of object to be enumerated. The value may be one or more of the following:

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_FFACTUATOR

An object that contains a force feedback actuator. In other words, forces may be applied to this object.

DIDFT_FFEFFECTTRIGGER

An object that can be used to trigger force feedback effects.

DIDFT_NODATA

An object that does not generate data. Although no data can be read from it, the object can be used as an output actuator in a force feedback effect (if the DIDFT_FFACTUATOR flag is set).

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.

Return Values

If the method succeeds, the return value is DI_OK.

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

DIERR_INVALIDPARAM

DIERR_NOTINITIALIZED

Remarks

The DIDFT_FFACTUATOR and DIDFT_FFEFFECTTRIGGER flags in the **dwFlags** member 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_FFEFFECTRIGGER | DIDFT_TGLBUTTON | DIDFT_PSHBUTTON) restricts enumeration to buttons of any kind that can be used as effect triggers.

IDirectInputDevice::GetCapabilities

The **IDirectInputDevice::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 may 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.

IDirectInputDevice::GetDeviceData

The **IDirectInputDevice::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, then 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 may be zero or the following flag.

DIGDD_PEEK
Do not remove the items from the buffer. A subsequent **IDirectInputDevice::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 may 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 **IDirectInputDevice::SetDataFormat** method, set the buffer size with **IDirectInputDevice::SetProperty** method, and acquire the device by using the **IDirectInputDevice::Acquire** method.

The following 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 = IDirectInputDevice_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 example illustrates how this can be done:

```
dwItems = INFINITE;
hres = IDirectInputDevice_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 fragment illustrates how this can be done:

```
dwItems = INFINITE;
hres = IDirectInputDevice_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
        // were successfully captured.
    }
}
```

To query about whether a buffer overflow has occurred, set the *rgdod* parameter to NULL and the *pdwInOut* parameter to zero. The following example illustrates how this can be done:

```
dwItems = 0;
hres = IDirectInputDevice_GetDeviceData(
            pdid,
            sizeof(DIDEVICEOBJECTDATA),
            NULL,
            &dwItems,
            0);
if (hres == DI_BUFFEROVERFLOW) {
    // Buffer overflow occurred
}
```

See Also

IDirectInputDevice2::Poll

Polling and Events

IDirectInputDevice::GetDeviceInfo

The **IDirectInputDevice::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 may 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.

IDirectInputDevice::GetDeviceState

The **IDirectInputDevice::GetDeviceState** method retrieves instantaneous 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 **IDirectInputDevice::SetDataFormat** method.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may 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 **IDirectInputDevice::SetCooperativeLevel** method, then set the data format by using **IDirectInputDevice::SetDataFormat**, and acquire the device by using the **IDirectInputDevice::Acquire** method.

The four predefined data formats require corresponding device state structures according to the following table:

Data Format	**State Structure**
c_dfDIMouse	**DIMOUSESTATE**
c_dfDIKeyboard	array of 256 bytes
c_dfDIJoystick	**DIJOYSTATE**
c_dfDIJoystick2	**DIJOYSTATE2**

For example, if you passed the *c_dfDIMouse* format to the **IDirectInputDevice::SetDataFormat** method, you must pass a **DIMOUSESTATE** structure to the **IDirectInputDevice::GetDeviceState** method.

See Also

IDirectInputDevice2::Poll

Polling and Events

IDirectInputDevice::GetObjectInfo

The **IDirectInputDevice::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 will 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_DEVICE	The *dwObj* parameter must be zero.
DIPH_BYOFFSET	The *dwObj* parameter is the offset into the current data format of the object whose information is being accessed.
DIPH_BYID	The *dwObj* parameter is the object type/instance identifier. This identifier is returned in the **dwType** member of the **DIDEVICEOBJECTINSTANCE** structure returned from a previous call to the **IDirectInputDevice::EnumObjects** method.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may 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 DirectInput.

IDirectInputDevice::GetProperty

The **IDirectInputDevice::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 **IDirectInputDevice::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 **IDirectInputDevice::GetDeviceData** method before data is lost. This value may be set to zero to indicate that the application will 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 you previously attempted to set.

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 **IDirectInputDevice::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 0 to 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 for the *pdiph* parameter for more information.

Granularity represents the smallest distance the object will report movement. Most axis objects have a granularity of one, 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 zero, then 20, then 40, and so on.

This is a read-only property; you cannot set its value by calling the **IDirectInputDevice::SetProperty** method.

DIPROP_RANGE

Retrieves the range of values an object can possibly report. The retrieved minimum and maximum values are set in the *lMin* and *lMax* members of the associated **DIPROPRANGE** structure. See the description for the *pdiph* parameter for more information.

For some devices, this is a read-only property; you cannot set its value by calling the **IDirectInputDevice::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 deadzone).

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 any other property, 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 may be one of the following error values:

DIERR_INVALIDPARAM
DIERR_NOTINITIALIZED
DIERR_OBJECTNOTFOUND
DIERR_UNSUPPORTED

Remarks

The following C example illustrates 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 = IDirectInputDevice_GetProperty(pdid, DIPROP_BUFFERSIZE, &dipdw.diph);
if (SUCCEEDED(hr)) {
   // The dipdw.dwData member contains the buffer size.
}
```

See Also

IDirectInputDevice::SetProperty

IDirectInputDevice::Initialize

The **IDirectInputDevice::Initialize** method initializes a DirectInputDevice object. The **IDirectInput::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 may 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 may be necessary.

dwVersion

Version number of DirectInput for which the application is designed. This value will normally be DIRECTINPUT_VERSION. Passing the version number of a previous version will cause DirectInput to emulate that version. For more information, see Designing for Previous Versions of DirectInput.

rguid

Identifier for the instance of the device for which the interface should be associated. The **IDirectInput::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 may 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.

IDirectInputDevice::RunControlPanel

The **IDirectInputDevice::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 may be one of the following error values:

DIERR_INVALIDPARAM

DIERR_NOTINITIALIZED

IDirectInputDevice::SetCooperativeLevel

The **IDirectInputDevice::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. This parameter can be one of the following values:

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.

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 will not be able to 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 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.

Return Values

If the method succeeds, the return value is DI_OK.

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

DIERR_INVALIDPARAM

DIERR_NOTINITIALIZED

Remarks

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 **IDirectInputDevice::Acquire** method.

IDirectInputDevice::SetDataFormat

The **IDirectInputDevice::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 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_dfDIJoystick*
- *c_dfDIJoystick2*

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may 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 **IDirectInputDevice::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

IDirectInputDevice::GetDeviceState

IDirectInputDevice::SetEventNotification

The **IDirectInputDevice::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 will use the Win32 **SetEvent** function on the handle when the state of the device changes. If the *hEvent* parameter is NULL, then notification is disabled.

The application may create the handle as either a manual-reset or automatic-reset event by using the Win32 **CreateEvent** function. If the event is created as an automatic-reset event, then the operating system will automatically reset the event once a wait has been satisfied. If the event is created as a manual-reset event, then it is the application's responsibility to call the Win32 **ResetEvent** function to reset it. DirectInput will not call the Win32 **ResetEvent** function for event notification handles. Most applications will create the event as an automatic-reset event.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may 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, then the application can use the event handle in the same manner as any other Win32 event handle.

The following example checks if 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
    // automatic-reset, then it has also been reset.
}
```

The following example illustrates blocking indefinitely until the event is set. Note that this behavior is strongly discouraged because the thread will not respond to the system until the wait is satisfied. In particular, the thread will not respond to Windows messages.

```
dwResult = WaitForSingleObject(hEvent, INFINITE);
if (dwResult == WAIT_OBJECT_0) {
    // Event has been set. If the event was created
    // as automatic-reset, then it has also been reset.
}
```

The following 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
        // automatic-reset, then it has also been reset.
        ProcessInputEvent1();
        break;

    case WAIT_OBJECT_0 + 1:
        // Event 2 has been set. If the event was created as
```

```
      // automatic-reset, then 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 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 automatic-reset, then it has also
      // been reset.
      ProcessInputEvent1();
      break;

   case WAIT_OBJECT_0 + 1:
      // Event 2 has been set. If the event was
      // created as automatic-reset, then 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, then tell the next wait
        // to wait indefinitely for input or a message.
        if (!DoGame()) {
            dwWait = INFINITE;
        }
        break;
    }
}
```

See Also

Polling and Events

IDirectInputDevice::SetProperty

The **IDirectInputDevice::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 may 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.

Note that the use of force feedback effects may 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 for 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 for 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 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_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 0 to 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 wishes 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_RANGE

Sets the range of values an object can possibly report. The minimum and maximum values are taken from the **lMin** and **lMax** members of the associated **DIPROPRANGE** structure.

For some devices, this is a read-only property.

You cannot set a reverse range; **lMax** must be greater than **lMin**.

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 deadzone).

This setting can be applied to either the entire device or to a specific axis.

pdiph

Address of the **DIPROPHEADER** structure contained within the **DIPROPDWORD** structure. If setting object range information, this is the address of the **DIPROPHEADER** structure contained within the **DIPROPRANGE** structure.

Return Values

If the method succeeds, the return value is DI_OK.

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

DI_PROPNOEFFECT

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 **IDirectInputDevice::GetDeviceData** method before data is lost. This value may be set to zero to indicate that the application will 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 you previously attempted to set.

See Also

IDirectInputDevice::GetProperty

IDirectInputDevice::Unacquire

The **IDirectInputDevice::Unacquire** method releases access to the device.

```
HRESULT Unacquire();
```

Return Values

The return value is DI_OK is the device was unacquired, or DI_NOEFFECT if the device was not in an acquired state to begin with.

IDirectInputDevice2::CreateEffect

The **IDirectInputDevice2::CreateEffect** method creates and initializes an instance of an effect identified by the effect GUID.

```
HRESULT IDirectInputDevice2::CreateEffect(
  REFGUID rguid,
  LPCDIEFFECT lpeff,
  LPDIRECTINPUTEFFECT * ppdeff,
  LPUNKNOWN punkOuter
);
```

Parameters

rguid

The identity of the effect to be created. This can be a predefined effect GUID, or it can be a GUID obtained from **IDirectInputDevice2::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

A **DIEFFECT** structure that provides parameters for the created effect. This parameter is optional. If it is NULL, then 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

Location of the pointer to the **IDirectInputEffect** interface, if successful.

punkOuter

The controlling unknown for COM aggregation. The value is NULL if the interface is not aggregated. Most callers will pass NULL.

Return Values

If the method succeeds, the return value is DI_OK or S_FALSE.

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

DIERR_DEVICENOTREG
DIERR_DEVICEFULL
DIERR_INVALIDPARAM
DIERR_NOTINITIALIZED

If the return value is S_FALSE, the effect was created and the parameters of the effect were updated, but the effect could not be downloaded because the associated device is not acquired in exclusive mode.

IDirectInputDevice2::EnumCreatedEffectObjects

The **IDirectInputDevice2::EnumCreatedEffectObjects** method enumerates all of the currently created effects for this device. Effects created by **IDirectInputDevice2::CreateEffect** are enumerated.

```
HRESULT IDirectInputDevice2::EnumCreatedEffectObjects(
  LPDIENUMCREATEDEFFECTOBJECTSCALLBACK lpCallback,
  LPVOID pvRef,
  DWORD fl
);
```

Parameters

lpCallback
Address of an application-defined callback function. DirectInput provides the prototype function **DIEnumCreatedEffectObjectsProc**.

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 may be one of the following error values:

DIERR_INVALIDPARAM

DIERR_NOTINITIALIZED

Remarks

The results will be 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.

IDirectInputDevice2::EnumEffects

The **IDirectinputDevice2::EnumEffects** method enumerates all of the effects supported by the force feedback system on the device. The enumerated GUIDs may represent predefined effects as well as effects peculiar to the device manufacturer.

```
HRESULT IDirectInputDevice2::EnumEffects(
  LPDIENUMEFFECTSCALLBACK lpCallback,
  LPVOID pvRef,
  DWORD dwEffType
);
```

Parameters

lpCallback
Address of an application-defined callback function. DirectInput provides the prototype function **DIEnumEffectsProc**.

pvRef
A 32-bit application-defined value to be passed to the callback function. This parameter may be any 32-bit value; 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 may 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.

In order to exploit an effect to its fullest, you must contact the device manufacturer to obtain information on the semantics of the effect and its effect-specific parameters.

IDirectInputDevice2::Escape

The **IDirectInputDevice2::Escape** method sends a hardware-specific command to the driver.

```
HRESULT IDirectInputDevice2::Escape(
  LPDIEFFESCAPE pesc
);
```

Parameters

pesc

A **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 may 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.

IDirectInputDevice2::GetEffectInfo

The **IDirectInputDevice2::GetEffectInfo** method obtains information about an effect.

```
HRESULT IDirectInputDevice2::GetEffectInfo(
  LPDIEFFECTINFO pdei,
  REFGUID rguid
);
```

Parameters

pdei
A 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 may 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);
```

IDirectInputDevice2::GetForceFeedbackState

The **IDirectInputDevice2::GetForceFeedbackState** method retrieves the state of the device's force feedback system.

```
HRESULT IDirectInputDevice2::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, 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

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.

Future versions of DirectInput may 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 may 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.

IDirectInputDevice2::Poll

The **IDirectInputDevice2::Poll** method retrieves data 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).

HRESULT IDirectInputDevice2::Poll()

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may 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 **IDirectInputDevice::SetDataFormat** method, and the device must be acquired by using the **IDirectInputDevice::Acquire** method.

See Also

Polling and Events

IDirectInputDevice2::SendForceFeedbackCommand

The **IDirectInputDevice2::SendForceFeedbackCommand** method sends a command to the device's force feedback system.

```
HRESULT IDirectInputDevice2::SendForceFeedbackCommand(
  DWORD dwFlags
);
```

Parameters

dwFlags

A single value indicating the desired change in state. The value may 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 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.

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 **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 may 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.

IDirectInputEffect

Applications use the methods of the **IDirectInputEffect** interface to manage effects of force feedback devices.

The interface is obtained by using the **IDirectInputDevice2::CreateEffect** method.

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

Effect information	**GetEffectGuid**
	GetEffectStatus
	GetParameters
Effect manipulation	**Download**
	Initialize

	SetParameters
	Start
	Stop
	Unload
Miscellaneous	**Escape**

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

AddRef

QueryInterface

Release

the **LPDIRECTINPUTEFFECT** type is defined as a pointer to the **IDirectInputEffect** interface:

```
typedef struct IDirectInputEffect    *LPDIRECTINPUTEFFECT;
```

IDirectInputEffect::Download

The **IDirectinputEffect::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 **IDirectInputEffect::SetParameters** method.

```
HRESULT IDirectInputEffect::Download(void);
```

Return Values

If the method succeeds, the return value is DI_OK or S_FALSE.

If the method fails, the return value may 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**.

IDirectInputEffect::Escape

The **IDirectInputEffect::Escape** method sends a hardware-specific command to the driver.

```
HRESULT IDirectInputEffect::Escape(
  LPDIEFFESCAPE pesc
);
```

Parameters

pesc

A **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 may 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.

IDirectInputEffect::GetEffectGuid

The **IDirectInputEffect::GetEffectGuid** method retrieves the GUID for the effect represented by the **IDirectInputEffect** object.

```
HRESULT IDirectInputEffect::GetEffectGuid(
  LPGUID pguid
);
```

Parameters

pguid

A **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 may 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 **IDirectInputDevice2::GetEffectInfo**.

IDirectInputEffect::GetEffectStatus

The **IDirectInputEffect::GetEffectStatus** method retrieves the status of an effect.

```
HRESULT IDirectInputEffect::GetEffectStatus(
  LPDWORD pdwFlags
);
```

Parameters

pdwFlags
Status flags for the effect. The value may be zero, 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 may be one of the following error values:

DIERR_INVALIDPARAM
DIERR_NOTINITIALIZED

IDirectInputEffect::GetParameters

The **IDirectInputEffect::GetParameters** method retrieves information about an effect.

```
HRESULT IDirectInputEffect::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 function.

dwFlags

Flags specifying which portions of the effect information is to be retrieved. The value may be zero, 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_AXES

The **cAxes** and **rgdwAxes** members should receive data. The **cAxes** member on entry contains the sizes (in **DWORD**s) of the buffer pointed to by the **rgdwAxes** member. If the buffer is too small, then 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 **DWORD**s) of the buffer pointed to by the **rglDirection** member. If the buffer is too small, then 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 will return 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 will be set in the **dwFlags** member, indicating which coordinate system DirectInput used. In particular, passing all three coordinate system flags will retrieve 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, then the **lpEnvelope** member will be set to NULL.

DIEP_GAIN

The **dwGain** member should receive data.

DIEP_SAMPLEPERIOD

The **dwSamplePeriod** 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 will be filled in with the type-specific data associated with the effect, and the **cbTypeSpecificParams** member will contain the number of bytes copied. If the buffer supplied by the application is too small to contain all the type-specific data, then the method returns DIERR_MOREDATA, and the **cbTypeSpecificParams** member will contain 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 may 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.

IDirectInputEffect::Initialize

The **IDirectInputEffect::Initialize** method initializes a DirectInputEffect object.

```
HRESULT IDirectInputEffect::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 may be necessary for backwards compatibility. It is an error for a DLL to pass the handle of the parent application.

dwVersion
Version number of DirectInput for which the application is designed. This value will normally be DIRECTINPUT_VERSION. Passing the version number of a previous version will cause 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 **IDirectInputDevice2::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 may 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 **IDirectInputDevice2::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);
```

IDirectInputEffect::SetParameters

The **IDirectInputEffect::SetParameters** method sets information about an effect.

```
HRESULT IDirectInputEffect::SetParameters(
  LPCDIEFFECT peff,
  DWORD dwFlags
);
```

Parameters

peff

A **DIEFFECT** structure that contains effect information. The **dwSize** member must be filled in by the application before calling this function, 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 may be zero, 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 valid.

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 in order 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_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 may 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, then you can manually download the effect by invoking the **IDirectInputEffect::Download** method.

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 **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, 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.

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, then it will be downloaded before being started. This default behavior can be suppressed by passing the DIES_NODOWNLOAD flag.

```
HRESULT IDirectInputEffect::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 its **dwDuration** member to INFINITE.

dwFlags

Flags 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.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may 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.

IDirectInputEffect::Stop

The **IDirectInputEffect::Stop** method stops playing an effect. The parent device must be acquired.

```
HRESULT IDirectInputEffect::Stop(void);
```

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may 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.

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 IDirectInputEffect::Unload(void);
```

Return Values

If the method succeeds, the return value is DI_OK.

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

DIERR_INPUTLOST
DIERR_INVALIDPARAM
DIERR_NOTEXCLUSIVEACQUIRED
DIERR_NOTINITIALIZED

Functions

This section is a reference for DirectInput functions other than COM interface methods and callback functions.

The following is the only function that falls into this category:

- **DirectInputCreate** is used to create the DirectInput system.

DirectInputCreate

The **DirectInputCreate** function creates a DirectInput object that supports the *IDirectInput* COM interface.

```
HRESULT 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 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 may be necessary.

dwVersion

Version number of DirectInput for which the application is designed. This value will normally be DIRECTINPUT_VERSION. Passing the version number of a previous version will cause DirectInput to emulate that version. For more information, see Designing for Previous Versions of DirectInput.

lplpDirectInput

Pointer to an address that will be initialized with 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 will pass NULL. If aggregation is requested, the object returned in **lplpDirectInput* will be 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 may be one of the following error values:

DIERR_BETADIRECTINPUTVERSION

DIERR_INVALIDPARAM

DIERR_OLDDIRECTINPUTVERSION

DIERR_OUTOFMEMORY

Remarks

Calling this 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 this 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.

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.

- **DIEnumCreatedEffectObjectsProc**
- **DIEnumDeviceObjectsProc**
- **DIEnumDevicesProc**
- **DIEnumEffectsProc**

DIEnumCreatedEffectObjectsProc

The **DIEnumCreatedEffectObjectsProc** function is an application-defined callback function that receives DirectInputDevice effects as a result of a call to the **IDirectInputDevice2::EnumCreatedEffectObjects** method.

```
BOOL CALLBACK DIEnumCreatedEffectObjectsProc(
  LPDIRECTINPUTEFFECT peff,
  LPVOID pvRef
);
```

Parameters

peff
Pointer to an effect object that has been created.

pvRef

The application-defined value given in the **IDirectInputDevice2::EnumCreatedEffectObjects** method.

Return Values

Returns DIENUM_CONTINUE to continue the enumeration or DIENUM_STOP to stop the enumeration.

DIEnumDeviceObjectsProc

The **DIEnumDeviceObjectsProc** function is an application-defined callback function that receives DirectInputDevice objects as a result of a call to the **IDirectInputDevice::EnumObjects** method.

```
BOOL CALLBACK DIEnumDeviceObjectsProc(
  LPCDIDEVICEOBJECTINSTANCE lpddoi,
  LPVOID pvRef
);
```

Parameters

lpddoi

A **DIDEVICEOBJECTINSTANCE** structure that describes the object being enumerated.

pvRef

The application-defined value given in the **IDirectInputDevice::EnumObjects** method.

Return Values

Returns DIENUM_CONTINUE to continue the enumeration or DIENUM_STOP to stop the enumeration.

DIEnumDevicesProc

The **DIEnumDevicesProc** function is an application-defined callback function that receives DirectInput devices as a result of a call to the **IDirectInput::EnumDevices** method.

```
BOOL CALLBACK DIEnumDevicesProc(
  LPDIDEVICEINSTANCE lpddi,
  LPVOID pvRef
);
```

Parameters

lpddi
Address of a **DIDEVICEINSTANCE** structure that describes the device instance.

pvRef
The application-defined value given in the **IDirectInput::EnumDevices** method.

Return Values

Returns DIENUM_CONTINUE to continue the enumeration or DIENUM_STOP to stop the enumeration.

DIEnumEffectsProc

The **DIEnumEffectsProc** function is an application-defined callback function used with the **IDirectInputDevice2::EnumEffects** method.

```
BOOL CALLBACK DIEnumEffectsProc(
  LPCDIEFFECTINFO pdei,
  LPVOID pvRef
);
```

Parameters

pdei
A **DIEFFECTINFO** structure that describes the enumerated effect.

pvRef
Address of application-defined data given to the **IDirectInputDevice2::EnumEffects** method.

Return Values

Returns DIENUM_CONTINUE to continue the enumeration, or DIENUM_STOP to stop it.

Macros

This section describes the following macros used in DirectInput:

- **DIDFT_GETINSTANCE**
- **DIDFT_GETTYPE**
- **DIEFT_GETTYPE**
- **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, here is a C call to the **IDirectInputDevice::Release** method:

```
lpdid->lpVtbl->Release(lpdid));
```

The equivalent macro call looks like this:

```
IDirectInputDevice_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.

```
BYTE DIDFT_GETINSTANCE(
  DWORD dwType
);
```

Parameters

dwType
The DirectInput data format type. The possible values for this parameter are identical to those found in the **dwType** member of the **DIOBJECTDATAFORMAT** structure.

DIDFT_GETTYPE

The **DIDFT_GETTYPE** macro extracts the object type code from a data format type.

```
BYTE DIDFT_GETTYPE(
  DWORD dwType
);
```

Parameters

dwType
The DirectInput data format type. The possible values for this parameter are identical to those found in the **dwType** member of the **DIOBJECTDATAFORMAT** structure.

DIEFT_GETTYPE

The **DIEFT_GETTYPE** macro extracts the effect type code from an effect format type.

```
BYTE DIEFT_GETTYPE(
  DWORD dwType
);
```

Parameters

dwType
The DirectInput effect format type. The possible values for this parameter are identical to those found in the **dwEffType** member of the **DIEFFECTINFO** structure.

DISEQUENCE_COMPARE

The **DISEQUENCE_COMPARE** macro compares two DirectInput sequence numbers, compensating for wraparound.

```
BOOL DISEQUENCE_COMPARE(
  DWORD dwSequence1,
  Operator cmp,
  DWORD dwSequence2
);
```

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 zero otherwise.

Remarks

The following example checks whether the *dwSequence1* parameter value precedes the *dwSequence2* parameter value chronologically:

```
BOOL Sooner = (DISEQUENCE_COMPARE(dwSequence1, <, dwSequence2));
```

GET_DIDEVICE_SUBTYPE

The **GET_DIDEVICE_SUBTYPE** macro extracts the device subtype code from a device type description code.

```
BYTE GET_DIDEVICE_SUBTYPE(
  DWORD 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.

This macro is defined in Dinput.h as follows:

```
#define GET_DIDEVICE_SUBTYPE(dwDevType) HIBYTE(dwDevType)
```

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.

```
BYTE GET_DIDEVICE_TYPE(
  DWORD 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.

Remarks

This macro is defined in Dinput.h as follows:

```
#define GET_DIDEVICE_TYPE(dwDevType)    LOBYTE(dwDevType)
```

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**
- **DIJOYSTATE**
- **DIJOYSTATE2**
- **DIMOUSESTATE**
- **DIOBJECTDATAFORMAT**
- **DIPERIODIC**
- **DIPROPDWORD**
- **DIPROPHEADER**
- **DIPROPRANGE**
- **DIRAMPFORCE**

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 {
    LONG lOffset;
    LONG lPositiveCoefficient;
    LONG lNegativeCoefficient;
    DWORD dwPositiveSaturation;
    DWORD dwNegativeSaturation;
    LONG lDeadBand;
} DICONDITION, *LPDICONDITION;

typedef const DICONDITION *LPCDICONDITION;
```

Members

lOffset
The offset for the condition, in the range -10,000 to +10,000.

lPositiveCoefficient
The coefficient constant on the positive side of the offset, in the range -10,000 to +10,000.

lNegativeCoefficient
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 **lNegativeCoefficient** is ignored and the value of **lPositiveCoefficient** is used as both the positive and negative coefficients.

dwPositiveSaturation
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 saturations, then the value of this member is ignored.

lNegativeSaturation
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 **lNegativeSaturation** is ignored and the value of **lPositiveSaturation** is used as both the positive and negative saturations.

lDeadBand

The region around **lOffset** where the condition is not active, in the range 0 to 10,000. In other words, the condition is not active between **lOffset** - **lDeadBand** and **lOffset** + **lDeadBand**.

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 - q0)* where *A* is a scaling coefficient, *q* is some metric, and *q0* is the neutral value for that metric.

The simplified formula give above must be adjusted if a nonzero dead band is provided. If the metric is less than **lOffset** - **lDeadBand**, then the resulting force is given by the following formula:

force = **lNegativeCoefficient** * (*q* - (**lOffset** - **lDeadBand**))

Similarly, if the metric is greater than **lOffset** + **lDeadBand**, then the resulting force is given by the following formula:

force = **lPositiveCoefficient** * (*q* - (**lOffset** + **lDeadBand**))

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, then 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 **lOffset** set to zero 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, then 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.

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 {
    LONG lMagnitude;
```

```
} DICONSTANTFORCE, *LPDICONSTANTFORCE;

typedef const DICONSTANTFORCE *LPCDICONSTANTFORCE;
```

Members

lMagnitude
The 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.

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 {
  DWORD cChannels;
  DWORD dwSamplePeriod;
  DWORD cSamples;
  LPLONG rglForceData;
} DICUSTOMFORCE, *LPDICUSTOMFORCE;

typedef const DICUSTOMFORCE *LPCDICUSTOMFORCE;
```

Members

cChannels
The 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, then nothing is associated with the extra axes.

If there is but a single channel, then the effect will be rotated in the direction specified by the **rglDirection** member of the **DIEFFECT** structure. If there is more than one channel, then rotation is not allowed.

Not all devices support rotation of custom effects.

dwSamplePeriod
The sample period in microseconds.

cSample
The total number of samples in the **rglForceData**. It must be an integral multiple of the **cChannels**.

rglForceData
Pointer to an array of force values representing the custom force. If multiple channels are provided, then the values are interleaved. For example, if **cChannels** is 3, then the first element of the array belongs to the first channel, the second to the second, and the third to the third.

DIDATAFORMAT

The **DIDATAFORMAT** structure carries information describing a device's data format. This structure is used with the **IDirectInputDevice::SetDataFormat** method.

```
typedef struct {
  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 **IDirectInputDevice::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 **IDirectInputDevice::SetProperty** method. This may not 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_dfDIKeyboard*, *c_dfDIJoystick*, or *c_dfDIJoystick2*.

The following declarations set 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  lX;               // x-axis goes here
    LONG  lY;               // 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, lX),
    DIDFT_AXIS | DIDFT_ANYINSTANCE, 0, },
  { &GUID_YAxis, FIELD_OFFSET(MYDATA, lY),
    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
    DIDF_ABSAXIS,               // absolute axis coordinates
    sizeof(MYDATA),             // device data size
    numObjects,                 // number of objects
    rgodf,                      // and here they are
};
```

DIDEVCAPS

The **DIDEVCAPS** structure contains information about a DirectInput device's capabilities. This structure is used with the **IDirectInputDevice::GetCapabilities** method.

```
typedef struct {
    DWORD dwSize;
    DWORD dwFlags;
    DWORD dwDevType;
    DWORD dwAxes;
    DWORD dwButtons;
    DWORD dwPOVs;
    DWORD dwFFSamplePeriod;
    DWORD dwFFMinTimeResolution;
    DWORD dwFirmwareRevision;
    DWORD dwHardwareRevision;
    DWORD dwDriverVersion;
} DIDEVCAPS, *LPDIDEVCAPS;
```

Members

dwSize

Size of this structure, in bytes. This member must be initialized by the application before a call to the **IDirectInputDevice::GetCapabilities** method.

dwFlags

Flags associated with the device. This value can be a combination of the following:

DIDC_ATTACHED

The device is physically attached.

DIDC_DEADBAND

The device supports deadband for at least one force feedback condition.

DIDC_EMULATED

Device functionality is emulated.

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 then the fade level and fade time parameters of the **DIENVELOPE** structure will be ignored by the device.

After a call to the **IDirectInputDevice2::GetEffectInfo** method, an individual effect will set the DIEFT_FFFADE flag if fade is supported for that effect.

DIDC_FFATTACK

The force feedback system supports the attack envelope parameter for at least one effect. If the device does not support attack then the attack level and attack time parameters of the **DIENVELOPE** structure will be ignored by the device.

After a call to the **IDirectInputDevice2::GetEffectInfo** method, an individual effect will set 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 **IDirectInputDevice2::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 **IDirectInputDevice2::Poll** method in order to obtain data.

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** structure will be ignored.

After a call to the **IDirectInputDevice2::GetEffectInfo** method, an individual condition will set the DIEFT_POSNEGCOEFFICIENTS flag if separate positive and negative coefficients are 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, then the negative saturation in the **DICONDITION** structure will be ignored.

After a call to the **IDirectInputDevice2::GetEffectInfo** method, an individual condition will set the DIEFT_POSNEGSATURATION flag if separate positive and negative saturations are 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, then the force generated by a condition is limited only by the maximum force which the device can generate.

After a call to the **IDirectInputDevice2::GetEffectInfo** method, an individual condition will set the DIEFT_SATURATION flag if saturation is supported for that condition.

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
The minimum time between playback of consecutive raw force commands.

dwFFMinTimeResolution
The minimum amount of 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, then the device would round any times to the nearest millisecond.

dwFirmwareRevision
Specifies the firmware revision of the device.

dwHardwareRevision
The hardware revision of the device.

dwDriverVersion
The 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 will have larger numbers.

See Also

DIDEVICEINSTANCE

DIDEVICEINSTANCE

The **DIDEVICEINSTANCE** structure contains information about an instance of a DirectInput device. This structure is used with the **IDirectInput::EnumDevices** and **IDirectInputDevice::GetDeviceInfo** methods.

```
typedef struct {
    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 may 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.

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 above categories

The following subtypes are defined for mouse-type devices.

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.

The following subtypes are defined for keyboard-type devices.

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.

The following subtypes are defined for joystick-type devices.
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

The high-order word of the device type description code contains flags that further identify the device.
DIDEVTYPE_HID
The device uses the Human Input 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 a HID device, then this member contains the HID usage page code.

wUsage
If the device is a HID device, then 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.

DIDEVICEOBJECTDATA

The **DIDEVICEOBJECTDATA** structure contains raw buffered device information. This structure is used with the **IDirectInputDevice::GetDeviceData** method.

```
typedef struct {
    DWORD dwOfs;
    DWORD dwData;
    DWORD dwTimeStamp;
    DWORD dwSequence;
} DIDEVICEOBJECTDATA, *LPDIDEVICEOBJECTDATA;

typedef const DIDEVICEOBJECTDATA *LPCDIDEVICEOBJECTDATA;
```

Members

dwOfs

Value specifying the offset into the current data format of the object whose data is being reported. That is, the location where the **dwData** would have been stored if the data had been obtained by a call to the **IDirectInputDevice::GetDeviceState** method.

If the device is accessed as a mouse, keyboard, or joystick, the **dwOfs** member will be one of the mouse device constants, keyboard device constants, or joystick device constants. If a custom data format has been set, then it will be an offset relative to the custom data format.

dwData

Data obtained from the device. The format of this data depends on the type of the device, but in all cases, the data is reported in raw form.

For axes, 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 buttons, 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 event was generated, in milliseconds. The current system tick count can be obtained by calling the Win32 **GetTickCount** function. Remember that this value wraps around approximately every 50 days.

dwSequence

DirectInput sequence number for this event. All DirectInput 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.

DIDEVICEOBJECTINSTANCE

The **DIDEVICEOBJECTINSTANCE** structure contains information about a device object instance. This structure is used with the **IDirectInputDevice::EnumObjects** method to provide the **DIEnumDeviceObjectsProc** callback function with information about a particular object associated with a device, like an axis or button. It is also used with the **IDirectInputDevice::GetObjectInfo** method to retrieve information about a device object.

```
typedef struct {
    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 wReserved;
} DIDEVICEOBJECTINSTANCE, *LPDIDEVICEOBJECTINSTANCE;

typedef const DIDEVICEOBJECTINSTANCE *LPCDIDEVICEOBJECTINSTANCE;
```

Members

dwSize

Size of the structure, in bytes. During enumeration, the application may inspect this value to determine how many members of the structure are valid. When the structure is passed to the **IDirectInputDevice::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 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

Unknown.

Other object types may be defined in the future.

dwOfs

Offset within the data format at which the data reported by this object is most efficiently obtained. This member is significant only for applications that build custom data formats. Most applications will not use this value.

dwType

Device type that describes the object. It is a combination of DIDFT_* flags that describe the object type (axis, button, and so forth) 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 **IDirectInputDevice::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_POLLED

The object does not return data until the **IDirectInputDevice2::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

Reserved.

wDesignatorIndex

Reserved.

wUsagePage

The HID usage page associated with the object, if known. HID 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.

wUsage

The HID usage associated with the object, if known. HID 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.

dwDimension

The dimensional units in which the object's value is reported, if known, or zero if not known. Applications can use this field to distinguish between, for example, the position and velocity of a control.

wExponent
The exponent to associate with the dimension, if known.

wReserved
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), then 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.

DIEFFECT

The **DIEFFECT** structure is used by the **IDirectInputDevice2::CreateEffect** method to initialize a new **IDirectInputEffect** object. It is also used by the **IDirectInputEffect::SetParameters** and **IDirectInputEffect::GetParameters** methods.

```
typedef struct {
    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;
} 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 via **IDirectInputDevice::EnumObjects**.

DIEFF_OBJECTOFFSETS

The values of **dwTriggerButton** and **rgdwAxes** are data format offsets, relative to the data format selected by **IDirectInput::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, then the effect has infinite duration. If an envelope has been applied to the effect, then the attack will be applied, followed by an infinite sustain.

dwSamplePeriod

The period at which the device should play back the effect, in microseconds. A value of zero 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 will choose 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 0 to 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, then 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, then each value in **rglDirection** is associated with the corresponding axis in **rgdwAxes**.

If polar, then 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, then the first angle is measured in hundredths of degrees 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 degrees towards (0, 0, 1). The third angle (if the number of axes is four or more) is measured in hundredths of degrees towards (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 zero.

lpEnvelope

Optional pointer to a **DIENVELOPE** structure that describes the envelope to be used by this effect. Note that not all effect types use envelopes. If no envelope is to be applied, then 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, then 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, then 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, then this member contains a pointer to a **DIPERIODIC** structure that defines the parameters for the effect.

If the effect is of type DIEFT_CONSTANTFORCE, then 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, then this member contains a pointer to a **DIRAMPFORCE** structure that defines the parameters for the ramp force.

DIEFFECTINFO

The **DIEFFECTINFO** structure is used by the **IDirectInputDevice2::EnumEffects** and **IDirectInputDevice2::GetEffectInfo** methods to return information about a particular effect supported by a device.

```
typedef struct {
    DWORD dwSize;
    GUID guid;
    DWORD dwEffType;
    DWORD dwStaticParams;
    DWORD dwDynamicParams;
    TCHAR tszName[MAX_PATH];
} DIEFFECTINFO, *LPDIEFFECTINFO;

typedef const DIEFFECTINFO  *LPCDIEFFECTINFO;
```

Members

dwSize

The size of the structure in bytes. During enumeration, the application may 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 **IDirectInputDevice2::GetEffectInfo** method.

guid

Identifier of the effect.

dwEffType

Zero or more of the following values:

DIEFT_ALL

Valid only for **IDirectInputDevice2::EnumEffects.** Enumerate all effects, regardless of type. This flag may not 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 **IDirectInputDevice2::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 **IDirectInputDevice2::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 **IDirectInputDevice2::EnumEffects** to restrict the enumeration to custom-force effects.

DIEFT_DEADBAND

The effect generator for this condition effect supports the **lDeadBand** parameter.

DIEFT_FFATTACK

The effect generator for this effect supports the attack envelope parameter. If the effect generator does not support attack then the attack level and attack time parameters of the **DIENVELOPE** structure will be ignored by the effect.

If neither DIEFT_FFATTACK nor DIEFT_FFFADE is set, then the effect does not support an envelope, and any provided envelope will be ignored.

DIEFT_FFFADE

The effect generator for this effect supports the fade parameter. If the effect generator does not support fade then the fade level and fade time parameters of the **DIENVELOPE** structure will be ignored by the effect.

If neither DIEFT_FFATTACK nor DIEFT_FFFADE is set, then the effect does not support an envelope, and any provided envelope will be 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 **IDirectInputDevice2::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 **IDirectInputDevice2::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, then the negative coefficient in the **DICONDITION** structure will be ignored and the positive coefficient will be 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, then the negative saturation in the **DICONDITION** structure will be ignored and the positive saturation will be 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 **IDirectInputDevice2::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, then 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, then 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 which are not supported by the device. The unsupported parameters are merely 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, then driver is permitted to stop the effect, update the parameters, then restart it. See **IDirectInputEffect::SetParameters** for more information.

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.

DIEFFESCAPE

The **DIEFFESCAPE** structure is used by the **IDirectInputEffect::Escape** method to pass hardware-specific data directly to the device driver.

```
typedef struct {
    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
The 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.

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 **lMagnitude** member of the **DICONSTANTFORCE** structure. The sustain time is represented by **dwDuration** member of the **DIEFFECT** structure.

```
typedef struct {
    DWORD dwSize;
    DWORD dwAttackLevel;
    DWORD dwAttackTime;
    DWORD dwFadeLevel;
    DWORD dwFadeTime;
} DIENVELOPE, *LPDIENVELOPE;

typedef const DIENVELOPE *LPCDIENVELOPE;
```

Members

dwSize
The 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 0 to 10,000. If the effect's type-specific data does not specify a baseline, then the amplitude is relative to zero.

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 0 to 10,000. If the effect's type-specific data does not specify a baseline, then the amplitude is relative to zero.

dwFadeTime
The time, in microseconds, to reach the fade level.

DIJOYSTATE

The **DIJOYSTATE** structure contains information about the state of a joystick device. This structure is used with the **IDirectInputDevice::GetDeviceState** method.

```
typedef struct DIJOYSTATE {
    LONG    lX;
    LONG    lY;
    LONG    lZ;
    LONG    lRx;
    LONG    lRy;
    LONG    lRz;
    LONG    rglSlider[2];
    DWORD   rgdwPOV[4];
    BYTE    rgbButtons[32];
} DIJOYSTATE, *LPDIJOYSTATE;
```

Members

lX
Information about the joystick x-axis (usually the left-right movement of a stick).

lY
Information about the joystick y-axis (usually the forward-backward movement of a stick).

lZ
Information about the joystick z-axis (often the throttle control). If the joystick does not have this axis, the value is zero.

lRx
Information about the joystick x-axis rotation. If the joystick does not have this, the value is zero.

lRy

Information about the joystick y-axis rotation. If the joystick does not have this axis, the value is zero.

lRz

Information about the joystick z-axis rotation (often called the rudder). If the joystick does not have this axis, the value is zero.

rglSlider[2]

Two additional axis values (formerly called the u-axis and v-axis) whose semantics depend on the joystick. Use the **IDirectInputDevice::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 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, *dwPOV* will be -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 **IDirectInputDevice::SetDataFormat** method, passing the *c_dfDIJoystick* global 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. Determine whether the indicator is centered as follows:

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

DIJOYSTATE2

The **DIJOYSTATE2** structure contains information about the state of a joystick device with extended capabilities. This structure is used with the **IDirectInputDevice::GetDeviceState** method.

```
typedef struct DIJOYSTATE2 {
    LONG    lX;
    LONG    lY;
    LONG    lZ;
    LONG    lRx;
    LONG    lRy;
    LONG    lRz;
    LONG    rglSlider[2];
```

```
    DWORD   rgdwPOV[4];
    BYTE    rgbButtons[128];
    LONG    lVX;
    LONG    lVY;
    LONG    lVZ;
    LONG    lVRx;
    LONG    lVRy;
    LONG    lVRz;
    LONG    rglVSlider[2];
    LONG    lAX;
    LONG    lAY;
    LONG    lAZ;
    LONG    lARx;
    LONG    lARy;
    LONG    lARz;
    LONG    rglASlider[2];
    LONG    lFX;
    LONG    lFY;
    LONG    lFZ;
    LONG    lFRx;
    LONG    lFRy;
    LONG    lFRz;
    LONG    rglFSlider[2];
} DIJOYSTATE2, *LPDIJOYSTATE2;
```

lX

Information about the joystick x-axis (usually the left-right movement of a stick).

lY

Information about the joystick y-axis (usually the forward-backward movement of a stick).

lZ

Information about the joystick z-axis (often the throttle control). If the joystick does not have this axis, the value is zero.

lRx

Information about the joystick x-axis rotation. If the joystick does not have this, the value is zero.

lRy

Information about the joystick y-axis rotation. If the joystick does not have this axis, the value is zero.

lRz

Information about the joystick z-axis rotation (often called the rudder). If the joystick does not have this axis, the value is zero.

rglSlider[2]

Two additional axis values (formerly called the u-axis and v-axis) whose semantics depend on the joystick. Use the **IDirectInputDevice::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 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, *dwPOV* will be -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.

VX
Information about the x-axis velocity.

lVY
Information about the y-axis velocity.

lVZ
Information about the z-axis velocity.

lVRx
Information about the x-axis angular velocity.

lVRy
Information about the y-axis angular velocity.

lVRz
Information about the z-axis angular velocity.

rglVSlider[2]
Information about extra axis velocities.

lAX
Information about the x-axis acceleration.

lAY
Information about the y-axis acceleration.

lAZ
Information about the z-axis acceleration.

lARx
Information about the x-axis angular acceleration.

lARy
Information about the y-axis angular acceleration.

lARz
Information about the z-axis angular acceleration.

rglASlider[2]
Information about extra axis accelerations.

lFX
Information about the x-axis force.

lFY

Information about the y-axis force.

lFZ
Information about the z-axis force.

lFRx
Information about the x-axis torque.

lFRy
Information about the y-axis torque.

lFRz
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 **IDirectInputDevice::SetDataFormat** method, passing the *c_dfDIJoystick2* global data format variable.

The **DIJOYSTATE2** structure has no special association with the **IDirectInputDevice2** interface. You can use either **DIJOYSTATE** or **DIJOYSTATE2** with either the **IDirectInputDevice** or the **IDirectInputDevice2** interface.

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. Determine whether the indicator is centered as follows:

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

DIMOUSESTATE

The **DIMOUSESTATE** structure contains information about the state of a mouse device or another device that is being accessed as if it were a mouse device. This structure is used with the **IDirectInputDevice::GetDeviceState** method.

```
typedef struct {
    LONG lX;
    LONG lY;
    LONG lZ;
    BYTE rgbButtons[4];
} DIMOUSESTATE, *LPDIMOUSESTATE;
```

Members

lX
Information about the mouse x-axis.

lY
Information about the mouse y-axis.

lZ
Information about the mouse z-axis (typically a wheel). If the mouse does not have a z-axis, then the value is zero.

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 **IDirectInputDevice::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. As a result, 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, then the appropriate member contains the change in position. If it is in absolute mode, then the member contains the absolute axis position.

DIOBJECTDATAFORMAT

The **DIOBJECTDATAFORMAT** structure contains information about a device object's data format for use with the **IDirectInputDevice::SetDataFormat** method.

```
typedef struct {
    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 will be stored. This value must be a multiple of four 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 **IDirectInput::SetDataFormat** method must be an absolute axis.

DIDFT_AXIS

The object selected by the **IDirectInput::SetDataFormat** method must be an absolute or relative axis.

DIDFT_BUTTON

The object selected by the **IDirectInput::SetDataFormat** method must be a push button or a toggle button.

DIDFT_FFACTUATOR

The object selected by the **IDirectInput::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 **IDirectInput::SetDataFormat** method must be a valid force feedback effect trigger.

DIDFT_POV

The object selected by the **IDirectInput::SetDataFormat** method must be a point-of-view controller.

DIDFT_PSHBUTTON

The object selected by the **IDirectInput::SetDataFormat** method must be a push button.

DIDFT_RELAXIS

The object selected by **IDirectInputDevice::SetDataFormat** must be a relative axis.

DIDFT_TGLBUTTON

The object selected by **IDirectInputDevice::SetDataFormat** must be a toggle button.

dwFlags

Zero or more of the following values:

DIDOI_ASPECTACCEL

The object selected by **IDirectInputDevice::SetDataFormat** must report acceleration information.

DIDOI_ASPECTFORCE

The object selected by **IDirectInputDevice::SetDataFormat** must report force information.

DIDOI_ASPECTPOSITION

The object selected by **IDirectInputDevice::SetDataFormat** must report position information.

DIDOI_ASPECTVELOCITY

The object selected by **IDirectInputDevice::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 **IDirectInputDevice::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_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 will fail.

```
DIOBJECTDATAFORMAT dfButton0 = {
    0,                              // Wildcard
    18,                              // Offset
    DIDFT_BUTTON | DIDFT_MAKEINSTANCE(0), // Button zero
    0,                              // Don't care about aspect
};
```

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 {
    DWORD dwMagnitude;
    LONG lOffset;
    DWORD dwPhase;
    DWORD dwPeriod;
} DIPERIODIC, *LPDIPERIODIC;

typedef const DIPERIODIC *LPCDIPERIODIC;
```

Members

dwMagnitude

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.

lOffset
The offset of the effect. The range of forces generated by the effect will be **lOffset** - **dwMagnitude** to **lOffset** + **dwMagnitude**. The value of the **lOffset** member is also the baseline for any envelope that is applied to the effect.

dwPhase
The position in the cycle of the periodic effect at which playback begins, in the range 0 to 35,999. See Remarks.

dwPeriod
The period of the effect in microseconds.

Remarks

A device driver may not provide support for all values in the **dwPhase** member. In this case the value will be rounded off to the nearest supported value.

DIPROPDWORD

The **DIPROPDWORD** is a generic structure used to access **DWORD** properties.

```
typedef struct {
    DIPROPHEADER    diph;
    DWORD           dwData;
} DIPROPDWORD, *LPDIPROPDWORD;

typedef const DIPROPDWORD *LPCDIPROPDWORD;
```

Members

diph
A **DIPROPHEADER** structure that must be initialized as follows:

Member	Value
dwSize	sizeof(**DIPROPDWORD**)
dwHeaderSize	sizeof(**DIPROPHEADER**)
dwObj	If the **dwHow** member is DIPH_DEVICE, this member must be zero. If the **dwHow** member is DIPH_BYID, this member must be the identifier for the object whose property setting is to be set or retrieved. If the **dwHow** member is DIPH_BYOFFSET, this member must be a data format offset for the object whose property setting is to be set or retrieved. For example, if the *c_dfDIMouse* data format is selected, it must be one of the DIIMOFS_* values.
dwHow	Specifies how the **dwObj** member should be interpreted. If

	dwObj is DIPROP_AXISMODE or DIPROP_BUFFERSIZE, **dwHow** should be DIPH_DEVICE.

dwData
The property-specific value being set or retrieved.

See Also

DIPROPRANGE, **IDirectInputDevice::GetProperty**, **IDirectInputDevice::SetProperty**

DIPROPHEADER

The **DIPROPHEADER** is a generic structure that is placed at the beginning of all property structures.

```
typedef struct {
    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 **dwObj** member must be zero.
DIPH_BYOFFSET	The **dwObj** member is the offset into the current data format of the object whose property is being accessed.
DIPH_BYID	The **dwObj** member is the object

type/instance identifier. This identifier is returned in the **dwType** member of the **DIDEVICEOBJECTINSTANCE** structure returned from a previous call to the **IDirectInputDevice::EnumObjects** member.

DIPROPRANGE

The **DIPROPRANGE** structure contains information about the range of an object within a device. This structure is used with the DIPROP_RANGE flag set in the **IDirectInputDevice::GetProperty** and **IDirectInputDevice::SetProperty** methods.

```
typedef struct {
    DIPROPHEADER diph;
    LONG        lMin;
    LONG        lMax;
} DIPROPRANGE, *LPDIPROPRANGE;

typedef const DIPROPRANGE *LPCDIPROPRANGE;
```

Members

diph

A **DIPROPHEADER** structure that must be initialized as follows:

dwSize	sizeof(DIPROPRANGE)
dwHeaderSize	sizeof(**DIPROPHEADER**)
dwObj	Identifier of the object whose range is being retrieved or set.
dwHow	How the **dwObj** member should be interpreted.

lMin

The lower limit of the range, inclusive. If the range of the device is unrestricted, this value will be DIPROPRANGE_NOMIN when the **IDirectInputDevice::GetProperty** method returns.

lMax

The upper limit of the range, inclusive. If the range of the device is unrestricted, this value will be DIPROPRANGE_NOMAX when the **IDirectInputDevice::GetProperty** method returns.

Remarks

The range values for devices whose ranges are unrestricted will wrap around.

See Also

DIPROPDWORD, **IDirectInputDevice::GetProperty**, **IDirectInputDevice::SetProperty**

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 {
    LONG lStart;
    LONG lEnd;
} DIRAMPFORCE, *LPDIRAMPFORCE;

typedef const DIRAMPFORCE *LPCDIRAMPFORCE;
```

Members

lStart
The magnitude at the start of the effect, in the range -10,000 to +10,000.

lEnd
The magnitude at the end of the effect, in the range -10,000 to +10,000.

Remarks

The **dwDuration** for a ramp force effect cannot be INFINITE.

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 will be 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_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 key
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

The following alternate names are available:

Alternate name	**Regular name**	**Note**
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_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 chart below 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

Mouse Device Constants

Mouse device constants, defined in Dinput.h, represent offsets within a mouse device's data packet, the **DIMOUSESTATE** structure. The data at a given offset is

associated with a device object (button or axis). Typically, these values will be 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_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 will be 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	A button.
or DIJOFS_BUTTON(*n*)	
DIJOFS_POV(*n*)	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(*n*)	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 will not collect any data and event notifications will not be signaled until the **IDirectInputDevice2::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 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_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 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_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.