Microsoft®

DirectInput™ 3.0

Application Programming Interface

September 11, 1996

Version 1.0

Table Of Contents

Introduction

The Windows DirectX SDK enables the creation of world class computer based games. DirectInput is a component of the DirectX SDK that allows fast, convenient access to input device data.

DirectInput 1.0, which shipped with the Windows 95, enabled access to digital joystick devices. It consisted of a few API, such as joyGetPosEx, a calibration applet (joy.cpl) and a driver model based on VJOYD.VXD which enabled support for digital joystick devices.

Since the release of version 1.0, one of the most frequently received requests for enhancements to the DirectX SDK has been to allow faster access to mouse and keyboard data than Windows currently provides. This is the issue being addressed with this version of DirectInput. DirectInput 3.0 provides faster access to mouse and keyboard data. Unlike DirectInput 1.0, the DirectInput 3.0 API for mouse and keyboard uses COM objects and interfaces.

In a future version of DirectInput, COM support will be added for joystick devices. Force feedback support will be added for joystick devices that support force feedback. Functionality will also be added to support generic input devices (devices that are not directly supported by a specific DirectInput interface).

Purpose of this Document

The purpose of this document is to describe the DirectInput Application Programming Interface and COM interfaces for version 3.0. Only the API and COM interfaces necessary to support mouse and keyboard input will be described. DirectInput 1.0 is not covered in this document.

Definition of Terms

API	Application Programmers Interface. This differs from an interface in that the function does not require a COM object to be instantiated first and the function does not generally have state associated with it.
COM	Component Object Model.
Device	Within the context of DirectInput, a device is a physical piece of hardware which the user can use to provide input to an application.
DirectInput	This is a COM object that represents the entire DirectInput subsystem.
GUID	Globally unique identifier. GUIDs are used by many parts of DirectInput as a way of uniquely identifying things.
IDirectInput	The COM interface exposed by the DirectInput object to enumerate input devices and create device objects.
Interface	The term used to describe the set of member functions accessible by an application which has instantiated a COM object. Member functions can be thought of as APIs that are specific to the object. An interface is essentially a C++ class with no data, in which all functions are virtual.
Object	Within the context of a DirectInput device, an “object” is a source of input on the device. For example, a mouse device may contain objects corresponding to the x-axis, the y-axis, and each of the buttons.

Chapter 1: Overview

The DirectInput Object

The **DirectInput** object represents the **DirectInput** subsystem. Applications can create a **DirectInput** object by calling the **DirectInputCreate** API, which returns an **IDirectInput** interface.

Once a pointer to the **IDirectInput** interface has been obtained, DirectInput enabled input devices can be enumerated. Input devices are enumerated through the **IDirectInput::EnumDevices** method.

The DirectInputDevice Object

The **DirectInputDevice** object represents an input device, be it a mouse or keyboard or some other type of device. Applications can create a **DirectInputDevice** object by calling the **IDirectInput::CreateDevice** method, which returns the **IDirectInputDevice** interface.

The first parameter to IDirectInput::CreateDevice is an instance GUID that identifies the instance of the device for which the interface is to be created. DirectInput comes with two predefined instance GUIDs, GUID_SysMouse and GUID_SysKeyboard which represent the user's primary mouse and primary keyboard, respectively.

Retrieving data from a mouse device

In order to retrieve data from a mouse device, call **IDirectInputDevice::SetDataFormat** with the *c_dfDIMouse* data format. The data returned for a mouse device is based on the number of units the mouse has moved, not screen coordinates. These mouse units are based on the actual values returned by the mouse hardware (also known as mickeys). DirectInput does not modify (cook) the data in any way. Only raw mouse data is returned.

The data returned from the mouse can be either relative or absolute data. Because a mouse is a relative device, relative data is returned by default. The axis mode of the mouse device, which specifies whether relative or absolute data should be returned, is a property of the device which can be changed via the **IDirectInputDevice::SetProperty** method. To set the axis mode to absolute, call **IDirectInputDevice::SetProperty** with the **REFGUID** parameter equal to **DIPROP_AXISMODE**. Set the *dwData* field of the **DIPROPDWORD** structure to **DIPROPAXISMODE_ABS**.

When the axis mode for the mouse device is set to relative, the axis coordinate represents the number of mouse units that the device was moved along that particular axis. A negative value indicates that the mouse was moved to the left for the X axis, upward for the Y axis and back for the Z axis. A positive value indicates that the mouse moved right for the X axis, downward for the Y axis and forward for the Z axis.

Note that a mouse has no concept of absolute position; as a result, absolute coordinates are simply a running total of all relative motions received by **DirectInput**. This means, in particular, that the numerical value of the absolute coordinate is meaningless; specifically, it is unrelated to the screen coordinates of the mouse pointer. Applications should treat absolute coordinates as relative to an unknown origin. For example, an application can record the current absolute position immediately after acquiring the device and save it as the "virtual origin". This virtual origin can then be subtracted from subsequent absolute coordinates retrieved from the device, via **IDirectInputDevice::GetDeviceState** or **IDirectInputDevice::GetDeviceData** (up until the next **IDirectInputDevice::Unacquire**) to compute the relative distance the mouse has moved from the point of acquisition. Absolute coordinates on purely relative devices (such as a mouse) are meaningful only when compared to some previously saved position.

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

To retrieve buffered data from the mouse, create an array of **DIDEVICEOBJECTDATA** structures and pass the pointer and a variable containing the size of the array to **IDirectInputDevice::GetDeviceData**. DirectInput will place the oldest mouse data into the array until there is no more data in the input queue or the array is filled. On return from **IDirectInputDevice::GetDeviceData**, the size variable will contain the number of array elements actually used. When collecting buffered data from the mouse, the data provided in a single **DIDEVICEOBJECTDATA** structure is a change in state for a single object on the mouse. For instance, a typical mouse will contain 4 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 of **DIDEVICEOBJECTDATA** structures passed to **IDirectInputDevice::GetDeviceData** will have three elements filled in: an element for button 0 going down, an element for the change in the X axis, and an element for the change in the Y axis. After the **IDirectInputDevice::GetDeviceData** call, the application can determine which object an element in the array refers to by checking the *dwOfs* field of the **DIDEVICEOBJECTDATA** structure against the following predefined constants: **DIMOFS_BUTTON0, DIMOFS_BUTTON1**, **DIMOFS_BUTTON2**, **DIMOFS_BUTTON3**, **DIMOFS_X**, **DIMOFS_Y**, and **DIMOFS_Z**. These constants refer to the offset of these values in the **DIMOUSESTATE** structure. Using these constants, you can tell exactly which object on the mouse the data in the **DIDEVICEOBJECTDATA** structure refers to. The actual data for that object is located in the *dwData* field of the structure. For button objects, only the low byte of *dwData* is significant. The high bit of this byte is set if the button went down and clear if the button went up.

The *Scrawl* sample application illustrates one way to collect buffered data and process the information reported by **IDirectInputDevice::GetDeviceData**.

Time-stamped mouse data is only available if data is being retrieved through **IDirectInputDevice::GetDeviceData**.

An application must set the cooperative level for the mouse device before acquiring the device and retrieving data. To set the cooperative level, call **IDirectInputDevice::SetCooperativeLevel** with the flags representing the desired cooperative levels. Under Windows 95, the following cooperative levels are supported for mouse devices: **DISCL_BACKGROUND | DISCL_NONEXCLUSIVE**, **DISCL_FOREGROUND | DISCL_NONEXCLUSIVE**, and **DISCL_FOREGROUND | DISCL_EXCLUSIVE**. DirectInput 3.0 does not support the **DISCL_BACKGROUND | DISCL_EXCLUSIVE** cooperative level for mouse devices. See the descriptions of these cooperative constants for more information on what each cooperative level means.

In a future version of DirectX, DirectInput will be supported on Windows NT. However, Windows NT will only support **DISCL_FOREGROUND | DISCL_EXCLUSIVE** for a mouse device. Depending on the level of mouse driver support, even DirectInput on Windows 95 may be restricted to **DISCL_FOREGROUND | DISCL_EXCLUSIVE** support. Therefore, if your application needs to run on the widest number of platforms and mouse drivers, it should use this cooperative level. Furthermore, the application should be tested with the prerelease version of DINPUT.DLL for Windows NT (provided in the Extras directory on the DirectX 3.0 SDK); the Windows NT version of DirectInput (and the Windows 95 version of DirectInput on a non-supported mouse driver) will report **DIERR_INPUTLOST** more frequently than the Windows 95 version, and your application should be written to handle these cases.

Before any data can be retrieved from the mouse device through **IDirectInputDevice::GetDeviceData** or **IDirectInputDevice::GetDeviceState**, the device must be acquired with a call to **IDirectInputDevice::Acquire**. It is recommended that the application release the mouse device by calling **IDirectInputDevice::Unacquire** when the application is paused or loses input focus. The device should also be unacquired when an application menu or the system menu is selected or if the window is

being resized or moved. When the application is no longer paused or regains input focus, the application should reacquire the mouse device by calling **IDirectInputDevice::Acquire**.

If an application is using the mouse in **DISCL_FOREGROUND** mode, it is recommended that application check for the **DIERR_INPUTLOST** return value from **IDirectInputDevice::GetDeviceData** or **IDirectInputDevice::GetDeviceState**. Because DirectInput automatically unacquires the mouse when the application loses focus, the application should try to reacquire the mouse device if it receives a **DIERR_INPUTLOST** return value. However, applications should not blindly attempt to reacquire DirectInput devices on any other type of error. Otherwise, the application may get stuck in an infinite loop repeatedly attempting to acquire a device that cannot be acquired.

If an application is accessing the mouse in **DISCL_NONEXCLUSIVE** mode, mouse data will be received both via DirectInput and through the Windows mouse messages. If an application is accessing the mouse in **DISCL_EXCLUSIVE** mode, mouse data will be available only via DirectInput; Windows mouse messages will not contain useful data.

The *Scrawl* sample application provides an example of the appropriate way to acquire and unacquire the mouse device.

Retrieving data from a keyboard device

In order to retrieve data from a keyboard device, call **IDirectInputDevice::SetDataFormat** with the *c_dfDIKeyboard* data format. DirectInput has defined a constant for each key on the enhanced keyboard as well as the additional keys found on international keyboards. In most cases, these constants are actually the PC enhanced scan codes. These key constants begin with **DIK_** and are defined in dinput.h. 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. See the section titled "**Special remarks on keyboard scan codes**" for more information.

To retrieve the current state of the keyboard, declare a structure of 256 bytes and pass the pointer to the **IDirectInputDevice::GetDeviceState** method. The **IDirectInputDevice::GetDeviceState** method behaves in the same way as the Windows **GetKeyboardState** function: The device state is stored in this array of 256 bytes, with each byte corresponding to the state of a key. For example, if high bit of the **DIK_ENTER**'th byte is set, then the Enter key is being held down. However, unlike **GetKeyboardState**, DirectInput only uses the high bit of the byte. If the high bit is set, then the key is down. Otherwise, the key is up.

To retrieve buffered data from the keyboard, create an array of **DIDEVICEOBJECTDATA** structures and pass the pointer and a variable containing the size of the array to **IDirectInputDevice::GetDeviceData**. DirectInput will place the oldest keyboard data into the array until there is no more data in the input queue or the array is filled. On return from **IDirectInputDevice::GetDeviceData**, the size variable will contain the number of array elements actually used. When collecting buffered data from the keyboard, the data provided in a single **DIDEVICEOBJECTDATA** structure is a change in state for a single object on the keyboard. Each key or button on the keyboard represents an object. If the user presses the "A" key, releases it and then presses the "R" key , the array of **DIDEVICEOBJECTDATA** structures passed to **IDirectInputDevice::GetDeviceData** will have three elements filled in: an element for the "A" key going down, an element for the "A" key going up, and an element for the "R" key going down. After the **IDirectInputDevice::GetDeviceData** call, the application can determine which object (or key) an element in the array refers to by checking the *dwOfs* field of the **DIDEVICEOBJECTDATA** structure against the predefined **DIK_*** constants. Using these constants, you can tell exactly which object on the keyboard the data in the **DIDEVICEOBJECTDATA** structure refers to. The actual data for that object is located in the *dwData* field of the structure. For button objects such as keys on the keyboard, only the low byte of *dwData* is significant. The high bit of this byte is set if the key went down and clear if the key went up.

Time-stamped keyboard data is only available if data is being retrieved through **IDirectInputDevice::GetDeviceData**.

An application must set the cooperative level for the keyboard device before acquiring the device and retrieving data. To set the cooperative level, call **IDirectInputDevice::SetCooperativeLevel** with the flags representing the desired cooperative levels. Under Windows 95, the following cooperative levels are supported for keyboard devices: **DISCL_BACKGROUND | DISCL_NONEXCLUSIVE**, **DISCL_FOREGROUND | DISCL_NONEXCLUSIVE**. DirectInput 3.0 does not support the **DISCL_BACKGROUND | DISCL_EXCLUSIVE** or **DISCL_FOREGROUND | DISCL_EXCLUSIVE** cooperative levels for keyboard devices. This means that keyboard data will always be received via DirectInput and through the Windows messages. See the descriptions of these cooperative constants for more information on what each cooperative level means.

In a future version of DirectX, DirectInput will be supported on Windows NT. However, Windows NT will only support **DISCL_FOREGROUND | DISCL_NONEXCLUSIVE** for a keyboard device. If your application needs to run on Windows NT, it should access the keyboard with this cooperative level. Depending on the level of keyboard driver support, even DirectInput on Windows 95 may be restricted to **DISCL_FOREGROUND | DISCL_NONEXCLUSIVE** support. Therefore, if your application needs to run on the widest number of platforms and keyboard drivers, it should use this cooperative level.

Before any data can be retrieved from the keyboard device through **IDirectInputDevice::GetDeviceData** or **IDirectInputDevice::GetDeviceState**, the device must be acquired with a call to **IDirectInputDevice::Acquire**. It is recommended that the application release the keyboard device by calling **IDirectInputDevice::Unacquire** when the application is paused or loses input focus. The device should also be unacquired when an application menu or the system menu is selected or if the window is being resized or moved. When the application is no longer paused or regains input focus, the application should reacquire the keyboard device by calling **IDirectInputDevice::Acquire**.

If an application is using the keyboard in **DISCL_FOREGROUND** mode, it is recommended that application check for **DIERR_INPUTLOST** return value from **IDirectInputDevice::GetDeviceData** or **IDirectInputDevice::GetDeviceState**. Because DirectInput automatically unacquires the keyboard when the application loses focus, the application should try to reacquire the keyboard device if it receives a **DIERR_INPUTLOST** return value. However, applications should not blindly attempt to reacquire DirectInput devices on any other type of error. Otherwise, the application may get stuck in an infinite loop repeatedly attempting to acquire a device that cannot be acquired.

Special remarks on keyboard scan codes

There are several aspects of keyboards which applications should be aware of. Applications are encouraged to allow users to reconfigure keyboard action keys to suit the physical keyboard layout.

For the purposes of this discussion, the baseline keyboard shall be the US PC Enhanced keyboard. When a key is described as missing, it means that the key is present on the US PC Enhanced keyboard but not on the keyboard under discussion. When a key is described as added, it means that the key is absent on the US PC Enhanced keyboard but present on the keyboard under discussion.

Not all PC Enhanced keyboards support the new Windows keys (DIK_LWIN, DIK_RWIN, and DIK_APPS). There is no way to determine whether the keys are physically available.

Note that there is no DIK_PAUSE key code. The PC Enhanced keyboard does not generate a separate DIK_PAUSE scan code; rather, it synthesizes a "Pause" from the DIK_LCONTROL and DIK_NUMLOCK scan codes.

Keyboards for laptops or other reduced-footprint computers frequently 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 or equal to 0x80). Furthermore, the PC XT keyboard lacks DIK_SYSRQ.

Japanese keyboards, particularly the NEC PC-98 keyboards, contain a substantially different set of keys from US keyboards. Please see **Appendix A** for more information.

Chapter 2: Constants, Global Variables & Structures

c_dfDIKeyboard global variable

A predefined **DIDATAFORMAT** structure which describes a keyboard device. This object is provided in the DINPUT.LIB library file as a convenience.

A pointer to this structure may be passed to **IDirectInputDevice::SetDataFormat** to indicate that the device will be accessed in the form of a keyboard.

c_dfDIMouse global variable

A predefined **DIDATAFORMAT** structure which describes a mouse device. This object is provided in the DINPUT.LIB library file as a convenience.

A pointer to this structure may be passed to **IDirectInputDevice::SetDataFormat** to indicate that the device will be accessed in the form of a mouse.

DIDATAFORMAT Structure

```
typedef struct {
    DWORD dwSize;
    DWORD dwObjSize;
    DWORD dwFlags;
    DWORD dwDataSize;
    DWORD dwNumObjs;
    LPDIOBJECTDATAFORMAT rgodf;
} DIDATAFORMAT;
```

The **DIDATAFORMAT** structure is used by the **IDirectInputDevice::SetDataFormat** method to set the data format for a device. An application typically does not need to create a **DIDATAFORMAT** structure; rather, it can use one of the predefined data formats, *c_dfDIMouse* or *c_dfDIKeyboard*.

Members

dwSize
The size of the **DIDATAFORMAT** structure.

dwObjSize
The size of the **DIDATAOBJECTFORMAT** structure.

dwFlags
Flags describing other attributes of the data format.

The following flags are defined:

DIDF_RELAXIS: Set the axes into relative mode. Setting this flag in the data format is equivalent to manually setting the axis mode property via **IDirectInputDevice::SetProperty**. The flag may not be combined with **DIDF_ABSAXIS**.

DIDF_ABSAXIS: Set the axes into absolute mode. Setting this flag in the data format is equivalent to manually setting the axis mode property via **IDirectInputDevice::SetProperty**. The flag may not be combined with **DIDF_RELAXIS**.

dwDataSize
The size of the device data that should be returned by the device. This value must be a multiple of four and must exceed the **dwOfs** value for all objects specified in the object list.

dwNumObjs
The number of objects in the **rgodf** array.

rgodf
Pointer to an array of **DIOBJECTDATAFORMAT** structures, each of which describes how one object's data should be reported in the device data. It is an error for the *rgodf* to indicate that two difference pieces of information be placed in the same location. It is also an error for the *rgodf* to indicate that the same piece of information be placed in two locations.

Examples

The following declarations set a data format which can be used for an application which is interested in two axes (reported in absolute coordinates) and two buttons.

```
// Suppose an application wishes to use 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),       0,   DIDFT_AXIS |
DIDFT_ANYINSTANCE, },
  { &GUID_YAxis,    FIELD_OFFSET(MYDATA, lY),       0,   DIDFT_AXIS |
DIDFT_ANYINSTANCE, },
  { &GUID_Button,   FIELD_OFFSET(MYDATA, bButtonA), 0, DIDFT_BUTTON |
DIDFT_ANYINSTANCE, },
  { &GUID_Button,   FIELD_OFFSET(MYDATA, bButtonB), 0, DIDFT_BUTTON |
DIDFT_ANYINSTANCE, },
};
#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 Structure

```
typedef struct {
    DWORD dwSize;
    DWORD dwDevType;
    DWORD dwFlags;
```

```
    DWORD dwAxes;
    DWORD dwButtons;
    DWORD dwPOVs;
} DIDEVCAPS;
```

The **DIDEVCAPS** structure is used by the **IDirectInputDevice::GetCapabilities** method to return the capabilities of the device.

Members

dwSize
Specifies the size, in bytes, of the structure. This field must be initialized by the application before calling **IDirectInputDevice::GetCapabilities**.

dwDevType
Device type specifier. See the section titled **DirectInput device type description codes** for a description of this field.

dwFlags
Flags associated with the device. The following flags are defined:

DIDC_ATTACHED: The device is physically attached.

DIDC_POLLEDDEVICE: The device is polled rather than interrupt-driven. The application must explicitly call **GetDeviceState** in order to obtain data; buffering and event notifications will not be effective.

dwAxes
Specifies the number of axes available on the device.

dwButtons
Specifies the number of buttons available on the device.

dwPOVs
Specifies the number of point-of-view controllers available on the device. This is not used in version 3.0 of DirectInput.

DIDEVICEINSTANCE Structure

```
typedef struct {
    DWORD dwSize;
    GUID guidInstance;
    GUID guidProduct;
    DWORD dwDevType;
    TCHAR tszInstanceName[MAX_PATH];
    TCHAR tszProductName[MAX_PATH];
} DIDEVICEINSTANCE;
```

The **DIDEVICEINSTANCE** structure is used by the **IDirectInput::EnumDevices** and **IDirectInputDevice::GetDeviceInfo** methods to return information about a particular device instance.

Members

dwSize
The size of the structure in bytes.

guidInstance
Unique identifier which identifies 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 machine. An instance GUID obtained from one machine is unrelated to instance GUIDs on another machine.

guidProduct
Unique identifier which identifies the product. This identifier is established by the manufacturer of the device.

dwDevType
Device type specifier. See the section titled **DirectInput device type description codes** for a description of this field.

tszProductName[MAX_PATH]
Friendly name for the product. For example, "Frobozz Industries SuperStick 5X"

tszInstanceName[MAX_PATH]
Friendly name for the instance. For example, "Joystick 1".

DIDEVICEOBJECTDATA Structure

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

The **DIDEVICEOBJECTDATA** structure is used by the **IDirectInputDevice::GetDeviceData** method to return raw buffered device information.

Members

dwOfs
Offset into the current data format of the object whose data is being reported. In other words, the location where the **dwData** would have been stored if the data had been obtained via **IDirectInputDevice::GetDeviceState**.

For the predefined data formats, the **dwOfs** field will be as follows:

If the device is accessed as a mouse, it will be one of the **DIMOFS_*** values.

If the device is accessed as a keyboard, it will be one of the **DIK_*** values.

If a custom data format has been set, then it will be an offset relative to the custom data format.

dwData
The 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.

DIDFT_AXIS: 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.

DIDFT_BUTTON: Only the low byte of the **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 in milliseconds at which the event was generated. The current system tick count can be obtained by calling the **GetTickCount** system 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 Structure

```
typedef struct {
    DWORD dwSize;
    GUID guidType;
    DWORD dwOfs;
    DWORD dwType;
    DWORD dwFlags;
    TCHAR tszName[MAX_PATH];
} DIDEVICEOBJECTINSTANCE;
```

The **DIDEVICEOBJECTINSTANCE** structure is used by the **IDirectInputDevice::EnumObjects** method to return information about a particular object (axis, button, etc.) on a device to the callback function.

Members

dwSize

The size of the structure in bytes. The application may inspect this value to determine how many fields of the structure are valid. For DirectInput 3.0, the value will be sizeof(DIDEVICEOBJECTINSTANCE). Future versions of DirectInput may return a larger structure.

guidType

Identifier which indicates the type of the object. This field is optional. If present, it may be one of the following values:

GUID_XAxis: This is the horizontal axis of a controller. For example, it may represent the horizontal motion of a mouse.

GUID_YAxis: This is the vertical axis of a controller. For example, it may represent the vertical motion of a mouse.

GUID_ZAxis: This is the forward/backwards axis of a controller. For example, it may represent rotation of the Z-wheel on a mouse.

GUID_Button: This is a button on a mouse.

GUID_Key: This is a key on a keyboard.

Other object types may be defined in the future. (For example, **GUID_Fire**, **GUID_Throttle**, **GUID_SteeringWheel**.)

dwOfs

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

dwType

Device type specifier which describes the object. It is a combination of **DIDFT_*** flags which describe the object type (axis, button, etc.) and contains the object instance number in the high byte. Use the **DIDFT_GETINSTANCE** macro to extract the object instance number.

dwFlags

No flags are currently defined.

tszName[MAX_PATH]

Name of the object. For example, "X-Axis" or "Right Shift".

DIMOFS_BUTTON0 constant

The offset of the mouse button 0 state relative to the beginning of the **DIMOUSESTATE** structure. This value is returned as the *dwOfs* field in the **DIDEVICEOBJECTDATA** structure to indicate that the data applies to mouse button 0.

DIMOFS_BUTTON1 constant

The offset of the mouse button 1 state relative to the beginning of the **DIMOUSESTATE** structure. This value is returned as the *dwOfs* field in the **DIDEVICEOBJECTDATA** structure to indicate that the data applies to mouse button 1.

DIMOFS_BUTTON2 constant

The offset of the mouse button 2 state relative to the beginning of the **DIMOUSESTATE** structure. This value is returned as the *dwOfs* field in the **DIDEVICEOBJECTDATA** structure to indicate that the data applies to mouse button 2.

DIMOFS_BUTTON3 constant

The offset of the mouse button 3 state relative to the beginning of the **DIMOUSESTATE** structure. This value is returned as the *dwOfs* field in the **DIDEVICEOBJECTDATA** structure to indicate that the data applies to mouse button 3.

DIMOFS_X constant

The offset of the mouse x-axis position relative to the beginning of the **DIMOUSESTATE** structure. This value is returned as the *dwOfs* field in the **DIDEVICEOBJECTDATA** structure to indicate that the data applies to the mouse x-axis position.

DIMOFS_Y constant

The offset of the mouse y-axis position relative to the beginning of the **DIMOUSESTATE** structure. This value is returned as the *dwOfs* field in the **DIDEVICEOBJECTDATA** structure to indicate that the data applies to the mouse y-axis position.

DIMOFS_Z constant

The offset of the mouse z-axis position relative to the beginning of the **DIMOUSESTATE** structure. This value is returned as the *dwOfs* field in the **DIDEVICEOBJECTDATA** structure to indicate that the data applies to the mouse z-axis position.

DIMOUSESTATE Structure

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

The **DIMOUSESTATE** structure is used by the **IDirectInputDevice::GetDeviceState** method to return the status of a mouse device or a non-mouse device that is being accessed as if it were a mouse. You must prepare the device for mouse-style access by calling **IDirectInputDevice::SetDataFormat**, passing the *c_dfDIMouse* data format.

Note that 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.

Members

lX

Contains information about the mouse x-axis. If the device is in relative axis mode, then this field contains the change in mouse x-axis position. If the device is in absolute axis mode, then this field contains the absolute mouse x-axis position.

lY

Contains information about the mouse y-axis. If the device is in relative axis mode, then this field contains the change in mouse y-axis position. If the device is in absolute axis mode, then this field contains the absolute mouse y-axis position.

lZ

Contains information about the mouse z-axis. If the device is in relative axis mode, then this field contains the change in mouse z-axis position. If the device is in absolute axis mode, then this field contains the absolute mouse z-axis position.

If the mouse does not have a z-axis, then the value is zero.

rgbButtons[4]

Array of button states. The high-order bit is set if the corresponding button is down.

DIOBJECTDATAFORMAT Structure

```
typedef struct {
    const GUID * pguid;
    DWORD dwOfs;
    DWORD dwType;
    DWORD dwFlags;
} DIOBJECTDATAFORMAT;
```

The **DIOBJECTDATAFORMAT** structure is used by the **IDirectInputDevice::SetDataFormat** method to set the data format for a single object within a device. A data format is made up of several **DIOBJECTDATAFORMAT** structures, one for each object (axis, button, etc). An array of these structures are 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* or *c_dfDIKeyboard*, which has predefined settings for **DIOBJECTDATAFORMAT**.

Members

pguid

The identifier for the axis, button, or other input source. When requesting a data format, leaving this field 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 4 for DWORD size data, such as axes. It can be byte aligned for buttons.

dwType

Device type specifier which describes the object. It is a combination of **DIDFT_*** flags which describe the object type (axis, button, etc.) and contains the object instance number in the high byte. When requesting a data format, the instance portion can be set to **DIDFT_ANYINSTANCE** to indicate that any instance is permissible.

dwFlags

No flags are currently defined. This field must be zero.

Examples

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,                      // Must be zero
};
```

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,                      // Must be zero
};
```

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,                      // Must be zero
};
```

The following object data format specifies that DirectInput should choose the first available "Fire" button and report its value in the high bit of the BYTE at offset 17 in the device data.
If the device does not have a "Fire" button, the attempt to set this data format will fail.

```
DIOBJECTDATAFORMAT dfAnyButton = {
    &GUID_FireButton,       // Object type
    17,                     // Offset
    DIDFT_BUTTON | DIDFT_ANYINSTANCE, // Any button is okay
    0,                      // Must be zero
};
```

The following object data format specifies that button zero 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 zero, the attempt to set this data format will fail.

```
DIOBJECTDATAFORMAT dfButton0 = {
    0,                      // Wildcard
    18,                     // Offset
    DIDFT_BUTTON | DIDFT_MAKEINSTANCE(0), // Button zero
    0,                      // Must be zero
};
```

DIPROP_AXISMODE constant

The **DIPROP_AXISMODE** constant is a predefined property used to set or retrieve the axis data mode. This setting applies to the entire device, rather than to any particular object, so the **dwHow** field must be **DIPH_DEVICE**.

This property uses the **DIPROPDWORD** structure. The **pdiph** field of the **DIPROPDWORD** structure must be a pointer to a **DIPROPHEADER** structure. The **dwData** field contains or receives the axis mode.

The **dwObj** field of the **DIPROPHEADER** structure must be zero, indicating that the property setting applies to the entire device and not to any particular object. The **dwSize** field must be set to the size of the **DIPROPDWORD** structure.

The **dwData** field of the **DIPROPDWORD** structure may be one of the following values:

DIPROPAXISMODE_ABS: Report axis positions in absolute coordinates. Axis motion accumulates over time.
DIPROPAXISMODE_REL: Report axis positions in relative coordinates. Axis motion is reported as differences from the previous request for the axis position.

DIPROP_BUFFERSIZE constant

The **DIPROP_BUFFERSIZE** constant is a predefined property used to set or retrieve the device input buffer size. The buffer size determines the amount of data that the buffer can hold between **GetDeviceData** calls before data is lost. This setting applies to the entire device, rather than to any particular object, so the **dwHow** field must be **DIPH_DEVICE**.

This property uses the **DIPROPDWORD** structure. The **pdiph** field of the **DIPROPDWORD** structure must be a pointer to a **DIPROPHEADER** structure. The **dwData** field contains or receives the buffer size.

The **dwObj** field of the **DIPROPHEADER** structure must be zero, indicating that the property setting applies to the entire device and not to any particular object. The **dwSize** field must be set to the size of the **DIPROPDWORD** structure.

The **dwData** field of the **DIPROPDWORD** structure may be set to zero to indicate that the application will not be reading buffered data from the device. Or it may be a nonzero value to indicate the size of the buffer to be used.

When setting the buffer size, if the buffer size in **dwData** 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_GRANULARITY constant

The **DIPROP_GRANULARITY** constant is a predefined property which retrieves the granularity of an object.

This property uses the **DIPROPDWORD** structure. The **pdiph** field of the **DIPROPDWORD** structure must be a pointer to a **DIPROPHEADER** structure. The **dwData** field receives the granularity.

The **dwObj** field of the **DIPROPHEADER** structure must be the identifier for the object whose granularity is to be retrieved. The **dwSize** field must be set to the size of the **DIPROPDWORD** structure.

The value of the granularity is the smallest distance the object will report movement. Most axis objects have a granularity of 1, meaning that all values are possible. Some axes may have a larger granularity. For example, the Z-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 Z-wheel slowly, the device reports a position of zero, then 20, then 40, etc.

This is a read-only property

DIPROP_RANGE constant

The DIPROP_RANGE constant is a predefined property which retrieves the range of values reported by an object.

This property uses the **DIPROPRANGE** structure. The **pdiph** field of the **DIPROPRANGE** structure must be a pointer to a **DIPROPHEADER** structure.

The **dwObj** field of the **DIPROPHEADER** structure must be the identifier for the object whose range is to be retrieved. The **dwSize** field must be set to the size of the **DIPROPRANGE** structure.

This is a read-only property.

DIPROPDWORD Structure

```
typedef struct {
    DIPROPHEADER    diph;
    DWORD           dwData;
} DIPROPDWORD;
```

Generic structure used to access DWORD properties.

Members

diph

Must be preinitialized as follows:

dwSize = sizeof(DIPROPDWORD).

dwHeaderSize = sizeof(DIPROPHEADER).

dwObj = object identifier.

dwHow = how the **dwObj** should be interpreted.

dwData

On **SetProperty**, this structure contains the value of the property to be set. On **GetProperty**, this structure receives the value of the property.

DIPROPHEADER Structure

```
typedef struct {
    DWORD           dwSize;
    DWORD           dwHeaderSize;
    DWORD           dwObj;
    DWORD           dwHow;
} DIPROPHEADER;
```

DIPROPHEADER is a generic structure which is placed at the beginning of all property structures.

Members

dwSize

Must be set to the size of the enclosing structure.

dwHeaderSize
Must be the size of the **DIPROPHEADER** structure.

dwObj
Identifies the object for which the property is to be accessed.

If the **dwHow** field is **DIPH_DEVICE**, then the **dwObj** field must be zero.

If the **dwHow** field is **DIPH_BYOFFSET**, then the **dwObj** field is the offset into the current data format of the object whose property is being accessed.

If the **dwHow** field is **DIPH_BYID**, then the **dwObj** field is the object type/instance identifier as returned in the *dwType* field of the **DIDEVICEOBJECTINSTANCE** returned from a prior call to **IDirectInputDevice::EnumObjects**.

dwHow
Specifies how the **dwObj** field should be interpreted.

DIPROPRANGE Structure

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

The **DIPROPRANGE** structure is used by the **DIPROP_RANGE** property to set or retrieve the range of an object such as an axis. If the device has an unrestricted range, the reported range will have **lMin** = DIPROPRANGE_NOMIN and **lMax** = DIPROPRANGE_NOMAX. Note that devices with unrestricted range will wrap around.

Members

diph
Must be preinitialized as follows:

dwSize = sizeof(DIPROPRANGE).

dwHeaderSize = sizeof(DIPROPHEADER).

dwObj = object identifier.

dwHow = how the **dwObj** should be interpreted.

lMin
The lower limit of the range, inclusive.

lMax
The upper limit of the range, inclusive.

DISCL_BACKGROUND constant

Parameter to **SetCooperativeLevel** to indicate that background access is desired. If background access is granted, then the device may be acquired at any time, even when the associated window is not the active window.

Exactly one of **DISCL_FOREGROUND** or **DISCL_BACKGROUND** must be passed to **IDirectInputDevice::SetCooperativeLevel**. It is an error to pass both or neither.
Note that the current version of DirectInput does not permit exclusive background access.

DISCL_EXCLUSIVE constant

Parameter to **SetCooperativeLevel** to indicate that exclusive access is desired. If exclusive access is granted, then 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. (The word "exclusive" is a bit of a misnomer here, but it is employed to parallel a similar concept in DirectDraw.)

It is strongly recommended that an application which acquires the mouse or keyboard device in exclusive mode unacquire the devices upon receipt of **WM_ENTERSIZEMOVE** and **WM_ENTERMENULOOP** messages; otherwise, the user will not be able to manipulate the menu or move or resize the window.

Exactly one of **DISCL_EXCLUSIVE** or **DISCL_NONEXCLUSIVE** must be passed to **SetCooperativeLevel**. It is an error to pass both or neither.
In the current version of DirectInput, exclusive access requires foreground access.

DISCL_FOREGROUND constant

Parameter to **SetCooperativeLevel** to indicate that foreground access is desired. If foreground access is granted, then the device is automatically unacquired when the associated window loses foreground activation.

Exactly one of **DISCL_FOREGROUND** or **DISCL_BACKGROUND** must be passed to **IDirectInputDevice::SetCooperativeLevel**. It is an error to pass both or neither.

DISCL_NONEXCLUSIVE constant

Parameter to **SetCooperativeLevel** to indicate that non-exclusive access is desired. Access to the device will not interfere with other applications which are accessing the same device.

Exactly one of **DISCL_EXCLUSIVE** or **DISCL_NONEXCLUSIVE** must be passed to **IDirectInputDevice::SetCooperativeLevel**. It is an error to pass both or neither.

GUID_SysKeyboard global variable

A pre-defined DirectInput instance GUID that always refers to the default system keyboard. This value may be passed to **IDirectInput::CreateDevice** to create an interface to the system keyboard.

GUID_SysMouse global variable

A pre-defined DirectInput instance GUID that always refers to the default system mouse. This value may be passed to **IDirectInput::CreateDevice** to create an interface to the system mouse.

DirectInput Device Type Description Codes

DirectInput device description codes are used in the **DIDEVICEINSTANCE** structure. The least-significant byte of the device type description code specifies the device type.

DIDEVTYPE_MOUSE: A mouse or mouse-like device (such as a trackball).
DIDEVTYPE_KEYBOARD: A keyboard or keyboard-like device.

The next-significant byte specifies the device subtype.

For mouse type devices, the following subtypes are defined:

DIDEVTYPEMOUSE_UNKNOWN: The subtype could not be determined.
DIDEVTYPEMOUSE_TRADITIONAL: A traditional mouse.
DIDEVTYPEMOUSE_FINGERSTICK: A fingerstick.
DIDEVTYPEMOUSE_TOUCHPAD: The device is a touchpad.
DIDEVTYPEMOUSE_TRACKBALL: The device is a trackball.

For keyboard type devices, the following subtypes are defined:

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.

DirectInput Data Format Types

DirectInput data format types describe the attributes of a single object in a device. An object in a device can be an axis, button, or other input source.

DIDFT_ALL

This flag is valid only for **IDirectInputDevice::EnumObjects**. All objects are enumerated, regardless of type. This flag may not be combined with any of the other flags.

DIDFT_RELAXIS

The object is a relative axis. A relative axis is one which reports its data as incremental changes from the previously reported position. Relative axes typically support an unlimited range.

An axis need not report a continuous range of values. For example, an axis may report its position in multiples of 20 indicating that the axis has a granularity of 20. The **DIPROP_GRANULARITY** property of an axis will report the axis granularity.

Relative axis devices do not have absolute coordinates. Rather, the reported absolute coordinates are simply the total of all relative coordinates reported by the device while it has been acquired. As a result, the absolute coordinates retrieved from a relative axis object are meaningful only when compared to other absolute coordinates. For example, an application may record the absolute values when a button is pressed, and retrieve it when the button is released. By subtracting the two, the application can compute the distance between the point the button was pressed and the point the button was released.

Since it is not possible to set or retrieve the origin for absolute values on a relative axis, an application should record the absolute position immediately after acquiring the device. This value then becomes the virtual origin. All subsequent calls can be considered absolute positions based on this origin.

DIDFT_ABSAXIS

The object is an absolute axis. An absolute axis is one that reports data as absolute positions.

Absolute axes typically support a finite range.

An axis need not report a continuous range of values. For example, an axis may report its position in multiples of 20 indicating that the axis has a granularity of 20. The **DIPROP_GRANULARITY** property of an axis will report the axis granularity.

DIDFT_AXIS

This flag is valid only for **IDirectInputDevice::EnumObjects**. All axes are enumerated, regardless of whether they are absolute or relative.

DIDFT_PSHBUTTON

The object is a pushbutton. A pushbutton is reported as down when the user presses it and as up when the user releases it.

DIDFT_TGLBUTTON

The object is a toggle button. A toggle button is reported as down when the user presses it and remains reported as down until the user presses the button a second time.

DIDFT_BUTTON

The object is a either a pushbutton or toggle button.

Chapter 3: DirectInput API & Macro Reference

DIDFT_GETINSTANCE

This macro extracts the object instance number code from a data format type. See **DirectInput Data Format Types** for more information.

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

Parameters

dwType
DirectInput data format type.

DIDFT_GETTYPE

This macro extracts the object type code from a data format type. See **DirectInput Data Format Types** for more information.

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

Parameters

dwType
DirectInput data format type.

DirectInputCreate

This function is called to create a **DirectInput** object which supports the **IDirectInput** COM interface. On success, the function returns a pointer to the new object in *lplpDirectInput.

Calling this function with *punkOuter* = NULL is equivalent to creating the object via **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 via **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 which supports the **IDirectInputA** interface, whereas the UNICODE version creates an object which supports the **IDirectInputW** interface. As with other system services which are sensitive to character set issues, macros in the header file map **DirectInputCreate** to the appropriate character set variation.

```
HRESULT DirectInputCreate(
        HINSTANCE       hinst,
        DWORD           dwVersion,
        LPDIRECTINPUT   * lplpDirectInput,
        LPUNKNOWN       punkOuter)
```

Parameters

hinst
Instance handle of the application or DLL that is creating the DirectInput object.

dwVersion
Version number of the dinput.h header file that was used. This value must be **DIRECTINPUT_VERSION**.
DirectInput uses this value to determine what version of DirectInput the application or DLL was designed for.

lplpDirectInput
Points to where to return the pointer to the **IDirectInput** interface, if successful.

punkOuter
Pointer to controlling unknown for OLE aggregation, or 0 if the interface is not aggregated. Most callers will pass 0.
Note that if aggregation is requested, the object returned in **lplpDirectInput* will be a pointer to an **IUnknown** rather than an **IDirectInput**, as required by OLE aggregation.

Return Values
Returns a COM error code. The following error codes are not necessarily comprehensive.
DI_OK = S_OK: The operation completed successfully.
DIERR_INVALIDPARAM = E_INVALIDARG: The lplpDirectInput parameter is not a valid pointer.
DIERR_OUTOFMEMORY = E_OUTOFMEMORY: Out of memory.
DIERR_OLDDIRECTINPUTVERSION: The application requires a newer version of DirectInput.
DIERR_BETADIRECTINPUTVERSION: The application was written for an unsupported prerelease version of DirectInput.

DISEQUENCE_COMPARE

Macro which compares two DirectInput sequence numbers, compensating for wraparound.

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

Parameters

dwSequence1
First sequence number to compare.

cmp
One of the following comparison operators: "==", "!=", "<", ">", "<=", ">=".

dwSequence2
Second sequence number to compare.

Return Values

Returns a nonzero value if the first sequence number is equal to, is not equal to, chronologically precedes, chronologically follows, chronologically precedes or is equal to, or chronologically follows or is equal to the second sequence number.

Example

The following example checks whether *dwSequence1* precedes *dwSequence2* chronologically:

```
if (DISEQUENCE_COMPARE(dwSequence1, <, dwSequence2)) {
```

```
                ...
}
```

The following example checks whether *dwSequence1* chronologically follows or is equal to *dwSequence2*:

```
if (DISEQUENCE_COMPARE(dwSequence1, >=, dwSequence2)) {
                ...
}
```

GET_DIDEVICE_SUBTYPE

This macro extracts the device subtype code from a device type description code. Note that the interpretation of the subtype code depends on the device primary type. See **DirectInput Device Type Description Codes** for more information.

BYTE GET_DIDEVICE_SUBTYPE(
DWORD dwDevType)

Parameters

dwDevType
DirectInput device type description code.

GET_DIDEVICE_TYPE

This macro extracts the device type code from a device type description code. See **DirectInput Device Type Description Codes** for more information.

BYTE GET_DIDEVICE_TYPE(
DWORD dwDevType)

Parameters

dwDevType
DirectInput device type description code.

MAKEDIPROP

Helper macro which creates an integer property.

Integer properties are defined by Microsoft. Vendors which wish to implement custom properties should use GUIDs.

Chapter 4: ClassFactory Interface Methods

Overview

The IClassFactory interface is required for OLE support. See the OLE documentation for more information. Most applications which use DirectInput will not need to communicate directly with the OLE class factory.

IClassFactory Interface

Member
AddRef(…)
CreateInstance(…)
LockServer(…)
QueryInterface(…)
Release(…)

Members

AddRef

Priority: 1

Increments the reference count for the interface. See the OLE documentation for **IUnknown::AddRef**.

```
HRESULT AddRef
        (LPCLASSFACTORY lpClassFactory)
```

Return Values

Returns the object reference count.

CreateInstance

This function creates a new DirectInput object with the specified interface. See the OLE documentation for **IClassFactory::CreateInstance**. Note that the newly-created object has not been initialized.

```
HRESULT CreateInstance
        (LPCLASSFACTORY   lpClassFactory,
        LPUNKNOWN         punkOuter,
        REFIID                  riid,
        LPVOID            * ppvOut)
```

Parameters

punkOuter

Pointer to controlling unknown for OLE aggregation, or 0 if the interface is not aggregated. Most callers will pass 0.

riid

Desired interface. This parameter must point to a valid interface identifier.

ppvOut

Points to where to return the pointer to the created interface, if successful.

Return Values

Returns a COM error code. The following error codes are not necessarily comprehensive.

S_OK: The operation completed successfully.
E_INVALIDARG: The *ppvOut* parameter is not a valid pointer.
CLASS_E_NOAGGREGATION: Aggregation not supported.
E_OUTOFMEMORY: Out of memory.
E_NOINTERFACE: The specified interface is not supported.

LockServer

This function increments or decrements the DLL lock count. While the DLL lock count is nonzero, it will not be removed from memory. See the OLE documentation for **IClassFactory::LockServer**.

```
HRESULT LockServer
        (LPCLASSFACTORY   lpClassFactory,
         BOOL             fLock)
```

Parameters

fLock
If **TRUE**, increments the lock count. If **FALSE**, decrements the lock count.

Return Values
Returns a COM error code. The following error codes are not necessarily comprehensive.
S_OK: The operation completed successfully.
E_OUTOFMEMORY: Out of memory.

QueryInterface

Gives a client access to other interfaces on an object. See the OLE documentation for **IUnknown::QueryInterface**.

```
HRESULT QueryInterface
        (LPCLASSFACTORY lpClassFactory,
        REFIID riid,
        LPVOID * ppvObj)
```

Parameters

riid
The requested interface's IID.

ppvObj
Receives a pointer to the obtained interface.

Return Values
Returns a COM error code.

Release

Decrements the reference count for the interface. If the reference count on the object falls to zero, the object is freed from memory. See the OLE documentation for **IUnknown::Release**.

```
HRESULT Release
        (LPCLASSFACTORY lpClassFactory)
```

Return Values

Returns the object reference count.

Chapter 5: DirectInput Interface Reference

Overview

The **DirectInput** object represents the DirectInput subsystem. It creates the **DirectInputDevice** object which represents a single input device.

IDirectInput Interface

Member
AddRef(...)
CreateDevice(…)
EnumDevices(…)
GetDeviceStatus(…)
Initialize(…)
QueryInterface(...)
Release(...)
RunControlPanel(…)

Members

AddRef

This member is part of the **IUnknown** interface inherited by **IDirectInput**. It is used to increase the reference count on the associate COM object. When the object is initially created, its reference count is set to one. Each time **AddRef** is called the reference count is incremented, and each time **Release** is called the reference count is decremented. The object deallocates itself when its reference count reaches zero.

DWORD AddRef
(LPDIRECTINPUT lpDirectInput)

Parameters

lpDirectInput
Points to the DirectInput object that this member is being called for.

Return Value
A DWORD containing the new reference count.

CreateDevice

Creates and initializes an instance of a device which is specified by the GUID. Calling this function with *punkOuter* = NULL is equivalent to creating the object via **CoCreateInstance**(&CLSID_DirectInputDevice, NULL, CLSCTX_INPROC_SERVER, *riid*, *lplpDirectInputDevice*); 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.

HRESULT CreateDevice

```
(LPDIRECTINPUT       lpDirectInput,
REFGUID              rguid,
LPDIRECTINPUTDEVICE  *lplpDirectInputDevice,
LPUNKNOWN            * pUnkOuter)
```

Parameters

lpDirectInput
Points to the DirectInput object that this member is being called for.

rguid
Reference to the **GUID** representing the desired input device. The GUID is retrieved through the **EnumDevices** method, or it can be one of the predefined GUIDs.

lplpDirectInputDevice
Points to the **IDirectInputDevice** interface pointer if successful.

pUnkOuter
Pointer to controlling unknown for OLE aggregation, or 0 if the interface is not aggregated. Most callers will pass 0.

Return Value
Returns a COM error code. The following error codes are not necessarily comprehensive.
DI_OK = S_OK: The operation completed successfully.
DIERR_INVALIDPARAM = E_INVALIDARG: The *ppvOut* parameter is not a valid pointer.
DIERR_OUTOFMEMORY = E_OUTOFMEMORY: Out of memory.
DIERR_NOINTERFACE = E_NOINTERFACE: The specified interface is not supported by the object.
DIERR_DEVICENOTREG: The device instance does not correspond to a device that is registered with DirectInput.

EnumDevices

This member is used to enumerate devices that are either currently attached or could be attached to the computer. For example, a flight stick may be installed on the system but not currently plugged in to the computer. A flag is set in the dwFlags parameter to indicate whether only attached or all installed devices should be enumerated. If the flag (**DIEDFL_ATTACHEDONLY**) is not present, all installed devices will be enumerated. A preferred device type can be passed as a filter so that only the devices of that type are enumerated.

An application-defined callback function is passed to **IDirectInput::EnumDevices** in the lpCallback parameter. This function is called 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.) For this release of DirectInput, only mouse and keyboard devices will be enumerated.

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

Parameters

lpDirectInput

Points to the DirectInput object that this member is being called for.

dwDevType

Device type filter. If 0, then all device types are enumerated. Otherwise, it is a **DIDEVTYPE_*** value, indicating the device type that should be enumerated. For this release of DirectInput, only mouse devices and keyboards are enumerated.

lpCallback

Points to an application-defined callback function that will be called with a description of each DirectInput device.

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

lpddi

Pointer to the structure that describes this device instance.

pvRef

Points to application-defined data given to **EnumDevices**.

Return Value

DIENUM_CONTINUE	**Continue the enumeration**
DIENUM_STOP	**Stop the enumeration**

pvRef

Points to a caller-defined 32 bit context that will be passed to the enumeration callback each time it is called.

dwFlags

Only one flag is currently defined.

DIEDFL_ATTACHEDONLY- Only enumerates devices that are currently attached.

Return Values

Returns a COM error code. The following error codes are not necessarily comprehensive.

DI_OK = **S_OK**: The operation completed successfully. Note that if the callback stops the enumeration prematurely, the enumeration is considered to have succeeded.

DIERR_INVALIDPARAM = **E_INVALIDARG**: The *fl* parameter contains invalid flags, or the callback procedure returned an invalid status code.

GetDeviceStatus

This member checks to see if the specified device is currently attached to DirectInput. Returns OK if device is attached or error if not attached.

```
HRESULT GetDeviceStatus
    (LPDIRECTINPUT      lpDirectInput,
    REFGUID             rguidInstance)
```

Parameters

lpDirectInput

Points to the DirectInput object that this member is being called for.

rguidInstance

Identifies the instance of the device whose status is being checked.

Return Values

Returns a COM error code. The following error codes are not necessarily comprehensive.

DI_OK = **S_OK**: The device is attached.

DI_NOTATTACHED = **S_FALSE**: The device is not attached.

E_FAIL: DirectInput could not determine whether the device is attached.

DIERR_INVALIDPARAM = **E_INVALIDARG**: The device does not exist.

Initialize

Initialize a DirectInput object. The **DirectInputCreate** method automatically initializes the DirectInput object device after creating it. Applications normally do not need to call this function.

```
HRESULT Initialize(
        LPDIRECTINPUT lpDirectInput,
        HINSTANCE hinst,
        DWORD dwVersion)
```

Parameters

hinst
Instance handle of the application or DLL that is creating the DirectInput object.

dwVersion
Version number of the dinput.h header file that was used. This value must be **DIRECTINPUT_VERSION**.
DirectInput uses this value to determine what version of DirectInput the application or DLL was designed for.

Return Values

Returns a COM error code. The following error codes are not necessarily comprehensive.
DI_OK = **S_OK**: The device is attached.
DIERR_DIERR_OLDDIRECTINPUTVERSION: The application requires a newer version of DirectInput.
DIERR_DIERR_BETADIRECTINPUTVERSION: The application was written for an unsupported prerelease version of DirectInput.

QueryInterface

This member is part of the **IUnknown** interface inherited by **IDirectInput**. This is the member that applications use to determine whether the object supports additional interfaces that they may be interested in. An application can ask the object if it supports a particular **COM** interface and if it does the application may begin using that interface immediately. If the requested interface is supported, a pointer to it is returned to the application in the **ppvObj** parameter. If the application does not want to use that interface or is finished with the interface it must call **Release** to free it. This member allows **DirectInput** objects to be extended by Microsoft without breaking, or interfering with, each others existing or future functionality. For additional information, see the OLE documentation for **IUnknown::QueryInterface**.

```
HRESULT QueryInterface
        (LPDIRECTINPUT lpDirectInput,
        REFIID riid,
        LPVOID FAR* ppvObj)
```

Parameters

lpDirectInput
Points to the DirectInput object that this member is being called for.

riid
Points to the interface id (IID) identifying the requested interface.

ppvObj

Points to a location that will be filled with the returned interface pointer if the query is successful.

Return Values

DI_OK
DIERR_INVALIDPARAM
DIERR_NOINTERFACE

Release

This member is part of the **IUnknown** interface inherited by **IDirectInput**. It is used to decrease the reference count on the associated COM object. When the object is initially created its reference count is set to one. Each time **AddRef** is called the reference count is incremented, and each time **Release** is called the reference count is decremented. The object deallocates itself when its reference count reaches zero. For additional information, see the OLE documentation for **IUnknown::QueryInterface**.

```
DWORD Release
    (LPDIRECTINPUT lpDirectInput)
```

Parameters

lpDirectInput
Points to the DirectInput object that this member is being called for.

Return Value
A DWORD containing the new reference count. Note that this value should be used only or debugging purposes.

RunControlPanel

This member is used to run the Windows' DirectInput control panel so that the user can install a new input device or modify the setup. This member will not run third party control panels.

```
HRESULT RunControlPanel
    (LPDIRECTINPUT    lpDirectInput,
    HWND              hwndOwner,
    DWORD             dwFlags)
```

Parameters

lpDirectInput
Points to the DirectInput object that this member is being called for.

hwndOwner
Identifies the windows handle that will be used as the parent window for the subsequent UI. NULL is a valid parameter, indicating that there is no parent window.

dwFlags
No flags currently defined. This parameter must be zero.

Return Values
Returns a COM error code. The following error codes are not necessarily comprehensive.
DI_OK = **S_OK**: The device is attached.

Chapter 6: DirectInputDevice Interface Reference

Overview

The IDirectInputDevice interface is used to retrieve data from an instance of an input device. The device can be either a mouse device or keyboard device in this release of DirectInput.

Interface

Member
Acquire(…)
AddRef(...)
EnumObjects(…)
GetCapabilities(…)
GetDeviceData(…)
GetDeviceInfo(…)
GetDeviceState(…)
GetObjectInfo(…)
GetProperty(…)
Initialize(…)
QueryInterface(…)
Release(...)
RunControlPanel(…)
SetCooperativeLevel(…)
SetDataFormat(…)
SetEventNotification(…)
SetProperty(…)
Unacquire(…)

Members

Acquire

Obtains access to the input device. A device must be acquired before **GetDeviceState** or **GetDeviceData** can be called on the device. Device acquisition does not have a reference count. If a device is acquired twice then unacquired once, the device is unacquired.

HRESULT Acquire
(LPDIRECTINPUTDEVICE lpDirectInputDevice)

Parameters

lpDirectInputDevice
Points to the DirectInput device object that this member is being called for.

Return Values
Returns a COM error code. The following error codes are not necessarily comprehensive.
DI_OK = **S_OK**: The operation completed successfully.
S_FALSE: The device has already been acquired. Note that this value is a success code.
DIERR_INPUTLOST: Access to the device was not granted.

DIERR_INVALIDPARAM = **E_INVALIDARG**: The device does not have a selected data format.

AddRef

This member is part of the **IUnknown** interface inherited by **IDirectInputDevice**. It is used to increase the reference count on the associated COM object. When the object is initially created, its reference count is set to one. Each time **AddRef** is called the reference count is incremented, and each time **Release** is called the reference count is decremented. The object deallocates itself when its reference count reaches zero. See the OLE documentation on **IUnknown::AddRef** for additional information.

```
DWORD AddRef
    (LPDIRECTINPUTDEVICE lpDirectInputDevice)
```

Parameters

lpDirectInputDevice
Points to the DirectInput device object that this member is being called for.

Return Value
A DWORD containing the new reference count.

GetCapabilities

Obtains information about the input device.

```
HRESULT GetCapabilities
    (LPDIRECTINPUTDEVICE    lpDirectInputDevice,
    LPDIDEVCAPS             lpDIDevCaps)
```

Parameters

lpDirectInputDevice
Points to the DirectInput device object that this member is being called for.

lpDIDevCaps
Points to a **DIDEVCAPS** structure that is filled in by the function. The **dwSize** field must be filled in by the application before calling this method. See the documentation of the **DIDEVCAPS** structure for additional information.

Return Value
Returns a COM error code. The following error codes are not necessarily comprehensive.
DI_OK = **S_OK**: The operation completed successfully.
DIERR_INVALIDPARAM = E_INVALIDARG: The lpDIDevCaps parameter is not a valid pointer.

GetDeviceData

Obtains buffered data from the DirectInput device.
Before device data can be obtained, the cooperative level must be set via **SetCooperativeLevel**, the data format must be set via **SetDataFormat**, and the device must be acquired via **Acquire**.

```
HRESULT GetDeviceData(
    LPDIRECTINPUTDEVICE     lpDirectInputDevice,
    DWORD cbObjectData,
    LPDIDEVICEOBJECTDATA rgdod,
    LPDWORD pdwInOut,
    DWORD fl)
```

Parameters

lpDirectInputDevice
Points to the DirectInput device object that this member is being called for.

cbObjectData
The size of the **DIDEVICEOBJECTDATA** structure in bytes.

rgdod
Array of **DIDEVICEOBJECTDATA** structures to receive the buffered data. It must consist of **pdwInOut* elements.
If this parameter is **NULL**, then the buffered data is not stored anywhere, but all other side-effects take place.

pdwInOut
On entry, contains the number of elements in the array pointed to by *rgdod*. On exit, contains the number of elements actually obtained.

fl
Flags which control the manner in which data is obtained. It may be zero or more of the following flags:

DIGDD_PEEK: Do not remove the items from the buffer. A subsequent **GetDeviceData** will read the same data. Normally, data is removed from the buffer after it is read.

Return Values

DI_OK = **S_OK**: All data were retrieved successfully. Note that the application needs to check the output value of **pdwInOut* to determine whether and how much data was retrieved: The value may be zero, indicating that the buffer was empty.

DI_BUFFEROVERFLOW = **S_FALSE**: Some data were retrieved successfully, but some data were lost because the device's buffer size was not large enough. The application should retrieve buffered data more frequently or increase the device buffer size. This status code is returned only on the first **IDirectInputDevice::GetDeviceData** call after the buffer has overflowed. Note that this is a success status code.

DIERR_NOTACQUIRED: The device is not acquired.

DIERR_INPUTLOST: Access to the device has been interrupted. The application should reacquire the device.

DIERR_INVALIDPARAM = **E_INVALIDARG**: One or more parameters was invalid.

Example

The following sample 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)) {
   // Buffer successfully flushed.
   // dwItems = number of elements flushed
   if (hres == DI_BUFFEROVERFLOW) {
```

```
        // Buffer had overflowed.
    }
}
```

If you pass **NULL** for the *rgdod* and request an infinite number of items, this has the effect of flushing the buffer and returning the number of items that were flushed.

```
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.
             }
}
```

If you pass **NULL** for the *rgdod*, request an infinite number of items, and ask that the data not be removed from the device buffer, this has the effect of querying for the number of elements in the device buffer.

```
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.
             }
}
```

If you pass **NULL** for the *rgdod* and request zero items, this has the effect of querying whether buffer overflow has occurred.

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

GetDeviceInfo

Obtains information about the device's identity.

```
HRESULT GetDeviceInfo(
    LPDIRECTINPUTDEVICE     lpDirectInputDevice,
    LPDIDEVICEINSTANCE      pdidi)
```

Parameters

lpDirectInputDevice
Points to the DirectInput device object that this member is being called for.

pdidi
Receives information about the device's identity. The caller must initialize the dwSize field of the **DIDEVICEINSTANCE** structure before calling this method.

Return Values

Returns a COM error code. The following error codes are not necessarily comprehensive.
DI_OK = **S_OK**: The operation completed successfully.
DIERR_INVALIDPARAM = **E_INVALIDARG**: One or more parameters was invalid.

GetDeviceState

Obtains instantaneous data from the DirectInput device.
Before device data can be obtained, the cooperative level must be set via **SetCooperativeLevel**, the data format must be set via **SetDataFormat**, and the device must be acquired via **Acquire**.

```
HRESULT GetDeviceState(
    LPDIRECTINPUTDEVICE     lpDirectInputDevice,
    DWORD cbData,
    LPVOID lpvData)
```

Parameters

lpDirectInputDevice
Points to the DirectInput device object that this member is being called for.

cbData
The size of the buffer pointed to by *lpvData*, in bytes.

lpvData
Points to a structure that receives the current state of the device. The format of the data is established by a prior call to **SetDataFormat**.

Return Values

Returns a COM error code. The following error codes are not necessarily comprehensive.
DI_OK = **S_OK**: The operation completed successfully.

E_PENDING: The device does not have data yet. Some devices (such as USB joysticks) require a delay between the time the device is turned on and the time the device begins sending data. During this warm-up time, **GetDeviceState** will return **E_PENDING**. When data becomes available, the event notification handle will be signaled.

DIERR_NOTACQUIRED: The device is not acquired.

DIERR_INPUTLOST: Access to the device has been interrupted. The application should reacquire the device.

DIERR_INVALIDPARAM = **E_INVALIDARG**: The *lpvData* parameter is not a valid pointer or the *cbData* parameter does not match the data size set by a previous call to **SetDataFormat**.

GetObjectInfo

Obtains information about an object.

```
HRESULT GetObjectInfo(
    LPDIRECTINPUTDEVICE       lpDirectInputDevice,
    LPDIDEVICEOBJECTINSTANCE pdidoi,
    DWORD dwObj,
    DWORD dwHow)
```

Parameters

lpDirectInputDevice
Points to the DirectInput device object that this member is being called for.

pdidoi
Receives information about the object. The caller must initialize the dwSize field of the **DIDEVICEOBJECTINSTANCE** structure before calling this method.

dwObj
Identifies the object for which the property is to be accessed. See the documentation of the **DIPROPHEADER** structure for additional information.

dwHow
Identifies how *dwObj* is to be interpreted. See the documentation of the **DIPROPHEADER** structure for additional information.

Return Values

Returns a COM error code. The following error codes are intended to be illustrative and not necessarily comprehensive.
DI_OK = **S_OK**: The operation completed successfully.
DIERR_INVALIDPARAM = **E_INVALIDARG**: One or more parameters was invalid.
DIERR_OBJECTNOTFOUND: The specified object does not exist.

GetProperty

This member retrieves information about the input device. Some properties can be set with a call to the **IDirectInputDevice::SetProperty** method; others are read-only. See the **IDirectInputDevice::SetProperty** method for a list of settable properties.

```
HRESULT GetProperty
    (LPDIRECTINPUTDEVICE    lpDirectInputDevice,
    REFGUID                 rguidProp,
    LPDIPROPHEADER          pdiph)
```

Parameters

lpDirectInputDevice
Points to the DirectInput device object that this member is being called for.

rguidProp
The identity of the property being retrieved. This can be one of the predefined **DIPROP_*** values, or it may be a pointer (reference, if using C++) to a GUID that identifies the property.
The following properties are predefined for an input device:

DIPROP_AXISMODE
DIPROP_BUFFERSIZE
DIPROP_GRANULARITY
DIPROP_RANGE
See the individual property descriptions in the **Structures and Constants** chapter for more information about each of these properties.

pdiph
Points to the **DIPROPHEADER** portion of a structure which depends on the property.

Return Value
Returns a COM error code. The following error codes are not necessarily comprehensive.
DI_OK = **S_OK**: The operation completed successfully.
DIERR_INVALIDPARAM = **E_INVALIDARG**: The *pdiph* parameter is not a valid pointer, or the *dwHow* field is invalid, or the *dwObj* field is not zero when *dwHow* is set to **DIPH_DEVICE**.
DIERR_OBJECTNOTFOUND: The specified object does not exist.
DIERR_UNSUPPORTED = **E_NOTIMPL**: The property is not supported by the device or object.

Example

The following "C" code fragment illustrates how to obtain the value of the **DIPROP_BUFFERSIZE** property.

```
DIPROPDWORD dipdw;
HRESULT hres;
dipdw.diph.dwSize = sizeof(DIPROPDWORD);
dipdw.diph.dwHeaderSize = sizeof(DIPROPHEADER);
dipdw.diph.dwObj  = 0;                    // device property
dipdw.diph.dwHow = DIPH_DEVICE;
hres = IDirectInputDevice_GetProperty(pdid, DIPROP_BUFFERSIZE,
&dipdw.diph);
if (SUCCEEDED(hres)) {
            // dipdw.dwData contains the value of the property
}
```

EnumObjects

Enumerate the input sources (axes, buttons, etc.) available on the input device.

HRESULT EnumObjects(

LPDIRECTINPUTDEVICE	**lpDirectInputDevice,**
LPDIENUMDEVICEOBJECTSCALLBACK	**lpCallback,**
LPVOID	**pvRef,**
DWORD	**fl)**

Parameters

lpDirectInputDevice
Points to the DirectInput device object that this member is being called for.

lpCallback
Points to an application-defined callback function that receives DirectInputDevice objects.

BOOL CALLBACK DIEnumDeviceObjectsProc(

LPCDIDEVICEOBJECTINSTANCE	**lpddoi,**
LPVOID	**pvRef)**

Parameters

lpddoi
A **DIDEVICEOBJECTINSTANCE** structure which describes the object being enumerated.

pvRef
Specifies the application-defined value given in the **IDirectInputDevice::EnumObjects** function.

Return Values

DIENUM_CONTINUE	**Continue the enumeration**
DIENUM_STOP	**Stop the enumeration**

pvRef
Reference data (context) for callback.

fl
Flags specifying the type(s) of objects to be enumerated. It may be a combination of the data format types. See **DirectInput Data Format Types**.

Return Values

Returns a COM error code. The following error codes are not necessarily comprehensive.
DI_OK = **S_OK**: The operation completed successfully. Note that if the callback stops the enumeration prematurely, the enumeration is considered to have succeeded.
DIERR_INVALIDPARAM = **E_INVALIDARG**: The *fl* parameter contains invalid flags, or the callback procedure returned an invalid status code.

Initialize

Initialize a DirectInputDevice object.

Note that if this method fails, the underlying object should be considered to be an an indeterminate state and needs to be reinitialized before it can be subsequently used.

The **IDirectInput::CreateDevice** method automatically initializes the device after creating it. Applications normally do not need to call this function.

```
HRESULT Initialize(
    LPDIRECTINPUTDEVICE     lpDirectInputDevice,
    HINSTANCE               hinst,
    DWORD                   dwVersion,
    REFGUID                 rguid)
```

Parameters

lpDirectInputDevice
Points to the DirectInput device object that this member is being called for.

hinst
Instance handle of the application or DLL that is creating the DirectInputDevice object. DirectInput uses this value to determine whether the application or DLL has been certified.

dwVersion
Version number of the dinput.h header file that was used. This value must be **DIRECTINPUT_VERSION**.
DirectInput uses this value to determine what version of DirectInput the application or DLL was designed for.

rguid
Identifies 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

Returns a COM error code. The following error codes are not necessarily comprehensive.
DI_OK = **S_OK**: The device is attached.
DIERR_DIERR_OLDDIRECTINPUTVERSION: The application requires a newer version of DirectInput.
DIERR_DIERR_BETADIRECTINPUTVERSION: The application was written for an unsupported prerelease version of DirectInput.
S_FALSE: The device had already been initialized with the instance GUID passed in **rguid**.
DIERR_ACQUIRED: The device cannot be initialized while it is acquired.

QueryInterface

This member is part of the **IUnknown** interface inherited by **IDirectInputDevice**. This is the member that applications use to determine whether the object supports additional interfaces that they may be interested in. An application can ask the object if it supports a particular **COM** interface and if it does the application may begin using that interface immediately. If the requested interface is supported, a pointer to it is returned to the application in the **ppvObj** parameter. If the application does not want to use that interface or is finished with the interface, it must call **Release** to free it. This member allows **DirectInput** objects to be extended by Microsoft without breaking, or interfering with, each others existing or future functionality. For more information, see the OLE documentation for **IUnknown::QueryInterface**.

```
HRESULT QueryInterface
    (LPDIRECTINPUTDEVICE lpDirectInputDevice,
    REFIID riid,
    LPVOID FAR* ppvObj)
```

Parameters

lpDirectInputDevice
Points to the DirectInput device object that this member is being called for.

riid
Points to the interface ID (IID) identifying the requested interface.

ppvObj
Points to a location that will be filled with the returned interface pointer if the query is successful.

Return Values
DI_OK
DIERR_INVALIDPARAM
DIERR_NOINTERFACE

Release

This member is part of the **IUnknown** interface inherited by **IDirectInputDevice**. It is used to decrease the reference count on the associated COM object. When the object is initially created its reference count is set to one. Each time **AddRef** is called the reference count is incremented, and each time **Release** is called the reference count is decremented. The object deallocates itself when its reference count reaches zero. For more information, see the OLE documentation for **IUnknown::QueryInterface**.

```
DWORD Release
    (LPDIRECTINPUTDEVICE lpDirectInputDevice)
```

Parameters

lpDirectInputDevice
Points to the DirectInput device object that this member is being called for.

Return Value
A DWORD containing the new reference count. Note that the return value should be used only for debugging purposes.

RunControlPanel

This member runs the 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
    (LPDIRECTINPUTDEVICE    lpDirectInputDevice,
    HWND                    hwndOwner,
    DWORD                   dwFlags)
```

Parameters

lpDirectInputDevice
Points to the DirectInput device object that this member is being called for.

hwndOwner
Identifies the window handle that will be used as the parent window for subsequent UI. NULL is a valid parameter, indicating that there is no parent window.

dwFlags
No flags currently defined. This parameter must be zero.

Return Values
Returns a COM error code. The following error codes are not necessarily comprehensive.
DI_OK = **S_OK**: The operation completed successfully.

SetCooperativeLevel

Establish the cooperativity level for the instance of the device. The cooperativity level determines how the instance of the device interacts with other instances of the device and the rest of the system.

Note that if the system mouse is acquired in exclusive mode, then the mouse cursor will be removed from the screen until the device is unacquired.

You must call this method before acquiring the device via **Acquire**.

```
HRESULT SetCooperativeLevel(
    LPDIRECTINPUTDEVICE    lpDirectInputDevice,
    HWND                   hwnd,
    DWORD                  dwFlags)
```

Parameters

hwnd

The window associated with the device. This parameter must be non-NULL if the **DISCL_FOREGROUND** flag is passed. The window must be a top-level window. It is an error to destroy the window while it is still active in a DirectInput device.

dwFlags

Flags which describe the cooperativity level associated with the device. The **DISCL_*** flags are documented separately.

Return Values

Returns a COM error code. The following error codes are not necessarily comprehensive.
DI_OK = **S_OK**: The operation completed successfully.
DIERR_INVALIDPARAM = **E_INVALIDARG**: The *hwnd* parameter is not a valid window handle, or invalid flags or combinations of flags were passed.

SetDataFormat

Set the data format for the DirectInput device. The data format must be set before the device can be acquired. It is necessary to set the data format only once. The data format may not be changed while the device is acquired. If the attempt to set the data format fails, all data format information is lost, and a valid data format must be set before the device may be acquired. Applications will typically use one of the predefined data formats: *c_dfDIMouse* or *c_dfDIKeyboard*.

```
HRESULT SetDataFormat(
    LPDIRECTINPUTDEVICE     lpDirectInputDevice,
    LPCDIDATAFORMAT         lpdf)
```

Parameters

lpDirectInputDevice

Points to the DirectInput device object that this member is being called for.

lpdf

Points to a structure that describes the format of the data the DirectInputDevice should return.

Return Values

Returns a COM error code. The following error codes are intended to be illustrative and not necessarily comprehensive.
DI_OK = **S_OK**: The operation completed successfully.
DIERR_INVALIDPARAM = **E_INVALIDARG**: The *lpdf* parameter is not a valid pointer.
DIERR_ACQUIRED: Cannot change the data format while the device is acquired.

SetEventNotification

This member specifies an event that is to be set when the device state changes. It is also used to turn off event notification.

It is an error to call **CloseHandle** on the event while it has been selected into an **IDirectInputDevice** object. You must call **IDirectInputDevice::SetEventNotification** 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. Examples of usage are shown below. For additional information on using Win32 wait functions, see the Win32 SDK and related documentation.

HRESULT SetEventNotification
(LPDIRECTINPUTDEVICE lpDirectInputDevice,
HANDLE hEvent)

Parameters

lpDirectInputDevice
Points to the DirectInput device object that this member is being called for.

hEvent
Handle to the event that is to be set when the device state changes. It must be an event handle. DirectInput will **SetEvent** the handle when the state of the device changes.

The application should create the handle as a manual-reset event via the **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 **ResetEvent** to reset it. DirectInput will not call **ResetEvent** for event notification handles. Most applications will create the event as an automatic-reset event.

If the *hEvent* is NULL, then notification is disabled.

Return Value
Returns a COM error code. The following error codes are not necessarily comprehensive.
DI_OK = **S_OK**: The operation completed successfully.

DIERR_ACQUIRED: The **IDirectInputDevice** object has been acquired. You must call **IDirectInputDevice::Unacquire** to unacquire the device before you can change the notification state.

DIERR_HANDLEEXISTS: The **IDirectInputDevice** object already has an event notification handle associated with it. DirectInput supports only one event notification handle per **IDirectInputDevice** object.

E_INVALIDARG:Not an event handle.

Example

To check 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;
                  }
        }
```

SetProperty

This member is used to set properties that define the device behavior. These properties which can be set include input buffer size and axis mode. The current value of these properties can be retrieved with a call to the **IDirectInputDevice::GetProperty** method.

HRESULT SetProperty

(LPDIRECTINPUTDEVICE	**lpDirectInputDevice,**
REFGUID	**rguid,**
LPCDIPROPHEADER	**pdiph)**

Parameters

lpDirectInputDevice

Points to the DirectInput device object that this member is being called for.

rguidProp

The identity of the property being set. This can be one of the predefined **DIPROP_*** values, or it may be a pointer (reference, if C++) to a GUID that identifies the property. The following properties are predefined and settable for a device:

DIPROP_AXISMODE
DIPROP_BUFFERSIZE
See the individual property descriptions in the **Structures and Constants** chapter for more information about each of these properties.

pdiph
Points to the **DIPROPHEADER** portion of a structure which depends on the property.

Return Value
Returns a COM error code. The following error codes are not necessarily comprehensive.
DI_OK = **S_OK**: The operation completed successfully.
DI_PROPNOEFFECT = **S_FALSE**: The operation completed successfully but had no effect. For example, changing the axis mode on a device with no axes will return this value.
DIERR_INVALIDPARAM = **E_INVALIDARG**: The *pdiph* parameter is not a valid pointer, or the *dwHow* field is invalid, or the *dwObj* field is not zero when *dwHow* is set to **DIPH_DEVICE**.
DIERR_OBJECTNOTFOUND: The specified object does not exist.
DIERR_UNSUPPORTED = **E_NOTIMPL**: The property is not supported by the device or object.

Unacquire

Release access to the device.

HRESULT Unacquire
(LPDIRECTINPUTDEVICE lpDirectInputDevice)

Parameters

lpDirectInputDevice
Points to the DirectInput device object that this member is being called for.

Return Values
Returns a COM error code. The following error codes are not necessarily comprehensive.
DI_OK = **S_OK**: The operation completed successfully.
S_FALSE: The object is not currently acquired. This may have been caused by a prior loss of input. Note that this is a success code.

Appendix A: Japanese Keyboards

There are substantial differences between Japanese and US 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 US 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. They 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, F14, 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