DirectSetup

[This is preliminary documentation and subject to change.]

This section provides information about the DirectSetup component of the DirectX® application programming interface (API) in the Platform Software Development Kit (SDK). Information is found under the following main headings:

- About DirectSetup
- DirectSetup Essentials
- DirectSetup Reference
- DirectSetup Samples

About DirectSetup

[This is preliminary documentation and subject to change.]

DirectSetup is a simple application programming interface (API) that provides you with one-call installation of the DirectX components. This is more than a convenience—DirectX is a complex product, and installing it is an involved task. You should not attempt to install DirectX manually.

DirectSetup also provides an automated way to install the appropriate Microsoft® Windows® registry information for applications that use the DirectPlayLobby object. This registry information is required for the DirectPlayLobby object to enumerate and start the application.

DirectSetup includes the following API functions:

DirectXRegisterApplication

DirectXUnRegisterApplication

DirectXSetup

DirectXSetupSetCallback

It also includes a prototype **DirectXSetupCallbackFunction** that can be defined by an application in order to customize the user interface during the installation process. The DirectX Programmer's Reference includes a sample application that demonstrates the use of this callback function; for more information, see Dinstall.

DirectSetup Essentials

[This is preliminary documentation and subject to change.]

This section contains general information about the DirectSetup component. The following topics are discussed:

- Setup Overview for DirectSetup
- Preparing the Setup Folder
- Customizing Setup
- Testing the Setup Program
- Enabling AutoPlay

Setup Overview for DirectSetup

[This is preliminary documentation and subject to change.]

Applications and games that depend on DirectX use the **DirectXSetup** function to install the necessary system components into an existing Windows installation. The function optionally updates the display and audio drivers for optimal support of DirectX. Typically you would call **DirectXSetup** from the program you are using to install your own application files.

Note

The **DirectXSetup** function overwrites system components from previous versions of DirectX. For example, if you install DirectX 6.0 on a system that already has DirectX 5.0 components, all DirectX 5.0 components will be overwritten. Because all DirectX components comply with Component Object Model (COM) backward compatibility rules, software written for DirectX 5.0 will continue to function properly.

The **DirectXSetup** function can tell when DirectX components, display drivers, and audio drivers need to be upgraded. It can also distinguish whether or not these components can be upgraded without adversely affecting the Windows operating system. This is said to be a "safe" upgrade. It is important to note that the upgrade is safe for the operating system, not necessarily for the applications running on the computer. Some hardware-dependent applications can be negatively affected by an upgrade that is safe for Windows.

By default, the **DirectXSetup** function performs only safe upgrades. If the upgrade of a device driver may adversely affect the operation of Windows, the upgrade is not performed.

During the setup process, DirectSetup creates a backup copy of the system components and drivers that are replaced. These can be restored if problems occur.

When display or audio drivers are upgraded, the **DirectXSetup** function uses a database created by Microsoft to manage the process. The database contains information on existing drivers that are provided by Microsoft, the manufacturers of the hardware, or the vendors of the hardware. This database describes the upgrade status of each driver, based on testing done at Microsoft and other sites.

You should check the value returned by **DirectXSetup**. If it is DSETUPERR_SUCCESS_RESTART, notify the user that changes will not take effect without a restart, and offer the choice of restarting immediately. See Dinstall.c in the Dinstall sample application for an example of how to do this.

Preparing the Setup Folder

[This is preliminary documentation and subject to change.]

The **DirectXSetup** function takes a parameter, *lpszRootPath*, that points to the root directory of the installation. Optionally, it can be NULL to indicate that the root path is the current directory—that is, the directory where your setup program resides.

The root directory must contain Dsetup.dll, Dsetup16.dll, and Dsetup32.dll. It must also have a folder named "DirectX" (not case sensitive) containing all the redistributable files and drivers. To create the proper structure on your application setup disc, you should simply copy the entire contents of the Redist\DirectX6 folder on the DirectX SDK disc into the root path of your setup program. Note that this folder was not copied to your development machine when you installed the SDK, so you have to get it from the original disc.

Customizing Setup

[This is preliminary documentation and subject to change.]

DirectSetup allows you to define a callback function for customizing the DirectX setup process. In the DirectSetup documentation, this callback function is referred to as **DirectXSetupCallbackFunction**, but you can give it any name you like.

If a callback function is not provided by the setup program, the **DirectXSetup** function displays status and error information in a dialog box and obtains user input by calling the Win32® **MessageBox** function. If a callback is provided, the information that would have been used to create the status dialog or message box is instead passed to the callback. The callback function is called once for each DirectX component and device driver that can be installed or upgraded.

You might use the callback function to do the following:

- *Employ a custom interface.* Your setup program might display messages in ways other than by using **MessageBox** or standard Windows dialog boxes. The callback function enables you to integrate messages for the DirectX component of your installation into your own interface.
- *Update a progress indicator.*
- *Suppress the display of status and error messages.* Designers of programs for novice users may want to suppress error messages and let the setup program handle errors and make upgrade choices silently. This approach requires a greater development effort for the setup program, but might be appropriate for the target audience.

The following topics help you customize your setup:

- Creating a DirectSetup Callback Function
- Setting the Callback Function
- Using Upgrade Flags in the Callback Function
- Overriding Defaults in the Callback Function

Creating a DirectSetup Callback Function

[This is preliminary documentation and subject to change.]

In order to customize the setup process, you must first create a callback function that conforms to the **DirectXSetupCallbackFunction** prototype, as in the following declaration:

```
DWORD WINAPI DirectXSetupCallbackFunction(
    DWORD dwReason,
    DWORD dwMsgType,
    LPSTR szMessage,
    LPSTR szName,
    void *pInfo);
```

In this example, adapted from the Dinstall sample application, the name of the function is the same as that of the prototype, but this is optional. The parameter names differ slightly from those in the prototype declared in Dsetup.h, and will be used throughout the following discussion.

Parameters

The *dwReason* parameter indicates why the callback function has been called. The possible values are listed and explained in the reference for **DirectXSetupCallbackFunction**.

The *dwMsgType* parameter receives flags equivalent to those that DirectSetup would, by default, pass to **MessageBox**, such as those controlling what buttons and icon are displayed. Of special interest to you are the button flags, which you use to determine what return values are expected. If this value is 0, the event never requires user input and DirectSetup would normally display a status message.

The *szMessage* parameter receives the same text that DirectSetup would otherwise pass to the status dialog or to **MessageBox**.

When a driver is a candidate for upgrading, its name is passed in the *szName* parameter and *pInfo* points to information about the how the upgrade will or should be handled—for example, whether DirectSetup recommends that the old driver be kept or upgraded. For more information, see Using Upgrade Flags in the Callback Function.

The way your callback function interprets the parameters is entirely up to you. Typically you would choose which messages to display (based on *dwReason*) and when to present the user with alternatives, and you would modify the interface accordingly.

Return Value

The value returned by your callback function must conform to the following rules:

- When *dwMsgType* is 0, the return value must be IDOK. In this case, there are no choices to be made by the user. (Your application, of course, is free to display a status message even though no input is required.)
- If *dwMsgType* is nonzero, the return value must be the same as would have been returned by **MessageBox**, given the equivalent choice by the user.

To determine the appropriate return value in the second case, you must test *dwMsgType* for the buttons that would normally have been put in a message box. The following example, from the GetReply function in Dinstall, shows how this can be done:

```
/* The global g_wReply identifies the custom dialog button
   that has been selected by the user. */

switch (dwMsgType & 0x0000000F)
{
  / * There would normally have been an OK and a Cancel button.
     Our IDBUT1 is equivalent to the OK button. If the user didn't
     choose that, it was a Cancel. */
   case MB_OKCANCEL:
      wDefaultButton = (g_wReply == IDBUT1) ? IDOK : IDCANCEL;
      break;

   / * And so on with the other button combinations. */
   case MB_OK:
      wDefaultButton = IDOK;
      break;
   case MB_RETRYCANCEL:
      wDefaultButton = (g_wReply == IDBUT1) ? IDRETRY : IDCANCEL;
      break;
   case MB_ABORTRETRYIGNORE:
      if (g_wReply == IDBUT1)
         wDefaultButton = IDABORT;
      else if (g_wReply == IDBUT2)
         wDefaultButton = IDRETRY;
      else
         wDefaultButton = IDIGNORE;
      break;
```

```
    case MB_YESNOCANCEL:
       if (g_wReply == IDBUT1)
          wDefaultButton = IDYES;
       else if (g_wReply == IDBUT2)
          wDefaultButton = IDNO;
       else
          wDefaultButton = IDCANCEL;
       break;
    case MB_YESNO:
       wDefaultButton = (g_wReply == IDBUT1) ? IDYES : IDNO;
       break;
    default:
       wDefaultButton = IDOK;
 }
```

This routine translates button clicks from the custom interface into the equivalent button identifiers in the standard dialog box that DirectSetup would otherwise create. The *wDefaultButton* variable is set to the identifier of the equivalent standard dialog box button, and it is this value that will ultimately be returned from the callback function.

Remember, you don't necessarily have to give the user a choice even when *dwMsgType* is nonzero. You might decide, for example, to upgrade drivers automatically in a case where DirectSetup would normally ask for confirmation from the user. The Dinstall sample does this when the user has asked to see only problem messages and the driver upgrade is considered safe:

```
 case DSETUP_CB_UPGRADE_SAFE:
    switch (dwMsgType & 0x0000000F)
    {
       case MB_YESNO:
       case MB_YESNOCANCEL:
          return IDYES;
       case MB_OKCANCEL:
       case MB_OK:
       default:
          return IDOK;
    }
    break;
```

The callback function returns either IDYES or IDOK, depending on which button would have represented a positive choice in a dialog box asking the user whether or not the upgrade should proceed.

Setting the Callback Function

[This is preliminary documentation and subject to change.]

Before calling **DirectXSetup**, you must notify DirectSetup that you wish to use a callback. You do so by calling the **DirectXSetupSetCallback** function, passing a pointer to the callback as a parameter. The following example shows how this is done:

```
DirectXSetupSetCallback(
    (DSETUP_CALLBACK) DirectXSetupCallbackFunction);
```

Using Upgrade Flags in the Callback Function

[This is preliminary documentation and subject to change.]

When the application-defined callback function **DirectXSetupCallbackFunction** is called by the **DirectXSetup** function, it is passed a parameter that contains the reason the callback function was invoked. If the reason is DSETUP_CB_MSG_CHECK_DRIVER_UPGRADE, the *pInfo* parameter points to a **DSETUP_CB_UPGRADEINFO** structure containing flags that summarize the **DirectXSetup** function's recommendations on how the upgrade of DirectX components and drivers should be performed. The structure member containing the flags is called **UpgradeFlags**.

The flags fall into the following categories:

Primary Upgrade Flags

These flags are mutually exclusive. One of them is always present in the **UpgradeFlags** structure member. You can extract it by performing a bitwise **AND** with DSETUP_CB_UPGRADE_TYPE_MASK.

- DSETUP_CB_UPGRADE_FORCE
- DSETUP_CB_UPGRADE_KEEP
- DSETUP_CB_UPGRADE_SAFE
- DSETUP_CB_UPGRADE_UNKNOWN

Secondary Upgrade Flags

Any or all of these flags may be present.

- DSETUP_CB_UPGRADE_CANTBACKUP
- DSETUP_CB_UPGRADE_HASWARNINGS

Device Active Flag

This flag is present if the device whose driver is being upgraded is active. It may be combined with any of the others.

- DSETUP_CB_UPGRADE_DEVICE_ACTIVE

Device Class Flags

These flags are mutually exclusive. One of them is always present.

- DSETUP_CB_UPGRADE_DISPLAY
- DSETUP_CB_UPGRADE_MEDIA

In summary: every time the *Reason* parameter has the value DSETUP_CB_MSG_CHECK_DRIVER_UPGRADE, the **UpgradeFlags** member of the structure pointed to by *pInfo* contains one Primary Upgrade Flag, zero or more Secondary Upgrade Flags, zero or one Device Active Flag, and one Device Class Flag.

If the **UpgradeFlags** member is set to DSETUP_CB_UPGRADE_KEEP, the DirectX component or device driver can't be upgraded. Performing an upgrade would cause Windows to cease functioning properly. The **DirectXSetup** function will not perform an upgrade on the component or driver.

A value of DSETUP_CB_UPGRADE_FORCE in **UpgradeFlags** means that the component or driver must be upgraded for Windows to function properly. The **DirectXSetup** function will upgrade the driver or component. It is possible that the upgrade may adversely affect some programs on the system. When the **DirectXSetup** function detects this condition, the **UpgradeFlags** member is set to (DSETUP_CB_UPGRADE_FORCE | DSETUP_CB_UPGRADE_HAS_WARNINGS). When this occurs, the **DirectXSetup** function will perform the upgrade but issue a warning to the user.

Components and drivers are considered safe for upgrade if they will not adversely affect the operation of Windows when they are installed. In this case, the **UpgradeFlags** member is set to DSETUP_CB_UPGRADE_SAFE. It is possible that the upgrade can be safe for Windows, but still cause problems for programs installed on the system. When **DirectXSetup** detects this condition, the **UpgradeFlags** member contains the value (DSETUP_CB_UPGRADE_SAFE | DSETUP_CB_UPGRADE_HAS_WARNINGS). If this condition occurs, the default action for the **DirectXSetup** function is not to perform the upgrade.

Overriding Defaults in the Callback Function

[This is preliminary documentation and subject to change.]

The application-defined function **DirectXSetupCallbackFunction** can override some of the default behaviors of the **DirectXSetup** function through its return value.

For example, the default behavior for **DirectXSetup** is not to install a component if the upgrade type in the structure pointed to by the *pInfo* parameter is set to (DSETUP_CB_UPGRADE_SAFE | DSETUP_CB_UPGRADE_HAS_WARNINGS). In this case, the *MsgType* parameter of the callback function is set to (MB_YESNO | MB_DEFBUTTON2). Without a callback function, DirectSetup would present the user with a dialog box whose default button was No. If the callback function does not seek user input but accepts the default, it returns IDNO. To override the default, the callback function returns IDYES. If it does override the default, the user will be notified by the **DirectXSetup** function.

Testing the Setup Program

[This is preliminary documentation and subject to change.]

To test the DirectSetup component of your setup program, compile a version in which the DSETUP_TESTINSTALL flag is passed to the **DirectXSetup** function. Set up the installation directory as described under Preparing the Setup Folder.

Now, when you run your setup program, DirectSetup will go through the motions of installing DirectX, including calls to a callback function if you have provided one, without actually installing any components.

Enabling AutoPlay

[This is preliminary documentation and subject to change.]

If you are building an AutoPlay compact disc title, copy the Autorun.inf file in the root directory of the DirectX SDK compact disc to the root of your application directory. This text file contains the following information:

```
[autorun]
OPEN=SETUP.EXE
```

If your application's setup program is called Setup.exe, you will not have to make any changes to this file; otherwise, edit this file to contain the name of your setup program. For more information, see **Autorun.inf**.

DirectSetup Reference

[This is preliminary documentation and subject to change.]

This section contains reference information for the API elements of DirectSetup. Reference material is divided into the following categories:

- Functions
- Structures
- Return Values

Functions

[This is preliminary documentation and subject to change.]

This section contains information on the following global functions used with DirectSetup:

- **DirectXRegisterApplication**
- **DirectXSetup**
- **DirectXSetupGetVersion**
- **DirectXSetupSetCallback**
- **DirectXSetupCallbackFunction**
- **DirectXUnRegisterApplication**

DirectXRegisterApplication

[This is preliminary documentation and subject to change.]

The **DirectXRegisterApplication** function registers an application as one designed to work with DirectPlayLobby.

```
int WINAPI DirectXRegisterApplication(
  HWND hWnd,
  LPDIRECTXREGISTERAPP lpDXRegApp
);
```

hWnd
Handle to the parent window. If this parameter is set to NULL, the desktop is the parent window.

lpDXRegApp
Address of the **DIRECTXREGISTERAPP** structure that contains the registry entries that are required for the application to function properly with DirectPlayLobby.

Return Values

If this function is successful, it returns TRUE.

If it is not successful, it returns FALSE. Use the **GetLastError** Win32 function to get extended error information.

Remarks

The **DirectXRegisterApplication** function inserts the registry entries needed for an application to operate with DirectPlayLobby. If these registry entries are added with **DirectXRegisterApplication**, they should be removed with **DirectXUnRegisterApplication** when the application is uninstalled.

Many commercial install programs will remove registry entries automatically when a program in uninstalled. However, such a program will only do so if it added the registry entries itself. If the DirectPlayLobby registry entries are added by **DirectXRegisterApplication**, commercial install programs will not delete the registry entries when the application is uninstalled. Therefore, DirectPlayLobby

registry entries that are created by **DirectXRegisterApplication** should be deleted by **DirectXUnRegisterApplication**.

Registry entries needed for DirectPlayLobby access can be created without the use of the **DirectXRegisterApplication** function. This, however, is not generally recommended. See *Registering Lobby-able Applications* in the DirectPlay® documentation.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in dsetup.h.
Import Library: Use dsetup.lib.

See Also

DirectXUnRegisterApplication

DirectXSetup

[This is preliminary documentation and subject to change.]

The **DirectXSetup** function installs one or more DirectX components.

```
int WINAPI DirectXSetup(
  HWND hWnd,
  LPSTR lpszRootPath,
  DWORD dwFlags
);
```

hWnd
Handle to the parent window for the setup dialog boxes.

lpszRootPath
Pointer to a string that contains the root path of the DirectX component files. This string must specify a full path to the directory that contains the files Dsetup.dll, Dsetup16.dll, and Dsetup32.dll, as well as a "DirectX" (not case sensitive) directory containing redistributable files. If this value is NULL, the current working directory is used.

dwFlags
One or more flags indicating which DirectX components should be installed. A full installation (DSETUP_DIRECTX) is recommended.

DSETUP_D3D	Obsolete. DirectX 3 programs that use this flag will install all DirectX components.
DSETUP_DDRAW	Obsolete. DirectX 3 programs that use this flag will install all DirectX components.

DSETUP_DDRAWDRV	Installs display drivers provided by Microsoft.
DSETUP_DINPUT	Obsolete. DirectX 3 programs that use this flag will install all DirectX components.
DSETUP_DIRECTX	Installs DirectX run-time components as well as DirectX-compatible display and audio drivers.
DSETUP_DIRECTXSETUP	Obsolete. DirectX 3 programs that use this flag will install all DirectX components.
DSETUP_DPLAY	Obsolete. DirectX 3 programs that use this flag will install all DirectX components.
DSETUP_DPLAYSP	Obsolete. DirectX 3 programs that use this flag will install all DirectX components.
DSETUP_DSOUND	Obsolete. DirectX 3 programs that use this flag will install all DirectX components.
DSETUP_DSOUNDDRV	Installs audio drivers provided by Microsoft.
DSETUP_DXCORE	Installs DirectX run-time components. Does not install DirectX-compatible display and audio drivers.
DSETUP_TESTINSTALL	Performs a test installation. Does not actually install new components.

Return Values

If this function is successful, it returns DSETUPERR_SUCCESS, or DSETUPERR_SUCCESS_RESTART if the user must restart the system in order for changes to take effect.

If it is not successful, it returns an error code. For a list of possible return codes, see Return Values.

Remarks

Before you use the **DirectXSetup** function in your setup program, you should ensure that there is at least 15 MB of available disk space on the user's system. This is the maximum space required for DirectX to set up the appropriate files. If the user's system already contains the DirectX files, this space is not needed.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in dsetup.h.
Import Library: Use dsetup.lib.

DirectXSetupGetVersion

[This is preliminary documentation and subject to change.]

The **DirectXSetupGetVersion** function retrieves the version number of the DirectX components that are currently installed.

```
INT WINAPI DirectXSetupGetVersion(
  DWORD   *pdwVersion,
  DWORD   *pdwRevision
);
```

pdwVersion
Address of a variable to receive the version number. Can be NULL if the version number is not wanted.

pdwRevision
Address of a variable to receive the revision number. Can be NULL if the revision number is not wanted.

Return Values

If the function is successful, it returns nonzero.

If it is not successful, it returns zero.

Remarks

The **DirectXSetupGetVersion** function can be used to retrieve the version and revision numbers before or after the **DirectXSetup** function is called. If it is called before the **DirectXSetup** function is invoked, it gives the version and revision numbers of the DirectX components that are currently installed. If it is called after the **DirectXSetup** function is called, but before the computer has been rebooted, it will give the version and revision numbers of the DirectX components that will take effect after the computer is restarted.

The version number in the *pdwVersion* parameter is composed of the major version number and the minor version number. The major version number will be in the 16 most significant bits of the **DWORD** when this function returns. The minor version number will be in the 16 least significant bits of the **DWORD** when this function returns. The version numbers can be interpreted as follows:

DirectX Version	**Value Pointed At By *pdwVersion***
DirectX 1	0x00040001
DirectX 2	0x00040002
DirectX 3	0x00040003
DirectX 5.0	0x00040005
DirectX 6.0	0x00040006

Note that there is no version 4 of DirectX.

The version number in the *pdwRevision* parameter is composed of the release number and the build number. The release number will be in the 16 most significant bits of the **DWORD** when this function returns. The build number will be in the 16 least significant bits of the **DWORD** when this function returns.

The following sample code fragment demonstrates how the information returned by **DirectXSetupGetVersion** can be extracted and used.

```
DWORD dwVersion;
DWORD dwRevision;
if (DirectXSetupGetVersion(&dwVersion, &dwRevision))
{
   printf("DirectX version is %d.%d.%d.%d\n",
        HIWORD(dwVersion), LOWORD(dwVersion),
        HIWORD(dwRevision), LOWORD(dwRevision));
}
```

Version and revision numbers can be concatenated into a 64-bit quantity for comparison. The version number should be in the 32 most significant bits and the revision number should be in the 32 least significant bits.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in dsetup.h.
Import Library: Use dsetup.lib.

See Also

DirectXSetup

DirectXSetupSetCallback

[This is preliminary documentation and subject to change.]

The **DirectXSetupSetCallback** sets a pointer to a callback function that is periodically called by **DirectXSetup**. The callback function can be used for setup progress notification and to implement a custom user interface for an application's setup program. For information on the callback function, see **DirectXSetupCallbackFunction**. If a setup program does not provide a callback function, the **DirectXSetupSetCallback** function should not be invoked.

INT WINAPI DirectXSetupSetCallback(
  DSETUP_CALLBACK *Callback*

```
);
```

Callback
Pointer to a callback function.

Return Values

Currently returns zero.

Remarks

To set a callback function, **DirectXSetupSetCallback** must be called before the **DirectXSetup** function is called.

The name of the callback function passed to **DirectXSetupSetCallback** is supplied by the setup program. However, it must match the prototype given in **DirectXSetupCallbackFunction**.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in dsetup.h.
Import Library: Use dsetup.lib.

See Also

DirectXSetupCallbackFunction, **DirectXSetup**

DirectXSetupCallbackFunction

[This is preliminary documentation and subject to change.]

DirectXSetupCallbackFunction is a placeholder name for an optional callback function supplied by the setup program. If present, it is called once for each step of the setup process.

```
DWORD DirectXSetupCallbackFunction(
  DWORD  Reason,
  DWORD  MsgType,
  char *szMessage,
  char *szName,
  void *pInfo
);
```

Reason

Reason for the callback. It can be one of the following values:

DSETUP_CB_MSG_BEGIN_INSTALL
DirectXSetup is about to begin installing DirectX components and device drivers.

DSETUP_CB_MSG_BEGIN_INSTALL_DRIVERS
DirectXSetup is about to begin installing device drivers.

DSETUP_CB_MSG_BEGIN_INSTALL_RUNTIME
DirectXSetup is about to begin installing DirectX components.

DSETUP_CB_MSG_BEGIN_RESTORE_DRIVERS
DirectXSetup is about to begin restoring previous drivers.

DSETUP_CB_MSG_CANTINSTALL_BETA
A pre-release beta version of Windows 95 was detected. The DirectX component or device driver can't be installed.

DSETUP_CB_MSG_CANTINSTALL_NOTWIN32
The operating system detected is not a Windows 32-bit operating system. DirectX is not compatible with Windows 3.*x*.

DSETUP_CB_MSG_CANTINSTALL_NT
The DirectX component or device driver can't be installed on versions of Windows NT prior to version 4.0.

DSETUP_CB_MSG_CANTINSTALL_UNKNOWNOS
The operating system is unknown. The DirectX component or device driver can't be installed.

DSETUP_CB_MSG_CANTINSTALL_WRONGLANGUAGE
The DirectX component or device driver is not localized to the language being used by Windows.

DSETUP_CB_MSG_CANTINSTALL_WRONGPLATFORM
The DirectX component or device driver is for another type of computer.

DSETUP_CB_MSG_CHECK_DRIVER_UPGRADE
Driver is being considered for upgrade. Verification from user is recommended.

DSETUP_CB_MSG_INTERNAL_ERROR
An internal error has occurred. Setup of the DirectX component or device driver has failed.

DSETUP_CB_MSG_NOMESSAGE
No message to be displayed. The callback function should return.

DSETUP_CB_MSG_NOTPREINSTALLEDONNT
The DirectX component or device driver can't be installed on the version of Windows NT/Windows 2000 in use.

DSETUP_CB_MSG_PREINSTALL_NT
DirectX is already installed on the version of Windows NT/Windows 2000 in use.

DSETUP_CB_MSG_SETUP_INIT_FAILED

Setup of the DirectX component or device driver has failed.

MsgType
Contains flags that control the display of a message box. These flags can be passed to the **MessageBox** function. An exception is when *MsgType* is 0. In that case, the setup program can display status information but should not wait for input from the user.

szMessage
A localized character string containing error or status messages that can be displayed in a dialog box created with the **MessageBox** function.

szName
The value of *szName* is NULL unless the *Reason* parameter is DSETUP_CB_MSG_CHECK_DRIVER_UPGRADE. In that case, *szName* contains the name of driver to be upgraded.

pInfo
Pointer to a structure containing upgrade information. When *Reason* is DSETUP_CB_MSG_CHECK_DRIVER_UPGRADE, the setup program is in the process of upgrading a driver and asking the user whether the upgrade should take place. In this case, *pInfo* points to a **DSETUP_CB_UPGRADEINFO** structure containing information about the upgrade.

Return Values

The return value should be the same as would be returned by the **MessageBox** function, with one exception. If this function returns zero, the **DirectXSetup** function will perform the default action for upgrade of the DirectX component or driver.

Remarks

The name of the **DirectXSetupCallbackFunction** is supplied by the setup program. The **DirectXSetupSetCallback** function is used to pass the address of the callback function to DirectSetup.

If *MsgType* is equal to zero, the setup program may display status information, but it should not wait for user input. In this case the function should return IDOK.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in dsetup.h.
Import Library: User-defined.

See Also

MessageBox, **DirectXSetupSetCallback**, Customizing Setup

DirectXUnRegisterApplication

[This is preliminary documentation and subject to change.]

The **DirectXUnRegisterApplication** function deletes the registration of an application designed to work with DirectPlayLobby.

```
int WINAPI DirectXUnRegisterApplication(
  HWND hWnd,
  LPGUID lpGUID
);
```

hWnd
Handle to the parent window. Set this to NULL if the desktop is the parent window.

lpGUID
Pointer to a GUID that represents the DirectPlay application to be unregistered.

Return Values

If the function succeeds, the return value is TRUE meaning that the registration is successfully deleted.

If the function fails, the return value is FALSE.

Remarks

The **DirectXUnRegisterApplication** function removes registry the entries needed for an application to work with DirectPlayLobby. An uninstall program should only use **DirectXUnRegisterApplication** if it used **DirectXRegisterApplication** when the application was installed.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in dsetup.h.
Import Library: Use dsetup.lib.

See Also

DirectXRegisterApplication

Structures

[This is preliminary documentation and subject to change.]

This section contains information about the following structures used with DirectSetup:

- **DIRECTXREGISTERAPP**
- **DSETUP_CB_UPGRADEINFO**

Note

The memory for all DirectX structures must be initialized to zero before use. In addition, all structures that contain a **dwSize** member must set the member to the size of the structure, in bytes, before use. The following example from DirectDraw performs these tasks on a common structure, **DDCAPS**:

```
DDCAPS ddcaps; // Can't use this yet.

ZeroMemory(&ddcaps, sizeof(ddcaps));
ddcaps.dwSize = sizeof(ddcaps);

// Now the structure can be used.
.
.
```

DIRECTXREGISTERAPP

[This is preliminary documentation and subject to change.]

The **DIRECTXREGISTERAPP** structure contains the registry entries needed for applications designed to work with DirectPlayLobby.

```
typedef struct _DIRECTXREGISTERAPP {
    DWORD  dwSize;
    DWORD  dwFlags;
    LPSTR  lpszApplicationName;
    LPGUID lpGUID;
    LPSTR  lpszFilename;
    LPSTR  lpszCommandLine;
    LPSTR  lpszPath;
    LPSTR  lpszCurrentDirectory;
} DIRECTXREGISTERAPP, *PDIRECTXREGISTERAPP,
*LPDIRECTXREGISTERAPP;
```

dwSize

Size of the structure. Must be initialized to the size of the **DIRECTXREGISTERAPP** structure.

dwFlags
Reserved for future use.

lpszApplicationName
Name of the application.

lpGUID
Globally unique identifier (GUID) of the application.

lpszFilename
Name of the executable file to be called.

lpszCommandLine
Command-line arguments for the executable file.

lpszPath
Path of the executable file.

lpszCurrentDirectory
Current directory. This is typically the same as **lpszPath**.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in dsetup.h.

DSETUP_CB_UPGRADEINFO

[This is preliminary documentation and subject to change.]

The **DSETUP_CB_UPGRADEINFO** structure is passed as a parameter to the application-defined **DirectXSetupCallbackFunction**. It contains valid information only when the *Reason* parameter is DSETUP_CB_MSG_CHECK_DRIVER_UPGRADE. Callback functions can use it to get status information on the upgrade that is about to be done.

```
typedef struct _DSETUP_CB_UPGRADEINFO {
    DWORD UpgradeFlags;
} DSETUP_CB_UPGRADEINFO;
```

UpgradeFlags
One or more flags indicating the status of the upgrade. The following values are defined:

DSETUP_CB_UPGRADE_CANTBACKUP
The old system components can't be backed up. Upgrade can be performed, but the components or drivers can't be restored later.

DSETUP_CB_UPGRADE_DEVICE_ACTIVE
The device is currently in use.

DSETUP_CB_UPGRADE_DEVICE_DISPLAY

The device driver being upgraded is for a display device.

DSETUP_CB_UPGRADE_DEVICE_MEDIA
The device driver being upgraded is for a media device.

DSETUP_CB_UPGRADE_FORCE
Windows may not function correctly if the component is not upgraded. The upgrade will be performed.

DSETUP_CB_UPGRADE_HASWARNINGS
DirectSetup can upgrade the driver for this device, but doing so may affect one or more programs on the system. The *szMessage* parameter contains the names of which programs may be affected. Upgrade not recommended.

DSETUP_CB_UPGRADE_KEEP
The system may fail if this device driver is upgraded. Upgrade not allowed.

DSETUP_CB_UPGRADE_SAFE
DirectSetup can safely upgrade this device driver. Upgrade recommended. A safe upgrade will not adversely affect the operation of Windows. Some hardware-dependent programs may be adversely affected.

DSETUP_CB_UPGRADE_UNKNOWN
DirectSetup does not recognize the existing driver for this device. This value will occur frequently. Upgrading may adversely affect the use of the device. It is strongly recommended that the upgrade not be performed.

Remarks

You can use the DSETUP_CB_UPGRADE_TYPE_MASK value to extract the upgrade type (FORCE, KEEP, SAFE, or UNKNOWN) from **UpgradeFlags**.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in dsetup.h.

See Also

DirectXSetupCallbackFunction

Return Values

[This is preliminary documentation and subject to change.]

The **DirectXSetup** function can return the following values. It can also return a standard COM error.

DSETUPERR_SUCCESS
Setup was successful and no restart is required.

DSETUPERR_SUCCESS_RESTART
Setup was successful and a restart is required.

DSETUPERR_BADSOURCESIZE
A file's size could not be verified or was incorrect.

DSETUPERR_BADSOURCETIME
A file's date and time could not be verified or were incorrect.

DSETUPERR_BADWINDOWSVERSION
DirectX does not support the Windows version on the system.

DSETUPERR_CANTFINDDIR
The setup program could not find the working directory.

DSETUPERR_CANTFINDINF
A required .inf file could not be found.

DSETUPERR_INTERNAL
An internal error occurred.

DSETUPERR_NEWERVERSION
A later version of DirectX has already been installed. Applications can safely ig nore this error, because the newer version will not be overwritten and is fully co mpatible with applications written for earlier versions.

DSETUPERR_NOCOPY
A file's version could not be verified or was incorrect.

DSETUPERR_NOTPREINSTALLEDONNT
The version of Windows NT/Windows 2000 on the system does not contain the current version of DirectX. An older version of DirectX may be present, or DirectX may be absent altogether.

DSETUPERR_OUTOFDISKSPACE
The setup program ran out of disk space during installation.

DSETUPERR_SOURCEFILENOTFOUND
One of the required source files could not be found.

DSETUPERR_UNKNOWNOS
The operating system on your system is not currently supported.

DSETUPERR_USERHITCANCEL
The **Cancel** button was pressed before the application was fully installed.

DirectSetup Visual Basic Reference

[This is preliminary documentation and subject to change.]

This section contains reference information for the API elements that DirectSetup provides. Reference material is divided into the following categories.

- Methods
- Types

Methods

[This is preliminary documentation and subject to change.]

The methods that implement DirectSetup are contained within the main **DirectX7** class. The documentation for the following methods are found under the **DirectX7** class reference section:

- **DirectX7.DirectXRegisterApplication**
- **DirectX7.DirectXSetup**
- **DirectX7.DirectXSetupGetVersion**
- **DirectX7.DirectXUnregisterApplication**

Types

[This is preliminary documentation and subject to change.]

There is one type that is used with DirectSetup, **DIRECTXREGISTERAPP**.

DIRECTXREGISTERAPP

[#] [This is preliminary documentation and subject to change.]

```
Type DIRECTXREGISTERAPP
    lFlags As Long
    strApplicationName As String
    strCommandLine As String
    strCurrentDirectory As String
    strFilename As String
    strGuid As String
    strPath As String
End Type
```

lFlags

strApplicationName

strCommandLine

strCurrentDirectory

[#] IDH__dx_DIRECTXREGISTERAPP_dsetup_vb

strFilename

strGuid

strPath

DirectSetup Samples

[This is preliminary documentation and subject to change.]

This section provides a summary of the application in the DirectX SDK that is primarily intended to demonstrate the DirectSetup component. The following sample program demonstrates the use and capabilities of DirectSetup :

- Dinstall Sample

Although DirectX samples include Microsoft® Visual C++® project workspace files, you might need to verify other settings in your development environment to ensure that the samples compile properly. For more information, see Compiling DirectX Samples and Other DirectX Applications.

Dinstall Sample

[This is preliminary documentation and subject to change.]

Description

Dinstall is an example of how to use DirectXSetup interfaces to install the DirectX subsystem and DirectX drivers. It shows how to use a callback function to present messages and get user input through a custom interface, in this case a simple modeless dialog box.

Path

Source: *(SDK root)*\Samples\Multimedia\Dxmisc\Src\Setup

Executable: *(SDK root)*\Samples\Multimedia\Dxmisc\Bin

User's Guide

First copy the entire contents of the Redist\DirectX6 folder from the DirectX SDK CD into the same folder as Dinstall.exe. In your development environment, set the working directory to this folder as well. (In Microsoft® Visual C++®, this setting is on the **Debug** page of the **Project Settings** dialog box.)

Run the program and select **Start Install** from the **File** menu. DirectSetup performs a simulated installation of DirectX (see Programming Notes) and advises you of its progress in a modeless dialog box. The **Options** menu allows you to change the level of messages shown. However, if you are performing only a simulated installation, you will never see problem or update messages.

Choose **Get Version** from the **File** menu. The program shows the version and revision number of DirectX currently installed on the system.

Programming Notes

The driver folders in \Redist\DirectX6\Directx\Drivers contain localized versions of Microsoft-provided DirectX drivers. You can delete any number of these folders from your working directory if you want to save disk space.

By default, the program passes DSETUP_TESTINSTALL to the **DirectXSetup** function. This means that no files are actually copied, nor is the registry modified. To perform a real installation, delete this flag from the call.

Dinstall employs a callback function to monitor the progress of installation and intercept messages. Depending on the user's preferred warning level, as tracked in *g_fStatus*, messages may be ignored or presented to the user in a modeless dialog box. If user input is required, the appropriate buttons are displayed and the GetReply function monitors the message queue until one of the buttons is pressed.