DirectPlay

The Microsoft® DirectPlay® application programming interface (API) is the component of Microsoft DirectX® that enables you to write network applications such as multiplayer games. DirectPlay performs all of the hard work associated with connecting players, even those behind Network Address Translation (NAT) devices, and managing sessions. It allows you to create, find, and connect to multiplayer games. When connected, DirectPlay enables you to send guaranteed or nonguaranteed messages to other players. A common framework for launching applications and in-game voice communications is also provided. In addition, DirectPlay provides support for Microsoft Windows® Powered Pocket PC 2002 and connectivity with DirectPlay 8.0 applications.

Information about DirectPlay is presented in the following sections.

Roadmap

What's New in DirectPlay. New features and functionality of this component in DirectX 9.0.

Basic Concepts in DirectPlay. An overview of what DirectPlay is and what it can do for your application, together with a first view of some key objects and the steps involved in creating a network application.

<u>Peer-to-Peer Sessions</u>. A guide to using the <u>IDirectPlay8Peer</u> interface for creating and managing peer-to-peer applications.

<u>Client/Server Sessions</u>. A guide to using the <u>IDirectPlay8Client</u> and <u>IDirectPlay8Server</u> interfaces for creating and managing client/server applications.

<u>DirectPlay Lobby</u>. A guide to using the <u>IDirectPlay8LobbyClient</u> and <u>IDirectPlay8LobbiedApplication</u> interfaces for setting up multiplayer games.

<u>DirectPlay Voice</u>. A guide to using the <u>IDirectPlayVoiceClient</u> and <u>IDirectPlayVoiceServer</u> interfaces for creating and managing voice sessions.

<u>DirectPlay for Pocket PC 2002</u>. Information about creating applications for the Pocket PC 2002.

Advanced Topics in DirectPlay. More advanced features of DirectPlay including addressing, NATs, callbacks, and multithreading.

<u>DirectPlay C/C++ Tutorials</u>. Step-by-step tutorials following sample code in the software development kit (SDK).

<u>DirectPlay C++ Samples</u>. A guide to the C/C++ sample applications in the SDK.

<u>DirectPlay C/C++ Reference</u>. Detailed information for the DirectPlay C++ API.

What's New in DirectPlay

This section introduces new features for Microsoft® DirectPlay® 9.0.

New Features in DirectPlay

- The DirectPlay application programming interfaces (APIs) are available for the Microsoft Windows® Powered Pocket PC 2002. See <u>DirectPlay for Pocket PC 2002</u> for more information.
- The <u>DPN_MSGID_SEND_COMPLETE</u> message structure has two new members that allow you to calculate the round-trip travel time of messages.
- DirectPlay has a new service provider for network simulation. You can use the <u>IDP8SimControl</u> methods to test applications under a variety of network conditions.
- DirectPlay has a new interface, <u>IDirectPlay8ThreadPool</u>, that allows you to manage threads in your application.
- Applications can cancel all messages sent by a particular player using the DPNCANCEL_PLAYER_SENDS flag when calling <u>IDirectPlay8Peer::CancelAsyncOperation</u>, <u>IDirectPlay8Server::CancelAsyncOperation</u>, and <u>IDirectPlay8Client::CancelAsyncOperation</u>.
- Players can receive their local player identifier (ID) in the <u>DPN_MSGID_CONNECT_COMPLETE</u> message.
- Hosts can prevent DirectPlay from processing enumeration queries by setting the DPNSESSION_NOENUMS flag in the <u>DPN_APPLICATION_DESC</u> structure when calling <u>IDirectPlay8Peer::Host</u> and <u>IDirectPlay8Server::Host</u>.
- Messages sent to a group with no players in it will now return DPNSUCCESS_NOPLAYERSINGROUP instead of DPNERR_GENERIC.
- Packet signing is available for all DirectPlay traffic.
- Applications can close immediately by setting the DPNCLOSE_IMMEDIATE flag when calling IDirectPlay8Peer::Close, IDirectPlay8Client::Close, and IDirectPlay8Server::Close.

- DirectPlay 9.0 has improved defense against connection spoofing.
- Use the DPNINITIALIZE_HINT_LANSESSION flag when calling <u>IDirectPlay8Peer::Initialize</u>, <u>IDirectPlay8Client::Initialize</u>, and <u>IDirectPlay8Server::Initialize</u>.
- Packet coalescence is available by setting the DPNSEND_COALESCE flag when calling <u>IDirectPlay8Peer::SendTo</u>, <u>IDirectPlay8Client::Send</u>, and <u>IDirectPlay8Server::SendTo</u>.
- Applications can tune the DirectPlay protocol using the DPN_CAPS_EX structure used when calling <u>IDirectPlay8Peer::GetCaps</u>, <u>IDirectPlay8Client::GetCaps</u>, and <u>IDirectPlay8Server::GetCaps</u> or <u>IDirectPlay8Peer::SetCaps</u>, <u>IDirectPlay8Client::SetCaps</u>, and <u>IDirectPlay8Server::SetCaps</u>.
- A group owner's context value has been added to the <u>DPNMSG_CREATE_GROUP</u> structure.
- If the DPNSESSION_NODPNSVR flag is not set in the DPN_APPLICATION_DESC structure when calling IDirectPlay8Peer::Host or IDirectPlay8Server::Host and dpnsvr.exe does not start, the call to Host will fail and return DPNERR_DPNSVRNOTAVAILABLE.
- Less reliable connections should now perform better with improved DirectPlay protocol behavior.
- Network Address Translation (NAT) support has improved. This includes the new <u>IDirectPlay8NATResolver</u> interface, which allows you to create a NAT resolver application.
- DirectPlay now supports Internet Protocol (IP) v6.

Basic Concepts in DirectPlay

The Microsoft® DirectPlay® API provides developers with the tools to develop multiplayer applications such as games or chat clients. For simplicity, this documentation will refer to all such applications as "games". A multiplayer application has two basic characteristics:

- Two or more individual users, each with a game client on their computer.
- Network links that enable the users' computers to communicate with each other, perhaps through a centralized server.

DirectPlay provides a layer that largely isolates your application from the underlying network. For most purposes, your application can just use the DirectPlay API, and enable DirectPlay to handle the details of network communication. DirectPlay provides many features that simplify the process of implementing many aspects of a multiplayer application, including:

- Creating and managing both peer-to-peer and client/server sessions
- Managing users and groups within a session
- Managing messaging between the members of a session over different network links and varying network conditions
- Enabling applications to interact with lobbies
- Enabling users to communicate with each other by voice

The following documentation provides a high-level overview of the capabilities of DirectPlay.

- DirectPlay Network Communication
- <u>Communicating with DirectPlay Objects</u>

- Creating and Managing Sessions
- <u>Getting DirectPlay Data</u>

 $\ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.

DirectPlay Network Communication

The primary function of Microsoft® DirectPlay® is to provide you with efficient and flexible messaging support that largely isolates your application from the underlying network hardware and software. If you need to send a status update, you can call the relevant DirectPlay application programming interface (API), regardless of what kind of network link is involved. DirectPlay network service providers support communication over Transmission Control Protocol/Internet Protocol (TCP/IP), Internetwork Packet Exchange (IPX), modem, and serial links.

Notes DirectPlay does not provide secure communications.

- DirectPlay Transport Protocol
- DirectPlay Addresses

DirectPlay Transport Protocol

The core of the DirectPlay networking capabilities is the DirectPlay protocol. This transport-layer protocol has been completely overhauled for DirectPlay 8, and is now used for all messaging. The DirectPlay protocol is focused on making it simple for you to send data from the sending application to the target application, without needing to worry about what happens in between. The protocol offers a number of features that are tailored to the needs of multiplayer games, including:

- Reliable and unreliable delivery of messages. Reliable messages will be resent until the target application receives them. You can assign the delivery type on a message-by-message basis.
- Sequential and non-sequential delivery of messages. Sequential messages will be passed to the target application in the order they were sent.
- Message fragmentation and reassembly. If message size exceeds the capacity of a particular network, DirectPlay automatically fragments and reassembles the message.
- Congestion control. DirectPlay automatically *throttles* your outgoing messages to a level that can be handled by the target. This feature prevents you from flooding the target with more messages than it can process.
- Send prioritization. To ensure that the most important messages get sent first, DirectPlay enables you to designate messages as low, medium, or high priority. The high priority messages are sent to the front of the output queue, followed by medium and low priority messages.
- Message timeouts. To prevent the outgoing message queue from being clogged with messages that have been superseded by more recent messages, DirectPlay enables you to assign a timeout value to all messages. When a message times out, it is removed from the outgoing message queue, regardless of whether it has

been sent or not.

DirectPlay Addresses

In order to deliver messages, each participant in a multiplayer game must have a unique address. Addresses can refer either to the computer that your application is running on (*device address*), or a computer that your application needs to communicate with (*host address*).

DirectPlay addresses are in the form of URL strings. These strings consist of a scheme, scheme separator, and data string in the following general format.

x-directplay:/[data string]

The data string contains several elements that specify everything that is needed to enable communication to take place between sender and target, over a variety of different types of network link.

In use, the URL strings are embedded in a DirectPlay address object which is passed to or from DirectPlay API methods. You have the option of either manipulating the URL string directly, or using the methods exposed by the address object to handle each element of the data string separately.

Communicating with DirectPlay Objects

Microsoft® DirectPlay® consists of a collection of Component Object Model (COM) objects. Each object exposes one or more interfaces that enable you to control various aspects of DirectPlay. For instance, the DirectPlay peer object (CLSID_DirectPlay8Peer) is used to manage peerto-peer games.

You communicate with a DirectPlay object by calling the methods exposed by its interfaces. For instance, to send some data to another user in a peer-to-peer game, you would send a message by calling the <u>IDirectPlay8Peer::SendTo</u> method. DirectPlay then takes care of getting the message to its target.

DirectPlay communicates with your application through one or more callback functions. These functions are similar in principle to the familiar Window procedure. Your application implements the callback function and passes a pointer to the function to DirectPlay during initialization. When DirectPlay needs to communicate with your application, it calls the callback function and passes in two key items of information:

- A message identifier (ID) that identifies the message type
- A pointer to a block of data, typically a structure, that provides any needed details.

For instance, when the message sent in the above example arrives at its target, the target application's callback function will receive a message with a DPNMSGID_RECEIVE message ID, indicating that a message has arrived from another user. The accompanying structure contains the data.

Because much of DirectPlay messaging is multithreaded, it is critical that

callback functions be properly implemented.

Creating and Managing Sessions

A game *session* is an instance of a particular multiplayer game. A session has two or more users playing simultaneously, each with the same game client on his or her computer. A *player* is an entity in the game itself, and is defined by the particular game. Each user may have more than one *player* in a game. However, the game application must manage these players itself, using separate Microsoft® DirectPlay® interfaces or objects for each player.

The first step in creating a session is to collect a group of users. There are two basic approaches.

- Many game sessions are arranged by a *lobby* application running on a remote computer. This approach is used by most Internet-based games.
- It is also possible to arrange games by having the individual users' computers communicate with each other. This approach is typically limited such situations as a group of potential users that are all on the same LAN.

Once the session has been arranged, the game is launched and gameplay begins. As the session proceeds, players may be eliminated from the session, or new players added. The details are up to the individual game.

With a multiplayer game, each user's user interface (UI) can be synchronized with that of all other users in the session. Managing a multiplayer session thus requires a continual stream of messages to and from each user. For example, every time a player moves, a message must be sent to update that player's position on all the other game clients in the session. The core of DirectPlay is that part of the application programming interface (API) that supports efficient and flexible messaging between all the computers in a session.

There are two basic ways to structure the messaging *topology* of a session: *peer-to-peer* and *client/server*. Both topologies have their advantages and limitations, so you will need to evaluate which is most appropriate for your game.

This section includes the following topics.

- <u>Peer-to-Peer Topology</u>
- <u>Client/Server Topology</u>

Peer-to-Peer Topology

A peer-to-peer game consists of the individual players' computers, connected by network links. Schematically, the topology of a four-player peer-to-peer game looks like:

Gameplay is handled by having each user's game client communicate directly with the other users' clients. For instance, when one user moves, the game client must send three update messages, one to each of the other users' computers.

A peer-to-peer game is normally arranged and launched through a *lobby client* application that resides on the user's computer. There are two basic ways the lobby client can arrange a session:

- The lobby client communicates directly with other potential users' lobby clients. This approach can be used, for instance, to arrange a game among users on the same LAN subnet.
- The lobby client acts as a link to lobby server application running on a remote computer. This is the way Internet-based games are normally arranged.

Once a session has been arranged and launched, most or all of the messaging will be user to user. If a lobby server is involved, it will only be handling such tasks as updating its list of session members when a player leaves the game, or enabling a new user to request entry to the session. Otherwise, the server stays in the background, and is typically not even aware of most of the messages that are being sent.

Because the server is either non-existent or at least not directly involved with the game play, one user is designated as the game *host*. They are

responsible for handling logistical details such as bringing new players into an ongoing session.

Peer-to-peer games have the advantage of simplicity. All that is needed is a collection of players with game clients, and a way to organize a session. The primary drawback of the peer-to-peer topology is scalability. As the number of users increase, the number of messages needed to facilitate game play increases geometrically. The maximum number of users that can be accommodated depends on the game and the network bandwidth, but is typically no more than 20-30.

Client/Server Topology

A client/server game consists of the individual players' computers (the "game clients") connected to a central server computer. The topology of a four-player client/server game is shown in the following graphic.

Game play is handled by having each user's game client communicate with the server. The server is responsible for passing information about to the other users. For instance, when one user moves, the user's computer sends a message to the server. The server then sends messages to the other players to inform them of a change in game state. The server can have a number of responsibilities; it can:

- Act as the session's messaging hub. Each computer needs to send messages only to the server. The server handles the logistics of synchronizing all the other users. This arrangement can substantially reduce message traffic, especially for large games.
- Host the game. The server typically takes care of the tasks that must be handled by the session host in a peer-to-peer game.
- Support many aspects of the game. The server often does much more than support game logistics. With many games, especially large ones, much of the processing that maintains the "game universe" takes place on the server. The game clients are primarily responsible for handling the user interface (UI).

A client/server game is typically arranged and launched through a lobby client application that resides on the user's computer. The lobby client acts as a link to a lobby server application, which usually runs on the same remote computer that is hosting the game. When the game has been launched, the game server application becomes the host and handles tasks such as admitting new users to the game. There are a number of advantages to client/server games.

- They are more efficient, especially for large-scale games. In particular, they scale much better than peer-to-peer games because additional players cause only a linear increase in the messaging traffic. The client/server topology is necessary for massively multiplayer games.
- You are not limited by the processing power of your users' computers. You can locate much of the processing required to maintain a large complex "game universe" on a single powerful computer, and let the users' computers handle the UI.
- You can control key aspects of your game at a central site. For instance, you can often update the game or fix bugs by modifying the server application, thereby avoiding the need to update large numbers of game clients.

Once you have developed and shipped a peer-to-peer game, you are essentially finished. The game clients are largely self-sufficient. However, with a client/server game, you have an ongoing responsibility to your users that goes beyond providing support services. You must also provide and maintain a game server computer and the associated software, along with the network links to handle all the messaging, for the lifetime of the application. In the case of massively multiplayer games, you may need to operate your servers for extended periods with few or no breaks in service, or risk angering users by disrupting their game play.

^{© 2003} Microsoft Corporation. All rights reserved.

Getting DirectPlay Data

Many times during a Microsoft® DirectPlay® session, your application needs to get information from DirectPlay. This can be to enumerate all the hosts available or to get a new player's data. The DirectPlay programming model for retrieving data usually involves passing a buffer to DirectPlay to be filled. However, the buffer size needed is usually unknown. Allocating a large enough block of memory to hold any conceivable array will work, but is inefficient. Therefore, to use these methods that return data, you should first call the method with a null buffer. The method will return to you the required size of the buffer. Then you can call the method again using the required buffer size. This system of getting data is used to:

- Get address information

 (IDirectPlay8Address::GetComponentByIndex,
 IDirectPlay8Address::GetComponentByName,
 IDirectPlay8Address::GetURLA, IDirectPlay8Address::GetURLW,
 IDirectPlay8Address::GetUserData).
- Enumerate session hosts (<u>IDirectPlay8Peer::EnumHosts</u>, <u>IDirectPlay8Client::EnumHosts</u>).
- Enumerate service providers

 (IDirectPlay8Peer::EnumServiceProviders,
 IDirectPlay8Client::EnumServiceProviders,
 IDirectPlay8Server::EnumServiceProviders).
- Get application descriptions

 (IDirectPlay8Client::GetApplicationDesc,
 IDirectPlay8Server::GetApplicationDesc,
 IDirectPlay8Peer::GetApplicationDesc).
- Fill structures (<u>Structures</u>).
- Enumerate devices (IDirectPlay8NATResolver::EnumDevices).

- Enumerate players and groups(<u>IDirectPlay8Peer::EnumPlayersAndGroups</u>, <u>IDirectPlay8Server::EnumPlayersAndGroups</u>).
- Enumerate group members

 (IDirectPlay8Peer::EnumGroupMembers,
 IDirectPlay8Server::EnumGroupMembers).
- Enumerate local programs (<u>IDirectPlay8LobbyClient::EnumLocalPrograms</u>).

The following procedure outlines how to enumerate the members of a group in a peer-to-peer game as an example. The same general procedure is followed by all other types of data retrieval, except for host enumerations. Because enumerations are often used to obtain a snapshot of information that might be changing, you should perform enumerations in a loop until you are successful.

- 1. Call **IDirectPlay8Peer::EnumGroupMembers**. This method returns an integer array in the *prgdpnid* parameter that contains the identifier (ID) of each player in the group. The *pcdpnid* parameter is used to indicate the number of elements in the array. Set the *pcdpnid* parameter to 0 to request the appropriate value. Set *prgdpnid* to NULL.
- 2. When the method returns, *pcdpnid* will point to the number of elements that will be in the array.
- 3. Allocate your array using the returned *pcdpnid* value, and assign the array to the *prgdpnid* parameter.
- 4. Set *pcdpnid* to the value that was returned in the first method call.
- 5. Call IDirectPlay8Peer::EnumGroupMembers again.
- 6. When the method returns the second time, check the return value. If successful, the method will return S_OK, and the array will contain the player's IDs.
- 7. If the method returns DPNERR_BUFFERTOOSMALL again, the number of players has increased since the previous method call.

Return to step three and use the new *pcdpnid* value to increase the array size. Be careful not to leak memory.

In some cases, the method returns an array of structures. In that case, you follow the same procedure, but the value returned from the first method call gives you the size of the array in bytes, instead of the number of elements in the array. Refer to the individual method references for details.

For more information, see <u>Enumerating Hosts</u>.

Architect Your DirectPlay Application

Microsoft® DirectPlay® offers many choices about how you structure the networking for your game. So, before you start writing a DirectPlay application, you need to make some decisions about the architecture of your game. The following list details the main issues you'll want to consider before getting started.

- Peer-to-peer or client/server game. Peer-to-peer is simpler to set up and, once deployed, has limited overhead; but client/server works better for large-scale multiplayer games and may provide improved Network Address Translation (NAT) support. For more information, see <u>Peer-to-Peer Topology</u> and <u>Client/Server</u> <u>Topology</u>.
- Multithreading. DirectPlay provides the <u>IDirectPlay8ThreadPool</u> interface, which allows you to control the number of DirectPlay threads in your game. You can set the thread count to zero and call <u>IDirectPlay8ThreadPool::DoWork</u> to perform DirectPlay tasks. This allows you to avoid complex synchronization issues. However, a multithreaded DirectPlay game scales better. For more information, see <u>DirectPlay Callback Functions and Multithreading</u> <u>Issues</u>.
- Connection types, such as local area network (LAN), broadband, or modem. Supporting just one connection type allows you to expect consistency among the send and receive rates of your players and the amount of data they can handle. However, if you want to support different connections, you'll need to include adjustments to keep the maximum send rate at the rate of the slowest connection and control the size of the data packets, or expect significantly higher latency and dropped packets for the slower connections. For more information, see <u>Optimizing Network Usage</u>.
- Reliable or unreliable messaging. Most applications will want to use unreliable messages, because they improve the speed of your

game. You can specify certain messages to be reliable, for data that cannot be lost. For more information, see <u>Message</u> <u>Categories</u>.
Peer-to-Peer Sessions

A peer-to-peer session consists of a collection of users connected by a network. While a lobby server may be used to arrange and launch the game, the messaging needed to run the game is sent directly from one user's to another. Any communication with the lobby server is for such limited purposes as updating the list of participants.

With a peer-to-peer game, everything that is needed to run the game is part of the client software. With no server involved, all the processing needed to create and maintain the game universe must be handled by the client applications. This document discusses the basic principles of a lobbyable Microsoft® DirectPlay® peer-to-peer game. For a simple working example of a peer-to-peer application, see the SimplePeer application included with the software development kit (SDK).

- Initiating a Peer-to-Peer Session
- Enumerating Hosts
- <u>Selecting a Service Provider for a Peer-to-Peer Session</u>
- <u>Selecting a Host for a Peer-to-Peer Session</u>
- <u>Connecting to a Peer-to-Peer Session</u>
- <u>Managing a Peer-to-Peer Session</u>
- Handling DirectPlay Messaging
- Host Migration
- <u>Normal Peer-to-Peer Game Play</u>
- Leaving a Peer-to-Peer Session
- <u>Terminating a Peer-to-Peer Session</u>

© 2003 Microsoft Corporation. All rights reserved.

Initiating a Peer-to-Peer Session

A peer-to-peer game can be launched directly by the user, or *lobby-launched* by a *lobby client* application that resides on the user's computer. This documentation will assume that the game is lobbyable, and can communicate with the lobby client.

One of the first steps you should take is to determine whether your game was lobby-launched. To do so, create and initialize a lobbied application object (CLSID_DirectPlay8LobbiedApplication). When you do so, you pass the object a pointer to your lobbied application message handler. This message handler receives messages directly from the lobbied application object, and indirectly from the lobby client and the lobby.

- If the application was lobby-launched, the <u>IDirectPlay8LobbiedApplication::Initialize</u> method returns a connection handle to the lobby client and a <u>DPL_MSGID_CONNECT</u> message is sent to your lobbied application message handler. The **pdplConnectionSettings** member of the associated structure points to a <u>DPL_CONNECTION_SETTINGS</u> structure that contains connection information such as address objects for the members of the session.
- If the application was not lobby launched, you will receive neither the connection handle, nor the message. However, if you call <u>IDirectPlay8LobbiedApplication::SetAppAvailable</u>, a lobby client can later connect your running application to a session by sending your lobbied application message handler a DPL_MSGID_CONNECT message.

You should also create and initialize a peer object (CLSID_DirectPlay8Peer). This object will be your primary means of communicating with Microsoft® DirectPlay®, and the other users in the session. If you want to have multiple players in the session, you must create a separate instance of this object for each player.

© 2003 Microsoft Corporation. All rights reserved.

Enumerating Hosts

One way to arrange a session is to have session hosts advertise themselves as available. Peers or clients can look for a game to join by enumerating the available hosts, selecting one, and then join the game by sending a connection request. See <u>Peer-to-Peer Sessions</u> or <u>Client/Server Sessions</u> for a detailed discussion.

Unlike other enumerations, the information needed to respond to a request for available hosts is not stored on the local computer. Instead, a client or peer must broadcast a request, for instance on their local subnet, and wait for available hosts to respond. Hosts, on the other hand, must wait for these requests, and then respond appropriately. There are thus two slightly different procedures, depending on whether you are a potential session member, or a session host.

The following procedure illustrates how to enumerate the available hosts for a peer-to-peer session. The procedure for a client/server session is essentially the same. <u>IDirectPlay8Peer::EnumHosts</u> is the method that starts the enumeration. The key parameters to set are *pApplicationDesc*, *pdpaddrDeviceinfo*, and *pdpaddrHost*.

- 1. Assign the globally unique identifier (GUID) of the game you are interested in playing to the **guidApplication** member of the <u>DPN_APPLICATION_DESC</u> structure and assign the structure pointer to the *pApplicationDesc* parameter.
- 2. Create an address object for your device and assign its pointer to *pdpaddrDeviceinfo*. This object contains the information needed to make a network connection.
- 3. To query a specific computer for available hosts, create a host address object for that computer and assign its pointer to *pdpaddrHost*. If you set this parameter to NULL, Microsoft®

DirectPlay® will create an address object from the information contained in *pdpaddrDeviceinfo*. See <u>DirectPlay Addressing</u> for further discussion of address objects. If you are using an Internet Protocol (IP) or Internetwork Packet Exchange (IPX) service provider, the query will then normally be broadcast to your local subnet. If you set the DPNENUMHOSTS_OKTOQUERYFORADDRESSING flag, the service provider may display a dialog box to the user to request address information.

- 4. Call IDirectPlay8Peer::EnumHosts.
- 5. Your callback message handler will then receive a series of <u>DPN_MSGID_ENUM_HOSTS_RESPONSE</u> messages, one for each host that responds.

Examine the information returned to your message handler, select a session, and ask to join it by calling <u>IDirectPlay8Peer::Connect</u>.

If you want to be the host of a session, advertise yourself as available, and wait for queries or connection requests. The following procedure applies to peer-to-peer hosts, but is essentially similar to the procedure for client/server hosts.

- 1. Call <u>IDirectPlay8Peer::SetPeerInfo</u> to specify the static settings for your player.
- 2. Specify the configuration of the game by assigning values the **DPN_APPLICATION_DESC** structure.
- 3. Call <u>IDirectPlay8Peer::Host</u> to advertise yourself as a potential host. Set the *pdnAppDesc* parameter to the **DPN_APPLICATION_DESC** structure defined in the previous step.
- 4. Wait for enumeration requests. They will take the form of a <u>DPN_MSGID_ENUM_HOSTS_QUERY</u> message sent to your callback message handler. If you want to respond to the enumeration request, fill in the **DPN_APPLICATION_DESC** and return S_OK. The peer will receive a

DPN_MSGID_ENUM_HOSTS_RESPONSE message with the information.

5. If the peer decides that they would like to join your session, you will receive a <u>DPN_MSGID_INDICATE_CONNECT</u> message.

See the **Peer-to-Peer Sessions** and **Client/Server Sessions** sections for further discussion of how to arrange and launch a game.

© 2003 Microsoft Corporation. All rights reserved.

Selecting a Service Provider for a Peer-to-Peer Session

The service provider is your network connection. Most games use either the Transmission Control Protocol/Internet Protocol (TCP/IP) or modem service provider, but Microsoft® DirectPlay® also provides support for serial and Internetwork Packet Exchange (IPX) connections.

If your user was connected to the session by a lobby client, you can determine the appropriate service provider by examining the DPL_CONNECTION_SETTINGS structure that accompanies the DPL_MSGID_CONNECT message. Otherwise, you may need to determine which service provider to use, perhaps by querying the user. You can use the peer object's IDirectPlay8Peer::EnumServiceProviders method to enumerate the available service providers. See Getting DirectPlay Data for further discussion.

Once you have selected a service provider, you can then create a DirectPlay address object for your user (a *device address*). You will use this address to identify your device with a number of DirectPlay methods. See <u>DirectPlay Addressing</u> for a detailed discussion of DirectPlay addresses and address objects.

© 2003 Microsoft Corporation. All rights reserved.

Selecting a Host for a Peer-to-Peer Session

Although most aspects of peer-to-peer games can be handled by the various users' communicating directly with each other, there are some tasks that must have a single owner. These tasks are handled by the game host. To join a session, you must know the address of the session's host. A common way to select a host is through a lobby server. In that case, when a user's application is connected to the session, the connection settings that you receive with the DPL_MSGID_CONNECT message include the host's address object. To find out who the session host is:

- Check the dwFlags member of the <u>DPL_CONNECTION_SETTINGS</u> structure that is returned. If that member is set to DPLCONNECTSETTINGS_HOST, your system is the host.
- If the DPLCONNECTSETTINGS_HOST flag is not set, then you can get the address of the host from the **pdp8HostAddress** member.

You can also create a session, perhaps on a local area network (LAN) subnet, by advertising yourself as a session host. To do so call IDirectPlay8Peer::SetPeerInfo to set the player's name and then call IDirectPlay8Peer::Host to advertise yourself as a potential host. You specify the configuration of the game by assigning values the DPN_APPLICATION_DESC structure that is passed through the pdnAppDesc parameter of IDirectPlay8Peer::Host.

To allow your user to examine the available sessions and hosts, you can enumerate the available hosts by calling <u>IDirectPlay8Peer::EnumHosts</u>. When the user has selected a session, you can request a connection.

 $\ensuremath{\textcircled{\sc 0}}$ 2003 Microsoft Corporation. All rights reserved.

Connecting to a Peer-to-Peer Session

Unless you are the session host, you will need to connect your player to the session. To do so, you must have the address of the session host. If your application was connected by a lobby client, you can obtain the host's address by calling

IDirectPlay8LobbiedApplication::GetConnectionSettings. You can also obtain the address by enumerating the available hosts. The information returned by the enumeration includes each host's addresses, and a DPN_APPLICATION_DESC structure that describes the associated session.

To ask to join a session, call <u>IDirectPlay8Peer::SetPeerInfo</u> to set your player's name, and then call <u>IDirectPlay8Peer::Connect</u> with the selected host's address to connect to the session.

When a player attempts to join a session, the host receives a <u>DPN_MSGID_INDICATE_CONNECT</u> message. To accept the player into the session, return S_OK. Returning any other value rejects the request. In either case, the player will receive a

DPN_MSGID_CONNECT_COMPLETE message that contains your response. If the host accepted the connection, the **hResultCode** member of the associated structure will be set to S_OK. If not, **hResultCode** will be set to DPNERR_HOSTREJECTEDCONNECTION.

The host can define a player context value when it receives the **DPN_MSGID_INDICATE_CONNECT** message, however the player identifier (ID) will not yet be defined. The host can also wait to define a player context value until it receives a <u>DPN_MSGID_CREATE_PLAYER</u> message, which includes the player ID. Ordinary players to not receive a

DPN_MSGID_INDICATE_CONNECT message.

Once the new player is connected, each member of the session, including the host, receives a **DPN_MSGID_CREATE_PLAYER** message announcing the new player. The structure associated with the message contains the player ID that you will use to send messages to that player. Peers that are not hosts must define the player context value when they handle this message. When a peer or host has returned from handling this message, that player context value is set for the session, and cannot be changed. See <u>Using Player Context Values</u> for more discussion of player context values.

 $\ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.

Managing a Peer-to-Peer Session

The session host is responsible for managing the session, including:

- Managing the list of session members and their network addresses
- Deciding whether a new user is allowed to join the session.
- Notifying all members when a new user joins the session, and passing them the new user's address.
- Providing new users with the current game state
- Notifying all users when a user leaves the session

When players attempt to join a session, the host will receive a <u>DPN_MSGID_INDICATE_CONNECT</u> message. To accept the player into the session return S_OK. Returning any other value rejects the request. In either case, the player will receive a

<u>DPN_MSGID_CONNECT_COMPLETE</u> message that contains your response.

The host can remove a player from the session by calling <u>IDirectPlay8Peer::DestroyPeer</u>. Other members of the session cannot call this method successfully. If you want to allow players to request that another player be removed from the session, you must send the request to the host with normal Microsoft® DirectPlay® messaging, and have the host handle the request.

^{© 2003} Microsoft Corporation. All rights reserved.

Handling DirectPlay Messaging

This set of documents provides information about how to handle Microsoft® DirectPlay® messaging. This particular document introduces general considerations for all DirectPlay messaging. Details for messaging with particular session types and roles can be found in the following documents.

- Handling Standard Peer-to-Peer Messages
- <u>Peer-to-Peer Host Messages</u>

General Messaging Considerations

The following general considerations apply to all DirectPlay messaging.

- You will frequently receive a message following a method call. For instance, when you ask to join a host's session by calling IDirectPlay8Peer::Connect, you receive a DPN_MSGID_CONNECT_COMPLETE message that provides the result of your request. Two events thus follow your method call: The method returns and you receive a message. The order in which these events occur depend on whether you call the method synchronously or asynchronously, and may not be predictable.
- By default, DirectPlay performs most operations asynchronously. To use synchronous operations, you typically need to set an *xxx*_SYNC flag in the method's *dwFlags* parameter. As long as an asynchronous method call does not fail for reasons such as parameter validation, it immediately returns
 DPNSUCCESS_PENDING. DirectPlay notifies you that the operation is complete by sending you a message such as DPN_MSGID_SEND_COMPLETE or
 DPN_MSGID_ASYNC_OP_COMPLETE. These messages can arrive before or after the method returns.
- DirectPlay signals the end of most asynchronous operations by sending you a completion message, usually
 DPN_MSGID_ASYNC_OP_COMPLETE. Two exceptions to this rule occur: Send/SendTo, and Connect operations. The end of those operations is signaled by DPN_MSGID_SEND_COMPLETE and DPN_MSGID_CONNECT_COMPLETE, respectively. Synchronous operations do not return until the operation is complete. For that reason, most do not generate a completion message. The exception is synchronous connection operations that may generate a DPN_MSGID_CONNECT_COMPLETE message. See Handling Standard Peer-to-Peer Messages for further discussion.
- You can force many asynchronous operations to terminate at any

time by calling <u>IDirectPlay8Peer::CancelAsyncOperation</u>. Group operations cannot be terminated.

- By default, DirectPlay serializes the messages associated with each player. As long as the DPNSEND_NONSEQUENTIAL flag is not set, you receive messages from a particular player in the order in which they are sent. This is true for messages sent to you directly, and to you as a member of a group. When a player sends a message to your group, the message appears to you as coming from the player, not from the group. You receive all messages from a particular player on one thread at a time, regardless of whether DPNSEND_NONSEQUENTIAL is set. You will not receive messages from a player before you have processed the corresponding <u>DPN_MSGID_CREATE_PLAYER</u> message, or after you have processed the corresponding <u>DPN_MSGID_DESTROY_PLAYER</u> message.
- Group messages are also serialized. For instance, you will not receive a <u>DPN_MSGID_ADD_PLAYER_TO_GROUP</u> message until you have processed <u>DPN_MSGID_CREATE_GROUP</u>. You will not receive any group-related messages after you have processed the corresponding <u>DPN_MSGID_DESTROY_GROUP</u> message.
- Create and destroy messages are always paired. For instance, every **DPN_MSGID_CREATE_GROUP** message will be matched by a corresponding **DPN_MSGID_DESTROY_GROUP** message.
- You should spend as little time in your message handler as possible. Executing a long blocking operation while handling a DirectPlay message may seriously impede network performance. For example, when you send data to another player in a peer-to-peer session by calling <u>IDirectPlay8Peer::SendTo</u> within your message handler, do so asynchronously. This allows you to return from the message handler as soon as the method returns. You can process the result of the operation when the DPN_MSGID_SEND_COMPLETE message arrives.

© 2003 Microsoft Corporation. All rights reserved.

Handling Standard Peer-to-Peer Messages

This document describes how to handle Microsoft® DirectPlay® messaging for a normal member of a peer-to-peer session. It does not discuss messages that are specific to a host. For a discussion of host-related messaging, see <u>Peer-to-Peer Host Messages</u>. For a discussion of general messaging issues, see <u>Handling DirectPlay Messaging</u>.

- <u>Startup Messages</u>
- <u>Messaging During Normal Game Play</u>
- <u>Session Termination Messages</u>

Startup Messages

The host can set up a peer-to-peer session in two ways.

- Arrange the session in advance. For example, you can use an online lobby to collect a group of players prior to starting the session. The player selected as host is responsible for calling <u>IDirectPlay8Peer::Host</u> to start the session. The lobby typically provides the host's address to the other players, and they use that address to connect directly to the session. Players usually need not do a host enumeration. See <u>DirectPlayLobby</u> for details on how to handle lobbies.
- Create a stand-alone session. When a player does a host enumeration, an enumeration query is sent to every host at the specified network location that matches the player's description of the type of session they want to join. If the host chooses to respond, the host passes the player the information the player needs to connect to the session. With a stand-alone session, there is no need for a lobby. The host calls **IDirectPlay8Peer::Host** and assembles the group of players by responding to enumeration queries and connection attempts.

This section describes the messages you may receive when selecting and joining a session.

<u>DPN_MSGID_ENUM_HOSTS_RESPONSE</u>

If you need to locate a standalone session, call <u>IDirectPlay8Peer::EnumHosts</u> to enumerate the available hosts. You will receive a **DPN_MSGID_ENUM_HOSTS_RESPONSE** message for each host that responds to your enumeration request. You may receive multiple

DPN_MSGID_ENUM_HOSTS_RESPONSE messages from the same host.

By default, host enumeration is performed asynchronously. When an asynchronous enumeration terminates, for instance after the retry count is reached, you will be notified with a DPN_MSGID_ASYNC_OP_COMPLETE message. You will receive no further DPN_MSGID_ENUM_HOSTS_RESPONSE messages after DPN_MSGID_ASYNC_OP_COMPLETE arrives. You can cancel the operation at any time by calling IDirectPlay8Peer::CancelAsyncOperation. You can also cancel a host enumeration by calling IDirectPlay8Peer::Connect to connect to a session. The enumeration is halted as soon as the connect request completes successfully.

Note You should call **IDirectPlay8Peer::Connect** outside of the **DPN_MSGID_ENUM_HOSTS_RESPONSE** message handler.

<u>DPN_MSGID_CONNECT_COMPLETE</u>

Typically, the next step you will take is to attempt to join a session by calling **IDirectPlay8Peer::Connect**. You normally receive a **DPN_MSGID_CONNECT_COMPLETE** message with the host's response, even if the host rejects your connection request. You can receive this message for asynchronous and synchronous calls, but there are differences in detail.

 Asynchronous calls: When the connection attempt is underway, IDirectPlay8Peer::Connect normally returns DPNSUCCESS_PENDING. When you get this return value, you will always receive a
 DPN_MSGID_CONNECT_COMPLETE message containing the outcome of the connection attempt. However, the message might be indicated on a different thread before
 IDirectPlay8Peer::Connect returns. If the method returns DPNSUCCESS_PENDING, you also receive an asynchronous operation handle that you can use to cancel the operation by calling

IDirectPlay8Peer::CancelAsyncOperation. This handle is valid only until you receive the DPN_MSGID_CONNECT_COMPLETE message. If IDirectPlay8Peer::Connect does not return DPNSUCCESS_PENDING, you will not receive a DPN_MSGID_CONNECT_COMPLETE message. Typically,

this occurs when there is a problem with parameter validation, memory allocation, or addressing.

• Synchronous calls: The method returns after the connection attempt is completed. You may receive a **DPN MSGID CONNECT COMPLETE** message, as well as a return value. However, the message is guaranteed to arrive before the method returns. If your connection attempt is successful, you will receive a **DPN MSGID CONNECT COMPLETE**, followed by a return value of DPN OK. You may also receive a **DPN MSGID CONNECT COMPLETE** message before other return values, such as **DPNERR HOSTREJECTEDCONNECTION.** You will not receive DPN MSGID CONNECT COMPLETE if the method call fails because of parameter validation, memory allocation, or addressing problems. You do not receive an asynchronous operation handle and cannot use IDirectPlay8Peer::CancelAsyncOperation to cancel a synchronous connection attempt.

• <u>DPN_MSGID_CREATE_PLAYER</u> and <u>DPN_MSGID_CREATE_GROUP</u>

If your connection attempt was successful, you will receive a **DPN_MSGID_CREATE_PLAYER** message for each player in the session at the time you joined. At a minimum, you will receive one message for the host and one for your player. If any groups have been created, you will receive a **DPN_MSGID_CREATE_GROUP** message for each group. You will also receive a series of

DPN_MSGID_ADD_PLAYER_TO_GROUP messages for each group, one for each player in the group. All of these messages will arrive before DPN_MSGID_CONNECT_COMPLETE.

You must define your player and group context values when you handle the associated creation messages. These context values can also be defined earlier, when IDirectPlay8Peer::Host, IDirectPlay8Peer::CreateGroup methods are called. You can change those player or group context values when you handle DPN_MSGID_CREATE_PLAYER or DPN_MSGID_CREATE_GROUP respectively. However, once that handler returns, the corresponding context value cannot be changed again. See Using Player Context Values for further discussion.

Messaging During Normal Game Play

During a game, you might receive any of the following messages.

• DPN_MSGID_CREATE_PLAYER and DPN_MSGID_DESTROY_PLAYER

Players can typically enter or leave while the game is in progress. You receive a DPN_MSGID_CREATE_PLAYER when a player enters a game, and a DPN_MSGID_DESTROY_PLAYER message when that player leaves. You might receive DPN_MSGID_CREATE_PLAYER and DPN_MSGID_DESTROY_PLAYER messages simultaneously on different threads, but only for different players. You will not receive a DPN_MSGID_DESTROY_PLAYER message before your callback function has returned from receiving the corresponding DPN_MSGID_CREATE_PLAYER message. If you want to create a player context value, you must do so before you return from the DPN_MSGID_CREATE_PLAYER message handler.

• DPN_MSGID_CREATE_GROUP and DPN_MSGID_DESTROY_GROUP

Many games simplify messaging by allowing players to be grouped. When a group is created, you receive a DPN_MSGID_CREATE_GROUP message, regardless of whether your player is a member. If you want to create a group context value, you must do so before you return from the DPN_MSGID_CREATE_GROUP message handler. When a group is destroyed, you receive a DPN_MSGID_DESTROY_GROUP message. You will not receive a DPN_MSGID_DESTROY_GROUP message before your callback function has returned from processing the corresponding **DPN_MSGID_CREATE_GROUP** message.

• DPN_MSGID_ADD_PLAYER_TO_GROUP and DPN_MSGID_REMOVE_PLAYER_FROM_GROUP

You receive **DPN_MSGID_ADD_PLAYER_TO_GROUP** messages for each group member after a group is created, and when a player is added to a group. You receive a

DPN_MSGID_REMOVE_PLAYER_FROM_GROUP message when a player is removed from a group, and for all the remaining players in the group when the group is destroyed. You will not receive a DPN_MSGID_REMOVE_PLAYER_FROM_GROUP message before your callback function has returned from processing the corresponding

DPN_MSGID_ADD_PLAYER_TO_GROUP message.

You are guaranteed to receive every

DPN_MSGID_REMOVE_PLAYER_FROM_GROUP message for a group before you receive the DPN_MSGID_DESTROY_GROUP message. When a player is destroyed, you will receive a DPN_MSGID_REMOVE_PLAYER_FROM_GROUP message for every group in which that player is a member, before you receive the player's DPN_MSGID_DESTROY_PLAYER message.

DPN_MSGID_SEND_COMPLETE

You send data to a player or group by calling <u>IDirectPlay8Peer::SendTo</u>. If you call this message asynchronously, the method will normally return <u>DPNERR_PENDING</u>. You receive a **DPN MSGID SEND COMPLETE** message when the data is sent. The **DPN_MSGID_SEND_COMPLETE** message can arrive before or after the method returns.

Receiving a DPN_MSGID_SEND_COMPLETE message does not necessarily mean that the target has received and processed the message. By default, if the network is overloaded, packets may be dropped. If you want to be certain that the message has arrived at the target, set the DPNSEND_GUARANTEED flag when you call IDirectPlay8Peer::SendTo. You will not receive a DPN_MSGID_SEND_COMPLETE message until DirectPlay has verified that the target has received the data. If you want to be certain that the message has also been processed by the target, set the DPNSEND_COMPLETEONPROCESS flag. If you set this flag, you will not receive a DPN_MSGID_SEND_COMPLETE message until the target's message handler has processed the message and returned.

Note Setting the DPNSEND_GUARANTEED and DPNSEND_COMPLETEONPROCESS flags add overhead to the Send process. You may want to design your communication process to be able to tolerate some data loss in return for increased performance.

DPN_MSGID_RECEIVE

When another player sends you data, it is delivered to your message handler with a **DPN_MSGID_RECEIVE** message. By default, the buffer containing the data is valid only until you return from the message handler. If you want to retain control over the data buffer after your message handler returns, have your message handler return DPNERR_PENDING. This return value will prevent DirectPlay from freeing or modifying the buffer. When you no longer need the buffer, you must return it to the control of

DirectPlay by calling <u>IDirectPlay8Peer::ReturnBuffer</u>.

• <u>DPN_MSGID_PEER_INFO</u>, <u>DPN_MSGID_GROUP_INFO</u>, and <u>DPN_MSGID_APPLICATION_DESC</u>

Information structures are associated with the application, and with each player and group in the session. If this information changes during a session, you will receive the corresponding message. To obtain up-to-date information, you must call <u>IDirectPlay8Peer::GetApplicationDesc</u>, <u>IDirectPlay8Peer::GetGroupInfo</u>, or <u>IDirectPlay8Peer::GetPeerInfo</u>.

• <u>DPN_MSGID_HOST_MIGRATE</u>

A session must have a host. If the host set the DPNSESSION_MIGRATE_HOST flag when it created the session and then leaves the session without terminating it, DirectPlay chooses a new host. DirectPlay notifies the remaining players of the change by sending them a **DPN_MSGID_HOST_MIGRATE** message with the new host's identifier (ID).

Session Termination Messages

Sessions terminate for a variety of reasons. Typically, the session terminates when the host calls <u>IDirectPlay8Peer::TerminateSession</u>. If host migration is not permitted, the session terminates when the host calls <u>IDirectPlay8Peer::Close</u> or is involuntarily disconnected.

<u>DPN_MSGID_TERMINATE_SESSION</u>

The DPN_MSGID_TERMINATE_SESSION message notifies you that the session is over. It is followed by a series of DPN_MSGID_DESTROY_PLAYER, DPN_MSGID_DESTROY_GROUP, and DPN_MSGID_REMOVE_PLAYER_FROM_GROUP messages.

• DPN_MSGID_DESTROY_PLAYER

You will receive a **DPN_MSGID_DESTROY_PLAYER** message for each player in the session. If the host has intentionally terminated the session, this is considered normal behavior and the **dwReason** member of the associated structure is set to DPNDESTROYPLAYERREASON_NORMAL. The DPNDESTROYPLAYERREASON_SESSIONTERMINATED value is set only for unexpected disconnections from a session that does not allow host migration.

• DPN_MSGID_DESTROY_GROUP and DPN_MSGID_REMOVE_PLAYER_FROM_GROUP

If any groups have been formed, you will receive DPN_MSGID_REMOVE_PLAYER_FROM_GROUP messages for each member of each group, and a DPN_MSGID_DESTROY_GROUP message for each group itself. The DPN_MSGID_REMOVE_PLAYER_FROM_GROUP messages will always arrive before the corresponding DPN_MSGID_DESTROY_PLAYER or DPN_MSGID_DESTROY_GROUP messages.

© 2003 Microsoft Corporation. All rights reserved.

Peer-to-Peer Host Messages

This document describes how to handle Microsoft® DirectPlay® messaging for the host of a peer-to-peer session. It discusses only those messages that are specific to a host. For a discussion of messaging for a normal peer, see <u>Handling Standard Peer-to-Peer Messages</u>. For a discussion of general messaging issues, see <u>Handling DirectPlay</u> <u>Messaging</u>.

- <u>Starting a Session</u>
- Host Migration and Session Termination
Starting a Session

You can set up a peer-to-peer session in two ways:

- Arrange the session in advance. For example, you can use an online lobby to collect a group of players prior to starting the session. The player selected as host is responsible for calling <u>IDirectPlay8Peer::Host</u> to start the session. The lobby typically provides the host's address to the other players, and they use that address to connect directly to the session. Players usually need not do a host enumeration. For details on how to handle lobbies, see <u>DirectPlay Lobby</u>.
- Create a stand-alone session. When a player does a host enumeration, an enumeration query is sent to every host at a specified network location, which matches the player's description of the type of session they want to join. If the host chooses to respond, the host passes the player the information needed to connect to the session. With a stand-alone session, there is no need for a lobby. The host calls IDirectPlay8Peer::Host and assembles the group of players by responding to enumeration queries and connection attempts.

In either case, you must start the session by calling IDirectPlay8Peer::Host. When you start the session, you must be prepared to handle the following messages:

• <u>DPN_MSGID_ENUM_HOSTS_QUERY</u>

If you have started a stand-alone session, a potential player must be able find it before the player can connect. Players who are looking for a stand-alone session call

IDirectPlay8Peer::EnumHosts to enumerate the available sessions. They indicate the type of session they are interested in by filling in the **guidApplication** member of the DPN_APPLICATION_DESC structure. They can also specify the network location they are interested in by adding the appropriate items to the host address object, which is passed to the *pdpaddrHost* parameter of **IDirectPlay8Peer::EnumHosts**. For more information about how to handle DirectPlay address objects, see <u>DirectPlay Addressing</u>.

During enumeration, DirectPlay sends each available host a **DPN_MSGID_ENUM_HOSTS_QUERY** that includes the address of the player requesting the enumeration. If the player specified an application globally unique identifier (GUID), this message will go to only those hosts that have specified the same GUID. If the player does not specify an application GUID, DirectPlay will send **DPN_MSGID_ENUM_HOSTS_QUERY** to all available hosts at the specified network location.

You can accept the enumeration request by having your handler return <u>DPN_OK</u>. DirectPlay will then pass information about the session, specifically your address and the

DPN_APPLICATION_DESC structure, to the player. You can also provide some application-specific data to the player by assigning a data buffer to the **DPNMSG_ENUM_HOSTS_QUERY** structure's **pvResponse** member. When this buffer is no longer needed, DirectPlay notifies you that it is safe to free the buffer by sending you a <u>DPN_MSGID_RETURN_BUFFER</u> message.

You can reject the enumeration request by having your message handler return a value other than DPN_OK. In that case, no information will be sent to the player. If you reject an enumeration request, there is no need for a response buffer, and you will not receive a **DPN_MSGID_RETURN_BUFFER** message. **Note** You can receive a **DPN_MSGID_ENUM_HOSTS_QUERY** message at any time while you are a host. If you are hosting an arranged session, or do not want to add any more players, you must still handle this message and reject the enumeration request.

DPN_MSGID_INDICATE_CONNECT

Once a player has the address of a host, the player can attempt to connect to that session by calling <u>IDirectPlay8Peer::Connect</u>. The session host then receives a **DPN_MSGID_INDICATE_CONNECT** message with information about the player.

To accept a player's connection request, have your DPN_MSGID_INDICATE_CONNECT message handler return DPN_OK. As with DPN_MSGID_ENUM_HOSTS_QUERY, you can also return a data buffer to the player by assigning it to the DPNMSG_INDICATE_CONNECT structure's pvReplyData member. When this buffer is no longer needed, DirectPlay sends you a DPN_MSGID_RETURN_BUFFER message to notify you that you can safely free the buffer.

You can also create a local player context value for the player at this time. That value will be passed back to you later when you receive the a <u>DPN_MSGID_CREATE_PLAYER</u> message for the player. You have the option of changing the player context value when you handle that message. See <u>Using Player Context Values</u> for more information about player context values.

You can reject a connection request by having your handler return any value other than DPN_OK. In that case, there is no need for a response buffer and you will not receive a **DPN_MSGID_RETURN_BUFFER** message. You can return a data buffer to the rejected player by assigning it to the **DPNMSG_INDICATE_CONNECT** structure's **pvReplyData** member. When this buffer is no longer needed, DirectPlay sends you a **DPN_MSGID_RETURN_BUFFER** message to notify you that you can safely free the buffer.

You can also prevent connections by limiting the number of players in the session. When you call **IDirectPlay8Peer::Host** or **IDirectPlay8Peer::SetApplicationDesc**, set the **dwMaxPlayers** member of the **DPN_APPLICATION_DESC** structure to the maximum number of players you will allow in the session. When the number of players in the session reaches your specified maximum, you will receive no more **DPN_MSGID_INDICATE_CONNECT** messages. Instead,

additional connection attempts will automatically fail, returning <u>DPNERR_SESSIONFULL</u>.

• DPN_MSGID_CREATE_PLAYER

After you have returned DPN_OK to accept a player's connection request, you will receive a corresponding

DPN_MSGID_CREATE_PLAYER message. If you want to create a player context value or modify the value you created when you handled DPN_MSGID_INDICATE_CONNECT, you must do so when you handle DPN_MSGID_CREATE_PLAYER. When your DPN_MSGID_CREATE_PLAYER message handler returns, DirectPlay permits no further changes to the player context value.

• <u>DPN_MSGID_INDICATED_CONNECT_ABORTED</u>

If a player drops the connection after the host has processed the **DPN_MSGID_INDICATE_CONNECT** message but before it has processed **DPN_MSGID_CREATE_PLAYER**, the host receives a

DPN_MSGID_INDICATED_CONNECT_ABORTED message. If you receive this message, free any memory that you allocated while processing DPN_MSGID_INDICATE_CONNECT. If you process DPN_MSGID_CREATE_PLAYER before the player disconnects, you will receive a DPN_MSGID_DESTROY_PLAYER message. You can free the memory you have allocated when you process that message. If the player disconnects while you are processing DPN_MSGID_CREATE_PLAYER, you will receive a DPN_MSGID_DESTROY_PLAYER message after your message handler returns from processing

DPN_MSGID_CREATE_PLAYER.

Host Migration and Session Termination

Every peer-to-peer session must have a host. However, the original host may leave the session either by calling <u>IDirectPlay8Peer::Close</u> or by being disconnected. This situation can be handled in one of two ways:

- When host leaves the session, it terminates.
- When the host leaves the session, a new host is chosen.

The original host specifies which of those two options is to be followed when it calls **IDirectPlay8Peer::Host** to start the session. If it enables "host migration" by setting the DPNSESSION_MIGRATE_HOST flag in the **DPN_APPLICATION_DESC** structure, a new host will be chosen when the current host leaves. However, even if host migration is enabled, the host can force the session to terminate by calling IDirectPlay8Peer::TerminateSession. If host migration is not enabled, the session terminates when the host leaves. The host will then receive the same messages as all other members of the session. See **Handling Standard Peer-to-Peer Messages** for details.

Host Migration

While the host must be one of the initial members of the session, they may choose to leave before session is finished. When the host leaves the session, there are two possible outcomes:

- The session terminates.
- The host *migrates*, and another user becomes host.

Sessions may or may not permit host migration. To enable host migration, the session organizer must set the DPNSESSION_MIGRATE_HOST flag in the **dwFlags** member of the <u>DPN_APPLICATION_DESC</u> structure when they set up the game. If this flag is not set, the session terminates when the host leaves.

If the DPNSESSION_MIGRATE_HOST flag is set, the host can still force the session to terminate by calling <u>IDirectPlay8Peer::TerminateSession</u>.

If DPNSESSION_MIGRATE_HOST flag is set and the host leaves the session, Microsoft® DirectPlay® will select a new session host. All remaining session members will receive a

<u>DPN_MSGID_HOST_MIGRATE</u> message that includes the identifier (ID) of the new host.

 $\ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.

Normal Peer-to-Peer Game Play

In Microsoft® DirectPlay®, a message is essentially a block of gamerelated data that you send to one or more members of the session. DirectPlay does not specify the contents or format of the data block, it just provides a mechanism to transmit the data from one user to another. When the game is underway, each session member will normally send a constant stream of messages to all other members of the session for the duration of the game. The primary purpose of these messages is to keep the game state synchronized, so that each user's application displays the same user interface (UI). However, messages can also be used for a variety of other game-specific purposes.

For many games, especially rapidly changing ones, you may have to manage your messaging carefully. DirectPlay throttles outgoing messages to a level that can be handled by the target. You will have be careful that you do not send messages too rapidly, and ensure that the most important messages get through. See <u>Basic Networking</u> for a discussion of how to effectively handle DirectPlay messaging.

To send a message to another session member, call <u>IDirectPlay8Peer::SendTo</u>. That member will receive a <u>DPN_MSGID_RECEIVE</u> message with the data. To send a message to a specific player, set the *dpnid* parameter to the player identifier (ID) that you received with the associated <u>DPN_MSGID_CREATE_PLAYER</u> message. You can also send a message to every player in the session by setting *dpnid* to DPNID_ALL_PLAYERS_GROUP. You can also define groups of players, and use a single **IDirectPlay8Peer::SendTo** call to send a message to all members of a group.

Note You can also use the <u>IDirectPlay8Peer::SetPeerInfo</u> method to

send information to other users. They will receive the information with a <u>DPN_MSGID_PEER_INFO</u> message. However, this way of transmitting information is not very efficient, and should not be used for normal messaging.

Using Groups

Many games allow players to be organized into groups. For instance, strategy games typically allow individual players to be organized into groups that can then be directed as a single entity. DirectPlay also allows the formation of groups of players. DirectPlay groups are essentially a way to simplify your messaging. When you have defined a group, you can send a message to every group member with a single call to **IDirectPlay8Peer::SendTo**. While DirectPlay groups normally correspond to the groups that are defined by the game, you are free to create a group for any reason.

To create a DirectPlay group, call <u>IDirectPlay8Peer::CreateGroup</u> All session members will then receive a <u>DPN_MSGID_CREATE_GROUP</u> message with the details. The message will include a group ID that is used to send messages to the group.

Once the group is created, you then add players by calling <u>IDirectPlay8Peer::AddPlayerToGroup</u>. Session members will then receive a <u>DPN_MSGID_ADD_PLAYER_TO_GROUP</u> message with the IDs of the group and the player that was just added.

Once the group is established, you can send data to the group by calling **IDirectPlay8Peer::SendTo**, with the *dpnid* parameter set to the group ID. All group members will then receive a **DPN_MSGID_RECEIVE** message with the data.

To remove a player from a group, call <u>IDirectPlay8Peer::RemovePlayerFromGroup</u>. The session members will receive a <u>DPN_MSGID_DESTROY_PLAYER</u> message with the player's ID. Finally, when you no longer need the group, you can destroy it by calling <u>IDirectPlay8Peer::DestroyGroup</u> All session members will then receive a <u>DPN_MSGID_DESTROY_GROUP</u> message with the group ID.

Leaving a Peer-to-Peer Session

To leave a session, terminate the connection by calling <u>IDirectPlay8Peer::Close</u>. The session members will be notified with a <u>DPN_MSGID_DESTROY_PLAYER</u> message.

If you are the session host, leaving also terminates the session unless you configured the session to allow host migration. See <u>Host Migration</u> for details.

Terminating a Peer-to-Peer Session

When the session is over, the host should terminate the session by calling <u>IDirectPlay8Peer::TerminateSession</u>. This method terminates the session even if host-migration is enabled. All session members will be notified by a <u>DPN_MSGID_TERMINATE_SESSION</u> message. You should then perform any necessary cleanup. To start another session, you must first call <u>IDirectPlay8Peer::Close</u>, and then <u>IDirectPlay8Peer::Initialize</u>.

If you registered your application as available for connection by calling <u>IDirectPlay8LobbiedApplication::SetAppAvailable</u>, a lobby client can offer to connect you to a new session by sending your lobbied application message handler a <u>DPL_MSGID_CONNECT</u> message. You must have first called **IDirectPlay8Peer::Close** and **IDirectPlay8Peer::Initialize**.

Client/Server Sessions

A client/server session consists of a collection of players, or clients, connected to a central server. As far as Microsoft® DirectPlay® is concerned, a client has no knowledge of any other clients, only the server. The messaging needed to run the game is between the individual clients and the server. DirectPlay does not provide direct client-to-client messaging, as it does for peer-to-peer sessions.

A client/server session requires two distinctly different applications.

- The server application runs on a remote server. At a minimum, it serves as a central messaging hub and game host. The server must receive and handle all incoming messages from the clients, and send appropriate messages back out. Any transfer of data from one client to another must be handled by the server application.
- A client application runs on each players' computer. The primary function of this application is to handle the UI, and keep the player's game state synchronized with the server.

There are certain aspects of the session that can be handled by only one of these applications. For instance, updating a player's video display can only be done by the client application. However, many aspects of the processing needed to maintain the game universe can, at least in principle, be done by either application. Writing an effective client/server game requires some careful consideration of how to divide the processing chores between the two applications.

This document describes the basic principles of client server games, and outlines how to implement client and server applications.

• Initiating a Client/Server Session

- <u>Selecting a Service Provider for a Client</u>
- <u>Selecting a Client/Server Host</u>
- <u>Connecting to a Client/Server Session</u>
- <u>Managing a Client/Server Session</u>
- Handling Client/Server Messages
- <u>Normal Client/Server Game Play</u>
- Leaving a Client/Server Session
- <u>Terminating a Client/Server Session</u>

A client/server game can be launched through a lobby, or directly by the server application.

The Server Application

Client/server games are often arranged through lobbies. The most straightforward way to launch the server is to implement it as a *lobbyable* application. This approach provides a way to launch the server, and supports communication between server and lobby during the course of the session. See <u>DirectPlay Lobby</u> for further discussion.

A server can also be directly launched, and then advertise itself as available and wait for clients to connect. See <u>Selecting a Client/Server</u> <u>Host</u> for details.

Once the server application has been launched, it should initialize itself by calling <u>IDirectPlay8Server::Initialize</u>. As with other similar Microsoft® DirectPlay® methods, the primary purpose of initialization is to provide DirectPlay with a pointer to your callback message handler. You should also call <u>IDirectPlay8Server::SetServerInfo</u> to describe the current game. Clients cannot connect to a server until this method has been called.

The Client Application

One of the first steps you should take is to determine whether your game was lobby-launched. To do so, create and initialize a lobbied application object (CLSID_DirectPlay8LobbiedApplication). When you do so, you pass the object a pointer to your lobbied application message handler. This message handler receives messages directly from the lobbied application object, and indirectly from the lobby client and the lobby.

- If the application was lobby-launched, the <u>IDirectPlay8LobbiedApplication::Initialize</u> method returns a connection handle to the lobby client and a <u>DPL_MSGID_CONNECT</u> message is sent to your lobbied application message handler. The **pdplConnectionSettings** member of the associated structure points to a <u>DPL_CONNECTION_SETTINGS</u> structure that contains connection information such as an address object for the server.
- If the application was not lobby launched, you will receive neither the connection handle, nor the message. However, if you call <u>IDirectPlay8LobbiedApplication::SetAppAvailable</u>, a lobby client can later connect your running application to a session by sending your lobbied application message handler a DPL_MSGID_CONNECT message.

You should also create and initialize a client object (CLSID_DirectPlay8Client). This object will be your primary means of communicating with DirectPlay and the server. If you want to have multiple players in the session, you must create a separate instance of this object for each player.

Selecting a Service Provider for a Client

The service provider is your network connection. Most games use either the Transmission Control Protocol/Internet Protocol (TCP/IP) or modem service provider, but Microsoft® DirectPlay® also provides support for serial and Internetwork Packet Exchange (IPX) connections.

If your user was connected to the session by a lobby client, you can determine the appropriate service provider by examining the DPL_CONNECTION_SETTINGS structure that accompanies the DPL_MSGID_CONNECT message. Otherwise, you may need to determine which service provider to use, perhaps by querying the user. You can use the client object's IDirectPlay8Client::EnumServiceProviders method to enumerate the available service providers. See Getting DirectPlay Data for further discussion.

Once you have selected a service provider, you can then create a DirectPlay address object for your user (a *device address*). You will use this address to identify your device with a number of DirectPlay methods. See <u>DirectPlay Addressing</u> for a detailed discussion of DirectPlay addresses and address objects.

Selecting a Client/Server Host

By definition, the server application hosts the session. To join a session, a client application must determine the host server's address. A common way to select a host is through a lobby server. In that case, when a user's application is connected to the session, the connection settings that you receive with the DPL_MSGID_CONNECT message include the host's address object. The **pdp8HostAddress** member of the associate structure points to an address object with the host's address.

Servers using an Internet Protocol (IP) or Internetwork Packet Exchange (IPX) service provider can also create a session perhaps on a local area network (LAN) subnet, by advertising themselves as session hosts. To create a session, call <u>IDirectPlay8Server::SetServerInfo</u> to specify the server settings. Then call <u>IDirectPlay8Server::Host</u> to advertise the server as a session host. You specify the configuration of the game by assigning values to the <u>DPN_APPLICATION_DESC</u> structure that is passed through the *pdnAppDesc* parameter of **IDirectPlay8Server::Host**.

To allow your user to view the available sessions and hosts, a client application can query for available hosts by calling <u>IDirectPlay8Client::EnumHosts</u>. When the user has selected a host, you can request a connection.

Connecting to a Client/Server Session

All clients must explicitly join the session by connecting to the host, even if the session has been arranged through a lobby. A connection establishes the client as a member of the session, and provides the host with the information it needs to communicate with the client. The host has the option of accepting or rejecting a connection request.

The Server Application

When a client attempts to join a session, the host receives a <u>DPN_MSGID_INDICATE_CONNECT</u> message. To accept the player into the session, return S_OK. Returning any other value rejects the request. In either case, the client will receive a <u>DPN_MSGID_CONNECT_COMPLETE</u> message that contains your response. You can define a player context value at this time, or wait until

you receive a <u>DPN_MSGID_CREATE_PLAYER</u> message. See <u>Using</u> <u>Player Context Values</u> for more discussion of player context values.

If the player is successfully added to the session, the server will receive a **DPN_MSGID_CREATE_PLAYER** message with the new player's identifier (ID) (DPNID). If you want to define a player context value and have not yet done so, you must define it before your message handler returns from handling this message. When it has done so, you cannot change the player context value.

The Client Application

To connect to a session, you must have the address of the session host. If your application was connected by a lobby client, you can obtain the host's address by calling

IDirectPlay8LobbiedApplication::GetConnectionSettings.

If you do not have the address of a session host and you are using either an Internet Protocol (IP) or Internetwork Packet Exchange (IPX) service provider, you can look for sessions by calling

<u>IDirectPlay8Client::EnumHosts</u> and enumerating the available hosts. You can also obtain the address by enumerating the available hosts. The information returned by the enumeration includes each host's address, the device use to reach the host, and a <u>DPN_APPLICATION_DESC</u> structure that describes the associated session.

To ask to join a session, call <u>IDirectPlay8Client::SetClientInfo</u> to set your player's name, and then call <u>IDirectPlay8Client::Connect</u> with the selected host's address to connect to the session.

Your message handler will receive a

DPN_MSGID_CONNECT_COMPLETE message with the host's response. If the host accepted the connection, the **hResultCode** member of the associated structure will be set to S_OK. If not, **hResultCode** will be set to DPNERR_HOSTREJECTEDCONNECTION.

Managing a Client/Server Session

As host, the server is responsible for managing the course of the session. The details will depend on how the application is designed, but a session host's duties include, at a minimum:

- Managing the list of session members and their network addresses. Microsoft® DirectPlay® handles some of this task, but server applications typically need to manage more player data than is provided by DirectPlay.
- Deciding whether a new user is allowed to join the session.
- Providing new users with the current game state.

When a player attempts to join a session, the host receives a <u>DPN_MSGID_INDICATE_CONNECT</u> message. To accept the player into the session, return S_OK. Returning any other value rejects the connection request. In either case, the player will receive a <u>DPN_MSGID_CONNECT_COMPLETE</u> message that contains your response.

The host can remove a player from the session by calling <u>IDirectPlay8Server::DestroyClient</u>.

Handling Client/Server Messages

This document describes how to handle Microsoft® DirectPlay® messaging for client/server sessions.

Messaging within client/server sessions is handled in much the same way as it is for peer-to-peer sessions (see <u>Handling DirectPlay</u> <u>Messaging, Handling Standard Peer-to-Peer Messages</u>, and <u>Peer-to-Peer Host Messages</u>). The primary differences are that within client/server sessions:

- A client receives no messages that carry information about other playersand no group-related messagesbecause DirectPlay provides no way for a client to know about or to communicate with other clients.
- DirectPlay provides no host-migration messaging because the *server* must be the host. A client/server session cannot be hosted by a client.

This document focuses on the issues specific to client/server sessions.

- <u>Client Startup Messages</u>
- <u>Server Startup Messages</u>
- <u>Client Messaging During Normal Game Play</u>
- <u>Server Messaging During Normal Game Play</u>
- <u>Client Session Termination Messages</u>
- <u>Server Session Termination Messages</u>
Client Startup Messages

To select and join a session, you must locate the server that is the session host, connect to the server, and handle the messaging used to set up the session. As with peer-to-peer sessions, the server hosting the session can either arrange the session in advance or create a stand-alone session.

To locate and connect to the server, you will need to handle someor allof the following messages. The process is essentially identical to that used for a peer-to-peer session. Refer to **Handling Standard Peer-to-Peer Messages** for details on how to handle individual messages.

• <u>DPN_MSGID_ENUM_HOSTS_RESPONSE</u>

To enumerate the available servers, call <u>IDirectPlay8Client::EnumHosts</u>, then handle **DPN_MSGID_ENUM_HOSTS_RESPONSE** messages from each server that responds to your enumeration request.

<u>DPN_MSGID_CONNECT_COMPLETE</u>

After you have located the server and called IDirectPlay8Client::Connect to attempt to connect to the session, you will normally need to handle a DPN_MSGID_CONNECT_COMPLETE message, containing the server's response.

Server Startup Messages

Since the server is, by definition, the host of a client/server session, the startup procedure and messaging is virtually identical to a peer-to-peer host. As with peer-to-peer sessions, the host server can either arrange the session in advance, or create a stand-alone session. You will need to handle some or all of the following messages while starting up your session. Refer to **Peer-to-Peer Host Messages** for details on how to handle the individual messages.

<u>DPN_MSGID_ENUM_HOSTS_QUERY</u>

If you are hosting a session, you will need to handle DPN_MSGID_ENUM_HOSTS_QUERY from potential players who are attempting to find a suitable host. Even if you are not hosting a session, you may still receive this message and must handle it.

<u>DPN_MSGID_INDICATE_CONNECT</u>

You receive this message when a player attempts to connect to your session. Return <u>DPN_OK</u> to accept connection attempt and allow the player into the session. Return any other value to reject the connection attempt.

• <u>DPN_MSGID_CREATE_PLAYER</u>

You receive this message for each player that you accept into your session.

<u>DPN_MSGID_INDICATED_CONNECT_ABORTED</u>

You receive this message if a client drops the connection after you have processed the **DPN_MSGID_INDICATE_CONNECT** message but before you have processed

DPN_MSGID_CREATE_PLAYER.

Client Messaging During Normal Game Play

The messages that you can receive during normal game play are basically a subset of the messages that are used by normal members of a peer-to-peer session. You will not receive messages that carry information about other players, and no group-related messages. You also will not receive a host-migration message, because the host cannot migrate.

You must process the following messages. Refer to **Handling Standard Peer-to-Peer Messages** for details on how to handle individual messages.

<u>DPN_MSGID_SEND_COMPLETE</u>

You can send messages to the server by calling <u>IDirectPlay8Client::Send</u>. As with peer-to-peer sessions, you receive a **DPN_MSGID_SEND_COMPLETE** message notifying you that the message has been sent.

DPN_MSGID_RECEIVE

When the server sends you data, it is delivered to your message handler with a **DPN_MSGID_RECEIVE** message.

• <u>DPN_MSGID_APPLICATION_DESC</u>, <u>DPN_MSGID_SERVER_INFO</u>

Information structures are associated with the application and the server. If this information changes during a session, you will receive a corresponding message. To retrieve the updated application and server information structures, you must must call IDirectPlay8Client::GetApplicationDesc, IDirectPlay8Client::GetServerInfo.

Server Messaging During Normal Game Play

The server receives essentially the same messages as a peer-to-peer host during normal game play. You will need to process the following messages. Refer to **Handling Standard Peer-to-Peer Messages** for details on how to handle the individual messages.

• DPN_MSGID_CREATE_PLAYER and DPN_MSGID_DESTROY_PLAYER

You receive **DPN_MSGID_CREATE_PLAYER** when a player enters a game, and a **DPN_MSGID_DESTROY_PLAYER** message when that player leaves the game.

• <u>DPN_MSGID_CREATE_GROUP</u> and <u>DPN_MSGID_DESTROY_GROUP</u>

Groups are visible only to the server. You receive a **DPN_MSGID_CREATE_GROUP** message when you create a group, and **DPN_MSGID_DESTROY_GROUP** when you destroy the group.

• <u>DPN_MSGID_ADD_PLAYER_TO_GROUP</u> and <u>DPN_MSGID_REMOVE_PLAYER_FROM_GROUP</u>

You receive a DPN_MSGID_ADD_PLAYER_TO_GROUP message each time you add a player to a group, and DPN_MSGID_REMOVE_PLAYER_FROM_GROUP each time you remove a player. When you destroy a group, you receive a DPN_MSGID_REMOVE_PLAYER_FROM_GROUP message for each group member.

• DPN_MSGID_SEND_COMPLETE

When you send messages to a client by calling

IDirectPlay8Server::SendTo, you receive a

DPN_MSGID_SEND_COMPLETE message to notify you that the message was sent.

• DPN_MSGID_RECEIVE

When a client sends you data, it is delivered to your message handler with a **DPN_MSGID_RECEIVE** message.

DPN_MSGID_APPLICATION_DESC, DPN_MSGID_CLIENT_INFO, DPN_MSGID_GROUP_INFO

Information structures are associated with the application, the server, and your groups. If this information changes during a session, you will receive the corresponding message. To retrieve the updated application, client, and group information structures, you must must call **IDirectPlay8Client::GetApplicationDesc**, IDirectPlay8Server::GetClientInfo, and IDirectPlay8Server::GetGroupInfo.

Client Session Termination Messages

A client/server session normally ends when the server calls <u>IDirectPlay8Server::Close</u>. A session can also end if the server is disconnected. When the session ends, each client receives the following messages. Refer to **Handling Standard Peer-to-Peer Messages** for details on how to handle the individual messages.

<u>DPN_MSGID_TERMINATE_SESSION</u>

This messages notifies you that the session is over.

Server Session Termination Messages

After you end the session, you will receive the following messages. Refer to **Handling Standard Peer-to-Peer Messages**.

• DPN_MSGID_DESTROY_PLAYER

You will receive a **DPN_MSGID_DESTROY_PLAYER** message for each player in your session.

• DPN_MSGID_DESTROY_GROUP and DPN_MSGID_REMOVE_PLAYER_FROM_GROUP

You will receive a **DPN_MSGID_REMOVE_PLAYER_FROM_GROUP** message for each player in each group, and a **DPN_MSGID_DESTROY_GROUP** message for each group.

Normal Client/Server Game Play

In Microsoft® DirectPlay®, a message is essentially a block of gamerelated data that is sent from client to server or vice versa. DirectPlay does not specify the contents or format of the data block, it just provides a mechanism to transmit the data. Once the game is underway, each client will normally send a constant stream of messages to the server, and vice versa, for the duration of the game. The primary purpose of these messages is to keep the game state synchronized, so that each user's application displays the same user interface (UI). However, messages can also be used for a variety of other game-specific purposes.

For many games, especially rapidly changing ones, you may have to manage your messaging carefully. DirectPlay throttles outgoing messages to a level that can be handled by the target. You will have be careful that you do not send messages too rapidly, and ensure that the most important messages get through. See <u>Basic Networking</u> for a discussion of how to effectively handle DirectPlay messaging.

The Server Application

To send a message to a client, call <u>IDirectPlay8Server::SendTo</u>. The client will receive a <u>DPN_MSGID_RECEIVE</u> message with the data.

The Client Application

To send a message to the server, call <u>IDirectPlay8Client::Send</u>. The server will receive a **DPN_MSGID_RECEIVE** message with the data.

Note DirectPlay does not provide a mechanism for clients to communicate with other clients, only with the server. Any client-client communication must be implemented by the server application.

Using Groups

Many games allow players to be organized into groups. For example, in a squad-based game, every player in the squad could be a member of a group. DirectPlay allows servers in a client/server game to create groups of players. While DirectPlay groups typically correspond to the groups that defined by the game, you are free to create a group for any reason. DirectPlay groups are essentially a way to simplify your messaging. When you have defined a group, you can send a message to every group member with a single **IDirectPlay8Server::SendTo** call.

To create a DirectPlay group, call <u>IDirectPlay8Server::CreateGroup</u>. Your message handler will then receive a <u>DPN_MSGID_CREATE_GROUP</u> message with the details. The message will include a group identifier (ID) that is used to send messages to the group. Once the group is created, you then add players by calling <u>IDirectPlay8Server::AddPlayerToGroup</u>.

Once the group is established, you can send data to the group by calling **IDirectPlay8Server::SendTo**, with the *dpnid* parameter set to the group ID. All group members will then receive a **DPN_MSGID_RECEIVE** message with the data.

To remove a player from a group, call <u>IDirectPlay8Server::RemovePlayerFromGroup</u>. Finally, when you no longer need the group, you can destroy it by calling <u>IDirectPlay8Server::DestroyGroup</u>.

 $\ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.

Leaving a Client/Server Session

A client can leave a session by calling <u>IDirectPlay8Client::Close</u>. The server is notified with a <u>DPN_MSGID_DESTROY_PLAYER</u> message.

Terminating a Client/Server Session

To terminate a client/server session, the server calls IDirectPlay8Server::Close. There is no host migration in a client/server session, so this method terminates all connections and closes the session. The clients are notified of the session end by a DPN_MSGID_TERMINATE_SESSION message.

The server will then receive a <u>DPN_MSGID_DESTROY_PLAYER</u> message for each player, including itself. **IDirectPlay8Server::Close** is synchronous, and will not return until all the **DPN_MSGID_DESTROY_PLAYER** messages have been processed. When **IDirectPlay8Server::Close** has returned, you can safely shut down the server application.

DirectPlay Lobby

A lobby is an application with the primary purpose of helping users arrange multiplayer games. The lobby is usually an application that is hosted on a remote server. The user visits the lobby, typically through the Internet, and either sets up a game session or joins a session started by someone else. The lobby application then launches the group's individual game applications, and the game is underway.

Because many multiplayer games are arranged through lobbies, most games based on Microsoft® DirectPlay® must be able to interact with lobby applications. Conversely, because most lobbies will want to support DirectPlay-based games, the lobby application must be able to interact with the game application. This document discusses how to enable a Microsoft DirectX® game to interact with a lobby and vice versa.

Overviews

- DirectPlay Lobby Architecture
- DirectPlay Lobby Support
- Implementing a Lobby Client
- Implementing a Lobbyable Application
- Lobby Clients
- Lobby Servers
- Lobbyable Applications

DirectPlay Lobby Architecture

The process of arranging and managing a session of a multiplayer game based on Microsoft® DirectPlay® involves five separate components. The lobby server application is an application that typically resides on a remote server and is accessed through the Internet. The remaining four components are installed on each player's computer.

- Lobby client. The lobby client is an application that communicates with the lobby server. It also communicates with the user's game application through the DirectPlay lobby client object.
- Lobbyable game application. The lobbyable game application is an application that uses the DirectPlay lobbied application object to communicate with the lobby client, and through the lobby client with the lobby server.
- DirectPlay lobby client object.
- DirectPlay lobbied application object.

The two DirectPlay objects act as links between the game application and the lobby client. They communicate with each other through private interfaces. The following graphic shows how these pieces are linked, and how they communicate.

 $\ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.

DirectPlay Lobby Support

A lobby is an application whose primary purpose is to enable players to meet and arrange games. It is typically located on a remote computer, and accessed over the Internet. Lobby servers often also perform a variety of other functions, such as hosting chat rooms, posting news and information, and selling merchandise. While lobby servers are convenient and commonly used to arrange multiplayer games, they aren't required. Multiplayer games can also be arranged by direct communication between lobby clients.

There are normally three components that are needed to enable a game to interact with a lobby:

- A lobby server
- A lobby client
- A lobbyable game.

Microsoft® DirectPlay® does not specify how you should implement a lobby server application. Instead, DirectPlay provides support for a lobby client. A lobby client is an application that is implemented by a lobby server vendor, and installed on each user's system. It serves as a link between the user and the lobby. While you could handle such communication directly, you would have to know the specific implementation details of every lobby that might launch your game.

The lobby client application handles the details of communicating with its associated lobby server, using whatever protocols are appropriate. The lobby client communicates with the user and their game applications through a DirectPlay interface. DirectPlay then passes messages to the application. The application can also use a DirectPlay interface to pass

messages to the lobby client.

A lobby can launch virtually any application. However, the application must have some specific lobby-aware components to take full advantage of lobby-launching. In particular, a lobbyable application can communicate with the lobby client throughout the course of the session. If an application is registered as lobbyable, the lobby client also automatically receives updates for various changes in game status, such as host migration.

Implementing a Lobby Client

A lobby client is an application that resides on a user's computer. It typically serves as a link between a game application on the user's computer and a lobby server on a remote computer. However, lobby clients can also function as stand-alone applications. For instance, they can be used to arrange a game session among the users of a particular local area network (LAN) subnet.

Lobby clients typically consist of three primary components that handle the following tasks:

- Communicating with the outside world, either a lobby server or other lobby clients.
- Communicating with the user, typically through a graphical user interface (GUI).
- Communicating with Microsoft® DirectPlay®.

DirectPlay does not specify how the first two items should be implemented. Lobby client vendors should use whatever approach is suitable to their product. What DirectPlay provides is a standard application programming interface (API) that a lobby client can use to communicate with DirectPlay, and through DirectPlay with the user's lobbyable game applications.

This section discusses the essential details of lobby client implementation.

- Initializing a Lobby Client
- Launching a Lobbied Application
- Implementing a Lobby Client Message Handler

- <u>Communicating with a Lobbied Application</u>
- <u>Closing Down a Lobby Client</u>

See the LobbyClient sample application for a fully implemented example of a simple lobby client.

Initializing a Lobby Client

Lobby clients are launched either by a lobby server or directly, by a user. When a lobby client is launched, it must be initialized before it can launch an application. Initialization involves the following tasks.

- Call CoCreateInstance to create a lobby client object (CLSID_DirectPlay8LobbyClient). Use the *riid* parameter to request an <u>IDirectPlay8LobbyClient</u> interface (IID_IDirectPlay8LobbyClient).
- Call the lobby client's <u>IDirectPlay8LobbyClient::Initialize</u> method. Pass the method a pointer to your lobby client's callback message handler.
- Use the <u>IDirectPlay8LobbyClient::EnumLocalPrograms</u> method to enumerate the lobbyable applications on the user's system.

The first two steps create the lobby client object, and set up a communication link between that object and your lobby client. The final step determines what lobbyable applications are available on the user's system. You need this information in order to launch the selected application.

The following code sample illustrates how to enumerate local applications. It is a simplified version of the *EnumRegisteredApplications* function in the LobbyClient sample found in the software development kit (SDK). Code related to error handling and to the dialog box has been deleted for clarity. See the LobbyClient sample in the SDK for the complete code.

```
HRESULT EnumRegisteredApplications()
{
    HRESULT hr;
```

```
DWORD dwSize = 0;
DWORD dwPrograms = 0;
DWORD iProgram;
BYTE* pData = NULL;
```

// g_pLobbyClient is a pointer to an IDirectPlay8LobbyClient interfac // Start with a NULL data buffer. The required buffer size is // returned through dwSize. hr = g_pLobbyClient->EnumLocalPrograms(NULL, pData, &dwSiz if(dwSize == 0) { // No registered applications. }

```
// Set the data buffer to the appropriate size
pData = new BYTE[dwSize];
hr = g_pLobbyClient->EnumLocalPrograms( NULL, pData, &dwSiz
```

```
// Cast the returned data to the appropriate structure type
DPL_APPLICATION_INFO* pAppInfo = (DPL_APPLICATION_II)
```

```
// Enumerate the names of the registered applications
for( iProgram=0; iProgram<dwPrograms; iProgram++ )
{
    TCHAR strAppName[MAX_PATH];
    DXUtil_ConvertWideStringToGeneric( strAppName, pAppInfo->p
    }
    SAFE_DELETE_ARRAY( pData );
    return S_OK;
}
```

Launching a Lobbied Application

When the user has selected an application and your lobby client has verified that it has been registered on the user's system, you can launch the application. To do so, call

IDirectPlay8LobbyClient::ConnectApplication. The first parameter is a DPL_CONNECT_INFO structure that contains a variety of information needed to launch the application, including the following:

- The globally unique identifier (GUID) that identifies the application.
- The connection settings, including the user's Microsoft® DirectPlay® address. See <u>DirectPlay Addressing</u> for a discussion of DirectPlay addresses.
- Whether the application will be a host.

When you call **IDirectPlay8LobbyClient::ConnectApplication**, you can set a connection context value in the *pvUserApplicationContext* parameter. If you do not set the connection context value when you call **IDirectPlay8LobbyClient::ConnectApplication**, you can set the *pvConnectionContext* parameter when you handle the <u>DPL_MSGID_CONNECT</u> message. After this value has been set, the connection context value will be passed along in the *pvConnectionContext* parameter with any message you receive from that connection. This is a useful parameter to set if you are receiving messages from multiple connections and using a common lobby client message handler.

The **IDirectPlay8LobbyClient::ConnectApplication** method returns an application handle that is used to identify the application in all further communication. When the application has launched and the connection is successfully established, your message handler receives a

DPL_MSGID_CONNECT message.

Note Your message handler may receive the **DPL_MSGID_CONNECT** before the **IDirectPlay8LobbyClient::ConnectApplication** method has confirmed the connection by returning a success code. Your message handler should be prepared to handle the message even if the method has not yet returned.

Implementing a Lobby Client Message Handler

The message handler is a callback function that is used by the lobby client object to communicate with the lobby client. The lobby client message handler has three parameters that pass in the following information.

- A message identifier (ID) that indicates the message type.
- A pointer to a message data block. You must cast this parameter to the structure that is used by the particular message.
- A pointer to an optional application-defined user-context data block.

The user context value is defined by the lobby client when it calls IDirectPlay8LobbyClient::Initialize. It can be used for such purposes as differentiating between messages that are sent from different objects. For more information about user-context values, see <u>Using Player Context</u> <u>Values</u>.

Your message handler must be able to handle the following five lobby client-specific messages.

- DPL_MSGID_CONNECT
- DPL_MSGID_CONNECTION_SETTINGS
- DPL_MSGID_DISCONNECT
- DPL_MSGID_RECEIVE
- DPL_MSGID_SESSION_STATUS

Most of these messages are generated by the lobby client object in response to changes in the game status, or when the lobby client requests information. The exception is **DPL_MSGID_RECEIVE**. This message is used to pass data directly from the game application to the
lobby client. See <u>PFNDPNMESSAGEHANDLER</u> for a complete description the message handler function.

Note Microsoft® DirectPlay® message handlers must be written to work properly in a multithreaded environment, or your application may not function well.

DPL_MSGID_CONNECT

This message is sent by the lobby client following the launch of a lobbyable application. The message indicates that the application has been successfully connected. The associated **DPL_MSGID_CONNECT** structure holds a variety of information, including:

- A connection ID. Use this ID when your lobby client needs to send data to the application with <u>IDirectPlay8LobbyClient::Send</u>, or release the connection with <u>IDirectPlay8LobbyClient::ReleaseApplication</u>.
- Lobby connection data.
- An optional connection context value.

DPL_MSGID_CONNECTION_SETTINGS

This message is sent by DirectPlay whenever an associated lobbyable application calls its

IDirectPlay8LobbiedApplication::SetConnectionSettings method to modify the session connections. The associated

DPL_MSGID_CONNECTION_SETTINGS structure contains the updated connection information.

DPL_MSGID_DISCONNECT

This message is sent when the lobbyable application disconnects from the session by calling <u>IDirectPlay8LobbiedApplication::Close</u>. Your lobby client application should delete the connection from its list and free any data that is associated with the application.

DPL_MSGID_RECEIVE

This message enables an application to pass data to the lobby client. DirectPlay passes the data block from the application to the lobby client in a **DPL_MSGID_RECEIVE** structure. It is up to the lobby client to process the data.

DPL_MSGID_SESSION_STATUS

This message is sent by DirectPlay whenever one of the following six changes in the session's status occurs.

- The session is connected.
- The session could not connect.
- The session has been disconnected.
- The session has been terminated.
- The session host has migrated.
- This computer has become the session host.

The type of status change is indicated by the value of the dwStatus field in the associated **DPL_MSGID_SESSION_STATUS** structure.

For more information, see <u>A Sample Lobby Client Message Handler</u>.

A Sample Lobby Client Message Handler

The following code is a simplified version of the message handler from the LobbyClient sample in the software development kit (SDK). Error handling code has been removed for clarity. See the sample for a complete version.

```
HRESULT WINAPI DirectPlayLobbyMessageHandler( PVOID pvUs
                        DWORD dwMessageId,
                        PVOID pMsgBuffer )
 switch( dwMessageId )
 ł
  case DPL_MSGID_DISCONNECT:
   PDPL_MESSAGE_DISCONNECT pDisconnectMsg;
   pDisconnectMsg = (PDPL_MESSAGE_DISCONNECT)pMsgBu
   // Free any data associated with the application and
   // remove the connection from the list.
   break:
  case DPL MSGID RECEIVE:
  ł
   PDPL_MESSAGE_RECEIVE pReceiveMsg;
   pReceiveMsg = (PDPL_MESSAGE_RECEIVE)pMsgBuffer;
   // The lobby application sent data. Process the data and
   // respond appropriately.
   break;
  case DPL MSGID SESSION STATUS:
```

```
{
```

```
PDPL_MESSAGE_SESSION_STATUS pStatusMsg;
pStatusMsg = (PDPL_MESSAGE_SESSION_STATUS)pMsgBuf
```

```
switch( pStatusMsg->dwStatus )
```

```
{
```

}

```
case DPLSESSION_CONNECTED: // Session connected.
break;
```

case DPLSESSION_COULDNOTCONNECT: // Session could
 break;

```
case DPLSESSION_DISCONNECTED: // Session disconnect
    break;
```

```
case DPLSESSION_TERMINATED: // Session terminated.
break;
```

```
case DPLSESSION_HOSTMIGRATED: // Host migrated.
break;
```

```
case DPLSESSION_HOSTMIGRATEDHERE: // Host migrated
    break;
```

```
case DPL_MSGID_CONNECTION_SETTINGS:
```

```
{
   PDPL_MESSAGE_CONNECTION_SETTINGS pConnectionSt
   pConnectionStatusMsg = (PDPL_MESSAGE_CONNECTION_S
   // The application has changed the connection settings.
   break;
}
```

```
}
return S_OK;
```

```
}
```

Communicating with a Lobbied Application

Much of the lobby client's interaction with an associated application is indirect. The lobby client does something that affects the application, and Microsoft® DirectPlay® sends an appropriate message, and vice versa. For instance, if the lobby client changes the connection settings, DirectPlay notifies the application and provides the new settings. However, the <u>IDirectPlay8LobbyClient::Send</u> method enables the lobby client to send a message directly to the application. DirectPlay passes the data to the application without modification. It is the responsibility of the application to process that data.

The lobbied application can also send data directly to the lobby client. The data is passed to the lobby client's message handler with a <u>DPL_MSGID_RECEIVE</u> message. DirectPlay passes the data to the lobby client without modification. It is up to the lobby client to process the data.

Closing Down a Lobby Client

When you are ready to close the session, disconnect the application by calling <u>IDirectPlay8LobbyClient::ReleaseApplication</u>. The application receives a <u>DPL_MSGID_DISCONNECT</u> message.

After releasing the application, perform any cleanup that is necessary, and close the session by calling <u>IDirectPlay8LobbyClient::Close</u>. You should then free the lobby client object by calling <u>IDirectPlay8LobbyClient::Release</u>.

Implementing a Lobbyable Application

A lobbyable application must be designed to work properly with a lobby client. While a lobby client can launch any application, only *lobbylaunched* applications can receive messages from Microsoft® DirectPlay® and from the lobby client. To be lobby launched, an application must be appropriately registered, and it must be able to use a DirectPlay lobbied application object to communicate with DirectPlay and the lobby client.

This section discusses how to implement lobbyable applications.

- <u>Registering a Lobbyable Application</u>
- Handling Lobby Launching
- Implementing a Lobbied Application Callback Message Handler
- Communicating with a Lobby Client
- <u>Closing Down a Lobbied Application</u>

Registering a Lobbyable Application

A lobbyable application must be registered before it can be lobby launched. This registration needs to be done only once and is typically handled by the application's setup procedure. Do not attempt to modify the registry directly to register an application as lobbyable. Instead, do the following:

- Call CoCreateInstance to create a lobbied application object (CLSID_DirectPlay8LobbiedApplication). Set the *riid* parameter to IID_IDirectPlay8LobbiedApplication to request an <u>IDirectPlay8LobbiedApplication</u> interface.
- Call <u>IDirectPlay8LobbiedApplication::RegisterProgram</u>.This method takes the information and creates appropriate registry keys and values.

You must provide **IDirectPlay8LobbiedApplication::RegisterProgram** with a variety of information, including the following:

- A globally unique identifier (GUID) that is used to identify the application.
- A friendly name for the application.
- The location and name of the application's executable file.
- The location and name of an optional *launcher* application.
- Any command-line arguments that need to be passed to the executable file when it is launched.

Instead of launching the game application, Microsoft® DirectPlay® launches a launcher application. The launcher application then launches the game. Launcher applications can be used, for example, as an antipiracy measure.

To unregister a registered program, call

IDirectPlay8LobbiedApplication::UnRegisterProgram. This method removes the registry entries created by IDirectPlay8LobbiedApplication::RegisterProgram.

Handling Lobby Launching

The first thing your lobbyable application should do when it is launched is to create and initialize a Microsoft® DirectPlay® lobbied application object. To do this, perform the following tasks.

- Call **CoCreateInstance** to create a lobbied application object (CLSID_DirectPlay8LobbiedApplication). Set the *riid* parameter to IID_IDirectPlay8LobbiedApplication to request an <u>IDirectPlay8LobbiedApplication</u> interface.
- Call <u>IDirectPlay8LobbiedApplication::Initialize</u> to initialize the lobbied application object. Pass the object a pointer to your lobbied application callback message handler.

Next, determine whether your application was lobby launched. If so, your application needs to set up a communication channel with DirectPlay so that you can effectively manage the session. Do the following to detect whether your application was lobby launched.

- When the **IDirectPlay8LobbiedApplication::Initialize** method returns, examine the *pdpnhConnection* parameter. If this parameter is set to a valid connection handle, the game was lobby launched.
- Examine the DPL_MSGID_CONNECT message you receive through your message handler. This message carries with it a variety of information, including the ID that you use to send messages to the lobby client.

Note Your message handler may receive the **DPL_MSGID_CONNECT** message before the **IDirectPlay8LobbiedApplication::Initialize** method returns. Your message handler should be prepared to handle the message appropriately.

If your application was not lobby launched, you can indicate that your application is available to lobby clients for connection by calling IDirectPlay8LobbiedApplication::SetAppAvailable. This method is typically called when the application has been launched by the user. However, it can also be used if the user has closed one session but the application is still running and available for another session. In either case, your message handler receives a **DPL_MSGID_CONNECT** message when the lobby client connects your application to a session.

The following sample code illustrates how to initialize a lobbied application, and how to detect whether an application was lobby launched. It is a simplified version of the *InitDirectPlay* function used by the SimplePeer application in the software development kit (SDK). Refer to that sample application for the complete code. In particular, errorhandling code has been deleted for clarity. The *g_bWasLobbyLaunched* variable is a global variable that is set to TRUE if the application was lobby launched.

```
HRESULT InitDirectPlay()
```

{

DPNHANDLE hLobbyLaunchedConnection = NULL; HRESULT hr;

```
0 );

//Check for a valid connection handle. If it is non-NULL

//the application was lobby launched.

g_bWasLobbyLaunched = ( hLobbyLaunchedConnection != NULL )

return S_OK;

}
```

Implementing a Lobbied Application Callback Message Handler

The message handler is a callback function that is used by the lobbied application object to communicate with a lobbied application. The lobbied application message handler has three parameters that pass in the following information.

- A message identifier (ID) that indicates the message type.
- A pointer to a message data block. You must cast this parameter to the structure that is used by the particular message.
- A pointer to an optional application-defined user-context data block.

The user context value is defined by the lobby client when it calls IDirectPlay8LobbyClient::Initialize. It can be used for such purposes as differentiating between messages that are sent from different objects. For more information about user-context values, see <u>Using Player Context</u> <u>Values</u>.

Your message handler must to be able to handle the following four lobbied application-specific messages.

- DPL_MSGID_CONNECT
- DPL_MSGID_CONNECTION_SETTINGS
- DPL_MSGID_DISCONNECT
- DPL_MSGID_RECEIVE

Most of these messages are generated by the lobbied application object in response to changes in the connection, or when the lobbied application requests connection information. The exception is

DPL_MSGID_RECEIVE. This message is used to pass data directly from

the lobby client to the game application. See

<u>PFNDPNMESSAGEHANDLER</u> for a complete description the message handler function.

Note Microsoft® DirectPlay® message handlers must be written to work properly in a multithreaded environment, or your application may not function well.

DPL_MSGID_CONNECT

This message is sent by the lobbied application object when the lobby client calls <u>IDirectPlay8LobbyClient::ConnectApplication</u> to connect an application to a session. The associated **DPL_MSGID_CONNECT** structure includes the following information.

- A connection ID. Use this ID when your application needs to send data to the lobby client with <u>IDirectPlay8LobbiedApplication::Send</u>, or update the session status with <u>IDirectPlay8LobbiedApplication::UpdateStatus</u>.
- Lobby connection data.
- An optional connection context value. You can set the connection context value when you handle the **DPL_MSGID_CONNECT** message. If this value is set for a connection, it will be returned whenever you receive a message from that connection in the *pvConnectionContext* parameter. This is a useful value to set if your application has multiple connections and a common lobbied application message handler.

DPL_MSGID_CONNECTION_SETTINGS

DirectPlay sends this message whenever an associated lobby client calls its <u>IDirectPlay8LobbyClient::SetConnectionSettings</u> method to modify the session connections. The associated

DPL_MSGID_CONNECTION_SETTINGS structure contains the updated connection information.

DPL_MSGID_DISCONNECT

This message is sent when the lobby client disconnects the application from the session by calling <u>IDirectPlay8LobbyClient::ReleaseApplication</u>. Your application should delete the connection from its list, and free any data that is associated with the session.

DPL_MSGID_RECEIVE

This message enables a lobby client to pass data to an application. DirectPlay passes the data block from the lobby client to the application in a **DPL_MSGID_RECEIVE** structure. It is up to the application to process the data.

For more information, see <u>A Sample Lobbied Application Message</u> <u>Handler</u>.

A Sample Lobbied Application Message Handler

The following code is a simplified version of the message handler from the SimplePeer sample in the software development kit (SDK). Error handling code has been removed for clarity. See the sample for a complete version.

```
HRESULT WINAPI DirectPlayLobbyMessageHandler( PVOID pvUs
                       DWORD dwMessageId,
                       PVOID pMsgBuffer )
 switch( dwMessageId )
 ł
  case DPL_MSGID_CONNECT:
   PDPL_MESSAGE_CONNECT pConnectMsg;
   pConnectMsg = (PDPL_MESSAGE_CONNECT)pMsgBuffer;
   // Connected. Start the session.
   break;
  case DPL_MSGID_DISCONNECT:
  ł
   PDPL_MESSAGE_DISCONNECT pDisconnectMsg;
   pDisconnectMsg = (PDPL_MESSAGE_DISCONNECT)pMsgBu
   // Disconnected. Free any data associated with
   // the lobby client.
   break;
  }
  case DPL MSGID RECEIVE:
  {
```

```
PDPL_MESSAGE_RECEIVE pReceiveMsg;
pReceiveMsg = (PDPL_MESSAGE_RECEIVE)pMsgBuffer;
// The lobby client sent data. Process the data and
// respond appropriately.
break;
}
case DPL_MSGID_CONNECTION_SETTINGS:
{
    PDPL_MESSAGE_CONNECTION_SETTINGS pConnectionSta
    pConnectionStatusMsg = (PDPL_MESSAGE_CONNECTION_S
    // The lobby client has changed the connection settings.
    break;
}
return S_OK;
```

 $\ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.

}

Communicating with a Lobby Client

Much of the lobbied application's interaction with an associated lobby client is indirect. The application does something that affects the lobby client, Microsoft® DirectPlay® sends an appropriate message, and vice versa. For instance, if the application changes the connection settings, DirectPlay notifies the lobby client, and provides the new settings. However, there are two methods that provide information directly to the lobby client: IDirectPlay8LobbiedApplication::UpdateStatus and IDirectPlay8LobbiedApplication::Send.

You must notify the lobby client when any of the following changes in the game status take place.

- The session is connected.
- The session could not connect.
- The session has been disconnected.
- The session has been terminated.
- The session host has migrated.
- This computer has become the session host.

To notify the lobby client of one of these status changes, call **IDirectPlay8LobbiedApplication::UpdateStatus**, and set the *dwStatus* parameter to the appropriate value. The lobby client receives a <u>DPL_MSGID_SESSION_STATUS</u> message to notify it of the status change.

The **IDirectPlay8LobbiedApplication::Send** method enables the application to send a message directly to the lobby client. DirectPlay passes the data to the lobby client without modification. It is the responsibility of the lobby client to process that data.

The lobby client can also send data directly to the application. The data is passed to the lobby client's message handler with a <u>DPL_MSGID_RECEIVE</u> message. DirectPlay passes the data to the application without modification. The lobby client must process the data.
Closing Down a Lobbied Application

To close a session, call <u>IDirectPlay8LobbiedApplication::Close</u>. The lobby client receives a <u>DPL_MSGID_DISCONNECT</u> message to notify it of the disconnection. If you want to keep the application running and connect to another session, indicate that your application is available by calling <u>IDirectPlay8LobbiedApplication::SetAppAvailable</u>. Otherwise, call <u>IDirectPlay8LobbiedApplication::Release</u> to free the lobbied application object, and shut the application down.

Lobby Clients

A lobby client is an application that is implemented by the lobby server vendor and installed on each player's computer. It handles communication between the players and their game applications, and the lobby server. A common way to install a lobby client is to have the user download it from the lobby server's Web site as part of the sign-up procedure.

The following is a typical scenario.

- 1. A new player goes to the Web site and signs up.
- 2. As part of the sign-up procedure, the lobby client is downloaded to the client's computer.
- 3. The player determines which to play and asks to join a session.
- 4. The Web site launches the lobby client on the player's computer. A typical launch mechanism is a URL that points to the lobby client's executable file.
- 5. The lobby client handles the mechanics of arranging the session, and then launches the user's game application.
- 6. If the game is a lobbyable application, the lobby client enables the game application to communicate with the lobby server. This connection enables the lobby server to keep track of events such as players entering and leaving the game and host migration.

Lobby clients do not necessarily have to be linked to a remote server. In another scenario the user launches the lobby client directly. The lobby client then lists the available games and sessions, perhaps among the people connected to the user's local area network (LAN) subnet. When the user chooses a game and session, the lobby client launches the game. This section discusses some the general features of a lobby client. For more information about communicating between a lobby client and its associated lobby server, see <u>Communicating with a Lobbied Application</u>.

For more information about implementation details, see <u>Implementing a</u> <u>Lobby Client</u> or the LobbyClient sample application included in the software development kit (SDK).

Communicating with a Lobby Client

Much of the lobbied application's interaction with an associated lobby client is indirect. The application does something that affects the lobby client, Microsoft® DirectPlay® sends an appropriate message, and vice versa. For instance, if the application changes the connection settings, DirectPlay notifies the lobby client, and provides the new settings. However, there are two methods that provide information directly to the lobby client: IDirectPlay8LobbiedApplication::UpdateStatus and IDirectPlay8LobbiedApplication::Send.

You must notify the lobby client when any of the following changes in the game status take place.

- The session is connected.
- The session could not connect.
- The session has been disconnected.
- The session has been terminated.
- The session host has migrated.
- This computer has become the session host.

To notify the lobby client of one of these status changes, call IDirectPlay8LobbiedApplication::UpdateStatus, and set the *dwStatus* parameter to the appropriate value. The lobby client receives a DPL_MSGID_SESSION_STATUS message to notify it of the status change.

The **IDirectPlay8LobbiedApplication::Send** method enables the application to send a message directly to the lobby client. DirectPlay passes the data to the lobby client without modification. It is the responsibility of the lobby client to process that data.

The lobby client can also send data directly to the application. The data is passed to the lobby client's message handler with a DPL_MSGID_RECEIVE message. DirectPlay passes the data to the application without modification. The lobby client must process the data.

Lobby Servers

The lobby server is an application whose primary purpose is to enable players to meet and arrange games. It is typically located on a remote computer, and accessed over the Internet. Lobby servers often perform a variety of other functions such as hosting chat rooms, posting news and information, and selling merchandise.

To manage multiplayer games, a lobby server typically handles a variety of tasks, including:

- Managing the network addresses of the various game sessions and players.
- Launching a session by launching the associated game applications on the players' computers.
- Adding players to an ongoing session.
- Connecting the various computers in a session to the correct network addresses.
- Keeping track of changes in the session, such as players leaving the game or changes in the game's host.

The details of the lobby server application depend on what kind of services the vendor wants to offer. Microsoft® DirectPlay® does not specify how a lobby server should be implemented nor how it should communicate with its users' computers. However, lobby vendors must implement and distribute to their users a lobby client that is compatible with DirectPlay.

^{© 2003} Microsoft Corporation. All rights reserved.

Lobbyable Applications

Lobbyable applications are designed to work with a lobby client based on Microsoft® DirectPlay®. While a lobby client can use DirectPlay to launch any application, lobbyable applications have a number of advantages.

- The lobby client receives automatic updates when game status changes.
- The lobby client can use a standard application programming interface (API) to communicate with the application.
- The application can use a standard API to communicate with the lobby client.

In short, DirectPlay virtually eliminates the need for game-specific or lobby client-specific communication code. You can use a standard API for everything with little or no modification for the particular game or lobby client.

For a discussion of implementation details, see <u>Implementing a</u> <u>Lobbyable Application</u> or software development kit (SDK) samples such as SimplePeer, or StagedPeer.

• Launching a Lobbyable Application

Launching a Lobbyable Application

One of the first things a lobbyable application should do after it is launched is create a lobbied application object. Among other things, this object enables your application to determine whether it was lobbylaunched. A lobbied application must also implement a message-handler callback function to receive messages from the lobby client. The basic procedure is:

- Create a lobbied application object.
- Initialize the object.
- If the initialization method returns a valid connection handle, your application was lobby launched.
- Examine the user context value that is returned by the initialization method. This value might contain game-specific information from the lobby client.
- Examine the connection message received through the lobbied application message handler. This message carries with it a variety of information, including the identifier (ID) that you will use to send messages to the lobby client.

Once an application has been successfully lobby launched, DirectPlay can automatically send status updates to the lobby client when events such as host migration occur. To enable automatic status updates, call the **RegisterLobby** method of the <u>IDirectPlay8Peer</u>, <u>IDirectPlay8Client</u>, or <u>IDirectPlay8Server</u> interface. You can also use the lobbied application interface to send messages to the lobby client.

Be aware that your message handler function might receive messages from the lobby client before the initialization method returns. In addition to the connection message, the callback function receives messages when the lobby client changes connection settings, or it breaks the connection. The lobby client can also send messages directly to your message handler that contain game-specific information.

Note It is possible to receive messages from more than one thread. To handle messaging properly, your lobbied application callback function should be re-entrant.

DirectPlay Voice

Microsoft® DirectPlay® Voice uses a DirectPlay session for mediaindependent network transport and player management. The DirectPlay Voice application programming interface (API) does not duplicate session control features from DirectPlay. A DirectPlay network transport session must also be created before DirectPlay Voice can transmit and receive voice communications. DirectPlay Voice can use either the IDirectPlay4 object or IDirectPlay8 object for network transport.

If DirectPlay Voice is being used in-process with a multiplayer game, the game will most likely also use the transport session to exchange its game-specific data. This makes it possible to optimize the use of network resources between the game and voice data.

It is also acceptable to create and use a transport session specifically for the voice session, as would be the case for a standalone voice conferencing application.

DirectPlay Voice is a full-voice communications API that is integrated with DirectPlay for network session management and network transport.

DirectPlay Voice is also integrated with DirectPlay Sound for voice recording and playback, and all DirectPlay Sound audio features are inherited including the ability to target voice data to different playback buffers and the use of special audio effects such as three-dimensional sound positioning.

Note DirectPlay and DirectPlay Voice sessions are separate entities. While there are guarantees about message order for each interface, there are no guarantees about message order between interfaces.

Overviews

- Audio Device Testing
- Automatic Gain Control
- <u>Capture Focus</u>
- <u>Configuring the Windows Sound Mixer</u>
- Creating a 3-D Voice Session
- <u>DirectPlay Voice Communication</u>
- DirectPlay Voice Topologies
- Fast User Switching
- Handling Voice Client Messages
- Handling Voice Host Messages
- Jitter Buffers
- <u>Sharing the Audio Capture Device</u>
- <u>Transmission Control</u>
- <u>Voice Codecs</u>
- <u>Voice Host Migration</u>
- <u>Working Set Guidelines</u>

Audio Device Testing

Microphone setup is supported by the Microsoft® DirectX® Voice Test Wizard. This wizard confirms that your system properly supports full duplex operation and ensures your microphone and playback settings are correct. You need to run the wizard only once for each combination of playback and capture device you select.

You can launch the Voice Test Wizard in two ways:

- Launch the wizard from your program by creating a voice test object (CLSID_DirectPlayVoiceTest) and calling its IDirectPlayVoiceTest::CheckAudioSetup method.
- In the Sound and Audio Devices Control Panel application for Microsoft Windows® Millennium Edition (Windows Me) or Windows XP, launch the wizard from the **Voice** tab.

Each time your application starts, you should test the configuration by calling **IDirectPlayVoiceTest::CheckAudioSetup** with the *dwFlags* parameter set to DVFLAGS_QUERYONLY. This enables you to quickly test whether the device configuration has changed since you last tested them. If your devices have not been tested, you should run **IDirectPlayVoiceTest::CheckAudioSetup** again to invoke the wizard. If you do not do so, <u>IDirectPlayVoiceClient::Connect</u> will return DVERR_RUNSETUP, and you will not be able to initialize Microsoft DirectPlay® Voice.

Many older computer systems still in use do not have a full duplex sound card. Without full duplex, a sound card can receivebut not sendvoice communications. Because games typically hold the sound card in playback mode, DirectX prevents problems by not allowing dynamic switching between playback and capture. The Voice Test Wizard provides users with information about the duplexing abilities of their system.

Automatic Gain Control

Microsoft® DirectPlay® Voice offers functionality to adjust the hardware input volume on the sound card automatically to provide the best recording input level possible. You can enable automatic gain control when you call IDirectPlayVoiceClient::Connect to connect to a voice session by setting the DVCLIENTCONFIG_AUTORECORDVOLUME flag in the **dwFlags** member of the <u>DVCLIENTCONFIG</u> structure. You can activate or deactivate automatic gain control during a session by calling IDirectPlayVoiceClient::SetClientConfig. Set the DVCLIENTCONFIG_AUTORECORDVOLUME flag to activate automatic gain control, and leave the flag unset to deactivate automatic gain control.

Most game applications should use automatic gain control because it requires a negligible amount of game resources and prevents the need for an in-game volume recording control. Users are not required to set the level themselves, yet they experience the highest quality of voice transmission and reception possible.

Capture Focus

The concept of capturing focus is integral to creating lobbyable game applications and lobby applications with Microsoft® DirectPlay® Voice support. If your game application does not properly implement focus capture, it is possible that voice communication will not function if your game was launched from a lobby application.

To illustrate this point, consider two players who meet in a lobby application that has DirectPlay Voice support. The two players agree to launch the game. After the game is launched, the lobby application loses focus on each player's computer, and each copy of the game application gains focus. If the game application does not properly gain focus from the lobby application, it is possible that the lobby application can still have focus while the game application is running.

For example, this will occur if the first player's lobby application retains focus while the game session is running while the second player's game session gains focus from the lobby application. From the second player's perspective, the first player's voice session has fallen back to half-duplex. The second player can hear the first player, but the first player cannot hear the second player. From the first player's perspective, the voice session has ended because the second player does not seem to be speaking. Also, the first player does not know that the second player can hear him or her.

Note that this behavior is by design. Consider the same scenario as above, but when the first player attempts to start the game session from the lobby application, there is a problem and the session fails to start. If the second player's session starts successfully, that player can hear the voice of the first player and the first player can inform the second player that their game session failed. Both players might then drop back to the lobby and attempt to start the session once again.

To handle capture focus properly, your game application must set the *hwndAppWindow* parameter in the <u>DVSOUNDDEVICECONFIG</u> structure to the window handle that will have focus when the game is running. The DirectPlay Voice session can then be created through a call to <u>IDirectPlayVoiceClient::Connect</u>. The game application must then handle the <u>DVMSGID_LOSTFOCUS</u> and <u>DVMSGID_GAINFOCUS</u> messages.

See <u>Sharing the Audio Capture Device</u> for more information.

Configuring the Windows Sound Mixer

The Microsoft® Windows® audio mixer must be correctly set up for Microsoft DirectPlay® Voice to function properly. This article describes how to set up the sound mixer to support a DirectPlay Voice session.

- <u>Selecting a DirectPlay Voice Device</u>
- <u>Configuring the Windows Sound Mixer for Playback</u>
- <u>Configuring the Windows Sound Mixer for Recording</u>
- <u>Microphone Boost</u>
- <u>Autoconfiguration</u>

Selecting a DirectPlay Voice Device

On Windows XP and Windows Millennium Edition (Windows Me) systems, the Sound and Audio Devices Control Panel application has a **Voice** tab. You can use this tab to select playback and recording devices for DirectPlay Voice. If you have multiple sound cards on your system, you can select a different sound card for DirectPlay Voice transmissions than the sound card you use for normal audio.

Configuring the Windows Sound Mixer for Playback

To set up your sound system for DirectPlay Voice playback, you might have to configure the Windows audio playback mixer. You can launch the playback mixer in several ways, including:

- Double-click the speaker icon in the status area of the taskbar.
- Launch the Voice Test wizard and click the **Volume** button below the **Playback** box on the microphone or speaker test pages.
- Launch the Sound and Audio Devices Control Panel application, select the **Audio Devices** tab, and click the playback section's **Volume** button.

The following screen shot shows the Windows 2000 playback mixer.

To configure the playback mixer

- 1. From the **Options** menu, choose **Properties**.
- 2. In the **Properties** dialog box that appears, click **Playback**.
- 3. At the bottom of the dialog box, make sure all available output lines are selected, as shown.
- 4. Click **OK** to display the playback mixer again.

The following screen shot shows the mixer's **Properties** dialog box.

The microphone has separate playback and recording lines. You must disable microphone playback line or your microphone input will be routed to your speakers and DirectPlay Voice will not be able to record your input.

To disable the microphone playback line

- 1. Click **OK** on the **Properties** dialog box to display the playback mixer again.
- 2. Make sure that the microphone section's **Mute** check box is selected.
- 3. If the microphone section has a check box labeled **Select**, clear the check box.

Configuring the Windows Sound Mixer for Recording

You may also need to manually configure the recording mixer. There are three ways to launch the recording mixer.

- Launch the Sound and Audio Devices Control Panel application, select the **Audio Devices** tab, and click the playback section's **Volume** button.
- Launch the Voice Test Wizard and click the **Volume** button below the **Recording** box on either the microphone or speaker test pages.
- Launch the playback mixer, choose the Properties command on the Options menu, select the Recording radio button, and click OK.

To configure the recording settings

- 1. From the **Options** menu, choose **Properties**.
- 2. Select **Recording**. The check boxes at the bottom of the dialog box will show all the available audio input sources for your selected devices.
- 3. Make sure that all sources are selected, and click **OK**. The recording mixer will then display a volume control for the microphone and all other selected sources.
- 4. Make sure that the microphone's Select check box is selected, and all other Select boxes are clear. In particular, do not select CD Audio. If CD Audio is selected and your user is listening to a CD while playing the game, DirectPlay Voice will transmit the output from the CD.

The following screen shot shows the Windows 2000 recording mixer.



Microphone Boost

Many sound cards have an option called "Microphone Boost" or "Mic Boost." This feature boosts the input volume so that weak input devices can produce stronger input. If you determine that the input level from a system is too low for your voice session, you can enable microphone boost.

To enable microphone boost

- 1. Display the recording mixer, as described in the previous section.
- 2. Enable the **Advanced** buttons on the mixer by choosing the **Advanced** command from the **Options** menu.
- 3. Click the microphone line's **Advanced** button to display the microphone's **Advanced Controls** dialog box.
- 4. Select the **Microphone Boost** check box.

If microphone boost makes your input too loud, you can use the same procedure to disable microphone boost. If your sound card does not support microphone boost, one or more of the following will be true.

- The **Advanced** command on the recording mixer's **Options** menu will be unavailable.
- The **Advanced** button in the recording mixer's microphone section will be unavailable.
- The Microphone Boost check box on the Advanced Control dialog box will be unavailable.

Autoconfiguration

If you set the DVSOUNDCONFIG_AUTOSELECT flag in the **dwFlags** member of the <u>DVSOUNDDEVICECONFIG</u> structure, DirectPlay Voice will attempt to configure the recording portion of the audio mixer for you. However, the attempt might fail. For example, DirectPlay Voice always chooses microphone 0 as the input source. If your user's system has more than one microphone, autoselection works only if the user has the microphone connected to the microphone 0 input. To use another microphone channel, the user must configure the mixer manually.

Note To ensure that your application works on all systems, you should allow your users to disable autoselection if necessary and configure the mixer manually.

Creating a 3-D Voice Session

You can add extra realism to a Microsoft® DirectPlay® Voice session by making the sound three-dimensional. Your players can then use audio cues to perceive the positions and movement of the other players. The basis for a 3-D voice session is the 3-D capabilities of Microsoft DirectSound®. This document discusses how to manage your voice buffers to create a 3-D voice session. For complete details about how to use the DirectSound 3-D application programming interface (API), see **3-D Sound**.

Managing 3-D Voice Buffers

When you receive a voice transmission, it is streamed into a DirectSound buffer. By default, everything is streamed into the main buffer. To implement 3-D sound, you must have a separate buffer for each distinct source. You can then use the DirectSound 3-D API to manage the buffers to create 3-D sound.

You can specify separate 3-D buffers for individual players and for groups of players. To specify a 3-D buffer for a particular player or group, call <u>IDirectPlayVoiceClient::Create3DSoundBuffer</u> and pass the method either the player or group DVID. All transmissions from that player or group will then go into the specified 3-D buffer.

DirectPlay Voice 3-D buffers have the following characteristics.

- If you do not specify a 3-D buffer for a player or group, those transmissions will be streamed into the main buffer.
- If you specify a 3-D buffer for a player or group, those transmissions will *not* be streamed into the main buffer.
- You can retrieve the 3-D buffer for the main buffer by calling IDirectPlayVoiceClient::Create3DSoundBuffer with DVID_REMAINING as the DVID.
- A player's transmission can be streamed into multiple buffers that can be managed separately. For instance, if a player has a 3-D buffer and is also a member of a group with a 3-D buffer, the player's transmissions will be streamed into both buffers. For example, you might have the group's transmissions coming from a radio on a table. If the player runs by shouting, you can hear the player through the radio, with a fixed position, as well as from a moving sound source as the player runs by.
- You can mute the main buffer by calling <u>IDirectPlayVoiceClient::SetClientConfig</u> and setting the

DVCLIENTCONFIG_MUTEGLOBAL flag in the DVCLIENTCONFIG structure. If the main buffer is muted, your player will hear only those voice streams that have a 3-D buffer.
DirectPlay Voice Communication

The current trend toward team-based multiplayer games makes playerto-player communication an essential part of gameplay. Historically this has been confined to text-based communication, where players type out the messages to their teammates. Although suitable for slower, turnbased games, text-based communication is at best an inconvenience for real-time games. Not only does it put slow typists at a disadvantage during gameplay but also it is a significant break in the reality that games attempt to create for the player. An obvious solution to the problem is the use of speech as a means for communication. It requires no training and increases the immersion of the game itself.

The windows platform provides all the tools required to provide real-time voice conferencing to video game developers, but it requires a significant amount of effort on the part of the game developer. This, combined with the cost and difficulty of obtaining the rights to compression technology capable of handling extremely low bandwidth situations, has prevented the wide-spread use of voice in games.

Microsoft® DirectPlay® provides the game developer with a robust realtime voice conferencing system that requires a minimal amount of effort to use.

 $[\]ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.

DirectPlay Voice Topologies

Microsoft® DirectPlay® Voice sessions require a DirectPlay network session to transport voice communication. When the network session has been created, a DirectPlay Voice object can be created to use one of three topologies.

- Forwarding Server Voice Topology
- <u>Mixing Server Sessions</u>
- <u>Peer-to-Peer Voice Topology</u>

The choice of topology is dependent on several factors, and these factors are discussed in the individual DirectPlay Voice topology topics. Note that not all voice topologies can be transported over all types of DirectPlay networking sessions.

Forwarding Server Voice Topology

In a Microsoft® DirectPlay® voice session using a forwarding server topology, one computer in the session acts as a forwarding server. Each client in the voice session streams voice data to the forwarding server, which then forwards the voice data to all other clients in the session. Each client receives all incoming audio streams forwarded from the forwarding server. Each client's computer then mixes the incoming streams and plays them back.

The outgoing bandwidth requirement on each client in a voice session using a forwarding server topology is constant because there is only one outgoing voice audio stream. The incoming bandwidth and processor requirements are identical to the requirements of a voice session using a peer-to-peer topology, but they vary depending on the number of incoming voice audio streams.

The server has much higher bandwidth requirements than the individual clients in a forwarding server DirectPlay voice session. However, the processor requirements are not high because no compression or decompression of voice data occurs on the server. This reduced load on the computer's processor also means that an individual client's computer with a high bandwidth connection can host the forwarding server without adversely affecting the performance of the individual client's computer or the performance of a game server and/or client program running on the same computer.

Note that in a voice session using a peer-to-peer topology, the outgoing bandwidth requirements on the individual clients are usually much higher

than the incoming bandwidth requirements. Therefore, reducing the outgoing bandwidth requirement to a single stream of audio can result in a significant reduction in total bandwidth usage. For example, if a client is taking part in an eight-person voice session in which all clients can hear one another, the client has seven outgoing voice streams each time voice data is captured and transmitted on his or her computer. However, it is rare that all clients talk at once, so there are most likely fewer than two or three incoming voice streams at any one time.

Mixing Server Sessions

In mixing server sessions, one computer in the session acts as a mixing server. Each client streams its voice data to the mixing server. The mixing server examines the targets of each voice stream, performs decompression, mixing, and recompression as appropriate to generate a mixed stream of audio data for each client. Each client receives this single stream of premixed audio data and plays it back.

The outgoing bandwidth, incoming bandwidth, and CPU requirement on the client in a mixing server session is easily predictable because each client has only one outgoing stream of audio to compress and send, and one incoming stream of audio to decompress and play back.

The mixing server has much higher bandwidth and CPU requirements than do the clients. Typically, the mixing server is either a completely dedicated computer, or it shares a computer with a dedicated game server.

Mixing server voice sessions do not support 3-D spatialization of the voice data through the <u>IDirectPlayVoiceClient::Create3DSoundBuffer</u> method.

You can run mixing server voice sessions using either a peer-to-peer or a client/server transport session.

Peer-to-Peer Voice Topology

In a Microsoft® DirectPlay® Voice session using a peer-to-peer topology, each voice-session client streams its voice audio data directly to every other voice-session client. Each client receives all individual incoming voice audio streams, mixes the received streams, and plays the resulting mixed signal on the client's computer.

The advantage of using a peer-to-peer topology is that no computer in the voice session requires high bandwidth or processor power. However, the bandwidth and processor usage on each client's computer varies according to the number of incoming and outgoing audio streams. The number of outgoing voice audio streams is equal to the number of targets participating in the voice session, unless the network provider is capable of true multicasting, as noted below. The number of incoming voice audio streams depends on how many other voice-session clients are targeting the client in question and also on how many of the other clients are speaking.

As a game design consideration, it is not useful for a voice-session client to be the target of more than about six to eight other clients. If all six to eight clients are speaking at once, the conversation can become confusing, and communication between clients can be difficult.

If the DirectPlay network session supports true multicasting, the number of outgoing voice audio streams can be reduced considerably. If all clients are part of a multicast network and the target of the voice stream is a DirectPlay group, there is only one outgoing stream. A DirectPlay voice session using a peer-to-peer voice topology supports 3-D spatialization of the voice data using the <u>IDirectPlayVoiceClient::Create3DSoundBuffer</u> method.

It is important to note that a voice session using a peer-to-peer voice topology cannot be used if the network transport is a client/server session.

Fast User Switching

Unlike previous versions of Microsoft® Windows®, Windows XP allows multiple users to log on simultaneously. Each user account has its own running applications and settings. The current user does not need to log off in order to switch to a new user. The *fast user switching* feature of Windows XP basically disconnects the current user account and brings the newly connected user account to the foreground. The disconnected user's applications and desktop settings remain loaded and running in the background.

While fast user switching offers a new level of convenience for users, it must be handled properly. Applications that use shared resources need to ensure they don't interfere with the currently connected application. Microsoft DirectPlay® Voice applications use shared resources and have the potential to cause problems when users switch. In particular, voice transmissions from the disconnected DirectPlay Voice application could still be audible, interfering with the current user's applications. To prevent such problems, DirectPlay Voice automatically mutes voice transmissions when the account is disconnected and unmutes transmissions when the account is connected again. This feature relieves your DirectPlay Voice application of the need to detect and handle a user switch.

Handling Voice Client Messages

This document describes how to handle messaging for a client of a Microsoft® DirectPlay® Voice session. For a discussion of host-related messaging, see <u>Handling Voice Host Messages</u>. For a discussion of general messaging issues, see <u>Handling DirectPlay Messaging</u>.

- <u>General Voice Messaging Considerations</u>
- <u>Startup Messages</u>
- <u>Messaging During Normal Game Play</u>
- <u>Session Termination Messages</u>

General Voice Messaging Considerations

Many of the messages used by DirectPlay Voice are similar to those used by the DirectPlay core application programming interface (API). However, be aware that similar core and voice messages sometimes differ in their usage.

For example, when you connect to either a core or voice session, DirectPlay returns a completion message with the results of your connection attempt. With a core session, you may get a <u>DPN_MSGID_CONNECT_COMPLETE</u> completion message under some circumstances, even when you connect synchronously. With DirectPlay Voice however, you will never get a completion message if you attempt to connect synchronously to a voice session. You get a <u>DVMSGID_CONNECTRESULT</u> completion message only if you connect to a voice session asynchronously.

One notable difference between core and voice message handling is that your core message handler receives every core message. With DirectPlay Voice, you have the option of specifying a list of messages that you want to receive by calling <u>IDirectPlayVoiceClient::SetNotifyMask</u>. You will receive only those messages that are on the notification list that you pass to this method through the *pdwMessageMask* parameter. This list must contain at least one message. You cannot use **IDirectPlayVoiceClient::SetNotifyMask** to disable all DirectPlay Voice messages.

Note DirectPlay and DirectPlay Voice sessions are separate entities. While there are guarantees about message order for each interface, there are no guarantees about message order between interfaces.

Startup Messages

DirectPlay Voice is an optional addition to a regular DirectPlay session. It enables voice communication between the session members. Before you can become a client of a voice session, you must have created the appropriate DirectPlay object and be connected to a regular peer-to-peer or client/server session. See <u>Peer-to-Peer Sessions</u> or <u>Client/Server</u> <u>Sessions</u> for details.

Every voice session must have a host. If you are in a peer-to-peer session, one of the members must be selected as the voice session host. The voice host need not be the same member as the one that is hosting the core session. If you are in a client/server session, the server must host the voice session as well as the core session. When a session host has been determined, the host must start the voice session by creating a voice server object and calling <u>IDirectPlayVoiceServer::StartSession</u>. When the session has been started, clients can connect.

This section outlines the messages that a voice client receives when joining a session.

• DVMSGID_CONNECTRESULT

Once the session is set up, clients must call <u>IDirectPlayVoiceClient::Connect</u> to connect to the voice session. If you call this method asynchronously and it returns <u>DV_PENDING</u>, you will receive a **DVMSGID_CONNECTRESULT** message with the result of the connection attempt. This message can arrive before or after **IDirectPlayVoiceClient::Connect** returns DV_PENDING. If you call the method synchronously, you will not receive a **DVMSGID_CONNECTRESULT** message. The result of a synchronous connection attempt is indicated by the method's return value.

<u>DVMSGID_CREATEVOICEPLAYER</u>

Voice clients receive this message only if they are a member of a peer-to-peer voice session. After you are connected and have processed **DVMSGID_CONNECTRESULT**, you will receive a **DVMSGID_CREATEVOICEPLAYER** message for yourself and each member of the voice session. The structure that accompanies this message contains the DVID value that you will use to identify the player during the session. The **dwFlags** member of this structure contains two flags that describe the player. If the player is the local player, the DVPLAYERCAPS_LOCAL flag will be set. If the player is half-duplex, the DVPLAYERCAPS_HALFDUPLEX flag will be set.

If you want to create a voice session player context value for the player, you must do so when you handle this message. For more information about player context values, see <u>Using Player Context</u> <u>Values</u>. You will receive no messages associated with a player until you have processed the corresponding **DVMSGID_CREATEVOICEPLAYER** message.

Messaging During Normal Game Play

Once you have successfully connected, there are a number of messages that you might receive during the course of a voice session. The list of possible messages depends on whether you are in a peer-to-peer or client/server voice session. As a general rule, voice clients in a client/server voice session do not receive any messages that are associated with a particular player.

Common Voice Messages

The following messages can be received by clients of both types of voice sessions.

DVMSGID_GAINFOCUS and DVMSGID_LOSTFOCUS

These messages let you know whether you have capture focus and are capturing audio. When you gain capture focus and audio capture starts, you receive a **DVMSGID_GAINFOCUS** message. When you lose capture focus and audio capture stops, you receive a **DVMSGID_LOSTFOCUS** message. Both messages are simple notifications and do not have an associated structure. For more information about capture focus, see <u>Capture Focus</u> and <u>Implementing a Callback Function in DirectPlay and DirectPlay</u> <u>Voice</u>.

Note Gaining or losing capture focus does not necessarily indicate the beginning or end of transmission. The beginning or end of transmission is indicated by the DVMSGID_RECORDSTART and DVMSGID_RECORDSTOP messages.

• DVMSGID_RECORDSTART and DVMSGID_RECORDSTOP

These messages notify you when the local player is transmitting.

You receive a **DVMSGID_RECORDSTART** message when transmission starts and a **DVMSGID_RECORDSTOP** when transmission stops. The structure that accompanies these messages contains the voice player context value for the local player. What causes transmission to start and stop depends on whether you are using push-to-talk or voice activation transmission control and whether your application has gained or lost audio capture focus.

The simplest case is when audio capture focus does not change. In that case, voice activated transmission starts when the user's voice exceeds the activation threshold. It stops when the voice drops below the activation threshold. With push-to-talk transmission, the user starts and stops transmission manually, typically by pushing and releasing a button.

Even if the player has exceeded the voice activation threshold or pushed **push-to-talk**, the application must have capture focus in order to transmit. If the player is talking but the application does not have audio capture focus, transmission begins only when the application gains audio capture focus. You will receive a **DVMSGID_GAINFOCUS** message followed by a **DVMSGID_RECORDSTART** message at that time. If you lose audio capture focus while the player is still talking, transmission immediately stops and you receive a **DVMSGID_LOSTFOCUS** message followed by a **DVMSGID_RECORDSTOP** message. If you get audio capture focus back and the player is still talking, transmission will start again.

You normally receive a **DVMSGID_RECORDSTOP** message when transmission stops, regardless of the reason. However, if you

disconnect from the voice session while you are transmitting, you are not guaranteed to receive a **DVMSGID_RECORDSTOP** message.

For details about capture focus, see **Capture Focus**. For details about transmission control, see <u>Transmission Control</u>.

DVMSGID_INPUTLEVEL and DVMSGID_OUTPUTLEVEL

The **DVMSGID_INPUTLEVEL** message gives you the current audio input level from your player's microphone. The **DVMSGID_OUTPUTLEVEL** message gives you the current audio output level from your player's speakers or headphones. These messages are sent at a regular time interval. They start after you have processed a **DVMSGID_RECORDSTART** message and stop after you process the corresponding **DVMSGID_RECORDSTOP** message. To specify the rate at which the messages are sent, set **dwNotifyPeriod** member of the <u>DVCLIENTCONFIG</u> structure to an appropriate time interval. To suppress these messages, set **dwNotifyPeriod** to 0.

Peer-to-Peer Messages

The following messages can be received only by clients of peer-to-peer voice sessions.

DVMSGID_CREATEVOICEPLAYER and DVMSGID_DELETEVOICEPLAYER

You receive a **DVMSGID_CREATEVOICEPLAYER** message each time a player is added to the voice session. You should handle this message in the same way as the player creation messages that you received when you connected to the session. When a player leaves the voice session, you receive a

DVMSGID_DELETEVOICEPLAYER message. You will receive a DVMSGID_DELETEVOICEPLAYER message for every DVMSGID_CREATEVOICEPLAYER message. When you have processed a DVMSGID_DELETEVOICEPLAYER message, you will receive no further messages for that player.

DVMSGID_SETTARGETS

The **DVMSGID_SETTARGETS** message is sent when the list of voice targets changes. It is generated any time a voice client calls IDirectPlayVoiceClient::SetTransmitTargets or a voice host calls IDirectPlayVoiceServer::SetTransmitTargets.

• <u>DVMSGID_PLAYERVOICESTART</u> and <u>DVMSGID_PLAYERVOICESTOP</u>

The **DVMSGID_PLAYERVOICESTART** message is sent when you start to receive a transmission from a player. If level notification is enabled, you will start receiving

DVMSGID_PLAYEROUTPUTLEVEL messages. The

DVMSGID_PLAYERVOICESTOP messages sent when that player's transmission stops. When you have processed this messages, the DVMSGID_PLAYEROUTPUTLEVEL messages stop. If audio capture focus changes, these messages are handled in essentially the same way as for DVMSGID_RECORDSTART and DVMSGID_RECORDSTOP. You will not receive these messages for a player until you have processed the corresponding DVMSGID_CREATEVOICEPLAYER message.

• DVMSGID_PLAYEROUTPUTLEVEL

The **DVMSGID_PLAYEROUTPUTLEVEL** gives you the current

audio output level on your player's speakers or headphones for a particular player's transmission. These messages are sent at a regular time interval. They start after you have processed a **DVMSGID_RECORDSTART** message and stop after you process the corresponding **DVMSGID_RECORDSTOP** message. To specify the rate at which the messages are sent, set **dwNotifyPeriod** member of the **DVCLIENTCONFIG** structure to an appropriate time interval. To suppress these messages, set **dwNotifyPeriod** to 0.

<u>DVMSGID_LOCALHOSTSETUP</u>

When the voice host migrates, the

DVMSGID_LOCALHOSTSETUP message is sent to the new voice host. This message allows the new voice host to set the callback function and context value that will be used when creating the new voice server object. When your application returns from handling this message, you will receive a <u>DVMSGID_HOSTMIGRATED</u> message.

DVMSGID_HOSTMIGRATED

When the host migrates, a **DVMSGID_HOSTMIGRATED** message is sent to all remaining members of the voice session. The associated structure contains the DVID of the new voice host. When the host migrates, DirectPlay Voice automatically creates a new voice server object. If your application is the new voice host, the **DVMSGID_HOSTMIGRATED** message's structure will also contain a pointer to the voice server's <u>IDirectPlayVoiceServer</u> interface. If you want to use this interface, you must call the interface's **AddRef** method to increment the interface's reference count. If you have called **AddRef** on **IDirectPlayVoiceServer**, be sure to call **Release** when you are finished with the interface.

Session Termination Messages

<u>DVMSGID_DISCONNECTRESULT</u>

Disconnect from a voice session by calling IDirectPlayVoiceClient::Disconnect. If you call this method asynchronously and it returns DV_PENDING, you will receive a DVMSGID_DISCONNECTRESULT to notify you of the outcome. This message can arrive before or after IDirectPlayVoiceClient::Disconnect returns. The DVMSGID_DISCONNECTRESULT message is not sent if you call IDirectPlayVoiceClient::Disconnect synchronously. The result of a synchronous disconnection attempt is indicated by the method's return value.

For peer-to-peer voice sessions, the client will receive a **DVMSGID_DELETEVOICEPLAYER** for every remaining player in the sessionbefore the client receives **DVMSGID_DISCONNECTRESULT**.

<u>DVMSGID_SESSIONLOST</u>

If the session terminates unexpectedly because of an unrecoverable error, clients will receive a **DVMSGID_SESSIONLOST** message. The reason for the failure is specified in the **hrResult** member of the associated structure.

For peer-to-peer voice sessions, the client will receive a **DVMSGID_DELETEVOICEPLAYER** for every remaining player in the sessionbefore the client receives **DVMSGID_SESSIONLOST**.

DVMSGID_DELETEVOICEPLAYER

For peer-to-peer sessions, you will receive a **DVMSGID_DELETEVOICEPLAYER** for every player remaining in the voice session when you are disconnected, regardless of whether you call **IDirectPlayVoiceClient::Disconnect** or the session is lost. This ensures that each **DVMSGID_CREATEVOICEPLAYER** that you receive has a corresponding **DVMSGID_DELETEVOICEPLAYER** message. You will receive all **DVMSGID_DELETEVOICEPLAYER** messages before the disconnect is completed.

Handling Voice Host Messages

Every Microsoft® DirectPlay® Voice session must have a host that is responsible for starting, managing, and terminating the session. A voice host must create a voice server object (CLSID_DirectPlayVoiceServer) and manage the voice session through that object's IDirectPlayVoiceServer interface. Like a voice client, a voice host must have a message handler and will receive messages during a session. A voice host message handler can receive any of the following three messages.

- <u>DVMSGID_CREATEVOICEPLAYER</u>
- <u>DVMSGID_DELETEVOICEPLAYER</u>
- DVMSGID_SESSIONLOST

Unlike core DirectPlay messaging, none of these messages are unique to a voice session host. Depending on the type of session, these messages may also be received by voice client message handlers. This document gives a brief description of how a voice session host handles messaging. For specific details on how to handle these messages, see <u>Handling</u> <u>Voice Client Messages</u>. For a discussion of general messaging issues, see <u>Handling DirectPlay Messaging</u>.

Messaging During Normal Game Play

A client/server voice host will normally receive the following messages after the voice session starts.

• DVMSGID_CREATEVOICEPLAYER

You will receive a **DVMSGID_CREATEVOICEPLAYER** message every time a player joins your voice session.

• DVMSGID_DELETEVOICEPLAYER

You will receive a **DVMSGID_DELETEVOICEPLAYER** message every time a player leaves your voice session.

Session Termination Messages

A client/server voice host may receive one or more of the following messages when the voice session terminates.

• DVMSGID_SESSIONLOST

You will receive a **DVMSGID_SESSIONLOST** message if the underlying core session is disconnected. For example, the core session will be disconnected when the underlying DirectPlay peer or server transport object is closed, or when your host is disconnected from the network.

• DVMSGID_DELETEVOICEPLAYER

You will receive one **DVMSGID_DELETEVOICEPLAYER** message for each client remaining in the session when the voice session is terminated.

Jitter Buffers

Microsoft® DirectPlay® Voice features a jitter buffer, an adaptive buffering algorithm that provides optimal voice quality with the least amount of latency.

On busy networks, individual packets of voice data information might arrive in a different sequence from that in which they were encoded on the host computer. Because voice data is sequential in nature, these incoming packets must be queued for a time so that delayed packets have an opportunity to arrive and be played back in order.

If the jitter buffer is set to maximize the quality of voice communication, it takes longer for the required number of voice packets to arrive and be queued for play. The result is voice latency, and the effect is that voice communication is not heard in real time. Instead, the voice data might be heard anywhere from a fraction of a second to several seconds after it was recorded. This can introduce problems during cooperative game play because events can occur in the game but players will not be able to communicate information based on those events in real time. For example, if a player in a first-person shooter is about to be attacked from behind and a teammate attempts to warn the player, the voice communication might not be heard until after the player has been attacked.

If the jitter buffer is set to a reduce latency, the number of packets required to fill the queue is reduced. However, it is possible that not all sequential packets will arrive in time and, as a result, voice data will be missing from the buffer when it is played. The voice communication will be heard much closer to the actual time it was recorded. However, it will have a "broken-up" quality.

The DirectPlay jitter buffer uses two methods to determine how to provide the best quality of voice communication with the least amount of latency. First, network conditions are monitored to determine the amount of lag or network congestion. The size of the jitter buffer, or queue, is then dynamically sized to keep latency as low as possible while providing the least amount of voice break up.

The default behavior of DirectPlay Voice jitter buffer is to automatically adjust to network conditions. You can manually adjust how closely the algorithm tracks network conditions using the **dwBufferAggressiveness** and **dwBufferQuality** members of the <u>DVCLIENTCONFIG</u> structure. The higher the level of "aggressiveness," the more closely the algorithm monitors network conditions. In general, the higher the quality value, the higher the quality of the voice, but the higher the latency as well. The lower the quality value, the lower the latency but the lower the quality of the voice. You can set these two members when you call IDirectPlayVoiceClient::Connect to connect to a session, and at any time during the session by calling IDirectPlayVoiceClient::SetClientConfig.

It is important to choose an appropriate level of aggressiveness for network conditions when your game application is running because selecting a high level of aggressiveness during times of steady network performance can cause the algorithm to misinterpret a transitory problem and overcompensate for a problem that might not exist.

Sharing the Audio Capture Device

The most recent versions of Microsoft® Windows®, Windows Millennium Edition (Windows Me), and Windows XP allow the audio capture device to be shared between different processes. However, in Windows 95, Windows 98, and Windows 2000 operating systems, the audio capture device is a *shared resource* and can be opened by only one process at a time. To use audio capture on these operating systems, your application must be designed to effectively handle resource ownership and coordination issues. This document discusses how multiple Microsoft DirectPlay® Voice applications can share the same audio capture device on any Windows operating system.

Coordinating Shared Resources

Resource ownership becomes a problem when a physical resource such the audio capture device is a shared resource. If one application owns the shared audio capture device, other applications cannot open it. In the past, this limitation rarely posed problems because few applications used audio capture. However, the increasing popularity of technologies such as voice recognition means that applications can no longer assume that they can have exclusive ownership of the audio capture device. For instance, a lobby operator might want to implement a voice chat feature to help clients set up game sessions. However, if the lobby client application owns the audio capture device, the game application will not be able to use audio capture and its voice features will be effectively disabled.

Resource coordination is the process by which two or more applications manage ownership of a shared resource. For example, if two voice communication applications are running, the foreground application is typically the only application that needs the audio capture device. When focus changes, ownership of the audio capture device needs to shift to the new foreground application. For instance, after a group of players have finished using a lobby's voice chat feature to set up their session, they will launch their game applications. At that point, the game application moves to the foreground and acquires ownership of the audio capture device for game-related voice features.

DirectPlay Voice handles the problem of sharing the audio capture device through a feature called *capture focus*. Capture focus operates on a perdevice basis, and allows multiple DirectPlay Voice applications to share a single audio capture device regardless of which version of Windows is
installed on the system. As long as capture focus is enabled, any number of DirectPlay Voice application instances can have the same audio capture device open. Capture focus also provides a way for DirectPlay Voice applications to coordinate their use of the audio capture device. Capture focus is not available to applications that are not using DirectPlay Voice.

Using Capture Focus

DirectPlay Voice automatically coordinates the use of the audio capture device for all DirectPlay Voice applications that have enabled capture focus. If multiple DirectPlay Voice applications are running with capture focus enabled, focus is normally coordinated in the following way.

- The application whose window is in the foreground has capture focus and can capture audio.
- An application retains capture focus until another DirectPlay Voice application that has enabled capture focus gains the foreground.
- None of the other applications can capture audio until they acquire capture focus.

The shift in capture focus normally occurs after the shift in window focus. Typically, your application will acquire capture focus one to two seconds after your window gains the foreground.

Capture focus is enabled by default. To disable capture focus, you must set the DVSOUNDCONFIG_NOFOCUS flag in the **dwFlags** member of the <u>DVSOUNDDEVICECONFIG</u> structure when you call <u>IDirectPlayVoiceClient::Connect</u> to connect to a voice session.

Note Use of the DVSOUNDCONFIG_NOFOCUS flag is not recommended. If you set this flag and an instance of your application successfully opens the audio capture device, no other DirectPlay Voice applications will be able to use that device.

To specify which window you want DirectPlay Voice to use for focus determination, assign its window handle to the **hwndApp** member of the **DVSOUNDDEVICECONFIG** structure when you connect to the voice session. You should use the window handle of your application's top-level window for focus determination. Do not use the desktop window. Doing

so may lead to inconsistent determination of focus.

When capture focus shifts, DirectPlay sends a <u>DVMSGID_LOSTFOCUS</u> message to the application that has lost focus. When an application receives this message, audio capture stops until the application regains focus. DirectPlay sends a <u>DVMSGID_GAINFOCUS</u> message to the application that has gained focus. This message indicates that the application has started capturing audio. It will continue to do so until it loses focus to another application.

Note This document describes how to share capture focus between different processes. However, it is possible to share capture focus among multiple instances within a single process if each instance has a unique window handle.

Strict Mode

By default, your application loses capture focus only if another capture focus-enabled DirectPlay Voice application gains the foreground. Your application can retain capture focus even if it is no longer the foreground application. It must only be higher in the z-order than any other capture focus-enabled application. If you want your application to have capture focus only when it is the foreground application, specify *strict mode*. In strict mode, a DirectPlay Voice application must be active and at the top of the z-order in order to have capture focus. If it loses that status to any other application, audio capture will cease. To specify strict mode, set the DVSOUNDCONFIG_STRICTFOCUS flag in the **dwFlags** member of the **DVSOUNDDEVICECONFIG** when you connect to the voice session.

Muting Capture Focus

You can give up capture focus programmatically by muting audio capture. To mute audio capture, call <u>IDirectPlayVoiceClient::SetClientConfig</u>, and set the DVCLIENTCONFIG_RECORDMUTE flag in the **dwFlags** member of the <u>DVCLIENTCONFIG</u> structure. When you mute audio capture, your application loses capture focus and the last application that had capture focus gains it again. If no other application has capture focus enabled, DirectPlay stops audio capture. You can unmute audio capture and attempt to regain capture focus by calling

IDirectPlayVoiceClient::SetClientConfig without the DVCLIENTCONFIG_RECORDMUTE flag. However, this action does not necessarily return capture focus to your application.

- If yours is the only capture focus-enabled application, you will immediately regain capture focus.
- If your application is higher in the z-order than any other capture focus-enabled application, you will immediately regain capture focus even if you are not the foreground application. In particular, when you mute audio capture, capture focus often passes an application application below yours in the z-order. If that order does not change, you will regain capture focus when you unmute audio capture.
- If capture focus has passed to another capture focus-enabled application that is higher than yours in the z-order, you do not immediately regain capture focus. Your application must become the foreground application in order to regain capture focus.
- If you are in strict mode, you will regain capture focus only if your application is at the top of the z-order. Your application must become the foreground application in order to regain capture focus.

^{© 2003} Microsoft Corporation. All rights reserved.

Transmission Control

To keep the performance requirements of Microsoft® DirectPlay® Voice low, voice data should be transmitted only when the user is speaking. There are three ways to control voice data transmission.

- Push-to-Talk
- Manual Voice Activation
- Automatic Voice Activation

You select which transmission control method to use when you call IDirectPlayVoiceClient::Connect to connect to a voice session. To specify the type of transmission control you want to use, set the appropriate flag in the DVCLIENTCONFIG structure's **dwFlags** member. You can change the method during the session by calling

IDirectPlayVoiceClient::SetClientConfig and changing the flag setting.

Push to Talk

Push-to-talk transmission control is analogous to pushing the Talk button on a two-way radio. It adds reality to certain game genres such as firstperson shooters. Push-to-talk requires users to actively select when they want to transmit voice data. There is no danger that anything besides voice data will activate transmission. In addition, requiring users to actively select when they want to speak reduces the number of users speaking at once.

Push-to-talk transmission control requires more design and development than voice activation. In particular, you must have some way of detecting when the user has chosen to start transmitting, typically by pushing a controller button or a key on the keyboard. When you detect that the user wants to start talking, you must start transmission by calling IDirectPlayVoiceClient::SetTransmitTargets, and provide an array of target identifiers (IDs) that will receive the transmission. When you detect that the user is finished, stop transmission by calling IDirectPlayVoiceClient::SetTransmitTargets again, with the target array set to NULL and the number of targets set to 0.

Push-to-talk is the default transmission control method. It is enabled unless you explicitly select voice activation by setting the DVCLIENTCONFIG_AUTOVOICEACTIVATED or DVCLIENTCONFIG_MANUALVOICEACTIVATED flag in the DVCLIENTCONFIG structure.

Voice Activation

With voice activated transmission control, the microphone input is constantly analyzed to determine if the user is speaking. When the input exceeds a threshold level, voice activation is triggered, and the user begins transmitting. Ideally, transmission starts when the user starts speaking, and stops when the user finishes.

Voice activation is simpler for the user than push-to-talk, because the user must only speak into the microphone. It is also more easily coded because you do not need to detect when the user wants to start or stop transmitting. You specify voice activation when you connect, and transmission control is handled by the system from that point on. However, one drawback of voice activation is that it can result in unwanted transmissions. In addition to speech, transmission can be triggered by sounds such as the user breathing directly on the microphone, high levels of ambient sound in a noisy environment, or a set of external speakers playing the game's audio background. Lowquality microphones can increase the probability of unwanted transmission.

Voice activation can be either automatic or manual. To specify one of these modes, set the DVCLIENTCONFIG_AUTOVOICEACTIVATED or DVCLIENTCONFIG_MANUALVOICEACTIVATED flag, respectively, when you connect to the voice session.

Automatic voice activation is the preferred transmission control method for most applications. In this mode, the threshold for transmission is determined automatically by the system. The threshold level is adaptive, adjusting itself automatically to the input signal. With manual voice activation, you must explicitly set a threshold when you connect to the voice session by assigning a value to the **dwThreshhold** member of **DVCLIENTCONFIG**. The system will not change this value for you. If conditions change, and the threshold value is no longer adequate, you must call

IDirectPlayVoiceClient::SetClientConfig and specify a new value for dwThreshhold.

© 2003 Microsoft Corporation. All rights reserved.

Voice Codecs

The compression/decompression (codec) algorithms provided with Microsoft® DirectPlay® are optimized for low-bandwidth voice compression and decompression. These codecs all operate on data based on an 8 kHz, 16-bit, mono format. However, DirectPlay Voice handles all the details of converting voice data to and from this intermediate format. Non-Microsoft codecs are not supported, and you cannot write proprietary codecs for use with DirectPlay Voice.

It is important to note that as the bandwidth requirements drop, the audio quality of the voice data also drops. The following table lists the supported codecs, the bandwidth in kilobits per second (Kbps), and the compression globally unique identifier (GUID) used to select them. The compression GUIDs are defined in Dvoice.h.

| Codec | Bandwidth | GUID |
|---|---------------------------|----------------------|
| Voxware VR12 | variable (1.2 Kbps, avg.) | DPVCTGUID_VR12 |
| Voxware SC03 | 3.2 Kbps | DPVCTGUID_SC03 |
| Voxware SC06 | 6.4 Kbps | DPVCTGUID_SC06 |
| TrueSpeech | 8 Kbps | DPVCTGUID_TRUESPEECH |
| Global System for Mobile Communications (GSM) | 13 Kbps | DPVCTGUID_GSM |
| Microsoft Adaptive Delta Pulse Code Modulation (MS- ADPCM) | 32 Kbps | DPVCTGUID_ADPCM |
| Pulse Code Modulation (PCM) | 64 Kbps | DPVCTGUID_NONE |

The first three codecs provide a high level of compression and have approximately the same resource demands. On a 500 MHz Pentium III class computer, these codecs use approximately 1.5 percent of the CPU capacity. The VR12 codec sounds tinny and robotic, but the SC03 and SC06 codecs provide reasonable fidelity. The PCM codec provides the highest sound quality and is essentially uncompressed, 8 kHz, 16-bit, mono-format audio data. **Note** The GSM, ADPCM, and PCM codecs are included with the Microsoft Windows® installation but might not have been installed by the user. You might need to prompt the user to install them. You can determine which codecs are available on a system by calling IDirectPlayVoiceServer::GetCompressionTypes. If a codec is not listed, the corresponding Audio Compression Manager (ACM) codec is not installed.

Selecting a Codec

As with all other game setup parameters, the host controls which codec is used for the voice session. All members of the voice session must use the same codec. Remember that in a peer-to-peer voice session, the voice-session host does not necessarily have to be the same as the game-data host. The host selects the codec when it calls IDirectPlayVoiceServer::StartSession. Set the **guidCT** member of the DVSESSIONDESC structure to the compression GUID of the codec that you want to use. A client can retrieve this structure by calling IDirectPlayVoiceClient::GetSessionDesc.

The same codec might not be ideal for the entire duration of a game. For instance, you might want to use one codec for the lobby chat feature that players use to set up the game, and another to handle voice communication after the game is launched. You cannot dynamically change codecs during a voice session. To switch to another codec, you must terminate the current voice session and create a new voice session with the new codec. However, you can stop and restart a voice session without terminating the underlying DirectPlay core session.

As with any form of network communication, it is important to analyze the cost of the voice communication to ensure that adequate bandwidth is available to support communication of the game data and voice data. Analyzing the voice bandwidth consumption is straightforward: Estimate the number of simultaneous voice streams that you anticipate and multiply that number by the sum of the bandwidth required by the codec and the protocol overhead. CPU consumption is another factor to consider when choosing a codec. As with network bandwidth, CPU resource consumption is additive, per stream.

 $\ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.

Voice Host Migration

In a peer-to-peer Microsoft® DirectPlay® network session, one client of the networking session acts as host. If that host exits the session or stops responding for any reason, another client in the session is elected as host.

In a DirectPlay voice session, a similar process of host migration occurs in peer-to-peer voice sessions, except that the voice host migrates independently of the DirectPlay network session. The voice host migrates when the server calls <u>IDirectPlayVoiceServer::StopSession</u> or if the voice host stops responding.

When the voice host migrates, the new host's voice client message handler receives a <u>DVMSGID_LOCALHOSTSETUP</u> message. The primary purpose of this message is to allow the new host to provide DirectPlay with a pointer to the callback message handler that will receive voice server messages. To specify your callback message handler, set the value of the **pMessageHandler** member of the

DVMSGID_LOCALHOSTSETUP structure to point to your voice server message handler before you return from the client message handler. The new host can also specify a voice server context value by setting the value of the **pvContext** member of the structure.

Each client in the voice session receives a DVMSGID_HOSTMIGRATED message with the DVID of the new host. The client that is chosen as the new host also receives a valid IDirectPlayVoiceServer pointer, which the client can use to call the voice server methods. The new host does not receive DVMSGID_HOSTMIGRATED until after it has processed DVMSGID_LOCALHOSTSETUP.

 $\ensuremath{\textcircled{\sc 0}}$ 2003 Microsoft Corporation. All rights reserved.

Working Set Guidelines

Determining the best configuration of transport topology, voice topology, transmission control, and codec depends on the type or genre of game you are creating, the number of players that will participate in a single game session, and the type of connection that will be targeted.

It's important to note that the number of players participating in the voice session is not necessarily the number of players actually participating in the game session. For example, if your game is a first-person shooter, voice communication can be represented in the game as a radio or communicator that is offered as a time-limited powerup. Also, the radio metaphor can be used to limit communication to radios in either vehicles or stationary command stations.

A second example to consider is an online bridge game, which involves four players at one time. Because this is a small working set, it is appropriate to choose a peer-to-peer voice topology transported over a peer-to-peer network topology. This small working set also allows for the use of voice activation as the mode of transmission control. The peer-topeer voice topology is easily implemented and does not require any player to act as a server. If all four players use the Voxware SC6 codec, the maximum resulting bandwidth is 4.2 Kbps per speech stream, including the codec protocol overhead. Further assuming that game data requires negligible bandwidth, the outgoing maximum bandwidth requirement for an individual speaker is three independent streams to the other three players, or 12.6 Kbps. The incoming stream to any client ranges from 0 if no other players are talking, to 12.6 Kbps if all three other players speak simultaneously. The CPU requirement is 8 percent for encoding and 0 to 12 percent for decoding. This results in a worstcase requirement of 25.2 Kbps. Therefore each player must have a minimum of a 14,400-baud modem.

Another example is a squad combat game that can involve up to 32 players split between 2 teams. Assume that the game data requires a 28,800 baud modem. In this example, there is a larger number of players and it is appropriate to choose a forwarding server voice topology. Again, if all players use the Voxware SC6 codec, the bandwidth requirements are the same as the bridge game above: 4.2 Kbps. In this example, note that there is 4.2 Kbps outgoing when speaking, and a maximum of 12.6 Kbps incoming from the squad. The maximum CPU requirement is 8 percent of a Pentium 200 for encode and 12 percent receiving. Therefore, each player requires 28.8 Kbps for game data, and the greater incoming bandwidth of 12.6 Kbps requires a minimum 41,400 baud rate from each player's modem.

The worst-case scenario for the forwarding server itself is if all 32 players talk at once, requiring 134.4 Kbps. The server CPU use is minimal because the server is not encoding or decoding the streams. It is merely redirecting them. More typically, there might be 16 players talking simultaneously for 67.2 Kbps.

To illustrate the difference between choosing a mixing server voice topology and a forwarding server voice topology, consider the same 32player squad combat game discussed above. If a mixing server voice topology is used, each client requires 4.2 Kbps to send and 4.2 Kbps to receive. The worst-case bandwidth requirements drop to 8.4 Kbps and 12 percent of the Pentium processor running at 200 MHz. This reduces the modem requirement to a 33,600 Kbps baud rate for the client.

For the server, the CPU burden changes. The server is now decoding

and re-encoding all incoming streams and is also mixing the streams as required. The CPU burden mix the stream is relatively low and is considered negligible. The worst case is the decoding and encoding of 32 simultaneous streams. This results in a requirement of at least a Pentium II processor running at 400 MHz for the voice service alone.

© 2003 Microsoft Corporation. All rights reserved.

DirectPlay for Pocket PC 2002

Beginning with Microsoft® DirectX® 8.1, the Microsoft DirectPlay® application programming interfaces (APIs) are available for the Microsoft Windows® Powered Pocket PC 2002. The implementation of the APIs is the same on all platforms, with the following exceptions for Pocket PC 2002.

- The DirectPlay Voice APIs are not available.
- Only the IPv.4 service provider and <u>Bluetooth Service Provider</u> are available. The IPv.4 service will function over the wireless 802.1*x* network. Currently, serial, modem, and Internetwork Packet Exchange (IPX) are not available.
- Because pop-up windows are not available on the Pocket PC 2002, the DPNHOST_OKTOQUERYFORADDRESSING, DPNCONNECT_OKTOQUERYFORADDRESSING, and DPNENUMHOSTS_OKTOQUERYFORADDRESSING flags cannot be used.

© 2003 Microsoft Corporation. All rights reserved.

Bluetooth Service Provider

The Bluetooth wireless technology service provider is a low-cost, lowpower wireless technology designed to simplify setting up network connections between devices. This topic describes how to use Bluetooth with the Microsoft® Windows® Powered Pocket PC 2002.

Note The Bluetooth service provider in Microsoft DirectPlay® is only supported with the Pocket PC 2002. It is not available for the desktop version. Applications wishing to operate between the desktop and the Pocket PC 2002 should use the DirectPlay Internet Protocol (IP) service provider.

Native Bluetooth support in DirectPlay is designed for hosting sessions where all players in the session are also using the Bluetooth service provider. In this scenario, all players must be located within Bluetooth radio range of each other to participate. DirectPlay does not support using a mixture of service providers within a single session. Note that Bluetooth can also be used as an access technology for other network types. For example, a player might obtain IP connectivity using a Bluetooth Network Access Point. In this case the Bluetooth device is essentially invisible to DirectPlay, and an application should use the IP service provider instead.

The Bluetooth service provider for DirectPlay provides feature parity with the IP service provider. All session types are supported client, server, and peertogether with the full set of features from each. However, some minor variations in setting up addresses and enumerations with Bluetooth are described here.

Addressing

Applications should always check to see if the Bluetooth service provider is available before attempting to use it or offering it as an option to the user. Use the IDirectPlay8Peer::EnumServiceProviders or IDirectPlay8Client::EnumServiceProviders or IDirectPlay8Server::EnumServiceProviders method to see if Bluetooth is available.

If the Bluetooth service provider is found, applications should set Bluetooth as the service provider for the address object by calling <u>IDirectPlay8Address::SetSP</u> as illustrated in the following example.

```
IDirectPlay8Address* pAddress;
hr = CoCreateInstance( CLSID_DirectPlay8Address, NULL,
CLSCTX_INPROC_SERVER,
IID_IDirectPlay8Address,
(LPVOID*) &pAddress);
```

pAddress->SetSP(&CLSID_DP8SP_BLUETOOTH);

When creating <u>IDirectPlay8Address</u> objects, applications need to specify the address of the host device with the <u>Host Name</u> address component. Each host name should be specified as a string using hex format as shown in the following example.

```
WCHAR * wszDeviceAddress=L" 50c20003c418";
pAddress->AddComponent(DPNA_KEY_HOSTNAME, wszDevice/
(wcslen(wszDeviceAddress)+1)*sizeof(WCHAR), DPNA_DATATYF
```

Address objects supplied to the application from DirectPlay as the result of an enumeration operation may be enhanced by adding the name information component. This specifies the name of the Pocket PC 2002 device the address represents and provides a more user friendly string to present rather than the Bluetooth device address. Therefore, applications should test for the presence of the DPNA_KEY_NAMEINFO field when displaying the result of enumerations and present it to the user if possible. Note that this field is purely an informative one and is not used by DirectPlay in anyway. Applications creating their own address objects rather than obtaining them using enumerations should always use the DPNA_KEY_HOSTNAME field to specify the address of a remote device. Connecting to a device using purely the DPNA_KEY_NAMEINFO component is not supported.

Enumeration

When enumerating Bluetooth devices, DirectPlay first attempts to discover all the Bluetooth devices in the area. Each device found is contacted one at a time and sent an enumeration request. This process is hidden from the application but the initial device discovery process can take 10 or more seconds to complete. Therefore, a call to IDirectPlay8Peer::EnumHosts or IDirectPlay8Client::EnumHosts will generate no responses initially and then a rapid sequence of responses after about 10 seconds.

Because of the time it takes to enumerate and because Bluetooth addresses are fixed for each device, it is recommended that applications provide a mechanism to store the last few addresses to which a player connected. This allows a user to pick from a list of recent connections rather than needing to type in a hexadecimal address each time or wait for the enumeration to complete.

A call to **EnumHosts** works slightly differently with the Bluetooth service provider. The following parameters are different.

- *dwEnumCount* is unused.
- *dwRetryInterval* determines the period in milliseconds that DirectPlay waits for a response from a single device before moving onto the next device.
- *dwTimeout* determines the total period allowed for the enumeration operation.

Note Setting *dwTimeout* to less than 15 seconds is pointless, given the minimum discovery period outlined above. The recommended approach is to set this value to Infinity, and wait for the user to either cancel the enumeration or select a session.

Attempting to connect to or host a session automatically cancels any running enumeration. When connecting causes the enumeration cancellation, the result code for the enumeration supplied in the <u>DPN_MSGID_ASYNC_OP_COMPLETE</u> message will be DPNERR_CONNECTING. When hosting causes the enumeration cancellation, the result code will be DPNERR_HOSTING.

Session Limitations

Due to the limits of the Bluetooth specification, the maximum number of devices supported in a peer-to-peer session is eight. For a client/server session, up to seven clients can connect to a single server.

Host migration can take several seconds in Bluetooth, and during this period no data transmission occurs between devices. Therefore, if the host player drops from the game, any messages sent by the remaining players will be queued during the time it takes to reestablish the network connectivity.

Bluetooth is essentially a client/server architecture. Although DirectPlay hides this and offers a peer-to-peer mode with host migration, all data still has to be routed through the host in this scenario. Applications should consider this when making assumptions about latency and available bandwidth.

^{© 2003} Microsoft Corporation. All rights reserved.

Advanced Topics in DirectPlay

This section of the Microsoft® DirectPlay® documentation is designed to show you how to use the DirectPlay application programming interface (API) to implement important aspects of multiplayer applications.

Overviews

- DirectPlay Addressing
- DirectPlay Callback Functions and Multithreading Issues
- DirectPlay Protocol
- DP8Sim Utility
- <u>Monitoring DirectPlay Network Traffic with Netmon</u>
- Network Address Translation, Firewalls, and Proxies
- <u>Optimizing Network Usage</u>
- Packet Signing
- <u>Testing Network Performance</u>
- <u>Using Player Context Values</u>
- Using the DirectPlay DPNSVR Application

© 2003 Microsoft Corporation. All rights reserved.

DirectPlay Addressing

To deliver messages, each participant in a multiplayer game must have a unique address. Addresses can refer to either the computer on which your application is running (*device address*), or a computer with which your application needs to communicate (*host address*).

Microsoft® DirectPlay® represents addresses in the form of a URL string. That address string is then encapsulated in a DirectPlay address object that is passed as a parameter in or out of methods such as IDirectPlay8Peer::Connect.

This section describes three ways of handling DirectPlay addresses.

- <u>DirectPlay Service Providers</u> discusses what service providers are, and which ones are available with DirectPlay.
- <u>DirectPlay and Ports</u> discusses what ports are, and how to use them.
- Internet Protocol Version 6 discusses how to use Internet Protocol, version 6 (IPv6) with DirectPlay.
- <u>DirectPlay URLs</u> discusses how to construct the address string directly.
- <u>Handling Addresses</u> discusses how to pass address objects in the **Host**, **EnumHosts**, or **Connect** methods.
- <u>DirectPlay Address Objects</u> discusses how to manipulate the address string using the methods exposed by the address object's <u>IDirectPlay8Address</u> interface.
- <u>Data Value Summary</u> contains a table listing all the possible data values associated with DirectPlay addresses and shows which values are required for each service provider.

© 2003 Microsoft Corporation. All rights reserved.
DirectPlay Service Providers

Microsoft® DirectPlay® provides your application with a virtual network connection that enables you to communicate with other computers in the same way, regardless of type of network involved. To provide this level of abstraction, network connections are made through a service provider. When you have selected a service provider, your application uses the appropriate DirectPlay methods to communicate with other computers in a session. The service provider handles the details of communicating over the selected network hardware.

DirectPlay includes service providers for four types of network connections: Transmission Control Protocol/Internet Protocol (TCP/IP), Internetwork Packet Exchange (IPX), modem, and serial. See <u>DirectPlay</u> <u>Address Objects</u> for a discussion of how to select a service provider.

Note DirectPlay uses the telephony application programming interface (API) (TAPI) to handle modem communication. The use of this API means that the code that is used to answer the phone must be in the message loop's thread.

© 2003 Microsoft Corporation. All rights reserved.

DirectPlay and Ports

This topic explains how to select a game port. Ports are an important part of addressing for Internet Protocol (IP) and Internetwork Packet Exchange (IPX) service providers. In cases where each computer has a unique IP address, the port specifies the application running on the computer. Microsoft® Windows® allows only one application per port. So if a user were playing a networked game, sending e-mail, and downloading files at the same time, each application would be bound to a different port. Each message received would have a port number in its address so that it could be passed to the correct application.

In cases where computers are behind a Network Address Translation (NAT) device and sharing a single public IP address, the port not only specifies the application but also the computer on which the application is running. So, if three computers sharing the same IP address were running the same application, each application would be bound to a different port, allowing messages to be delivered to the correct computer and application.

Also, if you specify a port, you don't need to use DPNSVR, which makes NAT device administration easier for users. Your application should choose a default port but allow users to override that port in case it is already in use. For more information about configuring game ports when players are behind NAT devices, see <u>Network Address Translation</u>, <u>Firewalls, and Proxies</u>.

Note Ports are not used for serial or modem service providers.

Setting a Port

Set a port by calling the <u>IDirectPlay8Address::AddComponent</u> method. In the following example, port 12345 is selected.

```
IDirectPlay8Address* pAddress;
DWORD dwPort;
...
dwPort = 12345;
hr = pAddress->AddComponent(DPNA_KEY_PORT, &dwPort, sized
```

The port is specified as a **DWORD**, although the port number is limited to 65535 (2 bytes) and is used in host byte order. When selecting a port, do not choose a reserved port.

Ports Selected by DirectPlay

If you do not specify a port as described above, Microsoft DirectPlay® will select a port between 2302 and 2400 for local use when hosting, enumerating, or connecting. If hosts enable DPNSVR, the helper application will use port 6073 to forward enumeration queries to the application. This is also the default port number used when no port is specified in the remote host address passed to

IDirectPlay8Peer::EnumHosts and IDirectPlay8Client::EnumHosts. And finally, if your application does not choose to disable the automated NAT features, you may notice additional ports being opened. The actual port numbers selected for this feature are not deterministic.

Reserved Ports

All ports under 1024 are reserved.

Also, DirectPlay has reserved the following ports.

| Port | Reserved for |
|-----------|--|
| 1900 | Universal Plug and Play (UPnP) - Simple Service Discovery Protocol |
| 2302-2400 | DirectPlay device address default |
| 2234 | Internet sharing and firewall support for Windows Millennium Edition (Windows Me) and Windows XP |
| 6073 | DirectPlay enumerations using DPNSVR |
| 47624 | Deprecated DirectPlay enumeration |

The Internet Assigned Numbers Authority (*) maintains a list of registered User Datagram Protocol (UDP) ports that you should avoid using.

© 2003 Microsoft Corporation. All rights reserved.

Internet Protocol Version 6

The Internet Protocol (IP) is the foundation upon which all Internet traffic is based. The current version, Internet Protocol, version 4 (IPv4), has remained unchanged for over two decades. While this is a testament to its original design, no one ever envisioned the unprecedented growth the Internet has seen. The demand for unique addresses used to route traffic through this network has grown so large that many are predicting the supply will be exhausted in a few short years. Technologies such as Network Address Translation (NAT) have only marginally extended the IPv4 lifetime at the cost of end-to-end connectivity for many applications such as games. Internet Protocol, version 6 (IPv6) is the next generation networking protocol designed to address this and many other concerns.

The Microsoft® DirectPlay® Transmission Control Protocol/Internet Protocol (TCP/IP) service provider supports both IPv4 and IPv6 transparently. This ensures that the application you write today will automatically take advantage of the IPv6 features as its adoption becomes more widespread. The only behavior required of your application is that it should treat the value of the DPNA_KEY_HOSTNAME address component returned by DirectPlay as a string. Your application should not expect it to be in an IPv4 address, formatted using decimal dotted notation. The DPNA_KEY_HOSTNAME component can be an IPv4 address, an IPv6 address, or a Domain Name System (DNS) hostname string. You should simply display DPNA_KEY_HOSTNAME as a string without parsing. Alternatively, use match-making services or friendly player names to hide the address complexity from the user altogether.

For more information about creating TCP/IP protocol DirectPlay

addresses, see <u>Creating TCP/IP Address Objects</u>. For more information about match-making services, see <u>DirectPlay Lobby</u>.

© 2003 Microsoft Corporation. All rights reserved.

DirectPlay URLs

Microsoft® DirectPlay® represents addresses as URLs. In general, URLs are strings that consist of three basic components in the following order: scheme, scheme separator, and data string.

All DirectPlay addresses use "x-directplay" as the scheme, and ":/" (a colon followed by a forward slash) as the scheme separator. Using ":/" as a separator implies that the data that follows is *opaque*. In other words, the data string does not conform to any Internet standard and should be passed to the receiving application without modification. All DirectPlay URLs thus have the following general form.

x-directplay:/[data string]

Note Do not use "://" (a colon followed by two forward slashes) as a scheme separator. That separator implies that the data that follows conforms to an Internet standard and can be interpreted as such. To prevent confusion, DirectPlay flags any URL containing "://" as invalid.

This section discusses the following topics.

- Data Strings
- Sample URLs

© 2003 Microsoft Corporation. All rights reserved.

Data Strings

The data string holds address information. The first part of a data string consists of a series of *keyname= value* elements separated by semicolons (;). You can include optional user data by putting a number sign (#) after the last value, followed by an application-defined string.

The key name is a lowercase string that identifies the data and implicitly indicates what type of data is contained in the value. For instance, the "provider" key name indicates that the value contains a Microsoft® DirectPlay® service provider globally unique identifier (GUID), in the form of a GUID string. The following characters are reserved and should not be used in value strings.

| Ampersand (&) | Forward slash (/) |
|----------------|-------------------|
| At sign (@) | Number sign (#) |
| Colon (:) | Question mark (?) |
| Equal sign (=) | Semicolon (;) |

The first element in the data string must be the provider. Other elements can follow in any order. A generic URL looks something like this.

```
x-directplay:/provider=Provider_GUID;[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=value1];[keyname1=v
```

The **Provider_GUID** should be of the form

{EBFE7BA0-628D-11D2-AE0F-006097B01411}

but using the escape characters for the invalid characters as shown in the following example.

```
%7BEBFE7BA0-628D-11D2-AE0F-006097B01411%7D
```

 $\ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.

Sample URLs

The following sample URLs illustrate what a Microsoft® DirectPlay® URL might look like for the four standard service providers. Each **GUID** expression should start with %7B and end with %7D, which are the escape characters for { and }.

Local IP Address

x-directplay:/provider=%7BEBFE7BA0-628D-11D2-AE0F-006097B

Local IPX Address

x-directplay:/provider=%7B53934290-628D-11D2-AE0F-006097B01

Local Serial Address

x-directplay:/provider=%7B743B5D60-628D-11D2-AE0F-006097B0

Remote Modem Address

x-directplay:/provider=%7B6D4A3650-628D-11D2-AE0F-006097B0

© 2003 Microsoft Corporation. All rights reserved.

Handling Addresses

If you call the **Host**, **EnumHosts**, or **Connect** methods exposed by the IDirectPlay8Peer, IDirectPlay8Client, or IDirectPlay8Server you must pass address objects as parameters. If Microsoft® DirectPlay® does not have sufficient address information, the method that you called will fail, and it will return DPNERR_ADDRESSING. However, it is not necessary to have all the information in the address object at the time you call the method.

All address objects must have the service provider globally unique identifier (GUID) set. However, it is possible to omit other data values.

- You can omit the device if the service provider supports all adapters.
- You can omit the port number for Internet Protocol (IP) and Internetwork Packet Exchange (IPX) service providers for the Host, EnumHosts, and Connect methods. DirectPlay will assign a port number. This number may vary.
- If you set the OKTOQUERYFORADDRESSING flag, the service provider can display a dialog box asking the user for the information needed to complete the address. If the user does not supply sufficient information, the method will fail. If the OKTOQUERYFORADDRESSING flag is not set, no dialog box will be displayed. If the address you pass to the method is insufficient, the method will fail. In the last two cases, the error value that is returned will be DPNERR_ADDRESSING.

There are two important issues for IP and IPX service providers that you need to be aware of. Failing to handle them properly may cause your application to fail.

• If you set the NOBROADCASTFALLBACK flag when you call an enumeration method, you must supply a hostname. If you do not

do so, the method will fail and return DPNERR_ADDRESSING.

• If you do not specify a port, do not assume that DirectPlay will always choose the same port number. The only way to be certain of the port number is to specify it in your address. If you do not specify a port number, you must retrieve the actual value later, after the command is in progress.

Note Application developers who choose to override the default DirectPlay 8 dialog for Transmission Control Protocol/Internet Protocol (TCP/IP) are strongly urged to implement a solution that allows the user to override the port used for a connection or enumeration. One possible solution is to enable users to follow the host name with a colon and then the port, as implemented by the default DirectPlay 8 TCP/IP protocol dialog, for example: host.domain.com:8090. Another possible solution is to add a field to the user interface (UI) that enables the user to enter a port.

© 2003 Microsoft Corporation. All rights reserved.

DirectPlay Address Objects

Microsoft® DirectPlay® does not handle URL strings directly. Instead, the string must be encapsulated in a DirectPlay address object (CLSID_DirectPlay8Address). This object exposes the IDirectPlay8Address interface that enables you to insert URL information into, or extract it from, the address object.

To create DirectPlay address, you must call **CoCreateInstance** to create a DirectPlay address object. You can then define the address in one of two ways:

- Create the URL string directly. Then use either <u>IDirectPlay8Address::BuildFromURLA</u> or <u>IDirectPlay8Address::BuildFromURLW</u> to insert the complete string.
- Use **IDirectPlay8Address** methods to insert the various pieces of data that make up the string directly into the object. For details see:
 - Creating TCP/IP Address Objects
 - Creating Modem Address Objects
 - Creating IPX Address Objects
 - Creating Serial Address Objects

When you receive an address object, you have a similar pair of options.

- Extract the entire URL string with either <u>IDirectPlay8Address::GetURLA</u> or <u>IDirectPlay8Address::GetURLW</u>. Then parse the string and extract the needed information
- Use other **IDirectPlay8Address** methods to extract the particular data you are interested in from the address object.

 $\ensuremath{\textcircled{\sc 0}}$ 2003 Microsoft Corporation. All rights reserved.

Creating TCP/IP Address Objects

This topic discusses how to create typical Address objects using the <u>IDirectPlay8Address</u> methods for Transmission Control Protocol/Internet Protocol (TCP/IP) service providers.

The first step in creating an address object is to call **CoCreateInstance** to create an **IDirectPlay8Address** object. The parameters include the class identifier of an address object (CLSID_DirectPlay8Address), the identifier of the interface (IID_IDirectPlay8Address), and the address of a pointer to an **IDirectPlay8Address** interface. The following example illustrates how to create an address object.

IDirectPlay8Address* g_pDeviceAddress;

For more information about using **CoCreateInstance**, see <u>Creating a</u> <u>COM Object</u>.

After creating the address object, you must set the service provider component, at a minimum. To do that, call <u>IDirectPlay8Address::SetSP</u>. The following example illustrates how to set the service provider to the Microsoft® DirectPlay® TCP/IP protocol service provider.

```
hr = g_pDeviceAddress->SetSP(&CLSID_DP8SP_TCPIP );
```

The **EnumHosts**, **Connect**, and **Host** methods each have an OKTOQUERYFORADDRESSING flag (DPNENUMHOSTS_OKTOQUERYFORADDRESSING, DPNCONNECT_OKTOQUERYFORADDRESSING, and DPNHOST_OKTOQUERYFORADDRESSING) that allow DirectPlay to prompt the user for any missing information in the address beyond the service provider component. However, for most applications you will want to override these standard dialogs to improve the user experience. The following sections describe the common and required components used for the TCP/IP protocol service providers.

Device Addresses

If you are creating a device address for the local player and using TCP/IP protocol as your service provider, you may also want to set the device port. For a list of other data values you may set, see <u>Data Value</u> <u>Summary</u>.

The following example illustrates how to set the port for a device address using the <u>IDirectPlay8Address::AddComponent</u> method.

DWORD dwPort; // Set dwPort to the port number for the device address.

If you set the port for the player hosting the session, it is recommended that you also set the DPNSESSION_NODPNSVR flag in the DPN_APPLICATION_DESC structure passed in the Host method. However, if you disable DPNSVR, you should do one of the following:

- Specify the host port in the *pdpaddrHost* parameter when you call **EnumHosts**.
- Specify the host port in the *pHostAddr* parameter when you call **Connect**.

For more information about DPNSVR, see <u>Using the DirectPlay DPNSVR</u> <u>Application</u>.

Host Addresses

If you are creating a host address and using TCP/IP protocol as your service provider, you may need to set the host name address component. For a list of other data values you may set, see **Data Value Summary**. If you did not use **EnumHosts** to find a session and do not pass the DPNCONNECT_OKTOQUERYFORADDRESSING flag to **Connect**, the host name must be specified for the host address.

The following example illustrates how to set the host name for a host address using the **IDirectPlay8Address::AddComponent** method.

In this example, wstrIP contains the Internet Protocol (IP) address or name of the session host in dotted notation, that is, the string "123.123.123.123" or "DirectPlayMaze.rte.microsoft.com".

Similarly, you may need to set the port for the host address. If you set the port component of the device address for the player calling **Host**, it is recommended that you set the port component of the host address passed in **EnumHosts** or **Connect**. If you did not use **EnumHosts** to find a session's host address and do not pass the

DPNCONNECT_OKTOQUERYFORADDRESSING flag, the port must be specified for the host address in order to connect to the session.

The following example illustrates how to set the port for a host address using the **IDirectPlay8Address::AddComponent** method.

DWORD dwPort; // Set dwPort to the port number of the host address

hr = g_pHostAddress->AddComponent(DPNA_KEY_PORT, // I &dwPort, sizeof(dwPort), // lpvData, dwDataSize DPNA_DATATYPE_DWORD); // dwDataType

After you have created the address objects, you connect to the session by passing the device address in the *pDeviceInfo* parameter and the host address in the *pHostAddr* parameter in the **Connect** method.

© 2003 Microsoft Corporation. All rights reserved.

Creating Modem Address Objects

This topic discusses how to create typical address objects using the <u>IDirectPlay8Address</u> methods for modem service providers.

The first step in creating an address object is to call **CoCreateInstance** to create an **IDirectPlay8Address** object. The parameters include the class identifier of an address object (CLSID_DirectPlay8Address), the identifier of the interface (IID_IDirectPlay8Address), and the address of a pointer to an **IDirectPlay8Address** interface. The following example illustrates how to create an address object.

For more information about using **CoCreateInstance**, see <u>Creating a</u> <u>COM Object</u>.

After creating the address object, you must set the service provider component, at a minimum. To do that, call <u>IDirectPlay8Address::SetSP</u>. The following example illustrates how to set the service provider to the Microsoft® DirectPlay® modem service provider.

```
hr = g_pDeviceAddress->SetSP(&CLSID_DP8SP_MODEM );
```

The **EnumHosts**, **Connect**, and **Host** methods each have an OKTOQUERYFORADDRESSING flag (DPNENUMHOSTS_OKTOQUERYFORADDRESSING, DPNCONNECT_OKTOQUERYFORADDRESSING, and DPNHOST_OKTOQUERYFORADDRESSING) that allow DirectPlay to prompt the user for any missing information in the address beyond the service provider component. However, for most applications you will want to override these standard dialogs to improve the user experience. The following sections describe the common and required components used for the modem service providers.
Device Addresses

If you are creating a device address for the local player and using the modem service provider, you may need to set the device. For a list of other data values you may set, see <u>Data Value Summary</u>. If you do not pass the respective OKTOQUERYFORADDRESSING flag to **EnumHosts**, **Connect**, or **Host**, the device must be specified for the device address.

Use the **EnumServiceProviders** method to obtain a list of available devices. The following example illustrates how to retrieve a list of available modem devices by calling the **EnumServiceProviders** method with the *pguidServiceProvider* parameter set to CLSID_DP8SP_MODEM.

DWORD dwSize; DWORD dwItems; PDPN_SERVICE_PROVIDER_INFO pSPInfoBuffer;

The *pSPInfoBuffer* parameter is the address of the buffer where DirectPlay places the service provider device information if the call is successful. Typically **EnumServiceProviders** is called twice, once to retrieve the buffer size required, and again to actually fill in the buffer.

After enumerating the available devices, select a modern device by using the <u>IDirectPlay8Address::AddComponent</u> method as is shown below.

GUID pGuid;

// Set pGuid to the GUID of the modem device.

In this example, *pGuid* is the address of the globally unique identifier (GUID) that represents the selected modem device.

Host Addresses

If you are creating a host address and using the modem service provider, you may need to set the phone number. For a list of other data values you may set, see **Data Value Summary**. If you do not pass the respective OKTOQUERYFORADDRESSING flag to **EnumHosts** or **Connect**, the phone number must be specified for the host address.

The following example illustrates how to set the phone number for a host address using the **IDirectPlay8Address::AddComponent** method.

In this example, *wstrPhone* contains the phone number of the host to which you will connect.

After you have created the address objects, you connect to the session by passing the device address in the *pDeviceInfo* parameter and the host address in the *pHostAddr* parameter in the **Connect** method.

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

Creating IPX Address Objects

This topic discusses how to create typical address objects using the <u>IDirectPlay8Address</u> methods for Internetwork Packet Exchange (IPX) service providers.

The first step in creating an address object is to call **CoCreateInstance** to create an **IDirectPlay8Address** object. The parameters include the class identifier of an address object (CLSID_DirectPlay8Address), the identifier of the interface (IID_IDirectPlay8Address) and the address of a pointer to an **IDirectPlay8Address** interface. The following example illustrates how to create an address object.

```
IDirectPlay8Address* g_pDeviceAddress;

.

.

.

hr = CoCreateInstance( CLSID_DirectPlay8Address, NULL,

CLSCTX_INPROC_SERVER,

IID_IDirectPlay8Address,

(LPVOID*) &g_pDeviceAddress);
```

For more information about using **CoCreateInstance**, see <u>Creating a</u> <u>COM Object</u>.

After creating the address object, you must set the service provider component, at a minimum. To do that, call <u>IDirectPlay8Address::SetSP</u>. The following example illustrates how to set the service provider to the Microsoft® DirectPlay® IPX service provider.

```
hr = g_pDeviceAddress->SetSP(&CLSID_DP8SP_IPX );
```

The **EnumHosts**, **Connect**, and **Host** methods each have an OKTOQUERYFORADDRESSING flag (DPNENUMHOSTS_OKTOQUERYFORADDRESSING, DPNCONNECT_OKTOQUERYFORADDRESSING, and DPNHOST_OKTOQUERYFORADDRESSING) that allow DirectPlay to prompt the user for any missing information in the address beyond the service provider component. However, for most applications you will want to override these standard dialogs to improve the user experience. The following sections describe the common and required components used for the IPX service providers.

Device Addresses

If you are creating a device address for the local player and using IPX as your service provider, you may also want to set the device port. For a list of other data values you may set, see <u>Data Value Summary</u>.

The following example illustrates how to set the port for a device address using the <u>IDirectPlay8Address::AddComponent</u> method.

DWORD dwPort; // Set dwPort to the port number for the device address.

If you set the port for the player hosting the session, it is recommended that you also set the DPNSESSION_NODPNSVR flag in the <u>DPN_APPLICATION_DESC</u> structure passed in the **Host** method. However, if you disable DPNSVR, you should do one of the following:

- Specify the host port in the *pdpaddrHost* parameter when you call **EnumHosts**.
- Specify the host port in the *pHostAddr* parameter when you call **Connect**.

Host Addresses

If you are creating a host address and using IPX as your service provider, you may need to set the host name address component. For a list of other data values you may set, see **Data Value Summary**. If you did not use **EnumHosts** to find a session, the host name must be specified for the host address.

The following example illustrates how to set the host name for a host address using the **IDirectPlay8Address::AddComponent** method.

In this example, pwszHost contains the IPX network and node number of the session host. If you were to use (hexadecimal) network 2702 and node 00-02-B3-10-87-64, the host name string would be "00002702,0002B3108764".

Similarly, you may need to set the port for the host address. If you set the port for the device address of the player who is hosting, it is recommended that you set the port for the host address passed in **EnumHosts** or **Connect**. If you did not use **EnumHosts** to find a session's host address, the port must be specified for the host address in

order to connect to the session.

The following example illustrates how to set the port for a host address using the **IDirectPlay8Address::AddComponent** method.

DWORD dwPort; //Set dwPort to the port number of the host address

hr = g_pHostAddress->AddComponent(DPNA_KEY_PORT, //
&dwPort, sizeof(dwPort), //lpvData, dwDataSiz
DPNA_DATATYPE_DWORD); //dwDataTyp

After you have created the address objects, you connect to the session by passing the device address in the *pDeviceInfo* parameter and the host address in the *pHostAddr* parameter in the **Connect** method.

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

Creating Serial Address Objects

This topic discusses how to create typical address objects using the <u>IDirectPlay8Address</u> methods for serial service providers.

The first step in creating an address object is to call **CoCreateInstance** to create an **IDirectPlay8Address** object. The parameters include the class identifier of an address object (CLSID_DirectPlay8Address), the identifier of the interface (IID_IDirectPlay8Address), and the address of a pointer to an **IDirectPlay8Address** interface. The following example illustrates how to create an address object.

For more information about using **CoCreateInstance**, see <u>Creating a</u> <u>COM Object</u>.

After creating the address object, you must set the service provider component, at a minimum. To do that, call <u>IDirectPlay8Address::SetSP</u>. The following example illustrates how to set the service provider to the Microsoft® DirectPlay® serial service provider.

```
hr = g_pDeviceAddress->SetSP(&CLSID_DP8SP_SERIAL );
```

The **EnumHosts**, **Connect**, and **Host** methods each have an OKTOQUERYFORADDRESSING flag (DPNENUMHOSTS_OKTOQUERYFORADDRESSING, DPNCONNECT_OKTOQUERYFORADDRESSING, and DPNHOST_OKTOQUERYFORADDRESSING) that allow DirectPlay to prompt the user for any missing information in the address beyond the service provider component. However, for most applications you will want to override these standard dialogs to improve the user experience. The following sections describe the common and required components used for the serial service providers.

Device Addresses

If you are creating a device address for the local player and using the serial service provider, you need to set the following components.

- baud
- stop bits
- parity
- flow control
- device (depends on your setup)

For a list of other data values you may set, see <u>Data Value Summary</u>.

Baud

The following example illustrates how to set the baud rate using the <u>IDirectPlay8Address::AddComponent</u> method.

DWORD dwBaudRate; // Set dwBaudRate

hr = g_pHostAddress->AddComponent(DPNA_KEY_BAUD, &dwBaudRate, sizeof(dwBaudRate), //lpvData, DPNA_DATATYPE_DWORD); //dwDat

In this example, the *dwBaudRate* variable can contain any of the DPNA_BAUD_RATE constants defined in dpaddr.h.

Stop bits

The following example illustrates how to set the stop bits using the **IDirectPlay8Address::AddComponent** method.

```
size_t cb;
hr = StringCbLengthW(DPNA_STOP_BITS_ONE, MAX_PATH, &c
```

```
hr = g_pHostAddress->AddComponent( DPNA_KEY_STOPBITS,
DPNA_STOP_BITS_ONE, //lpvData
cb, //dwDataSize, in bytes
DPNA_DATATYPE_STRING ); //dwDa
```

In this example, the stop bits setting is 1, as indicated by the use of the DPNA_STOP_BITS_ONE constant defined in dpaddr.h.

Parity

The following example illustrates how to set the parity using the **IDirectPlay8Address::AddComponent** method.

```
size_t cb;
hr = StringCbLengh(DPNA_PARITY_NONE, MAX_PATH, &cb);
hr = g_pHostAddress->AddComponent(DPNA_KEY_PARITY,
DPNA_PARITY_NONE, //lpvData
cb, //dwDataSize, in bytes
DPNA_DATATYPE_STRING ); //dwDat
```

In this example, the parity setting is none, as indicated by the use of the DPNA_PARITY_NONE constant defined in dpaddr.h.

Flow Control

The following example illustrates how to set the flow control using the **IDirectPlay8Address::AddComponent** method.

size_t cb;

hr = StringCbLengh(DPNA_FLOW_CONTROL_RTSDTR, MAX_P/

hr = g_pHostAddress->AddComponent(DPNA_KEY_FLOWCONTR DPNA_FLOW_CONTROL_RTSDTR, cb, //dwDataSize, in bytes DPNA_DATATYPE_STRING); //dw

In this example, the flow control is set to RTS/DTR, as indicated by the use of the DPNA_FLOW_CONTROL_RTSDTR constant defined in dpaddr.h.

Device

You may need to set the device if you do not pass the respective OKTOQUERYFORADDRESSING flag to **EnumHosts**, **Connect**, or **Host**.

Use the **EnumServiceProviders** method to obtain a list of available devices. The following example illustrates how to retrieve a list of available serial devices by calling the **EnumServiceProviders** method with the *pguidServiceProvider* parameter set to CLSID_DP8SP_SERIAL.

```
DWORD dwSize;
DWORD dwItems;
PDPN_SERVICE_PROVIDER_INFO pSPInfoBuffer;
```

The *pSPInfoBuffer* parameter is the address of the buffer where DirectPlay places the service provider device information if the call is successful. Typically **EnumServiceProviders** is called twice, once to retrieve the buffer size required, and again to actually fill in the buffer.

After enumerating the available devices, select a serial device by using the **IDirectPlay8Address::AddComponent** method as is shown below.

```
GUID pGuid;
// Set pGuid
hr = g_pDeviceAddress->AddComponent( DPNA_KEY_DEVICE,
pGuid, sizeof(GUID), //lpvData, dwDataSize
DPNA_DATATYPE_GUID); //dwDataType
```

In this example, *pGuid* is the address of the globally unique identifier (GUID) that represents the selected serial device.

Host Addresses

Serial host addresses do not need to contain any additional information beyond the serial provider component.

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

Data Value Summary

The following two tables outline the standard data values and indicate which values are used by each type of service provider for both host and device addresses. For more information about how to create address objects, see <u>DirectPlay Address Objects</u>.

Host Addresses

| | IP | IPX | Serial | Modem |
|----------------------|----------|----------|----------|----------|
| Application Instance | Optional | Optional | Optional | Optional |
| Baud | Not used | Not used | Required | |
| <u>Device</u> | Not used | Optional | Required | Required |
| Flow Control | Not used | Not used | Required | |
| <u>Host Name</u> | Required | Required | Optional | Optional |
| <u>Parity</u> | Not used | Not used | Required | |
| Phone Number | Not used | Not used | Not used | Required |
| Port | Required | Required | Not used | Not used |
| <u>Program</u> | Optional | Optional | Optional | Optional |
| <u>Provider</u> | Required | Required | Required | Required |
| Stop Bits | Not used | Not used | Required | |

Device Addresses

| | IP | IPX | Serial | Modem |
|----------------------|----------|----------|----------|----------|
| Application Instance | Optional | Optional | Optional | Optional |
| Baud | Not used | Not used | Required | Not used |
| Flow Control | Not used | Not used | Required | Not used |
| <u>Host Name</u> | Optional | Optional | Optional | Optional |
| <u>Device</u> | Optional | Required | Required | Required |
| <u>Parity</u> | Not used | Not used | Required | Not used |
| Phone Number | Not used | Not used | Not used | Not used |
| Port | Optional | Required | Not used | Not used |
| <u>Program</u> | Optional | Optional | Optional | Optional |
| <u>Provider</u> | Required | Required | Required | Required |
| Stop Bits | Not used | Not used | Required | Not used |

 $\ensuremath{\textcircled{\sc 0}}$ 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

Data Values

The values that need to be included in the data string depend on the particular service provider. Modem providers, for instance, need a telephone number in their address, whereas local area network (LAN) providers might need a port number. This section provides a detailed description of the standard data values. It also includes a key name that can be used in place of the literal string. These names are defined in Dpaddr.h. For more information about using these values, see <u>DirectPlay</u> <u>Address Objects</u>.

- <u>Application Instance</u>
- <u>Baud</u>
- <u>Device</u>
- Flow Control
- Host Name
- Parity
- Phone Number
- Port
- Program
- <u>Provider</u>
- Stop Bits
- <u>NAT Resolver</u>
- <u>NAT Resolver User String</u>
- <u>Name Information</u>
- Processor
- <u>Scope</u>
- Traversal Mode

Application Instance

An optional globally unique identifier (GUID) that identifies an application instance. This value is used when specifying the game that is to be connected to.

Key Name: DPNA_KEY_APPLICATION_INSTANCE

Key String: "applicationinstance"

Data Type: GUID

Providers: All

Valid Values: Any valid application instance GUID.

Baud

The baud rate.

Key Name: DPNA_KEY_BAUD

Key String: "baud"

Data Type: **DWORD**

Providers: Modem and serial

Valid Values: Any valid baud rate. You can set this value to the appropriate integer, or you can use one of the following predefined values from Dpaddr.h.

DPNA_BAUD_RATE_9600 DPNA_BAUD_RATE_14400 DPNA_BAUD_RATE_19200 DPNA_BAUD_RATE_38400 DPNA_BAUD_RATE_56000 DPNA_BAUD_RATE_57600 DPNA_BAUD_RATE_115200

Device

A GUID that identifies the device on the local computer that will be used. If the service provider supports all adapters, you do not need to specify a device.

Key Name: DPNA_KEY_DEVICE

Key String: "device"

Data Type: GUID

Providers: All, but for device addresses only, not host addresses

Valid Values: Any valid device GUID.

Flow Control

The type of flow control to be used.

Key Name: DPNA_KEY_FLOWCONTROL

Key String: "flowcontrol"

Data Type: String

Providers: Serial and modem

Valid Values: Any of the following predefined values from Dpaddr.h.

DPNA_FLOW_CONTROL_NONE DPNA_FLOW_CONTROL_DTR DPNA_FLOW_CONTROL_XONXOFF DPNA_FLOW_CONTROL_RTSDTR DPNA_FLOW_CONTROL_RTS

Host Name

The name of a remote host computer.

Key Name: DPNA_KEY_HOSTNAME

Key String: "hostname"

Data Type: String

Providers: All, but for host addresses only, not device addresses

Valid Values: A fully-qualified host name, or a dotted address.

Parity

The parity of the connection.

Key Name: DPNA_KEY_PARITY

Key String: "parity"

Data Type: String

Providers: Serial and modem

Valid Values: Any of the following predefined values from Dpaddr.h.

DPNA_PARITY_NONE DPNA_PARITY_MARK DPNA_PARITY_EVEN DPNA_PARITY_SPACE DPNA_PARITY_ODD

Phone Number

A phone number.

Key Name: DPNA_KEY_PHONENUMBER

Key String: "phonenumber"

Data Type: String

Providers: Modem

Valid Values: Any valid phone number.

Port

An optional port number. For more information, see <u>DirectPlay and Ports</u>.

Key Name: DPNA_KEY_PORT

Key String: "port"

Data Type: **DWORD**

Providers: Internet Protocol (IP) and Internetwork Packet Exchange (IPX)

Valid Values: Any 16-bit integer. Only the lower 16 bits of the value are valid. If you do not specify a port, Microsoft® DirectPlay® will choose one for you.

Program

An optional application GUID.

Key Name: DPNA_KEY_PROGRAM

Key String: "program"

Data Type: GUID

Providers: All

Valid Values: Any valid application GUID.

Provider

A GUID that identifies the DirectPlay service provider that will be used. For more information, see <u>DirectPlay Service Providers</u>.

Key Name: DPNA_KEY_PROVIDER

Key String: "provider"

Data Type: GUID

Providers: All

Valid Values: Any valid service provider GUID.

Stop Bits

The number of stop bits.

Key Name: DPNA_KEY_STOPBITS

Key String: "stopbits"

Data Type: String

Providers: Serial and modem

Valid Values: Any of the following predefined values from Dpaddr.h.

DPNA_STOP_BITS_ONE DPNA_STOP_BITS_TWO DPNA_STOP_BITS_ONE_FIVE

NAT Resolver

A name of a <u>IDirectPlay8NATResolver</u> server. See <u>Network Address</u> <u>Translation, Firewalls, and Proxies</u> for more information.

Key Name: DPNA_KEY_NAT_RESOLVER

Key String: "natresolver"

Data Type: String

Providers: All, but for device addresses only, not host addresses

Valid Values: A fully-qualified name, or a dotted address.

NAT Resolver User String

A password to allow access to the **IDirectPlay8NATResolver** server. See **Network Address Translation, Firewalls, and Proxies** for more information.

Key Name: DPNA_KEY_NAT_RESOLVER_USER_STRING

Key String: "natresolveruserstring"

Data Type: String

Providers: All, but for device addresses only, not host addresses

Valid Values: Any valid password.
Name Information

The name of a Microsoft Windows® Powered Pocket PC 2002 device. See <u>Bluetooth Service Provider</u> for more information.

Key Name: DPNA_KEY_NAMEINFO

Key String: "nameinfo"

Data Type: String

Providers: None. This component is only for information returned from an enumeration. To connect to the device, use the DPNA_KEY_HOSTNAME component.

Valid Values: A friendly name.

Processor

The processor number.

Key Name: DPNA_KEY_PROCESSOR

Key String: "processor"

Data Type: String

Providers: All

Valid Values: A valid processor number.

Scope

Not implemented.

Traversal Mode

Enable or disable traversal mode. See **Network Address Translation**, **Firewalls, and Proxies** for more information.

Key Name: DPNA_KEY_TRAVERSALMODE

Key String: "traversalmode"

Data Type: String

Providers: All

Valid Values: Any of the following predefined values from Dpaddr.h.

DPNA_TRAVERSALMODE_PORTRECOMMENDED DPNA_TRAVERSALMODE_PORTREQUIRED DPNA_TRAVERSALMODE_NONE

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

DirectPlay Callback Functions and Multithreading Issues

Microsoft® DirectPlay® and DirectPlay Voice both require you to implement and register several callback functions to handle the events fired by DirectPlay. If you choose to use multiple DirectPlay threads in your game, it is possible that your application will receive multiple overlapping callbacks. Alternatively, you can use DirectPlay's DoWork mode to avoid multithreading issues.

DirectPlay DoWork Mode

DirectPlay offers a single-threaded environment so that application developers don't have to worry about data corruption or deadlocking due to synchronization problems. Applications simply set the DirectPlay thread count to zero by calling the

IDirectPlay8ThreadPool::SetThreadCount method. Then call IDirectPlay8ThreadPool::DoWork within the game loop and DirectPlay will perform all DirectPlay tasks during the time period specified. For more instructions about creating a single-threaded application, see <u>Tutorial 10: DirectPlay Thread Pool</u>.

DirectPlay Thread Pool

For most large-scale, multiplayer networked applications, you will want to implement multithreading because it enables greater scalability. For this, DirectPlay maintains a thread pool, controlled with an IDirectPlay8ThreadPool object. Your callback is invoked on a thread from this pool. The size of this thread pool is configurable on a per-process basis by using the IDirectPlay8ThreadPool::SetThreadCount method.

To correctly and reliably access data in DirectPlay callbacks, you are required to implement a method of multithreading synchronization. This is known as making your callback *re-entrant* or *threadsafe*.

The Microsoft Windows® family of operating systems currently offers three methods of synchronizing data in multithreaded environments:

- Mutex objects (mutually exclusive synchronization objects).
- Semaphore objects (flag variables used to indicate to potential users that a shared file or other resource is in use).
- Critical section objects (also provide mutually exclusive synchronization but used only by the threads of a single process).

The DirectPlay voice samples that ship with the Microsoft DirectX® 9.0 software development kit (SDK) demonstrate synchronization using critical section objects. If you want to implement a mutex or semaphore object, read about these topics in the Microsoft Platform Software Development Kit (SDK) as well as in many reference books. Implementing any of these synchronization methods requires an expert knowledge level in these areas due to the level of complexity and difficulty in debugging should any issues arise.

The DirectPlay threading model is optimized for maximum efficiency and

there are no thread context switches during "indication" messages, including receive messages.

See <u>Implementing a Callback Function in DirectPlay and DirectPlay Voice</u> for more information.

DirectPlay Networking Callbacks

DirectPlay networking callback functions are of type <u>PFNDPNMESSAGEHANDLER</u>. Depending on the type of networking session, you register the address of your callback function with <u>IDirectPlay8Peer::Initialize</u>, <u>IDirectPlay8Client::Initialize</u>, or <u>IDirectPlay8Server::Initialize</u>.

Synchronization Issues

You must employ one of the three thread synchronization objects in order to maintain the integrity of your game data during processing in a DirectPlay callback.

To understand how your game data could be corrupted, consider that your callback inserts a packet of game data into a structure. Because the callback is reentrant, another thread can enter the callback before the first callback has completed. It is possible that this second thread could also attempt to access the structure at the same location in memory and change the data. Therefore, the data placed in the structure by the first thread is overwritten by the data placed in the structure by the second thread. Please note that this is an oversimplified example of multithreading and there are many other implications to not properly synchronizing multiple threads.

See <u>Implementing a DirectPlay Networking Callback Using Critical</u> <u>Section Objects</u> for an example of how to synchronize data in a DirectPlay networking session.

Worker Threads

You have the option of creating your own "worker threads". A worker

thread is another multithreaded application defined callback that is created to process game data independently of the DirectPlay callbacks. The most common way of accomplishing this is to buffer data received during a DirectPlay networking callback thread. Then, a new thread is created and a message is sent to your worker thread callback to notify it to process the buffered data.

Multithreading Performance Issues and Asynchronous Operations

It is important to carefully consider how much time is spent processing messages in DirectPlay callbacks. If you process a lot of data within the DirectPlay callbacks and you employ a data locking mechanism to synchronize threads, you will run into blocking problems as other threads wait to enter the callback.

If you choose to implement a worker thread and offset the processing of game data to another callback, you run the risk of adding a lot of overhead processing time as the CPU switches context between the threads you create and the threads created by DirectPlay. This should be done only if the game data requires a large amount of processing time, and the data is not critical to the real time operation of the game. For example, it is not recommended to process player location data in a worker thread because this data is critical to positioning players in real time within the game.

You can also return **DPNSUCCESS_PENDING** from the callback, create a pointer to the data buffer, and make that pointer available the worker thread. When the worker thread is finished processing the game data, it calls the ReturnBuffer method of either <u>IDirectPlay8Peer</u>, <u>IDirectPlay8Client</u>, or <u>IDirectPlay8Server</u>, depending on the topology used.

Holding Locks Across API Calls

In general, you should avoid holding shared resource locks across application programming interface (API) calls. This is because it can be hard to envision all the possible interactions with other threads. In the following code, the sending thread is incorrectly holding the **pObj**-**>csSomeLock** critical section while calling <u>IDirectPlay8Peer::SendTo</u> synchronously.

```
typedef struct _MYOBJECT{
    CRITICAL_SECTION csSomeLock;
    DWORD dwFlags;
    .
    .
    .
    .
    MYOBJECT, *PMYOBJECT;
IDirectPlay8Peer *pDP8Peer;
PMYOBJECT pObj;
.
.
EnterCriticalSection(&pObj->csSomeLock);
pDP8Peer->SendTo(DPNID_ALL_PLAYERS_GROUP,
        &dpnBuffer, 1, 0,
        NULL, NULL, DPNSEND_SYNC);
LeaveCriticalSection(&pObj->csSomeLock);
```

The local player will receive a copy of the message with a call to the

application's message handler on a different thread because the DPNSEND_NOLOOPBACK flag was not used. If the message handler tried to acquire **pObj->csSomeLock** in response to this message, it would deadlock, because the sending thread cannot return from **IDirectPlay8Peer::SendTo** (and thus cannot drop the lock) until the message handler returns, but the message handler can't return until the sending thread drops the lock. Instead, use a flag or indexing system so that you can release the lock while you make the API call.

```
typedef struct _MYOBJECT{
    CRITICAL_SECTION csSomeLock;
                   dwFlags;
    DWORD
} MYOBJECT, *PMYOBJECT;
IDirectPlay8Peer
                *pDP8Peer;
PMYOBJECT
                 pObj;
EnterCriticalSection(&pObj->csSomeLock);
pObj->dwFlags |= FLAGS_SENDING;
LeaveCriticalSection(&pObj->csSomeLock);
pDP8Peer->SendTo(DPNID_ALL_PLAYERS_GROUP,
        &dpnBuffer, 1, 0,
        NULL, NULL, DPNSEND_SYNC);
EnterCriticalSection(&pObj->csSomeLock);
pObj->dwFlags &= ~FLAGS SENDING;
```

LeaveCriticalSection(&pObj->csSomeLock);

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

Implementing a Callback Function in DirectPlay and DirectPlay Voice

Microsoft® DirectPlay® and DirectPlay Voice both require you to implement and register several callback functions to handle the events raised by DirectPlay. DirectPlay is multithreaded and will raise multiple events concurrently. Therefore, in order to correctly and reliably access data in DirectPlay callbacks, you are required to implement a method of multithreading synchronization. This is known as making your callback *re-entrant* or *threadsafe*.

Callback Function Structure

The structure of the callback follows standard Microsoft Win32® application programming interface (API) programming guidelines.

HRESULT WINAPI Callback(PVOID pvUserContext, DWORD dwMessageType, PVOID pMessage);

pvUserContext is the a context value you supply when you register the callback function with DirectPlay. If you pass this value to DirectPlay when you register your callback, the context value will be returned when DirectPlay invokes your callback.

dwMessageType is one of the identifier (ID) values passed to your callback by DirectPlay.

pMessage will contain the message passed by DirectPlay.

Registering Your Callback

DirectPlay networking callback functions are of type <u>PFNDPNMESSAGEHANDLER</u>. Depending on the type of networking session, you register the address of your callback function with <u>IDirectPlay8Peer::Initialize</u>, <u>IDirectPlay8Client::Initialize</u>, or <u>IDirectPlay8Server::Initialize</u>. If you are registering a DirectPlay voice callback function, register the address of your callback with <u>IDirectPlayVoiceClient::Initialize</u> or <u>IDirectPlayVoiceServer::Initialize</u>, depending on the type of DirectPlay voice session you want to create.

The following code snippet demonstrates how to register a callback function with the <u>IDirectPlay8Peer</u> interface.

HRESULT WINAPI Callback(PVOID, DWORD, PVOID);

IDirectPlay8Peer* pdp8Peer;

```
// Get the server interface
hr = CoCreateInstance( CLSID_DirectPlay8Peer, ...)
...
```

// Register the callback
hr = pdp8Peer->Initialize(NULL, Callback, 0);

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

Implementing a DirectPlay Networking Callback Using Critical Section Objects

Microsoft® DirectPlay® networking and voice callback are multithreaded. Therefore, in order to correctly and reliably access data in DirectPlay callbacks, you are required to implement a method of multithreading synchronization.

Currently, there are three methods of synchronizing mulithreaded callback data.

- Mutex objects
- Semaphore objects
- Critical section objects

The DirectPlay voice samples that ship with the Microsoft DirectX® SDK demonstrate synchronization using critical section objects, and the following topics will also demonstrate how critical section objects are used. If you want to implement a mutex or semaphore object, these topics are discussed in the Microsoft Platform Software Development Kit (SDK) as well as in many reference books. Implementing any of these synchronization methods requires an expert knowledge level in these areas due of the level of complexity and difficulty in debugging should any issues arise.

CRITICAL_SECTION g_csPlayerContext; InitializeCriticalSectionAndSpinCount(&g_csPlayerContext, 0);

Next, implement the DirectPlay message callback handler.

HRESULT WINAPI DirectPlayMessageHandler(PVOID pvUserCon

```
DWORD dwMessageId,
PVOID pMsgBuffer )
{
    switch( dwMessageId )
    {
        case DPN_MSGID_CREATE_PLAYER:
        {
            EnterCriticalSection( &g_csPlayerContext );
            // callback is now locked
            // perform operation on player data
            LeaveCriticalSection( &g_csPlayerContext );
        }
    }
}
```

Finally, during application exit, ensure that you call the **DeleteCriticalSection** function to free the memory associated with your critical section object.

DeleteCriticalSection(&g_csPlayerContext);

 $\ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

DirectPlay Protocol

Multiplayer games require efficient and flexible network messaging services for optimal performance. The Microsoft® DirectPlay® protocol is a transport-layer messaging protocol that is used for all DirectPlay messaging. It provides your application with the messaging support it needs to make everything run smoothly. The DirectPlay protocol includes the following messaging support.

- Reliable and unreliable delivery of messages
- Sequential and non-sequential delivery of messages
- Message fragmentation and reassembly
- Congestion control
- Send prioritization
- Message timeouts

Note Before DirectPlay 8.1, the DirectPlay protocol was optional, and had to be specified explicitly. Since DirectPlay 8.1, this protocol is used for all DirectPlay messaging.

This document provides a general description of how the Microsoft DirectX® protocol works, and how you can use it in your application.

- Basic Message Handling
- <u>Message Categories</u>
- <u>Congestion Control</u>
- <u>Send Prioritization</u>
- Packet Signing
- <u>Monitoring Messaging Statistics</u>
- <u>Monitoring the Pending Message Queues</u>
- <u>Using the DirectX Protocol in an Application</u>

 $\ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

Basic Message Handling

A message, as the term is used in this document, is a block of data that needs to be sent to another computer. A network protocol creates a *packet* by adding some bits to the data block that hold information such as the target's network address. This packet is the basic unit of network data. When the target receives the packet, the target's network protocol removes the extra bits and passes the data block to the receiving application.

Although similar in usage, the terms *message* and *packet* are not strictly interchangeable. This document uses the term *message* to refer to the unit of information that is passed to and received from the Microsoft® DirectPlay® application programming interface (API). *Packet* refers to the unit of information handled by the network. DirectPlay handles packets internally. With rare exceptions, DirectPlay applications need to deal only with messages.

The primary reason for the distinction between *message* and *packet* is that networks generally limit the maximum size of the packets they handle. This size is referred to as a Maximum Transmission Unit (MTU). If a message is small, it is sent in a single packet and the two terms are effectively synonymous. However, large messages might need to be fragmented into two or more packets and then be reassembled by the receiver. The DirectPlay protocol automatically handles fragmentation and reassembly of messages as needed. As far as your application is concerned, you send a message, and the target receives it.

Note DirectPlay delivers messages of any size. However, the more packets that are required for a single message, the greater the odds that one or more packets will be lost and have to be retransmitted. Messages

that are large enough to require fragmentation and reassembly thus typically have more network latency than single-packet messages. If you need to keep network latency to a minimum, avoid sending large messages, especially on lossy networks. You can determine the maximum size that your connection can accommodate in a single packet by calling the **GetSPCaps** method exposed by the <u>IDirectPlay8Peer</u>, <u>IDirectPlay8Client</u>, and <u>IDirectPlay8Server</u> interfaces.

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

Message Categories

The Microsoft® DirectPlay® protocol is designed to handle the following two basic types of network messaging.

- *Reliable* versus *unreliable* messaging determines whether messages are guaranteed to be delivered to the target application.
- *Non-sequential* versus *sequential* messaging determines whether messages are received by the target application in the same order they are sent.

Games use messaging for a variety of purposes, each with different demands. To support this range of messaging needs, the DirectPlay protocol enables you to designate a message as belonging to one of four categories:

- Reliable and sequential
- Unreliable and sequential
- Reliable and non-sequential
- Unreliable and non-sequential

The DirectPlay protocol enables you to optimize your messaging strategy by assigning categories on a message-by-message basis.

Reliable and Unreliable Messaging

Messages are sometimes lost in transit. *Reliable* messaging provides a guarantee that the target will receive every message. This type of messaging is required when data loss cannot be tolerated. Most reliable messaging schemes require the target to acknowledge receipt of each message. If the sender does not receive an acknowledgment within a specified timeout period, it resends the message. This process typically continues until the sender receives an acknowledgment, confirming that the message has arrived.

The DirectPlay protocol imposes a limit on the number of resend attempts. If no acknowledgment is received after a reasonable number of attempts, DirectPlay assumes that the connection has been lost, and closes it.

Unreliable messaging is the simplest form of network communication. It might be faster than reliable messaging because there is no guarantee that the message will be delivered to the target. The sender transmits the message. If the target does not receive the message, the sender will not transmit the message again, and the packet is lost.

Unreliable messaging is used primarily when speed or bandwidth is more important than an occasional lost message. For example, high-bandwidth streaming media applications often use unreliable messaging. They cannot afford to take up bandwidth with acknowledgments and retransmissions, nor can they wait for a lost message to be retransmitted. An occasional lost message normally has only a minor impact on quality, so it can be ignored.

Sequential and Non-Sequential Messaging

Messages leave the sender in a particular sequence. However, there is no guarantee that messages will arrive at the target's computer in the same order that they are sent. For example, if a message is lost and must be retransmitted, that message will typically arrive later than messages that followed it in the original sequence.

Sequential messaging uses sequencing information embedded in the message to ensure that the messages are presented to the target application in the correct order. This type of messaging is required when the target application must receive messages in the correct order. Out-of-order messages are buffered until the missing messages arrive.

Non-sequential messaging presents the received messages to the target as soon as they arrive at the target computer, regardless of the order in which they were sent. Because there is no need to wait for a missing packet, applications often use non-sequential messaging when speed is more important than an occasional out-of-order message. The out-oforder message is ignored.

Choosing the Best Message Category

Choosing the best category for messages is a core issue for multiplayer game developers. While DirectPlay provides the tools to manage your messaging, the choice of a message category ultimately depends on the semantic content of the message and the nature of the game.

The following are general guidelines for choosing the best message category.

- Use non-guaranteed messaging whenever the content permits. For example, your game might send frequent player-location updates. Each update is independent, and it supersedes any previous updates. If an update is lost, the next update is sufficient to maintain the player's game state. A lost and retransmitted message might arrive later than the subsequent update message.
- Use guaranteed messaging when data loss cannot be tolerated. For example, a text-based chat feature depends on every character being delivered to its target.
- Use sequential messaging when the order of the messages is important. For example, streaming media typically uses sequentialunreliable messaging. An occasional dropped message can be tolerated, but an out-of-order message would cause problems.

 $\ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

Congestion Control

In an ideal world, your game can send messages as often as it needs to. They arrive at the target immediately and are processed instantaneously. If all of the computers in your game have ample processing power and are connected by a lightly used high-bandwidth network link, you might approach this ideal situation. You can then send messages as often as you like. However, a number of factors can create *congestion* and cause messaging to work more slowly than this ideal:

- Network latency. Even under ideal conditions, messages take a finite time to traverse the network from sender to target, especially over the Internet. There might be further delays for acknowledgments, retransmission of lost packets, or reassembly of out-of-order packets.
- Network bandwidth. The network bandwidth controls the rate at which a message can be sent or received by a computer. Network links have a wide range of bandwidths, and even high-bandwidth networks might be slowed by high traffic levels. If one or more of your players has a low-bandwidth connection, they will be able to send and receive messages only at a limited rate.
- Processing speed. Even if network bandwidth is high and latency low, the target application still needs some time to process a received message. If one or more of the players in a session is using a relatively slow computer, the rate at which they can process received messages might be below the rate at which messages can be sent.

Message Throttling

If there is no control over the rate at which messages are sent, a target can be flooded by more messages than it can handle. To prevent this situation, the Microsoft® DirectPlay® protocol *throttles* the rate at which messages are sent. The net effect of throttling is that the rate at which messages are sent is controlled by the rate at which the target can handle them.

Throttling is implemented with a *sliding window* mechanism. The sliding window is basically a queue with a limited number of slots that holds messages that have been sent but not yet received. All outgoing messages are placed in this queue, regardless of their category. When the sent-message queue is full, it accepts no more outgoing messages until one of the messages in the queue has been received.

For optimal performance, the size of the sliding window must be matched to current network conditions. The DirectPlay protocol automatically monitors such factors as the number of messages and the total number of bytes in the sent-message queue. This information is then used to dynamically adjust the size of the sliding window to optimize messaging for the current network conditions.

Connection Checking

If there is no activity on a link, the DirectPlay protocol periodically tests the connection by sending an empty reliable packet. If no acknowledgment is received from the target after a reasonable number of attempts, DirectPlay concludes that the link has been disconnected.

© 2003 Microsoft Corporation. All rights reserved.
Send Prioritization

Messages often vary widely in importance. Some are time-critical and must be delivered as quickly as possible. Others can be delayed if necessary or, possibly, not sent at all. One issue with congestion control algorithms is that an application might create messages faster than they can be sent. Unsent messages must then be held in a queue until an outgoing slot opens up. If all unsent messages are held in a single pending-message queue, high priority messages might be blocked while waiting for lower priority messages to be sent first.

The Microsoft® DirectPlay® protocol solves this problem by having three pending message queues: low, medium, and high priority. When a slot opens up in the sent-message queue, the protocol selects the next message to be sent as follows:

- 1. Send the oldest message in the high-priority queue.
- 2. If there are no messages in the high-priority queue, send the oldest message in the medium-priority queue.
- 3. If there are no messages in the medium-priority queue, send the oldest message in the low-priority queue.

This priority mechanism enables you to get your time-critical messages out as quickly as possible, even when less important messages have already been submitted.

Note All DirectPlay Voice messages are sent with medium priority and a timeout value of 1.5 seconds. Any messages that should not be preempted by voice traffic should be sent with high priority.

Send Timeouts

One of the consequences of throttling is that messages might spend a relatively long time in a pending-message queue, especially if they are low priority. Some messages might stay long enough to have been superseded by subsequent messages. These messages are no longer relevant. For example, your application might periodically send player-location update messages. Each update is independent of the others and supersedes any previous updates. If you have two player-location updates in the pending message queue, only the most recent one needs to be sent.

The DirectPlay protocol enables you to handle the problem of outdated messages by adding an optional timeout value to the message. If the message is still in a pending-message queue when the timeout expires, the message will be canceled.

Disconnection

When an application sends a disconnect message, the message is placed at the end of the low-priority pending-message queue, and the protocol stops accepting outgoing messages. This practice guarantees that all pending messages are sent before the link is disconnected. The disconnect message is sent as a reliable sequential message to guarantee that it arrives, but not before all other messages in the queue have been delivered.

© 2003 Microsoft Corporation. All rights reserved.

Packet Signing

Microsoft® DirectPlay® provides packet signing to prevent malicious users from sending spoofed Internet Protocol (IP) packets that corrupt the link between peers or between clients and a server. Packet signing is performed by attaching a "signature" to each packet before sending. This signature is 8 bytes. Any packet received without the correct signature is silently ignored by DirectPlay.

Packet signing does not encrypt the data that is sent over the link, nor does it provide a form of authentication. It should be used to prevent third parties from inserting invalid data into a DirectPlay game session. Some other mechanism needs to be used if you want the application to authenticate the users connecting to the game.

Packet signing is set when the session is started by the host. The host determines the type of signing by setting either the DPNSESSION_FAST_SIGNED or DPNSESSION_FULL_SIGNED flag in the DPN_APPLICATION_DESC structure when calling IDirectPlay8Peer::Host or IDirectPlay8Server::Host. Players connecting to the session can set either flag when calling IDirectPlay8Peer::Connect or IDirectPlay8Client::Connect, but they will use whichever type of signing the host has selected. When connected, clients can determine which type of signing the session is using by calling

IDirectPlay8Peer::GetApplicationDesc or

IDirectPlay8Client::GetApplicationDesc. The dwFlags member of the DPN_APPLICATION_DESC structure will contain one of the signing flags.

Note All players must be using Microsoft DirectX® 9.0 or later in order to use packet signing in a session.

Fast Signing

Fast signing creates the lowest overhead on your game. It merely adds the same 8-byte value to each packet. Fast signing is recommended for all DirectPlay sessions created. However, it is vulnerable to a malicious attack if another user has access to the packets being exchanged and can determine the 8-byte value.

Full Signing

Full signing adds computational overhead to your game. Each packet sent and received has to be hashed over in order to compute its signature. Full signing uses the Secure Hash Algorithm version 1.0. When the session has been established, access to the packets does not enable another user to compromise the link. However, if another user reads the initial connection sequence between two computers, that user may be able to compromise the link.

© 2003 Microsoft Corporation. All rights reserved.

Monitoring Messaging Statistics

While the Microsoft® DirectPlay® protocol handles many aspects of messaging automatically, your application should still monitor messaging behavior. For example, if you are consistently sending messages faster than they can be delivered, you might need to modify your messaging scheme.

Because network conditions change continuously, your application should periodically check the behavior of the network and adjust its messaging scheme accordingly. To do so, call the **GetConnectionInfo** method that is exposed by every DirectPlay interface that supports messaging. **GetConnectionInfo** returns a structure that contains a wide variety of statistical information that you can use to refine your messaging scheme, including the following:

- Round trip latency
- Throughput
- Packets sent
- Packets received
- Packets resent
- Packets dropped
- Messages transmitted at different priority levels

Note The messaging statistics are obtained by monitoring the actual network traffic. If you call **GetConnectionInfo** immediately after you initialize the connection, there will have been little time to collect data and the statistics might be misleading.

© 2003 Microsoft Corporation. All rights reserved.

Monitoring the Pending Message Queues

You should monitor your pending message queues to ensure that they do not become too large. The <u>IDirectPlay8Peer</u>, <u>IDirectPlay8Client</u>, and <u>IDirectPlay8Server</u> interfaces all expose a **GetSendQueueInfo** method that can be used to check the number of messages and the number of bytes currently in the queue. By default, the method returns the total for all three queues, but you can also obtain values for each of the three priority levels.

© 2003 Microsoft Corporation. All rights reserved.

Using the DirectX Protocol in an Application

This section covers how you can use the features of the Microsoft® DirectPlay® protocol in your application.

You can use five DirectPlay interfaces to send messages.

- IDirectPlay8Peer
- IDirectPlay8Client
- IDirectPlay8Server
- IDirectPlay8LobbyClient
- IDirectPlay8LobbiedApplication

Depending on which interface your application is using to send messages, you send a message by calling a method named either **Send**, or **SendTo**. While the usage of these five methods is similar, they vary in detail. Refer to the appropriate reference pages for further discussion.

The **Send/Sendto** method's parameters might allow you to control many of the DirectPlay protocol's features. For example, the *dwFlags* field of <u>IDirectPlay8Peer::SendTo</u> allows you to specify:

- The message's priority level.
- Whether the message is reliable or unreliable.
- Whether the message is sequential or non-sequential.

Refer to the appropriate method reference for further details.

When your application receives a message, your callback function will receive a <u>DPN_MSGID_RECEIVE</u> message. The associated structure contains a pointer to the data block, along with information such as the source of the message.

 $\ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.

DP8Sim Utility

The DP8Sim utility is an executable that uses the methods of <u>IDP8SimControl</u> to allow you to test your application under a variety of network conditions. Alternatively, you can use the DP8Sim service provider and **IDP8SimControl** interface to create your own test environment. For more information, see <u>Testing Network Performance</u>.

Note DP8Sim is implemented on top of the existing DirectPlay8 TCP/IP Service Provider. The settings are also applied on top of the existing network characteristics. Therefore, it is intended to be used on a highspeed local area network (LAN) where normal latency and packet loss are negligible.

Path

Executable: (SDK root)\bin\dp8simui.exe

DLL: (SDK root)\bin\dp8sim.dll

User's Guide

The configuration utility, Dp8simui.exe, is the user interface for controlling Dp8sim.dll. Both files must reside in the same directory. When you start the configuration utility, it will automatically register the Dp8sim.dll Component Object Model (COM) objects. You can also manually register the dynamic-link library (DLL) by typing the following command at your Microsoft® MS-DOS® prompt.

regsvr32.exe dp8sim.dll

After the Dp8sim.dll is registered, the DP8Sim service provider will be available. Select the "DirectPlay8 TCP/IP Service Provider (Network Simulator)" service provider from the service providers returned by the IDirectPlay8Peer::EnumServiceProviders,

IDirectPlay8Client::EnumServiceProviders, and

IDirectPlay8Server::EnumServiceProviders methods. Note that Microsoft DirectPlay® sessions cannot use the network simulator if the DirectPlay interface was created before running the simulator.

Once the simulator is running, you can control various network options. There are also predefined network settings that may be useful.

The options for sending (receiving) are:

- *Bandwidth* (bytes/second). The total available outbound (inbound) bandwidth for all players in bytes per second. All packets have their latency increased in proportion to their size according to this value. If the application sends (receives) more than this amount, later packets are queued behind earlier ones. Use 0 (zero) for unlimited bandwidth, up to the real underlying network bandwidth.
- Drop percentage (0 to 100 percent). The random frequency as a

percentage for an individual outbound (inbound) packet to be dropped. Each packet stands the same chance of being dropped, regardless of other packets. Note that this does not necessarily model the behavior of all networks. Packet loss on the Internet, for example, tends to be erratic, with packet loss high at some times and low at others. A value of 1 means that an average of 1 out of every 100 packets is dropped. A value of 100 means every packet is dropped. Use 0 if you do not want to drop any packets other than loss due to the real underlying network.

- Minimum latency (milliseconds). The minimum delay in milliseconds for outbound (inbound) packets. The actual delay for an individual packet is chosen randomly between this minimum value and the maximum latency value. Note that the delay is applied on top of any delay imposed by bandwidth limitations. Use 0 if you do not want to have a lower bound for artificial latency beyond the real underlying network.
- *Maximum latency* (milliseconds). The maximum delay in milliseconds for outbound (inbound) packets. The actual delay for an individual packet is chosen randomly between the minimum latency value and this maximum value. If this value is lower than the minimum latency value, it is automatically set to equal the minimum value. Note that the delay is applied on top of any delay imposed by bandwidth limitations. Use 0 if you do not want to have an upper bound for artificial latency beyond the real underlying network.

Note These options apply for in-game data only. Host enumerations and responses are not subject to the simulation.

To make modifications, use the **Apply** and **Revert** buttons. Click **Apply** to change the settings and **Revert** to restore the previous settings. To save you settings, click the **Save As** button. You will be prompted to give a name for your settings.

At the bottom of the configuration utility window, the send and receive statistics for all affected DirectPlay interfaces are displayed. The **Refresh**

button updates the statistics and **Clear** resets the statistics to 0.

To unregister Dp8sim.dll, type the following at the MS-DOS command prompt.

regsvr32.exe /u dp8sim.dll

Note The DP8Sim utility provided with Microsoft DirectX® 8.1 will not work with applications using the DirectX 9.0 DLLs.

For more information about network performance, see <u>Optimizing</u> <u>Network Usage</u>.

© 2003 Microsoft Corporation. All rights reserved.

Monitoring DirectPlay Network Traffic with Network Monitor

During game development, you might find it useful to monitor Microsoft® DirectPlay® network traffic, especially when trying to understand bugs. The <u>Network Monitor</u> is a standard utility for analyzing network traffic. DirectPlay includes a set of parsers, that allow you to use the **Network Monitor** to analyze four components of DirectPlay messaging: the service provider layer, the transport layer, the session layer, and the voice layer.

How Network Monitor Works With DirectPlay

The DirectPlay protocol stack has three basic layers.

- The voice and session layers share the top level of the stack. Normal messaging passes through the session layer, and voicerelated messaging passes through the voice layer.
- The transport layer is the middle of the stack. Both voice and session traffic passes through this layer, which is responsible for such tasks as fragmentation and reassembly of messages and retransmission of lost packets.
- The service provider layer is at the bottom of the stack. All messaging is handled by this layer, which is responsible for communicating with the network. For example, for Transmission Control Protocol/Internet Protocol (TCP/IP) networking, the service provider uses the Windows Sockets (Winsock) application programming interface (API) to communicate with the network stack. The Network Monitor can only parse network traffic that is carried on an Internet Protocol (IP) or Internetwork Packet Exchange (IPX) service provider.

By installing the DirectPlay parsers, you can use the **Network Monitor** to analyze the network traffic as it passes through any of these four layers. You can see all DirectPlay traffic by selecting the service provider parser. However, by selecting one of the higher-level parsers, you can filter out traffic that might not be of interest.

With the transport layer parser, you see all voice and session traffic, but not low-level traffic such as connection handshaking. Be aware that the transport layer breaks messages that are longer than the network's Maximum Transmission Unit (MTU) into one or more fragments.

The session and voice layer parsers enable you to analyze session and voice-related traffic separately. Both of these parsers are can detect

fragmentation, and notify the user, but cannot parse fragmented packets.

Configuring Network Monitor for DirectPlay

If you have a Microsoft Windows® 2000 Server system, **Network Monitor** is already installed. For Windows 2000 Professional, you must purchase a copy of Systems Management Server (SMS). For a general discussion of how to use **Network Monitor**, see <u>About Network Monitor</u> <u>2.0</u>.

To configure the **Network Monitor** to handle DirectPlay traffic:

- Copy Dp8parse.dll from (SDK root)\bin\DXUtils to the appropriate folder. The Network Monitor root folder is normally installed in the \Winnt\System32 folder. If you have installed SMS, the root folder will be called NetMonFull. For Windows 2000 Server, the root folder will be called NetMon. Depending on which version of the Network Monitor you are using, copy the parser dynamic-link library (DLL) to either the ...\NetMonFull\Parsers, or ...\NetMon\parsers folder.
- 2. Start the **Network Monitor**.
- 3. Set the adapter to capture from (Capture, Networks, Local Computer). Be sure to choose the adapter with the "Dial-up Connection" property set to FALSE.

You are now ready to start capturing traffic.

Capturing DirectPlay Network Traffic

To start the capture process, click **Start Capture** on the **Network Monitor** toolbar to open the capture view. Initially, you will see all the traffic that is passing through your adapter. You can filter that raw traffic stream to focus on only those packets that are of interest. By installing the DirectPlay parsers, you essentially add four DirectPlay-oriented filters to **Network Monitor** that enable you to filter everything but DirectPlay traffic from your capture view.

To select a filter:

- 1. Click the **Edit Display Filter** button on the **Network Monitor** toolbar.
- 2. Double-click **Protocol == Any**.
- 3. Click **Disable All**.
- 4. Under **Disabled Protocols**, double-click **DPLAYSESSION**, **DPLAYSP**, **DPLAYTRANSPORT**, and **DPLAYVOICE**.

Click **OK** twice to return to the capture view, and you are ready to start viewing DirectPlay traffic.

You can also apply a filter to the capture process itself, rather than to the capture view. This allows you, for instance, to capture only IP packets with specified source and destination ports. For details, see the **Network Monitor** documentation.

Tips for Using Network Monitor with DirectPlay

Here are a few tips to using the **Network Monitor** with DirectPlay.

- By default, the **Network Monitor** captures only 1 MB of the most recent traffic. You will probably want to increase this value to at least 10 to 20 MB.
- The **Network Monitor** doesn't stream to the hard drive, so all you can see is what is in the capture buffer. To stream captured traffic to a hard drive, you need to implement your own capturer. For details, see MSDN®.
- By default, DirectPlay parsing uses the [2302,2400]U{6073} port/socket range to filter IP and IPX packets. To parse ports other than the default DirectPlay ports, create two new **DWORD** values under the

HKEY_CURRENT_USER\Software\Microsoft\DirectPlay\Parsers key, as shown in the following example.

MinUserPort = x

MaxUserPort = y

The x- and y-data values define the range to parse in addition to the default DirectPlay ports. They can be the same value if you only need one custom port.

 DirectPlay parsers support both signed and unsigned traffic. By default, the parsers assume that packets are unsigned. To enable monitoring of signed packets, set the DWORD value under the HKEY_CURRENT_USER\Software\Microsoft\DirectPlay\Parsers key, as shown in the following example. AssumeSigned = 1

• Because the DirectPlay and Real-time Transport Protocol (RTP) are both layered on top of the User Datagram Protocol (UDP), their parsers might conflict. You should disable the RTP parser when analyzing DirectPlay traffic, and vice versa.

© 2003 Microsoft Corporation. All rights reserved.

Network Address Translation, Firewalls, and Proxies

Network Address Translation (NAT) is a mechanism with which one network can be connected to another. This is commonly used to connect a private home or office network to the Internet. The gateway between these two networks modifies packets sent from the private network to computers on the Internet so that they appear to have been sent by the gateway. When packets are sent back from the Internet to the gateway, the gateway forwards the packet on to the associated private computer.

The two main reasons these NAT gateways are used are as follows:

- Improved security and access control. NAT devices provide a central point through which all traffic from the unmanaged Internet must flow to reach the presumably secure home network. The NAT software frequently has capabilities to filter out any packets that are potentially harmful, as described below.
- Increased address space. The explosion in popularity of the Internet means an incredible demand for addresses. Internet Protocol, version 4 (IPv4), the underlying protocol for today's Internet, is rapidly running out of available unique addresses. NAT devices allow the private computers to "share" a single Internet Protocol (IP) address. Each computer does actually have its own IP address, but that address is only valid within the home network. Any computer outside the home network uses the NAT device's public address to communicate with those inside the home network.

Firewalls are devices or software that inspect incoming or outgoing packets, and reject those that are not allowed by the firewall administrator. Most of them drop incoming packets that did not have a previous outgoing packet to the same port for security reasons. In this respect they behave like NAT devices, which can't forward packets without knowing their intended target. Many NAT devices also implement firewall capabilities.

Proxies relay requests to the external network on behalf of computers on the internal network. They can cache some requests like World Wide Web traffic for improved response time. They also typically work in conjunction with proxy client software installed on the internal computers for increased access control. Because external computers only see the proxy's external address, proxies can be thought of as performing NAT for the internal computers.

Unfortunately, all of these mechanisms are often at odds with providing a seamless network gaming experience. For example, having both a private address as well as a shared public address can make it hard to send packets to the appropriate destination. Sometimes the user is forced to enable forwarding for a particular <u>port</u> in order to play online. But until the next version of the Internet Protocol, version 6 (IPv6) becomes widely deployed, issues like address sharing will only grow more common.

Microsoft® DirectPlay® provides many features such as Universal Plug and Play (UPnP) support that take the hard work out of supporting NAT. This section includes the following topics.

- <u>Quick NAT Compatibility Guidelines</u>. Quick guidelines that all game developers should consider to maximize NAT compatibility.
- <u>Topology Specific NAT Issues</u>. Breakdown of the issues affecting each topology architecture.
- <u>Using the IDirectPlay8NATResolver Interface</u>. Help implementing <u>IDirectPlay8NATResolver</u> servers.
- Notes Regarding Firewalls and Proxies. A few notes about how

NATs affect firewalls and proxies.

• <u>NAT Troubleshooting Techniques for Developers and End Users</u>. Troubleshooting techniques for developers and end users.

© 2003 Microsoft Corporation. All rights reserved.

Quick NAT Compatibility Guidelines

Following these guidelines will help you maximize your application's compatibility with Network Address Translation (NAT) devices, proxies, and firewalls.

• Keep host enumerations active while connecting.

Ending a call to **EnumHosts** before calling IDirectPlay8Peer::Connect or IDirectPlay8Client::Connect on the same interface might close a virtual connection that was established with some NAT devices. This could prevent the call to **Connect** from succeeding. Instead, specify INFINITE for the *dwEnumCount* parameter when calling **EnumHosts** and let Microsoft® DirectPlay® stop the enumeration for you when the call to **Connect** completes successfully.

• Avoid relying on DPNSVR.

DPNSVR can simplify discovering multiple hosts on a single computer or a migrated host, but packets sent to the DPNSVR port can seem like separate connections to many NAT devices. They might not be forwarded in the same way as packets sent directly to the game's port. If you use DPNSVR, you should enumerate both the DPNSVR port and the game port when possible. See <u>Using</u> <u>the DirectPlay DPNSVR Application</u> for more information.

Use default ports but allow user to override if necessary.
You should let DirectPlay select the local device port to use by default. The addresses returned by
IDirectPlay8Peer::GetLocalHostAddresses and
IDirectPlay8Server::GetLocalHostAddresses can be used to

identify the port actually selected for hosts. Your game can then use external means, such as <u>DirectPlay Lobby</u> match-making, as a way to get the host's addresses to clients. However, applications should still provide the option for users to select a particular port. Some NAT scenarios require the end user to manually set up portforwarding and they will appreciate the greater flexibility. See <u>DirectPlay and Ports</u> for more information.

• Use the client/server topology.

Every player in the peer-to-peer topology must be able to communicate directly with every other player. This increases the chances that any particular player is prevented from joining a session successfully because of one or more NAT devices. Use the <u>IDirectPlay8Client</u> and <u>IDirectPlay8Server</u> interfaces to be sure that as long as the client can reach the host, it can join the session.

• Avoid hosting behind NAT devices that are not Universal Plug and Play (UPnP) compatible when possible.

Almost all NAT issues involve accepting inbound connections through a NAT. <u>IDirectPlay8Peer</u> or **IDirectPlay8Server** hosts require inbound connections by definition, and are therefore most susceptible. Hosting only on the open Internet or behind devices that support UPnP simplifies the NAT limitations presented to the user.

• Use match-making services to hide address complexity from user. When NAT devices are involved, a player might have multiple addresses even though the computer only has a single physical network interface. Most users will not know which address is
correct for a given situation. Your application should use matchmaking services to pass a combined address returned by **IDirectPlay8Peer::GetLocalHostAddresses** or **IDirectPlay8Server::GetLocalHostAddresses** to the client application for a better user experience. You can also use matchmaking to facilitate advanced NAT traversal techniques. See <u>DirectPlay Lobby Support</u> for more information on match-making.

These guidelines and example implementations are described in more detail for each topology.

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

Topology Specific NAT Issues

Each topology has different Network Address Translation (NAT) requirements, as does each role within the topology. This section has been into organized into the following categories.

- <u>NAT Issues for Peer Hosts</u>
- <u>NAT Issues for Peer_Clients</u>
- <u>NAT Issues for Servers</u>
- NAT Issues for Clients

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

NAT Issues for Peer Hosts

Hosting with the <u>IDirectPlay8Peer</u> interface has many special Network Address Translation (NAT) considerations, which are described in the following pages.

- Basic NAT Issues for Peer Hosts
- Advanced NAT Techniques for Peer Hosts
- Peer Host NAT Compatibility Reference

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

The best Network Address Translation (NAT) compatibility for hosting with the <u>IDirectPlay8Peer</u> interface is when the NAT device supports the use of Universal Plug and Play (UPnP) by Microsoft® DirectPlay®. Peer hosts should avoid relying on DPNSVR and use a particular set of ports to improve support for NAT devices without UPnP compatibility.

DPNSVR

The DPNSVR helper application is launched when IDirectPlay8Peer::Host is called with a DPN_APPLICATION_DESC structure that does not have the DPNSESSION_NODPNSVR flag set. The DPNSVR process listens for enumeration queries on a "well known" port, which is the same port that is assumed when the DPNA_KEY_PORT component is not specified in the IDirectPlay8Address host object passed to IDirectPlay8Peer::EnumHosts.

When DPNSVR receives an enumeration query, it is forwarded to all DPNSVR-enabled hosts on the local computer. Each host application then replies to the enumerator directly from its own port. However, some clients' NAT devices expect these replies to come from the port to which the client originally sent, and might drop this enumeration response. Therefore if your application uses DPNSVR to help with session discovery, it should also attempt to enumerate the game port directly. This requires that the client knows the game's addresses in advance. See Client Issues for more information on handling peer clients.

Determining the Host's Addresses

A host can determine the addresses on which it is listening by using the <u>IDirectPlay8Peer::GetLocalHostAddresses</u> method.

```
IDirectPlay8Address *pDP8AddressHost = NULL;
DWORD dwNumAddresses = 1;
```

hr = pDP8Peer->GetLocalHostAddresses(&pDP8AddressHost, &dwN

Specifying the DPNGETLOCALHOSTADDRESSES_COMBINED flag with the Transmission Control Protocol/Internet Protocol (TCP/IP) service provider returns a single address object that allows

IDirectPlay8Peer::EnumHosts or <u>IDirectPlay8Peer::Connect</u> to attempt all of the host's addresses simultaneously. This address can be converted into a string for easy transmission through a match-making service by using the <u>IDirectPlay8Address::GetURLA</u> method as shown in the following example.

Once received, the client converts the string back into an object using <u>IDirectPlay8Address::BuildFromURLA</u>. See Client Issues for more information on handling peer clients.

Using a Particular Port

Applications should generally let DirectPlay select a port when hosting. However, there are some NAT scenarios where the user might want to change the port on which the game is hosting. Also, if your application does not pass addresses using <u>DirectPlay Lobby</u> or other match-making service, it should have a default game port to use for direct discovery. This can be added to a device address using the DPNA_KEY_PORT component as shown in the following example.

DWORD dwUserSelectedPort; // value retrieved from user input

```
if (dwUserSelectedPort != 0)
{
    // User specified a port value; use it.
    hr = pDP8AddressDevice->AddComponent(DPNA_KEY_PORT
}
else
{
    // Let DirectPlay select; don't add port component.
}
```

This address object is then passed to **IDirectPlay8Peer::Host** as a device address on which to host.

Automatic UPnP Port Forwarding

When the host has the Windows Internet Connection Firewall enabled or is behind a UPnP NAT device, DirectPlay will attempt to enable port forwarding for your application automatically. This asks the device to accept all packets received from the Internet on a particular port and forward them to a particular address and port inside the private network.

If DirectPlay selected the local port for the host, then it will select an unused external port for the NAT device to forward. The actual public port number chosen will vary, and might not be the same as the local port.

If the DPNA_KEY_PORT component was set in the device address specified to **IDirectPlay8Peer::Host**, then DirectPlay will ask the NAT device to forward the same external port number. If that public port number is in use then the call to **IDirectPlay8Peer::Host** will fail with DPNERR_INVALIDDEVICEADDRESS. This can happen when another instance of the application is already hosting behind the same NAT.

You should design your application and match-making so that they do not require the same port to be used both locally and on the NAT device. You can then allow DirectPlay to try alternate external ports when the matching port is not available by using the DPNA_KEY_TRAVERSALMODE device address component as shown in the following example.

DWORD dwTraversalMode = DPNA_TRAVERSALMODE_PORTR hr = pDP8AddressDevice->AddComponent(DPNA_KEY_TRAVERS

Disabling Automatic Traversal

Some users know that the hosting application is not behind a UPnP NAT device and the Windows Internet Connection Firewall is not enabled. Others might want to manually control any mappings made for the host. You can decrease the time required by **IDirectPlay8Peer::Host** and prevent automated traversal by setting the DPNA_KEY_TRAVERSALMODE component to DPNA_TRAVERSALMODE_NONE as shown in the following example.

DWORD dwTraversalMode = DPNA_TRAVERSALMODE_NONE;

hr = pDP8AddressDevice->AddComponent(DPNA_KEY_TRAVERS

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

Microsoft® DirectPlay® cannot provide 100 percent connectivity for hosts behind Network Address Translation (NAT) devices that do not support Universal Plug and Play (UPnP). However, your application might be able to improve support for some of these cases using more elaborate methods. The techniques described here require additional development effort, as well as external server resources.

NAT Resolver

NAT devices create an implicit port mapping when an internal computer sends a packet to an external computer. Some NAT devices allow any external computer to use this port mapping to send to the internal computer, instead of only forwarding replies sent by the original external target. These are sometimes referred to as "loose NATs." DirectPlay can take advantage of this behavior using the <u>IDirectPlay8NATResolver</u> interface and address components to allow hosting behind these devices.

You must start by implementing an **IDirectPlay8NATResolver** server that will be accessible from the Internet. Then your game application can add the DPNA_KEY_NAT_RESOLVER component to the device address passed to <u>IDirectPlay8Peer::Host</u>. The component data is a string containing the Internet Protocol (IP) address and port for the NAT resolver to use, separated by a colon. For example, the following code specifies to use a server located on port 5678 at the address 123.123.123.123.

```
WCHAR * wszNATResolver = L"123.123.123.123:5678";
DWORD dwNATResolverSize = (wcslen(wszNATResolver) + 1) * si
```

```
hr = pDP8DeviceAddress->AddComponent(DPNA_KEY_NAT_RES
wszNATResolver,
dwNATResolverSize,
DPNA_DATATYPE_STRING);
```

You can also specify a hostname instead of a numerical IP address, as shown in the following example.

```
WCHAR * wszNATResolver = L"resolver.mydomain.com:5678";
DWORD dwNATResolverSize = (wcslen(wszNATResolver) + 1) * si
```

For robustness, you might want to have more than one resolving servers to try. You can specify multiple addresses separated by commas as shown in the following example.

```
WCHAR * wszNATResolvers = L"123.123.123.123:5678,backupreso
DWORD dwNATResolverSize = (wcslen(wszNATResolvers) + 1) * s
```

Each resolver is tried simultaneously for speed, and the first response is used. If no server responds, the **IDirectPlay8Peer::Host** call still succeeds.

Because hosting these resolving servers requires resources, you might want to prevent arbitrary players from using the resolver. This can be achieved with the DPNA_KEY_NAT_RESOLVER_USER_STRING

address component. This value is passed directly to the resolver for verification in the <u>DPN_MSGID_NAT_RESOLVER_QUERY</u> message, and it can choose to respond or ignore as appropriate. The following example shows how to do this.

Note The user string is passed in clear text over the network, so if the text could contain sensitive information, you should encrypt it in some fashion.

The host's NAT device generates a port mapping for the NAT resolver query. If the NAT resolver elects to respond to the query, DirectPlay will send the mapping's public address back to the querying host. The host will then have it included in the addresses returned by the IDirectPlay8Peer::GetLocalHostAddresses method.

External peer clients use the mapping address when connecting to the internal host. However the NAT device will expire the mapping if it is inactive, which will prevent the clients from using it. The actual expiration timeout varies by NAT device. Generally, the external peer clients are expected to begin connecting within 30 seconds of the **IDirectPlay8Peer::Host** call completing.

An example usage of NAT resolver address components can be found in the <u>NATPeer</u> sample.

Enumerating Clients to Create Implicit NAT Mappings

An expanded technique can sometimes be used to host behind so-called "strict NATs" that only allow the target of an outbound packet to send packets in to the private network. If the strict NAT device uses the same external port number for sending to different external addresses, even though they are using different implicit mappings, the host might be able to use the <u>IDirectPlay8Peer::EnumHosts</u> method to generate mappings for external peer clients that want to join.

The host must know that the external peer client intends join, as well as the address that the client will use when connecting. This is determined through match-making or similar means. The client's address is then passed as the *pdpaddrHost* parameter for **IDirectPlay8Peer::EnumHosts** by the host itself. See <u>Enumerating</u>

Hosts for more information.

The joining peers do not need to send an enumeration response and will not even receive a query notification message callback. The NAT device will merely create a mapping for the outbound enumeration packet from the host that allows the client's inbound enumeration or connect requests to be forwarded to the internal host. The host's reverse **IDirectPlay8Peer::EnumHosts** operation should remain active until the peer client successfully joins or stops attempting to join so that the NAT

mapping remains active.

Because the address of the external peer clients must be determined beforehand, the clients also must not be behind NAT devices that select different port numbers for different external targets.

^{© 2003} Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

The following section outlines peer host Network Address Translation (NAT) compatibility for applications that follow the guidelines recommended in previous sections.

Windows 98 Second Edition Internet Connection Sharing

The following table shows the default support for hosts using Microsoft® Windows® 98 Second Edition Internet Connection Sharing (ICS).

| | Join from behind ICS computer | Join from ICS computer | Join from external network |
|--------------------------|-------------------------------|---------------------------|---|
| Host behind ICS computer | Yes | Yes | Not without manual port forwarding |
| Host on ICS computer | Yes | Yes | Not without manual port forwarding or by discovering game via broadcast |

Windows 98 Second Edition ICS supports joining a host on or behind the ICS computer from the Internet only with manual port forwarding.

The <u>Advanced NAT Techniques for Peer Hosts</u> section describes methods that might be used to allow external clients to join the host without manual port forwarding in some scenarios.

Windows 2000 Internet Connection Sharing

The following table shows the default support for hosts using Windows 2000 ICS.

| | Join from behind ICS computer | Join from ICS computer | Join from external network |
|--------------------------|-------------------------------|---------------------------|---------------------------------------|
| Host behind ICS computer | Yes | Yes | Not without manual port forwarding |
| Host on ICS computer | Yes | Yes | Yes |

Windows 2000 ICS supports joining a host on the ICS computer from the Internet. Joining a host behind the ICS computer requires manual port forwarding.

The **Advanced NAT Techniques for Peer Hosts** section describes methods that can be used to allow external clients to join the host without manual port forwarding in some scenarios.

Windows Millennium Edition Internet Connection Sharing

The following table shows the default support for hosts using Windows Millennium Edition (Windows Me) ICS.

| | Join from behind ICS computer | Join from ICS computer | Join from external network |
|--------------------------|-------------------------------|------------------------|----------------------------|
| Host behind ICS computer | Yes | Yes | Yes |
| Host on ICS computer | Yes | Yes | Yes |

Hosting behind or on a Windows Me ICS computer is fully supported using Microsoft DirectPlay®'s automatic traversal.

Windows XP Internet Connection Sharing

The following table shows the default support for hosts using Windows XP ICS.

| | Join from behind ICS computer | Join from ICS computer | Join from external network |
|--------------------------|-------------------------------|------------------------|----------------------------|
| Host behind ICS computer | Yes | Yes | Yes |
| Host on ICS computer | Yes | Yes | Yes |

Hosting behind or on a Windows XP ICS computer is fully supported using DirectPlay's automatic traversal.

Non-Microsoft NAT Device with Universal Plug and Play

Hosting behind a non-Microsoft NAT device that is compatible with DirectPlay's automatic traversal using Universal Plug and Play (UPnP) is supported. Refer to the NAT device's documentation for UPnP compatibility information.

Non-Microsoft NAT Device without Universal Plug and Play

Clients can join a host behind a non-Microsoft NAT device without UPnP support if the clients are behind the same device. External clients cannot join by default unless port forwarding is manually configured. Refer to the NAT device's documentation for information on enabling port forwarding.

The **Advanced NAT Techniques for Peer Hosts** section describes methods that can be used to allow external clients to join the host without manual port forwarding in some scenarios.

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

NAT Issues for Peer Clients

Clients using the <u>IDirectPlay8Peer</u> interface have many special Network Address Translation (NAT) considerations, which are described in the following pages.

- Basic NAT Issues for Peer Clients
- Advanced NAT Techniques for Peer Clients
- <u>Peer Client NAT Compatibility Reference</u>

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

The best Network Address Translation (NAT) compatibility for joining a session with the <u>IDirectPlay8Peer</u> interface is when the NAT device supports Microsoft® DirectPlay®'s use of Universal Plug and Play (UPnP). Peer clients should keep enumerations active and avoid relying on DPNSVR to improve support for NAT devices without UPnP compatibility.

Keeping EnumHosts Active While Connecting

Most NAT devices create virtual connections each time an internal client starts sending packets to a new external address and port, such as a new <u>IDirectPlay8Peer::EnumHosts</u> operation. These virtual connections will sometimes be destroyed as soon as the

IDirectPlay8Peer::EnumHosts operation completes. If your client application enumerates hosts before connecting, you should neither cancel the **IDirectPlay8Peer::EnumHosts** operation nor allow it to time out so that the call to <u>IDirectPlay8Peer::Connect</u> can reuse the virtual connection. This requirement can be met by specifying INFINITE for the *dwEnumCount* parameter as shown in the following example.

hr = pDP8Peer->EnumHosts(&dpnad, pDP8AddressHost, pDP8AddressDevice, NULL, 0, INFINITE, 0, 0, 0, NULL, &dpnhEnumHosts, 0);

DirectPlay will automatically cancel all outstanding

IDirectPlay8Peer::EnumHosts operations for you when the connect operation completes successfully. If the connect fails, the enumerations will continue running. You can then cancel them using the IDirectPlay8Peer::CancelAsyncOperation method.

To ensure that the connect operation uses the same virtual connection as

the enumeration, the host and device address objects specified to **IDirectPlay8Peer::Connect** should be the same as those included in the corresponding <u>DPN_MSGID_ENUM_HOSTS_RESPONSE</u> message. For examples of how to add a reference or duplicate these addresses, see the <u>DirectPlay C++ Samples</u>.

DPNSVR

The application sends enumeration queries to the DPNSVR port when the DPNA_KEY_PORT component is not specified in the <u>IDirectPlay8Address</u> host object passed to

IDirectPlay8Peer::EnumHosts. However the hosts reply from their respective game ports. Some NAT devices expect replies to come from the port to which the client originally sent, and might therefore drop the enumeration response. If your application uses DPNSVR to help with session discovery, it should also attempt to enumerate the game port directly.

Enumerating the Host's Game Port Directly

The host can retrieve its addresses using

IDirectPlay8Peer::GetLocalHostAddresses and pass it in string format to the client using IDirectPlay8Address::GetURLA through a match-making service. The client can then convert the string back into an address using the IDirectPlay8Address::BuildFromURLA method as shown in the following example.

IDirectPlay8Address *pDP8AddressHost = NULL;

hr = CoCreateInstance(CLSID_DirectPlay8Address, CLSCTX_INPR

hr = pDP8AddressHost->BuildFromURLA(szHostAddress);

This address object can then be passed to

IDirectPlay8Peer::EnumHosts or IDirectPlay8Peer::Connect.

Enumerating or Connecting From a Particular Port

Applications should almost always let DirectPlay choose the local port or ports when enumerating or connecting. However, the local ports selected by DirectPlay to connect to the original host are the same ones used to listen for new players once the host migrates. If your application uses host migration but does not pass updated addresses for the new host by a <u>DirectPlay Lobby</u> or other match-making service, you might want to allow the user to select a local port. This can be added to a device address using the DPNA_KEY_PORT component as shown in the following example.

```
DWORD dwUserSelectedPort; // value retrieved from user input
```

```
if (dwUserSelectedPort != 0)
{
    // User specified a port value; use it.
    hr = pDP8AddressDevice->AddComponent(DPNA_KEY_PORT, &
}
else
{
    // Let DirectPlay select; don't add port component.
}
```

This address object is then passed to **IDirectPlay8Peer::EnumHosts** or **IDirectPlay8Peer::Connect** for the device address parameter.
Automatic UPnP Port Forwarding

When the client has Microsoft Windows® Internet Connection Firewall enabled or is behind a UPnP NAT device, DirectPlay will attempt to enable port forwarding for your application automatically. This asks the device to accept all packets received from the Internet on a particular port and forward them to a particular address and port inside the private network.

If DirectPlay selected the local port for the client, then it will select an unused external port for the NAT device to forward. The actual public port number chosen will vary, and might not be the same as the local port.

If the DPNA_KEY_PORT component was set in the device address specified to **IDirectPlay8Peer::EnumHosts** or

IDirectPlay8Peer::Connect, then DirectPlay will ask the NAT device to forward the same external port number. If that public port number is in use then the call to IDirectPlay8Peer::EnumHosts or IDirectPlay8Peer::Connect will fail with

DPNERR_INVALIDDEVICEADDRESS. This might happen when another instance of the application behind the same NAT is already connected. You should design your application and match-making so that they do not require the same port to be used both locally and on the NAT device. You can then allow DirectPlay to try alternate external ports when the matching port is not available by using the

DPNA_KEY_TRAVERSALMODE device address component. This is shown in the following example.

DWORD dwTraversalMode = DPNA_TRAVERSALMODE_PORTR

hr = pDP8AddressDevice->AddComponent(DPNA_KEY_TRAVERS

Disabling Automatic Traversal

Some users know that the client is not behind a UPnP NAT device and Windows Internet Connection Firewall is not enabled. Others might want to manually control any mappings made for the client. You can decrease the time required by **IDirectPlay8Peer::EnumHosts** or **IDirectPlay8Peer::Connect** and prevent automated traversal by setting the DPNA_KEY_TRAVERSALMODE component to DPNA_TRAVERSALMODE_NONE as shown in the following example.

DWORD dwTraversalMode = DPNA_TRAVERSALMODE_NONE;

hr = pDP8AddressDevice->AddComponent(DPNA_KEY_TRAVERS

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

Advanced NAT Techniques for Peer Clients

Microsoft® DirectPlay® cannot provide 100 percent connectivity for peer clients behind Network Address Translation (NAT) devices that do not support Universal Plug and Play (UPnP). However, your application might be able to improve support for some of these cases using more elaborate methods. The techniques described here require additional development effort as well as external server resources.

NAT Resolver

NAT devices create an implicit port mapping when an internal computer sends a packet to an external computer. Some NAT devices allow any external computer to use this port mapping to send to the internal computer, instead of only forwarding replies sent by the original external target. These are sometimes referred to as "loose NATs." DirectPlay automatically takes advantage of this behavior when the peer host is outside of the NAT. Your application can use the IDirectPlay8NATResolver interface and address components to also allow peer clients behind a loose NAT to join a session hosted behind the same NAT device with external peer clients already connected to the same session.

You must start by implementing an **IDirectPlay8NATResolver** server that will be accessible from the Internet. Then your game application can add the DPNA_KEY_NAT_RESOLVER component to the device address passed to <u>IDirectPlay8Peer::EnumHosts</u> or <u>IDirectPlay8Peer::Connect</u>. The component data is a string containing the Internet Protocol (IP) address and port for the NAT resolver to use, separated by a colon. For example, the following code specifies to use a server located on port 5678 at the address 123.123.123.123.

```
WCHAR * wszNATResolver = L"123.123.123.123:5678";
DWORD dwNATResolverSize = (wcslen(wszNATResolver) + 1) * si
```

You can also specify a hostname instead of a numerical IP address as shown in the following example.

```
WCHAR * wszNATResolver = L"resolver.mydomain.com:5678";
DWORD dwNATResolverSize = (wcslen(wszNATResolver) + 1) * si
```

For robustness, you might want to have more than one resolving servers to try. You can specify multiple addresses separated by commas as shown in the following example.

```
WCHAR * wszNATResolvers = L"123.123.123.123.5678,backupreso
DWORD dwNATResolverSize = (wcslen(wszNATResolvers) + 1) * s
```

Each resolver is tried simultaneously for speed, and the first response is used. If no server responds, the **IDirectPlay8Peer::EnumHosts** or **IDirectPlay8Peer::Connect** call still succeeds.

Because hosting these resolving servers requires resources, you might want to prevent arbitrary players from using the resolver. This can be achieved with the DPNA_KEY_NAT_RESOLVER_USER_STRING address component. This value is passed directly to the resolver for verification in the DPN_MSGID_NAT_RESOLVER_QUERY message, and it can choose to respond or ignore as appropriate. The following example shows how to do this.

```
WCHAR * wszPassword = L"MyPassword";
DWORD dwPasswordSize = (wcslen(wszPassword) + 1) * sizeof(WC
hr = pDP8DeviceAddress->AddComponent(DPNA_KEY_NAT_RES
wszPassword,
dwPasswordSize,
DPNA DATATYPE STRING);
```

Note The user string is passed in clear text over the network, so if the text could contain sensitive information, you should encrypt it in some fashion.

The peer client's NAT device generates a port mapping for the NAT resolver query. If the NAT resolver elects to respond to the query, DirectPlay will send the mapping's public address back to the querying peer client. The address will then be reported to the host when connecting to the session so external peer clients can contact this joining internal client.

An example usage of NAT resolver address components can be found in the <u>NATPeer</u> sample.

 $\ensuremath{\textcircled{\sc 0}}$ 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

Peer Client NAT Compatibility Reference

The following section outlines peer client Network Address Translation (NAT) compatibility for applications that follow the guidelines recommended in previous sections.

Windows 98 Second Edition Internet Connection Sharing

Any number of peer clients behind or on the same Microsoft® Windows® 98 Second Edition Internet Connection Sharing (ICS) system is supported. Peer clients behind different Windows 98 Second Edition ICS computers are not supported.

A player behind or on the Windows 98 Second Edition ICS computer that receives a DPN_MSGID_HOST_MIGRATE message indicating that the host has migrated to the local player becomes a peer host. See the Windows 98 Second Edition Internet Connection Sharing section in the Peer Host NAT Compatibility Reference topic.

Windows 2000 Internet Connection Sharing

Any number of peer clients behind or on the same Windows 2000 ICS system is supported. Peer clients behind different Windows 2000 ICS computers are not supported.

A player behind or on the Windows 2000 ICS computer that receives a **DPN_MSGID_HOST_MIGRATE** message indicating that the host has migrated to the local player becomes a peer host. See the <u>Windows 2000</u> Internet Connection Sharing section in the **Peer Host NAT Compatibility Reference** topic.

Windows Millennium Edition Internet Connection Sharing

Any number of peer clients behind or on the same or different Windows Millennium Edition (Windows Me) ICS computers is fully supported using Microsoft DirectPlay®'s automatic traversal.

Windows XP Internet Connection Sharing

Any number of peer clients behind or on the same or different Windows XP ICS computers is fully supported using DirectPlay's automatic traversal.

Non-Microsoft NAT Device with Universal Plug and Play

Any number of peer clients behind the same or different non-Microsoft NAT devices that are compatible with DirectPlay's automatic traversal using Universal Plug and Play (UPnP) is supported. Refer to the NAT device's documentation for UPnP compatibility information.

Non-Microsoft NAT Device without Universal Plug and Play

Any number of peer clients behind the same NAT device without UPnP is supported. Peer clients behind different NAT devices without UPnP are not supported but might work, depending on the device implementation.

The <u>Advanced NAT Techniques for Peer Clients</u> describes methods that can be used to allow clients to join sessions in some scenarios that do not work using default settings.

A player behind a non-Microsoft NAT device that receives a **DPN_MSGID_HOST_MIGRATE** message indicating that the host has migrated to the local player becomes a peer host. See the <u>Third-Party</u> NAT Device Without Universal Plug-and-Play section in the **Peer Host NAT Compatibility Reference** topic.

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

NAT Issues for Servers

The <u>IDirectPlay8Server</u> interface has some special Network Address Translation (NAT) considerations, which are described in the following pages.

- Basic NAT Issues for Servers
- Advanced NAT Techniques for Servers
- <u>Server NAT Compatibility Reference</u>

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

The best Network Address Translation (NAT) compatibility for hosting with the <u>IDirectPlay8Server</u> interface is when the NAT device supports the use of Universal Plug and Play (UPnP) by Microsoft® DirectPlay®. Servers should avoid relying on DPNSVR and use a particular set of ports to improve support for NAT devices without UPnP compatibility.

DPNSVR

The DPNSVR helper application is launched when IDirectPlay8Server::Host is called with a DPN_APPLICATION_DESC structure that does not have the DPNSESSION_NODPNSVR flag set. The DPNSVR process listens for enumeration queries on a "well known" port, which is the same port that is assumed when the DPNA_KEY_PORT component is not specified in the IDirectPlay8Address host object passed to IDirectPlay8Client::EnumHosts.

When DPNSVR receives an enumeration query it is forwarded to all DPNSVR-enabled hosts on the local computer. Each host application then replies to the enumerator directly from its own port. However, some clients' NAT devices expect these replies to come from the port to which the client originally sent, and might drop this enumeration response. Therefore if your application uses DPNSVR to help with session discovery, it should also attempt to enumerate the game port directly. This requires that the client knows the game's addresses in advance. See Client Issues for More Information on Handling Peer Clients.

Determining the Host's Addresses

A host can determine the addresses on which it is listening by using the <u>IDirectPlay8Server::GetLocalHostAddresses</u> method.

```
IDirectPlay8Address *pDP8AddressHost = NULL;
DWORD dwNumAddresses = 1;
```

hr = pDP8Server->GetLocalHostAddresses(&pDP8AddressHost, &dv

Specifying the DPNGETLOCALHOSTADDRESSES_COMBINED flag with the Transmission Control Protocol/Internet Protocol (TCP/IP) service provider returns a single address object that allows

IDirectPlay8Client::EnumHosts or <u>IDirectPlay8Client::Connect</u> to attempt all of the host's addresses simultaneously. This address can be converted into a string for easy transmission through a match-making service by using the <u>IDirectPlay8Address::GetURLA</u> method as shown in the following example.

```
char *szHostAddress = NULL;
DWORD dwNumHostAddressChars = 0;
hr = pDP8AddressHost->GetURLA(NULL, &dwNumHostAddressCl
szHostAddress = LocalAlloc(LPTR, dwNumHostAddressChars * size
hr = pDP8AddressHost->GetURLA(szHostAddress, &dwNumHostAddress)
```

Once received, the client converts the string back into an object using <u>IDirectPlay8Address::BuildFromURLA</u>. See Client Issues for more information on handling peer clients.

Using a Particular Port

Applications should generally let DirectPlay select a port when hosting. However, there are some NAT scenarios where the user might want to change the port on which the game is hosting. Also, if your application does not pass addresses using <u>DirectPlay Lobby</u> or other match-making service, it should have a default game port to use for direct discovery. This can be added to a device address using the DPNA_KEY_PORT component as shown in the following example.

DWORD dwUserSelectedPort; // value retrieved from user input

```
if (dwUserSelectedPort != 0)
{
    // User specified a port value; use it.
    hr = pDP8AddressDevice->AddComponent(DPNA_KEY_PORT
}
else
{
    // Let DirectPlay select; don't add port component.
}
```

This address object is then passed to **IDirectPlay8Server::Host** as a device address on which to host.

Automatic UPnP Port Forwarding

When the host has the Microsoft Windows® Internet Connection Firewall enabled or is behind a UPnP NAT device, DirectPlay will attempt to enable port forwarding for your application automatically. This asks the device to accept all packets received from the Internet on a particular port and forward them to a particular address and port inside the private network.

If DirectPlay selected the local port for the host, then it will select an unused external port for the NAT device to forward. The actual public port number chosen will vary, and might not be the same as the local port.

If the DPNA_KEY_PORT component was set in the device address specified to **IDirectPlay8Server::Host**, DirectPlay will ask the NAT device to forward the same external port number. If that public port number is in use then the call to **IDirectPlay8Server::Host** will fail with DPNERR_INVALIDDEVICEADDRESS. This can happen when another instance of the application is already hosting behind the same NAT.

You should design your application and match-making so that they do not require the same port to be used both locally and on the NAT device. You can then allow DirectPlay to try alternate external ports when the matching port is not available by using the DPNA_KEY_TRAVERSALMODE device address component as shown

in the following example.

DWORD dwTraversalMode = DPNA_TRAVERSALMODE_PORTR hr = pDP8AddressDevice->AddComponent(DPNA_KEY_TRAVERS

Disabling Automatic Traversal

Some users know that the hosting application is not behind a UPnP NAT device and the Windows Internet Connection Firewall is not enabled. Others might want to manually control any mappings made for the host. You can decrease the time required by **IDirectPlay8Server::Host** and prevent automated traversal by setting the DPNA_KEY_TRAVERSALMODE component to DPNA_TRAVERSALMODE_NONE as shown in the following example.

DWORD dwTraversalMode = DPNA_TRAVERSALMODE_NONE;

hr = pDP8AddressDevice->AddComponent(DPNA_KEY_TRAVERS

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

Microsoft® DirectPlay® cannot provide 100 percent connectivity for hosts behind Network Address Translation (NAT) devices that do not support Universal Plug and Play (UPnP). However, your application might be able to improve support for some of these cases using more elaborate methods. The techniques described here require additional development effort as well as external server resources.

NAT Resolver

NAT devices create an implicit port mapping when an internal computer sends a packet to an external computer. Some NAT devices allow any external computer to use this port mapping to send to the internal computer, instead of only forwarding replies sent by the original external target. These are sometimes referred to as "loose NATs." DirectPlay can take advantage of this behavior using the <u>IDirectPlay8NATResolver</u> interface and address components to allow hosting behind these devices.

You must start by implementing an **IDirectPlay8NATResolver** server that will be accessible from the Internet. Then your game application can add the DPNA_KEY_NAT_RESOLVER component to the device address passed to <u>IDirectPlay8Server::Host</u>. The component data is a string containing the Internet Protocol (IP) address and port for the NAT resolver to use, separated by a colon. For example, the following code specifies to use a server located on port 5678 at the address 123.123.123.123.

```
WCHAR * wszNATResolver = L"123.123.123.123:5678";
DWORD dwNATResolverSize = (wcslen(wszNATResolver) + 1) * si
```

You can also specify a hostname instead of a numerical IP address as shown in the following example.

```
WCHAR * wszNATResolver = L"resolver.mydomain.com:5678";
DWORD dwNATResolverSize = (wcslen(wszNATResolver) + 1) * si
```

For robustness, you might want to have more than one resolving servers to try. You can specify multiple addresses separated by commas as shown in the following example.

WCHAR * wszNATResolvers = L"123.123.123.123.5678,backupreso DWORD dwNATResolverSize = (wcslen(wszNATResolvers) + 1) * s

```
hr = pDP8DeviceAddress->AddComponent(DPNA_KEY_NAT_RES
wszNATResolvers,
dwNATResolverSize,
DPNA_DATATYPE_STRING);
```

Each resolver is tried simultaneously for speed, and the first response is used. If no server responds, the **IDirectPlay8Server::Host** call still succeeds.

Because hosting these resolving servers requires resources, you might want to prevent arbitrary players from using the resolver. This can be achieved with the DPNA_KEY_NAT_RESOLVER_USER_STRING

address component. This value is passed directly to the resolver for verification in the <u>DPN_MSGID_NAT_RESOLVER_QUERY</u> message, and it can choose to respond or ignore as appropriate. The following example shows how to do this.

Note The user string is passed in clear text over the network, so if the text could contain sensitive information, you should encrypt it in some fashion.

The host's NAT device generates a port mapping for the NAT resolver query. If the NAT resolver elects to respond to the query, DirectPlay will send the mapping's public address back to the querying host. The host will then have it included in the addresses returned by the IDirectPlay8Server::GetLocalHostAddresses method.

External clients use the mapping address when connecting to the internal host. However the NAT device will expire the mapping if it is inactive, which will prevent external clients from using it. The actual expiration timeout varies by NAT device. Generally, the external clients are expected to begin connecting within 30 seconds of the **IDirectPlay8Server::Host** call completing.

An example usage of NAT resolver address components can be found in the <u>NATPeer</u> sample.

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

Server NAT Compatibility Reference

The following section outlines server Network Address Translation (NAT) compatibility for applications that follow the guidelines recommended in previous sections.

Windows 98 Second Edition Internet Connection Sharing

The following table shows the default support for hosts using Microsoft® Windows® 98 Second Edition Internet Connection Sharing (ICS).

| | Join from behind ICS computer | Join from ICS computer | Join from external network |
|--------------------------|-------------------------------|---------------------------|---|
| Host behind ICS computer | Yes | Yes | Not without manual port forwarding |
| Host on ICS computer | Yes | Yes | Not without manual port forwarding or by discovering game via broadcast |

Windows 98 Second Edition ICS supports joining a host on or behind the ICS computer from the Internet only with manual port forwarding.

The <u>Advanced NAT Techniques for Peer Hosts</u> section describes methods that can be used to allow external clients to join the host without manual port forwarding in some scenarios.
Windows 2000 Internet Connection Sharing

The following table shows the default support for hosts using Windows 2000 ICS.

| | Join from behind ICS computer | Join from ICS computer | Join from external network |
|--------------------------|-------------------------------|---------------------------|---------------------------------------|
| Host behind ICS computer | Yes | Yes | Not without manual port forwarding |
| Host on ICS computer | Yes | Yes | Yes |

Windows 2000 ICS supports joining a host on the ICS computer from the Internet. Joining a host behind the ICS computer requires manual port forwarding.

Windows Millennium Edition Internet Connection Sharing

The following table shows the default support for hosts using Windows Millennium Edition (Windows Me) ICS.

| | Join from behind ICS computer | Join from ICS computer | Join from external network |
|--------------------------|-------------------------------|------------------------|----------------------------|
| Host behind ICS computer | Yes | Yes | Yes |
| Host on ICS computer | Yes | Yes | Yes |

Hosting behind or on a Windows Me ICS computer is fully supported using Microsoft DirectPlay®'s automatic traversal.

Windows XP Internet Connection Sharing

The following table shows the default support for hosts using Windows XP ICS.

| | Join from behind ICS computer | Join from ICS computer | Join from external network |
|--------------------------|-------------------------------|------------------------|----------------------------|
| Host behind ICS computer | Yes | Yes | Yes |
| Host on ICS computer | Yes | Yes | Yes |

Hosting behind or on a Windows XP ICS computer is fully supported using DirectPlay's automatic traversal.

Non-Microsoft NAT Device with Universal Plug and Play

Hosting behind a non-Microsoft NAT device that is compatible with DirectPlay's automatic traversal using Universal Plug and Play (UPnP) is supported. Refer to the NAT device's documentation for UPnP compatibility information.

Non-Microsoft NAT Device without Universal Plug and Play

Clients can join a host behind a non-Microsoft NAT device without UPnP support if the clients are behind the same device. External clients cannot join by default unless port forwarding is manually configured. Refer to the NAT device's documentation for information on enabling port forwarding.

The <u>Advanced NAT Techniques for Servers</u> section describes methods that can be used to allow external clients to join the host without manual port forwarding in some scenarios.

© 2003 Microsoft Corporation. All rights reserved.

NAT Issues for Clients

Clients using the <u>IDirectPlay8Client</u> interface have some limited Network Address Translation (NAT) considerations, which are described in the following pages.

- Basic NAT Issues for Clients
- <u>Client NAT Compatibility Reference</u>

© 2003 Microsoft Corporation. All rights reserved.

The <u>IDirectPlay8Client</u> interface has very few Network Address Translation (NAT) concerns and works in almost all scenarios. Clients should however keep enumerations active and avoid relying on DPNSVR to ensure NAT compatibility.

Keeping EnumHosts Active While Connecting

Most NAT devices create virtual connections each time an internal client starts sending packets to a new external address and port, such as a new <u>IDirectPlay8Client::EnumHosts</u> operation. These virtual connections will sometimes be destroyed as soon as the

IDirectPlay8Client::EnumHosts operation completes. If your client application enumerates hosts before connecting, you should neither cancel the **IDirectPlay8Client::EnumHosts** operation nor allow it to time out so that the call to <u>IDirectPlay8Client::Connect</u> can reuse the virtual connection. This requirement can be met by specifying INFINITE for the *dwEnumCount* parameter as shown in the following example.

hr = pDP8Client->EnumHosts(&dpnad, pDP8AddressHost, pDP8AddressDevice, NULL, 0, INFINITE, 0, 0, 0, NULL, &dpnhEnumHosts, 0);

Microsoft® DirectPlay® will automatically cancel all outstanding IDirectPlay8Client::EnumHosts operations for you when the connect operation completes successfully. If the connect fails, the enumerations will continue running. You can then cancel them using the IDirectPlay8Client::CancelAsyncOperation method.

To ensure that the connect operation uses the same virtual connection as

the enumeration, the host and device address objects specified to **IDirectPlay8Client::Connect** should be the same as those included in the corresponding <u>DPN_MSGID_ENUM_HOSTS_RESPONSE</u> message. For examples of how to add a reference or duplicate these addresses, see the <u>DirectPlay C++ Samples</u>.

DPNSVR

The application sends enumeration queries to the DPNSVR port when the DPNA_KEY_PORT component is not specified in the <u>IDirectPlay8Address</u> host object passed to

IDirectPlay8Client::EnumHosts. However the hosts reply from their respective game ports. Some NAT devices expect replies to come from the port to which the client originally sent, and might therefore drop the enumeration response. If your application uses DPNSVR to help with session discovery, it should also attempt to enumerate the game port directly.

Enumerating the Host's Game Port Directly

The host can retrieve its addresses using <u>IDirectPlay8Server::GetLocalHostAddresses</u> and pass it in string format to the client using <u>IDirectPlay8Address::GetURLA</u> through a matchmaking service. The client can then convert the string back into an address using the <u>IDirectPlay8Address::BuildFromURLA</u> method as shown in the following example.

IDirectPlay8Address *pDP8AddressHost = NULL;

hr = CoCreateInstance(CLSID_DirectPlay8Address, CLSCTX_INPR

hr = pDP8AddressHost->BuildFromURLA(szHostAddress);

This address object can then be passed to

IDirectPlay8Client::EnumHosts or IDirectPlay8Client::Connect.

Enumerating or Connecting from a Particular Port

Applications should almost always let DirectPlay choose the local port or ports when enumerating or connecting. However, you might want to allow the user to select a local port. This can be added to a device address using the DPNA_KEY_PORT component as shown in the following example.

```
DWORD dwUserSelectedPort; // value retrieved from user input
if (dwUserSelectedPort != 0)
{
    // User specified a port value; use it.
    hr = pDP8AddressDevice->AddComponent(DPNA_KEY_PORT, &
    else
    {
        // Let DirectPlay select; don't add port component.
    }
```

This address object is then passed to **IDirectPlay8Client::EnumHosts** or **IDirectPlay8Client::Connect** for the device address parameter.

Disabling Automatic Traversal

When the client has Microsoft Windows® Internet Connection Firewall enabled or is behind a Universal Plug and Play (UPnP) NAT device, DirectPlay will attempt to enable port forwarding for your application automatically. This asks the device to accept all packets received from the Internet on a particular port and forward them to a particular address and port inside the private network. In many cases, this is unnecessary, particularly when the server is known to be on the open Internet. You can decrease the time required by **IDirectPlay8Client::EnumHosts** or **IDirectPlay8Client::Connect** and prevent automated traversal by setting the DPNA_KEY_TRAVERSALMODE component to DPNA_TRAVERSALMODE_NONE as shown in the following example.

DWORD dwTraversalMode = DPNA_TRAVERSALMODE_NONE;

hr = pDP8AddressDevice->AddComponent(DPNA_KEY_TRAVERS

© 2003 Microsoft Corporation. All rights reserved.

Client NAT Compatibility Reference

When following the guidelines listed in the previous sections, the <u>IDirectPlay8Client</u> interface supports all Network Address Translation (NAT) device types.

© 2003 Microsoft Corporation. All rights reserved.

Using the IDirectPlay8NATResolver Interface

The <u>IDirectPlay8NATResolver</u> interface can be used in conjunction with the DPNA_KEY_NAT_RESOLVER and

DPNA_KEY_NAT_RESOLVER_USER_STRING address components to improve automated Network Address Translation (NAT) support in some cases. This section describes how to implement a NAT resolver application that is deployed on the Internet. Hosts and clients use this server to identify their public address as described in <u>Advanced NAT</u> <u>Techniques for Peer Hosts</u>, <u>Advanced NAT Techniques for Peer Clients</u>, and <u>Advanced NAT Techniques for Servers</u>.

Creating an IDirectPlay8NATResolver Interface

To prepare the NAT resolver you must first create the interface object and call <u>IDirectPlay8NATResolver::Initialize</u> to specify a callback function. This is shown in the following example.

```
hr = CoCreateInstance(CLSID_DirectPlay8NATResolver,
NULL,
CLSCTX_INPROC_SERVER,
IID_IDirectPlay8NATResolver,
(LPVOID*) &g_pDPNATResolver);
```

```
hr = g_pDPNATResolver->Initialize(NULL, DirectPlayMessageHand
```

The <u>IDirectPlay8ThreadPool</u> interface can be used to control threading for the **IDirectPlay8NATResolver** interface. If you choose to use **IDirectPlay8ThreadPool**, that interface should be initialized and configured first.

Starting a NAT Resolver

Once the IDirectPlay8NATResolver interface has been initialized, you should call IDirectPlay8NATResolver::Start to begin listening for NAT resolver queries. The device addresses passed to IDirectPlay8NATResolver::Start are similar to those passed to IDirectPlay8Peer::Host or IDirectPlay8Server::Host, except that they must only be for the Transmission Control Protocol/Internet Protocol (TCP/IP) service provider, and Internet Protocol, version 6 (IPv6) device globally unique identifiers (GUIDs) are not accepted. You can leave the device GUID unspecified and listen using all devices. You might also choose to not specify an address object at all, and Microsoft® DirectPlay® will automatically select a port for listening on all TCP/IP protocol devices. The following example shows how to optionally start listening on a particular port specified by the user.

hr = g_pDPNATResolver->Start(&pDP8AddrLocal, 1, 0);

Handling queries

The **IDirectPlay8NATResolver** interface only generates one callback to the application's message handler,

DPN_MSGID_NAT_RESOLVER_QUERY. The message contains the address of the query sender and the device on which it was received. The application can allow DirectPlay to respond to the query by returning DPN_OK from its message handler, or ignore it by returning a failure code. This is shown in the following example.

```
case DPN_MSGID_NAT_RESOLVER_QUERY:
{
   DPNMSG_NAT_RESOLVER_QUERY *pMsgNATResolverQuery
   .
   .
   if (bIgnoreQuery)
   {
    // Return failure to ignore the query.
    return DPNERR_GENERIC;
   }
   // Return OK so that DirectPlay will reply.
   .
   .
   .
   .
}
```

The **DPNMSG_NAT_RESOLVER_QUERY** structure also contains a pointer to the user string, or NULL if the string was not specified. This

string is the same as the DPNA_NAT_RESOLVER_USER_STRING component included in the querying application's device address specified to **IDirectPlay8Peer::Host**, <u>IDirectPlay8Peer::EnumHosts</u>, <u>IDirectPlay8Peer::Connect</u>, **IDirectPlay8Server::Host**, <u>IDirectPlay8Client::EnumHosts</u>, or <u>IDirectPlay8Client::Connect</u>. The user string is sent in clear text, so you should encrypt the string if it contains sensitive data.

See <u>NATResolver</u> for an example usage of this interface.

© 2003 Microsoft Corporation. All rights reserved.

Notes Regarding Firewalls and Proxies

Following the guidelines outlined for maximizing Network Address Translation (NAT) support will maximize your application's firewall and proxy support as well. A few issues specific to firewalls and proxies are described here.

Windows Internet Connection Firewall

Microsoft® DirectPlay® works with the Internet Connection Firewall capabilities of Microsoft Windows® XP. When your application begins hosting, enumerating, or connecting, the appropriate ports will automatically open and when your application shuts down, these ports will automatically close. However, the user launching the application must be a member of the administrator group for the computer so that DirectPlay has the security permissions to open the ports. If the user is not an administrator, then only <u>IDirectPlay8Client</u> and <u>IDirectPlay8Peer</u> clients are supported.

You can disable the automatic Internet Connection Firewall traversal using the DPNA_KEY_TRAVERSALMODE device address component described in <u>Basic NAT Issues for Peer Hosts</u>, <u>Basic NAT Issues for Peer Clients</u>, <u>Basic NAT Issues for Servers</u>, and <u>Basic NAT Issues for Clients</u>.

Non-Microsoft Firewall Solutions

DirectPlay cannot automatically configure non-Microsoft firewalls. However, **IDirectPlay8Client** and **IDirectPlay8Peer** clients are generally supported without user intervention, unless the application relies on DirectPlay Server (DPNSVR). See **Basic NAT Issues for Peer Clients** or **Basic NAT Issues for Clients** for more information on avoiding DPNSVR.

The user should consult the product's documentation for information on how to allow hosts to traverse a particular non-Microsoft firewall.

Microsoft Internet Security and Acceleration Server

DirectPlay works with Microsoft Internet Security and Acceleration (ISA) Server. Clients that have the ISA Firewall Client software installed should not cancel the enumeration before connecting and should pass the address objects from the DPN_MSGID_ENUM_HOSTS_RESPONSE callback as described in **Basic NAT Issues for Peer Clients** and **Basic NAT Issues for Clients**. If the enumeration is cancelled, the ISA Server can close the virtual connection established by the enumeration and the call to <u>IDirectPlay8Peer::Connect</u> or <u>IDirectPlay8Client::Connect</u> will fail.

DPNSVR should also be avoided as described in **Basic NAT Issues for Peer Hosts**, **Basic NAT Issues for Peer Clients**, **Basic NAT Issues for Servers**, and **Basic NAT Issues for Clients**.

Hosts behind an ISA Server must manually configure the proxy to redirect traffic received externally to the internal host. Refer to the ISA Server documentation for information on defining server publishing and protocol rules.

Non-Microsoft Proxies

Applications should follow the guidelines described in **Basic NAT Issues** for Peer Hosts, Basic NAT Issues for Peer Clients, Basic NAT Issues for Servers, and Basic NAT Issues for Clients.

IDirectPlay8Client and **IDirectPlay8Peer** clients are generally supported, however the user should refer to the product's documentation for possible restrictions.

Hosts behind a non-Microsoft proxy must manually configure the proxy to redirect traffic received externally to the internal host. The user should refer to the product's documentation for directions on how to enable port forwarding.

 $\ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.

NAT Troubleshooting Techniques for Developers and End Users

This topic contains techniques to help developers and end users who are trying to troubleshoot Network Address Translation (NAT) problems.

- Verify simple connectivity by attempting to access the World Wide Web.
- Use the Ping system utility to verify that packets can be sent to and from the computer's Internet Protocol (IP) gateway. Also use Ping to test the host or player that cannot join. Note that some firewalls might disallow the Internet Control Message Protocol (ICMP) echo packets used by Ping even though the two computers can communicate.
- Non-Microsoft NAT device owners should download and install the latest firmware revision available from the manufacturer.
- Microsoft® Windows® XP Internet Connection Sharing (ICS) users should ensure that the Allow other network users to control or disable the shared Internet connection box in the shared connection's properties Advanced tab is set.
- Running a CPU-intensive application such as a game on the Windows ICS computer can impact NAT performance. Clients behind the ICS computer might experience higher latency and packet loss when the computer is heavily loaded.
- Many NAT devices have a Demilitarized Zone option that forwards all packets to a particular computer. This can be used to ensure connectivity for that computer if all other traversal mechanisms fail.
- When the <u>IDirectPlay8Peer::Connect</u> method completes with the <u>DPNERR_PLAYERNOTREACHABLE</u> error code, this means that the host was contacted, but one or more peers already in the session could not communicate with the joining player. Having the players join the session in a different order can prevent this, but it might mean some of the players are behind NAT devices that are

not compatible.

© 2003 Microsoft Corporation. All rights reserved.

Optimizing Network Usage

Providing the best gaming experience normally means sending updates and other information as rapidly as possible without flooding the target with more messages than it can handle. The Microsoft® DirectPlay® protocol combined with asynchronous messaging enables you to dynamically optimize your messaging strategy to provide your users with the best possible game experience.

The bulk of your messaging will use the <u>IDirectPlay8Peer::SendTo</u>, <u>IDirectPlay8Client::Send</u>, or <u>IDirectPlay8Server::SendTo</u> methods. These methods normally work asynchronously for all message categories. They return immediately, and your message handler receives a <u>DPN_MSGID_SEND_COMPLETE</u> message when the message is actually sent. You can choose to send messages synchronously by setting the DPNSEND_SYNC flag. If you do so, the method will block until the message is actually sent.

The DirectPlay protocol's throttling mechanism guarantees that the client will not receive messages faster than they can be handled. However, the throttling protocol does not control how frequently you submit messages to the outgoing queue. You can easily end up with a large backlog of messages in your unsent message queues. You can avoid this situation by sending messages as infrequently as possible, but then you might unnecessarily degrade the user's experience. An optimal messaging strategy sends messages as fast as possible without exceeding the target's ability to handle them.

The following are tips for optimizing your messaging strategy.

• Send most if not all of your messages asynchronously. If you send

a message synchronously, the method will block until the throttling mechanism allows the message to be sent.

- Monitor the pending message queues and the network statistics. If there are few or no messages in the queue, you can increase your transmission rate. If the queues are large or growing rapidly, decrease your transmission rate and perhaps cancel some messages. See the discussion of send timeouts in <u>Send</u> <u>Prioritization</u> for more information.
- Analyze the pending message queues on a player-by-player basis. Some players might be able to receive messages at a much higher rate than others. The bulk statistics might be misleading. Consider using directed sends rather than group sends.
- Choose the appropriate category for each message. Reserve the categories with the highest overhead for the most important messages.
- Prioritize your messages, so that the most important are assured of being sent promptly and not delayed by relatively unimportant messages.
- Do not let the pending message queues grow too large. In addition to delaying the transmission of your messages, a large number of pending messages might consume significant memory resources.
- Use the timeout feature of the **Send** and **SendTo** methods to automatically clear outdated messages from the pending message queue.
- Minimize the amount of data per message. It is usually better to send frequent small messages than a smaller number of large messages.
- Do not loop tightly when checking the pending message queue. Doing so wastes CPU cycles. Instead, use a sleep period based on how long it typically takes the queue to get down to the level that it will be ready for another send.

^{© 2003} Microsoft Corporation. All rights reserved.
Testing Network Performance

Microsoft® DirectPlay® offers the DP8Sim service provider that allows you to test your DirectPlay application in a variety of network conditions, such as high latency and packet loss. The network simulator is not a replacement for testing in the real deployment environment, but it can help you predict how your application will perform.

DirectPlay also offers the <u>DP8Sim Utility</u> if you don't want to create your own testing environment. The main advantage of using the <u>IDP8SimControl</u> interface and the DP8Sim service provider in your application is that it allows you to integrate the network settings into your application's debugging user interface (UI), command line, or scripting tools instead of having to manually change the settings through the **DP8Sim Utility**.

Note The network simulator is implemented on top of the existing DirectPlay8 TCP/IP Service Provider. The settings are also applied on top of the existing network characteristics. Therefore, it is intended to be used on a high-speed local area network (LAN) where normal latency and packet loss are negligible.

Using the DP8Sim service provider

You can switch between the standard Transmission Control Protocol/Internet Protocol (TCP/IP) service provider to the DP8Sim service provider to run network simulation tests on your application. To set the network simulator as your service provider, you have three choices.

- Enumerate all service providers and then select the "DirectPlay8 TCP/IP Service Provider (Network Simulator)."
- Explicitly set the service provider globally unique identifier (GUID) with <u>IDirectPlay8Address::SetSP</u>.
- Call <u>IDirectPlay8Peer::EnumServiceProviders</u>, <u>IDirectPlay8Server::EnumServiceProviders</u>, or <u>IDirectPlay8Client::EnumServiceProviders</u> with the *pguidServiceProvider* set to CLSID_NETWORKSIMULATOR_DP8SP_TCPIP.

Note The DP8Sim service providers enumerated will have the DPNSPINFO_NETWORKSIMULATORDEVICE flag set in the **dwFlags** member of the <u>DPN_SERVICE_PROVIDER_INFO</u> structure.

The following example shows how to explicitly set the service provider to the network simulator.

hr = g_pDeviceAddress->SetSP(&CLSID_NETWORKSIMULATOR

When using the DP8Sim service provider, DirectPlay will set the DPNSPCAPS_NETWORKSIMULATOR in <u>DPN_SP_CAPS</u>.

Note If you host a DPNSVR session with the network simulator service provider, you cannot host a DPNSVR session with the Internet Protocol (IP) service provider, and vice versa. All DPNSVR sessions must be with either the IP or the network simulator service provider. This does not affect your ability to host an Internetwork Packet Exchange (IPX) session or sessions without DPNSVR.

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

Using Player Context Values

Most applications will want to associate some data with each player. However, when you receive a message that is associated to a player, you need some way to access that data quickly. Player context values are designed to provide you with an efficient way to access your player data.

Note Only the <u>IDirectPlay8Peer</u> and <u>IDirectPlay8Server</u> interfaces use player context values. They are not needed for the <u>IDirectPlay8Client</u> interface because clients use this interface to communicate only with the server, not other clients.

Defining a Player Context Value

To user player context values, you need to have a block of data on your system for each player, typically in the form of a structure. A player context value is normally an index into an array of pointers to the various players' data blocks. When you receive a message from a player, there is no need for time-consuming operations such as searching for the player's identifier (ID) in a table. The index contained in the player context value allows you to quickly obtain the necessary pointer.

You define a player context value when you handle the DPN_MSGID_CREATE_PLAYER message that notifies you that a player has been added to the game. Host's can also define a player context value when they handle the DPN_MSGID_INDICATE_CONNECT message. That player context value will be set in the subsequent DPN_MSGID_CREATE_PLAYER message. When the host processes that message, it has the option of changing the player context value. To create a player context value:

- Allocate a structure to hold the player's data.
- Add the structure pointer to your player data array.
- Assign the index of that pointer to the **pvPlayerContext** member of the message's **DPN_MSGID_CREATE_PLAYER** structure.

Microsoft® DirectPlay® does not specify how you should obtain the data to populate the structure. Each game is responsible for handling that issue in its own way.

Note The only place you can define a player context value is in a DPN_MSGID_CREATE_PLAYER or DPN_MSGID_INDICATE_CONNECT message handler. When the DPN_MSGID_CREATE_PLAYER message handler returns, the player context value is set. For each subsequent message associated with that player, the player context value will be the same value that was set by the **DPN_MSGID_CREATE_PLAYER** message handler. You can modify the contents of the associated data structure, but you cannot change the player context value itself.

Managing Player Context Data

While player context values are fairly straightforward to handle, there are a couple of issues that you need to be careful with.

The player context value provides you with a quick way to obtain a valid memory address that will presumably be accessed each time a message arrives. However, you must be careful that different parts of your application do not access the data at the same time. DirectPlay serializes messages associated with a particular player, which guarantees that you will never be handling two messages from the same player at the same time. As long as you only access the data structure from your callback message handler, you can safely access the structure. However, most applications will need to access player data outside the message handler.

If your application accesses the data outside the callback message handler, you must prevent concurrent access by providing some sort of global mechanism to lock the structure. Even if your application does not require such locking in the early stages of development, you should assume that locking will eventually be required, and build it in from the beginning. If your player context values that are indexes into an array, you should also make sure that you read and update that array safely.

Don't deallocate a player's data structure prematurely. When a player leaves the game, you will normally want to deallocate their data structure and free the associated memory. However, be careful about deallocating the structure as soon as you receive a

<u>DPN_MSGID_DESTROY_PLAYER</u> message. If your application accesses that structure outside the callback message handler, that data may still be in use when the message arrives. If you deallocate the

structure as soon as the message arrives, you may cause other parts of your application to fail.

To avoid prematurely deallocating the structure, you should not only provide an application-level locking mechanism, you should also implement some sort of reference counting. Increment this reference count when you create the structure, and every time you use it. Decrement the reference count every time you have finished with the structure, including in your **DPN_MSGID_DESTROY_PLAYER** message handler. As long as the reference count is nonzero, some part of your application is accessing the structure. Do not deallocate the structure until the reference count drops to zero.

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

Using the DirectPlay DPNSVR Application

Microsoft® Windows® allows one process per Internet Protocol (IP) or Internetwork Packet Exchange (IPX) port. Windows does not allow multiple processes to share a single port. To act as a communication host, each application must use a separate port. This restriction creates several issues, especially when doing such tasks as enumerating running games.

- Avoiding port conflicts. You must choose a port that does not conflict with other applications.
- Managing multiple communications hosts on a single system. Each instance of a host must use a different port. Client applications then have to determine which port a particular host is using.
- Avoiding ports that are already in use. If your preferred port is in use, your application must be able to use another port.

The DPNSVR application addresses these issues by acting as a forwarding service for enumeration requests. When an application begins hosting, it informs DPNSVR on which port it is running. DPNSVR listens on a well-known port and forwards any enumeration requests to all Microsoft DirectPlay® hosts on the system. Responses to enumeration requests contain the port number to which the host is actually connected.

The following diagram illustrates how DPNSVR, on a computer with two active host applications, handles an enumeration query from a remote client.

DPNSVR offers developers the following advantages.

- You can write generic enumeration routines that enumerate all the games running on a particular system.
- You can use DirectPlay to select the host's port. Client applications can use the services of DPNSVR to enumerate the running games on a well-known port, and the responses will contain the port number to which the host is actually connected.
- You need not allow for the situation where your application does not get the port it requests.
- You need not be concerned about conflicts with other applications on the system

While most applications will want to use the services of DPNSVR, there are some circumstances where you may want to disable it. Two examples are:

- You know what port you want to use, and only one instance of your application will be running on the computer.
- You want to restrict the ability of players to enumerate your session. If you disable DPNSVR, only those players who know which port your host is connected to will be able to enumerate your host.

How to Use DPNSVR

To determine whether DPNSVR is supported by your service provider, call the **GetSPCaps** methods supported by the <u>IDirectPlay8Peer</u>, <u>IDirectPlay8Client</u>, or <u>IDirectPlay8Server</u> interface. If the service provider supports DPNSVR, the DPNSPCAPS_SUPPORTSDPNSRV flag will be set in the **dwFlags** member of the returned <u>DPN_SP_CAPS</u> structure. Only IP, IPX, and the network simulator service providers currently support DPNSVR.

Note If you host a DPNSVR session with the network simulator service provider, you cannot host a DPNSVR session with the IP service provider and vice versa. All DPNSVR sessions must be with either the IP or the network simulator service provider. This does not affect your ability to host IPX session or sessions without DPNSVR.

Using DPNSVR requires no special effort, because it is selected by default. If you do not want enumeration requests forwarded to your host, you must explicitly disable DPNSVR by setting the DPNSESSION_NODPNSVR flag in the **dwFlags** member of the <u>DPN_APPLICATION_DESC</u> structure. Some additional characteristics of DPNSVR are:

- DPNSVR does not have a window but it does appear in the user's task list.
- If no applications are using DPNSVR, it shuts down after 30 seconds of inactivity. DPNSVR is automatically started up when an application begins hosting.
- You can shut down DPNSVR by running DPNSVR /kill from the command line and it will shut down immediately. However, if you have any host applications open when you shut down DPNSVR, they will no longer be able to receive enumerations on the DPNSVR port.

Note Applications can always enumerate your host if they know the port on which it is running, even if the DPNSESSION_NODPNSVR flag is set.

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

DirectPlay C/C++ Tutorials

This section contains tutorials that provide step-by-step instructions for implementing basic Microsoft® DirectPlay® functionality in a C or C++ application. Each tutorial builds on the previous tutorials and it is recommended that you complete the tutorials in the order in which they are listed. The first eight tutorials use the <u>IDirectPlay8Peer</u> interface methods. Tutorial 9 uses the <u>IDirectPlay8Client</u> and <u>IDirectPlay8Server</u> interface methods.

Tutorials

- <u>Tutorial 1: Creating a DirectPlay Object and Enumerating</u>
 <u>Service Providers</u>
- Tutorial 2: Hosting a Session
- Tutorial 3: Enumerating Hosted Sessions
- Tutorial 4: Connecting to a Session
- <u>Tutorial 5: Sending Messages to Other Peers</u>
- <u>Tutorial 6: Handling Host Migration</u>
- <u>Tutorial 7: Creating a Lobbyable Application</u>
- Tutorial 8: Direct Play Voice
- <u>Tutorial 9: Creating a DirectPlay Client/Server Session</u>
- <u>Tutorial 10: DirectPlay Thread Pool</u>

Note These tutorials are written using C++ style function calls. For instructions on writing Microsoft DirectX® function calls in a C environment, see <u>Using C to Access COM Objects</u>.

For other examples, see <u>DirectPlay C++ Samples</u>.

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

Tutorial 1: Creating a DirectPlay Object and Enumerating Service Providers

This series of tutorials demonstrate how to implement a simple peer-topeer multiplayer game using Microsoft® DirectPlay®. This tutorial outlines the initial steps: creating a DirectPlay object and enumerating the available network service providers. Subsequent tutorials build on this foundation to take you through the key steps needed to construct a simple peer-to-peer console application. Much of the information in these tutorials can also be applied to client/server applications. The complete sample code for this tutorial is included with the Microsoft DirectX® software development kit (SDK) and can be found at (SDK root)\Samples\C++\DirectPlay\Tutorials\Tut01_EnumSP.

For a general discussion of peer-to-peer applications, see <u>Peer-to-Peer</u> <u>Sessions</u>.

- <u>User's Guide</u>
- Creating a DirectPlay Peer Object
- Enumerating Service Providers
- <u>Terminating a DirectPlay Application</u>

Note The error handling code for the examples in this document has been deleted for clarity. See the tutorial sample for a complete version of the code.

User's Guide

When you run this tutorial sample, a window opens and the application displays the service providers found on your computer. To end the tutorial, click **Exit**.

Creating a DirectPlay Peer Object

One of the first things a peer-to-peer application must do is create and initialize a DirectPlay peer object (CLSID_DirectPlay8Peer). To create a DirectPlay peer object, call **CoCreateInstance**, and request a pointer to an <u>IDirectPlay8Peer</u> interface. All of your subsequent interaction with DirectPlay and your fellow players will be through this interface.

Once you have created the peer object, you must initialize it by calling the <u>IDirectPlay8Peer::Initialize</u> method. This method takes three parameters:

- A player context value
- A pointer to your callback message handling function
- A flag

The message handler is a key part of a DirectPlay application. You communicate with DirectPlay by calling methods on a DirectPlay interface. DirectPlay in turn communicates with your application by sending messages to your message handler, much like Microsoft Windows® sends messages to a window procedure. See <u>DirectPlay</u> <u>Callback Functions and Multithreading Issues</u> for details on how to implement a message handler.

The player context value is an application defined value that is returned each time a message is sent to your message handler. See <u>Using Player</u> <u>Context Values</u> for a discussion of how to handle player context values.

There is only one flag value that can be set,

DPNINITIALIZE_DISABLEPARAMVAL. Setting this flag disables parameter validation for all DirectPlay methods. While setting this flag

improves your application's performance, you should only do so with an application that has been thoroughly tested.

The following excerpt from the tutorial sample illustrates how to create and initialize a DirectPlay peer object.

```
// Initialize DirectPlay
hr = g_pDP->Initialize(NULL, DirectPlayMessageHandler, 0 );
```

Enumerating Service Providers

DirectPlay is an application programming interface (API) that allows game players to easily connect with each other and exchange messages, regardless of the underlying transport protocol. It enables you to use one or more service providers when hosting, connecting to, or enumerating sessions. To determine which service providers are available on your system, DirectPlay allows your application to enumerate your system's valid installed service providers as well as their adapters.

To enumerate your service providers, call

IDirectPlay8Peer::EnumServiceProviders. This method fills an application-supplied buffer with information about the valid service providers on your system. As you do not know the minimum size of the buffer prior to calling this method, you should call it twice. The first time, specify a zero-length buffer and the method returns the required size. Use this size value in your second method call and DirectPlay fills the buffer with valid data. See <u>Getting DirectPlay Data</u> for further information about DirectPlay enumeration.

The following excerpt from the tutorial sample illustrates how to enumerate available service providers.

DPN_SERVICE_PROVIDER_INFO* pdnSPInfo = NULL; DPN_SERVICE_PROVIDER_INFO* pdnSPInfoEnum = NULL; DWORD dwItems = 0; DWORD dwSize = 0; DWORD i;

// Determine the required buffer size
hr = g_pDP->EnumServiceProviders(NULL, NULL, NULL, &dwS

```
pdnSPInfo = (DPN_SERVICE_PROVIDER_INFO*) new BYTE[
```

```
//Fill the buffer with service provider information
hr = g_pDP->EnumServiceProviders(NULL, NULL, pdnSPInfo, &
```

```
// Print the provider descriptions
pdnSPInfoEnum = pdnSPInfo;
for (i = 0; i < dwItems; i++)
{
    DXUtil_ConvertWideStringToGenericCch( strBuf, pdnSPInfoEn
    SendMessage( GetDlgItem( g_hDlg, IDC_PROVIDERS ),
        LB_ADDSTRING, 0, (LPARAM) strBuf );
}</pre>
```

```
pdnSPInfoEnum++;
```

```
}
```

Terminating a DirectPlay Application

If a DirectPlay peer object was successfully initialized by calling IDirectPlay8Peer::Initialize, it should be closed by calling IDirectPlay8Peer::Close. You should then release all active objects before terminating the application. A DirectPlay object can be reused once it has been closed, provided that it is re-initialized by calling IDirectPlay8Peer::Initialize.

The following excerpt from the tutorial sample illustrates how to close and release DirectPlay objects.

hr = g_pDP->Close(0);

g_pDP->Release(); g_pDP = NULL;

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

Tutorial 2: Hosting a Session

This tutorial extends Tutorial 1, and discusses how to create an address object and advertise your application as a session host. Refer to <u>Tutorial</u> <u>1: Creating a DirectPlay Object and Enumerating Service Providers</u> for the initial steps in this tutorial. The complete sample code for this tutorial is included with the Microsoft® DirectX® software development kit (SDK) and can be found at (SDK

root)\Samples\C++\DirectPlay\Tutorials\Tut02_Host.

- User's Guide
- Creating an Address Object
- Hosting a Session
- <u>Terminating the Application</u>

Note The error handling code for the examples in this document has been deleted for clarity. See the tutorial sample for a complete version of the code.

User's Guide

When you run this tutorial sample, a window opens which contains information about the session status, which initially will be "Not connected to a session". To begin hosting a session, click the **Start Hosting** button. Once you are hosting a session, the session status will change to "Hosting a Session." To stop hosting, click **Exit**.

Creating an Address Object

The first step in hosting a session is to create a Microsoft DirectPlay® address object (CLSID_DirectPlay8Address) that contains the address of of the device that you host the session on. At a minimum, each address object must contain a service provider. The simplest way to specify a service provider is to call the address object's

IDirectPlay8Address::SetSP method. You can optionally specify a particular adapter. If no adapter is specified, and it is allowed, DirectPlay attempts to host the session on all adapters associated with the given service provider. See <u>DirectPlay Addressing</u> for further information about DirectPlay addressing.

The following excerpt from the tutorial sample illustrates how to create an address object for a Transmission Control Protocol/Internet Protocol (TCP/IP) protocol service provider.

Hosting a Session

Once you have created the address object, you must then create a description of your application. To do so, assign appropriate values to the members of a <u>DPN_APPLICATION_DESC</u> structure. The information that you can provide includes:

- The application's globally unique identifier (GUID)
- The maximum number of players (optional)
- The session name (optional)
- A session password (optional)
- Application-specific data (optional)

The application GUID uniquely identifies an application, not a particular session. Different applications should not have the same GUID. If you specify a maximum player count of 0, there is no limit on the number of players that can join the session.

To begin hosting a session, call <u>IDirectPlay8Peer::Host</u>. The following excerpt from the tutorial sample illustrates how host a session with an unlimited number of players.



// Set up the application description.
ZeroMemory(&dpAppDesc, sizeof(DPN_APPLICATION_DESC))
dpAppDesc.dwSize = sizeof(DPN_APPLICATION_DESC);
dpAppDesc.guidApplication = g_guidApp;

// You are now ready to host the application.

hr = g_pDP->Host(&dpAppDesc, // AppDesc &g_pDeviceAddress, 1,// Device Address NULL, NULL, // Reserved NULL, // Player Context 0); // dwFlags

Hosting a DirectPlay session causes a host player to be created with the player name and data specified when you called the <u>IDirectPlay8Peer::SetPeerInfo</u> method. When the player is created, you are notified through your DirectPlay message handler.

Terminating the Application

If a DirectPlay peer object was successfully initialized, you should first close the object by calling <u>IDirectPlay8Peer::Close</u>; then release all active objects and terminate the application. See Tutorial 1 for further discussion.

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

Tutorial 3: Enumerating Hosted Sessions

Before your application can join a peer-to-peer session, you need the address of the session host. In some cases, lobby-launched applications might give you the host address. With Microsoft® DirectPlay®, you can use that address to connect to the session. There is no need to enumerate the available hosted sessions. However, if you do not have the address of a session host, you must obtain it by enumerating the available hosted sessions. This tutorial extends Tutorial 2, and discusses how to enumerate available hosts. The complete sample code for this tutorial is included with the Microsoft DirectX® software development kit (SDK) and can be found at (SDK

root)\Samples\C++\DirectPlay\Tutorials\Tut03_EnumHosts.

- User's Guide
- Initiating Host Enumeration
- Handling Host Responses
- Terminating the Application

Refer to the preceding tutorials for a discussion of the initial steps in the process:

- <u>Tutorial 1: Creating a DirectPlay Object and Enumerating Service</u>
 <u>Providers</u>
- Tutorial 2: Hosting a Session

Note The error handling code for the examples in this document has been deleted for clarity. See the tutorial sample for a complete version of the code.

User's Guide

When you run this tutorial sample, a window opens and you can choose to either begin hosting a session or enumerate existing sessions.

To host, click the **Host...** button and the **Host New Session** window opens. Enter a session name and click **OK**. the session status will change to 'Hosting Session "YourSessionName".'

To enumerate existing sessions, enter an Internet Protocol (IP) address in the **Search Address** box and click **Search**. The application prints all the sessions found at the address in the **Detected Sessions** box. The **Search** button will turn grey while the search is taking place. If the address does not exist, a message box opens containing the error. If no session is found at the address, the **Detected Sessions** box will have a message that says "No hosts found."

You can run this sample twiceonce to host a session and once to enumerate sessions. When enumerating, enter your computer's IP address.

To end the sample, click **Exit**.

Initiating Host Enumeration

To enumerate hosts, you must basically advertise a description of the type of session that you are interested in, your target address, and the device you perform the enumeration on. You then wait for any available hosts to respond.

You describe your application by assigning values to a <u>DPN_APPLICATION_DESC</u> structure, as discussed in Tutorial 2. Tutorial 2 also describes how to create an address object to contain your address.

To start the host enumeration, pass the DPN_APPLICATION_DESC structure and your address object to IDirectPlay8Peer::EnumHosts. You can specify a host address if you want to direct your enumeration query to a particular address. Otherwise, host enumeration queries are broadcast to every address in your Transmission Control Protocol/Internet Protocol (TCP/IP) subnet.

The following excerpt from the tutorial sample illustrates how to start enumerating the available hosts at a specified address.

| ZeroMemory(&dpAppDesc, sizeof(DPN_APPLICATION_DESC)) |
|--|
| dpAppDesc.dwSize = sizeof(DPN_APPLICATION_DESC); |
| dpAppDesc.guidApplication = g_guidApp; |

| hr = g_pDP->EnumHosts(| &dpAppDesc, | <pre>// pApplicationDesc</pre> |
|------------------------|-----------------------|--------------------------------|
| g_pHostAdd | lress, // Host A | ddress |
| g_pDeviceA | ddress, // Devic | e Address |
| NULL, 0, | // pvUserEn | umData, size |
| 4, | // dwEnumCour | nt |
| 0, | // dwRetryIntery | zal |
0, // dwTimeOut NULL, // pvUserContext NULL, // pAsyncHandle DPNENUMHOSTS_SYNC); // dwFlags

Handling Host Responses

A host receives your query in the form of a <u>DPN_MSGID_ENUM_HOSTS_QUERY</u> message sent to the message handler. The associated structure includes the information that you passed to **IDirectPlay8Peer::EnumHosts**. After examining this information, the host can respond to your query by returning DPN_OK, or reject the query by returning a different value.

When a host responds affirmatively, your application's message handler receives a <u>DPN_MSGID_ENUM_HOSTS_RESPONSE</u> message. The structure associated with the message contains information describing the session.

The following excerpt from the tutorial sample illustrates how to process a **DPN_MSGID_ENUM_HOSTS_RESPONSE** message. Essentially, the message handler places the associated structure in a list, to be examined after the enumeration is finished. Refer to the tutorial sample for details.

```
HRESULT WINAPI DirectPlayMessageHandler( PVOID pvUserCon
{
.
.
.
switch( dwMessageId )
{
    case DPN_MSGID_ENUM_HOSTS_RESPONSE:
    {
        PDPNMSG_ENUM_HOSTS_RESPONSE pEnumHostsRes
        const DPN_APPLICATION_DESC* pAppDesc;
        HOST_NODE* pHostNode = NULL;
    }
}
```

```
WCHAR*
```

pEnumHostsResponseMsg = (PDPNMSG_ENUM_HOSTS_F
pAppDesc = pEnumHostsResponseMsg->pApplicationDescrip
.
.
.
// Copy the Host Address
pEnumHostsResponseMsg->pAddressSender->Duplicate(&pl
pHostNode->pAppDesc = new DPN_APPLICATION_DESC
ZeroMemory(pHostNode->pAppDesc, sizeof(DPN_APPLICATION_APP

When enumeration is finished, the **IDirectPlay8Peer::EnumHosts** method returns, and your application can decide which session to join.

Terminating the Application

If a DirectPlay peer object was successfully initialized you should first close the object by calling <u>IDirectPlay8Peer::Close</u>. Then release all active objects and terminate the application. See Tutorial 1 for further discussion.

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

Tutorial 4: Connecting to a Session

If you have found a session, you can then connect to it. This tutorial extends Tutorial 3 and discusses how to connect to a session. The complete sample code for this tutorial is included with the Microsoft® DirectX® software development kit (SDK) and can be found at (SDK root)\Samples\C++\DirectPlay\Tutorials\Tut04_Connect.

- User's Guide
- <u>Connecting to a Session</u>
- Terminating the Application

Refer to the preceding tutorials for a discussion of the initial steps in the process:

- <u>Tutorial 1: Creating a DirectPlay Object and Enumerating Service</u>
 <u>Providers</u>
- <u>Tutorial 2: Hosting a Session</u>
- <u>Tutorial 3: Enumerating Hosted Sessions</u>

Note The error handling code for the examples in this document has been deleted for clarity. See the tutorial sample for a complete version of the code.

User's Guide

When you run this tutorial sample, a window opens and you can choose to either begin hosting a session or connect to an existing session.

To host, click the **Host...** button and the **Host New Session** window opens. Enter a session name and click **OK**. the session status will change to 'Hosting Session "YourSessionName".'

To connect to an existing session, enter an Internet Protocol (IP) address in the **Search Address** box and click **Search**. The application prints all the sessions found at the address in the **Detected Sessions** box. The **Search** button will turn grey while the search is taking place. Select one of the **Detected Sessions** and click **Connect**.

You can run this sample twiceonce to host a session and once to connect. When connecting, enter your computer's IP address.

To end the sample, click **Exit**.

Connecting to a Session

Once you have selected a host, you can connect to the session. To do so, you will need:

- A <u>DPN_APPLICATION_DESC</u> structure. The structure should include the application's globally unique identifier (GUID) and a password if one is required for joining the session.
- The host address that you received when you enumerated the available hosts.
- An address object with your device address. See Creating an Address Object for a discussion of how to create an address object.

To connect to a session, call <u>IDirectPlay8Peer::Connect</u>. The host receives a <u>DPN_MSGID_INDICATE_CONNECT</u> message with your information. The host might reject the connection at this point by returning a value other than DPN_OK. In that case, if

IDirectPlay8Peer::Connect is called synchronously, as it is in the tutorial, the method returns an error value. If the connection is accepted, your Microsoft DirectPlay® message handler receives a DPN_MSGID_CONNECT_COMPLETE message. You also receive a DPN_MSGID_CREATE_PLAYER messages for yourself and each player already in the session.

The following excerpt from the tutorial sample illustrates how to connect to a selected session.

DPN_APPLICATION_DESC dpnAppDesc; IDirectPlay8Address* pHostAddress = NULL;

```
ZeroMemory(&dpnAppDesc, sizeof(DPN_APPLICATION_DESC)
dpnAppDesc.dwSize = sizeof(DPN_APPLICATION_DESC);
dpnAppDesc.guidApplication = g_guidApp;
hr = g_pDP->Connect(&dpnAppDesc, // Application Description
          pHostAddress,
                        // Host Address
          g_pDeviceAddress,
          NULL,
          NULL,
          NULL, 0,
          NULL,
          NULL,
          NULL,
          DPNCONNECT_SYNC);
  if(FAILED(hr))
    //Failed Connecting to Host
}
```

Terminating the Application

If a DirectPlay peer object was successfully initialized, you should first close the object by calling <u>IDirectPlay8Peer::Close</u>; then release all active objects and terminate the application. See Tutorial 1 for further discussion.

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

Tutorial 5: Sending Messages to Other Peers

Once you have connected to a session, you can begin game play by sending messages to the other peers in the session for whom you have received a <u>DPN_MSGID_CREATE_PLAYER</u> message. This tutorial extends Tutorial 4 and discusses how to send messages. The complete sample code for this tutorial is included with the Microsoft® DirectX® software development kit (SDK) and can be found at (SDK root)\Samples\C++\DirectPlay\Tutorials\Tut05_Send.

- User's Guide
- <u>Sending a Message</u>
- <u>Receiving a Message</u>
- <u>Terminating the Application</u>

Refer to the preceding tutorials for a discussion of the initial steps in the process:

- Tutorial 1: Creating a DirectPlay Object and Enumerating Service
 <u>Providers</u>
- Tutorial 2: Hosting a Session
- <u>Tutorial 3: Enumerating Hosted Sessions</u>
- <u>Tutorial 4: Connecting to a Session</u>

Note The error handling code for the examples in this document has been deleted for clarity. See the tutorial sample for a complete version of the code.

User's Guide

When you run this tutorial sample, a window opens and you have the choice to either **Host** or **Connect**.

If you choose **Host**:

- A window will open and you should enter a session name, then click OK. Your session status will change to 'Hosting Session "YourSessionName".'
- 2. You can now choose to send a message or disconnect. If you choose to send a message, enter a text string and click **Send**. If you choose to disconnect, click **Disconnect** and the session ends.
- 3. Click **Exit** to end the sample

If you choose Connect:

- 1. The **Connect to Session** window will open and you should enter an Internet Protocol (IP) address and click **Search**. If any sessions are found at that address, they will be listed in the **Detected Sessions** box. Select a session and click **Connect**. If the address does not exist, a message box opens with an error message.
- 2. Once connected, your session status will change to 'Connected to Session "YourSessionName".' You can now choose to send a message or disconnect. If you choose to send a message, enter a text string and click **Send**. If you choose to disconnect, click **Disconnect**.
- 3. Click **Exit** to end the sample.

You can run this sample twiceonce to host a session and once to connect. When connecting, enter your computer's IP address. When connected, you can send messages between the host and the client application.

Sending a Message

Once the session is initiated, the peers manage the progress of the game by sending a stream of messages to each other. The content of the message is up to the application but typically includes such information as game state updates or player-to-player communication. Sending a message is straightforward. Select the player you want to send the message to, package your data in a buffer, and pass the data to the target player by calling <u>IDirectPlay8Peer::SendTo</u>. You also have the option of using a single **IDirectPlay8Peer::SendTo** call to broadcast a message to all players.

The following excerpt from the tutorial sample illustrates how to send a message to every player in the session.

DPN_BUFFER_DESC dpnBuffer; WCHAR wszData[256]; dpnBuffer.pBufferData = (BYTE*) wszData; dpnBuffer.dwBufferSize = 2 * (wcslen(wszData) + 1); hr = g_pDP->SendTo(DPNID_ALL_PLAYERS_GROUP, // dpnid &dpnBuffer, // pBufferDesc 1, // cBufferDesc 0, // dwTimeOut NULL, // pvAsyncContext NULL, // pvAsyncHandle DPNSEND_SYNC | DPNSEND_NOLOOPBACK); // dwFlags

Receiving a Message

When another player sends you a message, your Microsoft DirectPlay® message handler receives a <u>DPN_MSGID_RECEIVE</u> message containing the data buffer.

The following excerpt from the tutorial sample illustrates a simple message handler for received messages.

```
//In the receiving player's message handler:
HRESULT WINAPI DirectPlayMessageHandler(void *pvUserContex
ł
switch( dwMessageId )
ł
case DPN_MSGID_RECEIVE:
ł
    case DPN_MSGID_RECEIVE:
    {
      PDPNMSG_RECEIVE
                             pMsg;
      pMsg = (PDPNMSG_RECEIVE) pMsgBuffer;
            //process message
      break;
    }
}
return(DPN_OK);
```

}

Terminating the Application

If a DirectPlay peer object was successfully initialized, you should first close the object by calling <u>IDirectPlay8Peer::Close</u>; then release all active objects and terminate the application. See Tutorial 1 for further discussion.

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

Tutorial 6: Handling Host Migration

A peer-to-peer session must have a host. This tutorial extends Tutorial 5 and discusses how to handle the situation that occurs when the host leaves the session. The complete sample code for this tutorial is included with the Microsoft® DirectX® software development kit (SDK) and can be found at (SDK

root)\Samples\C++\DirectPlay\Tutorials\Tut02_HostMigration.

- User's Guide
- Host Migration
- Terminating the Application

Refer to the preceding tutorials for a discussion of the initial steps in the process:

- <u>Tutorial 1: Creating a DirectPlay Object and Enumerating Service</u>
 <u>Providers</u>
- Tutorial 2: Hosting a Session
- <u>Tutorial 3: Enumerating Hosted Sessions</u>
- Tutorial 4: Connecting to a Session
- <u>Tutorial 5: Sending Messages to Other Peers</u>

Note The error handling code for the examples in this document has been deleted for clarity. See the tutorial sample for a complete version of the code.

User's Guide

When you run this tutorial sample, a window opens and you have the choice to either **Host** or **Connect**.

If you choose **Host**:

- 1. A window will open and you should enter a session name. Select the **Migrate Host** box to allow host migration to take place in this session. Then click **OK**. Your session status will change to 'Hosting Session "YourSessionName".'
- 2. You can now choose to send a message or disconnect. If you choose to send a message, enter a text string and click **Send**. If you choose to disconnect, click **Disconnect** and the session ends.
- 3. Click **Exit** to end the sample

If you choose Connect:

- The Connect to Session window will open and you should enter an Internet Protocol (IP) address and click Search. If any sessions are found at that address, they will be listed in the Detected Sessions box. Select a session and click Connect. If the address does not exist, a message box opens with an error message.
- 2. Once connected, your session status will change to 'Connected to Session "YourSessionName".' You can now choose to send a message or disconnect. If you choose to send a message, enter a text string and click **Send**. If you choose to disconnect, click **Disconnect**.
- 3. Click **Exit** to end the sample.

You can run this sample twiceonce to host a session and once to connect. When connecting, enter your computer's IP address. When connected, you can send messages between the host and the client application. To test host migration, close the host application. The client application's session status will change to 'Hosting Session 'YourSessionName''.'.

Host Migration

In a peer-to-peer game, there is nothing to prevent the session host from leaving before the game is finished. This situation typically occurs when the player decides to leave the game but might also result from a network problem that disconnects the player. Because a game must have a host, there are two possible ways to deal with this problem: stop the game, or choose a new host. The process of choosing a new host in the middle of a game is referred to as "host migration."

When a session host initially advertises a session, it must choose whether to allow host migration. If host migration is not allowed, the game terminates when the host leaves. To enable host migration in a session, the original session host must set the DPNSESSION_MIGRATE_HOST flag in the **dwFlag** member of the <u>DPN_APPLICATION_DESC</u> structure that is passed to the <u>IDirectPlay8Peer::Host</u>. See **Tutorial 2: Hosting a Session** for more discussion of how to host a session.

If host migration is enabled, the host can still force the session to terminate by calling the <u>IDirectPlay8Peer::TerminateSession</u> method. Otherwise, Microsoft DirectPlay® selects a new session host. When the host migrates, each remaining member's DirectPlay message handler receives a <u>DPN_MSGID_HOST_MIGRATE</u> message. The associated structure includes the identifier of the new host, which may be any remaining player in the session including you.

The following excerpt from the tutorial sample illustrates how to handle **DPN_MSGID_HOST_MIGRATE**.

HRESULT WINAPI DirectPlayMessageHandler(void *pvUserContex {

```
switch( dwMessageId )
{
 case DPN_MSGID_HOST_MIGRATE:
    {
      PDPNMSG_HOST_MIGRATE pHostMigrateMsg;
      pHostMigrateMsg = (PDPNMSG_HOST_MIGRATE) pMsgB
      // See if you are the new host.
      if( pHostMigrateMsg->dpnidNewHost == g_dpnidLocalPlaye
        //You are the New Host;
      else
        //The new host is pHostMigrateMsg->dpnidNewHost
      break;
   }
}
}
```

Terminating the Application

If a DirectPlay peer object was successfully initialized, you should first close the object by calling <u>IDirectPlay8Peer::Close</u>; then release all active objects and terminate the application. See Tutorial 1 for further discussion.

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

Tutorial 7: Creating a Lobbyable Application

A lobby is an application whose primary purpose is to enable players to meet and arrange games. It involves two separate application:

- A lobby server application. This application is implemented by the lobby operator and resides on a remote computer.
- A lobby client application. This application is implemented by the lobby operator and runs on client systems. It allows the lobby server to communicate with client systems.

In order for a game application to take full advantage of lobby launching, it must be able to communicate with the lobby client application. This tutorial extends Tutorial 6 and discusses how to use Microsoft® DirectPlay® to make a peer-to-peer application "lobbyable". Much of this information also applies to client/server applications. For further information about DirectPlay lobby support, see <u>DirectPlay Lobby</u>. The complete sample code for this tutorial is included with the Microsoft DirectX® software development kit (SDK) and can be found at (SDK root)\Samples\C++\DirectPlay\Tutorials\Tut07_LobbyLaunch.

- User's Guide
- Detecting a Lobby Launch
- Handling the DPL_MSGID_CONNECT Message
- <u>Obtaining Connection Settings</u>
- <u>Starting the Session</u>
- Terminating a Lobbied Session
- <u>Registering an Application as Lobbyable</u>

Refer to the preceding tutorials for a discussion of the initial steps in the process:

- <u>Tutorial 1: Creating a DirectPlay Object and Enumerating Service</u>
 <u>Providers</u>
- <u>Tutorial 2: Hosting a Session</u>
- Tutorial 4: Connecting to a Session
- <u>Tutorial 5: Sending Messages to Other Peers</u>
- Tutorial 6: Handling Host Migration

The error handling code for the examples in this document have been deleted for clarity. See the tutorial sample for a complete version of the code.

User's Guide

This sample has the same user interface (UI) as **Tutorial 6: Handling Host Migration**. See the <u>Users Guide</u> in **Tutorial 6: Handling Host Migration** for how to use this sample.

Detecting a Lobby Launch

Once your application is launched, you must determine whether you were launched by a lobby, or by some other means. To detect a lobby launch, you must first create a DirectPlay lobbied application object (CLSID_DirectPlay8LobbiedApplication).

Once the lobbied application object has been created, you must initialize it by calling <u>IDirectPlay8LobbiedApplication::Initialize</u>. This call takes four parameters:

- A context value. This value is included with each message that the lobbied application object sends to your message handler.
- A pointer to your lobbied application message handling function. This is a standard DirectPlay message handler, which processes messages from the lobbied application object.
- A pointer to a lobby handle.
- A flag field.

There is only one flag value that can be set,

DPNINITIALIZE_DISABLEPARAMVAL. Setting this flag disables parameter validation for methods on this instance of the DirectPlay lobbied application object. While setting this flag improves your application's performance, you should only do so with an application that has been thoroughly tested.

After the call to **IDirectPlay8LobbiedApplication::Initialize** returns, examine the lobby handle. If your application was lobby-launched, the variable is set to a valid lobby handle. If your application was not lobbylaunched, the lobby handle is set to NULL.

The following excerpt from the tutorial sample illustrates how to detect a

lobby launch.

```
#include <dplay8.h>
#include <dplobby8.h>
BOOL g_bLobbyLaunched = FALSE; // TRUE if lobby launched.
IDirectPlay8lobbiedApplication * g_pLobbyApp = NULL;
hr = CoCreateInstance(CLSID_DirectPlay8LobbiedApplication, NUL
            CLSCTX_INPROC_SERVER,
            IID_IDirectPlay8LobbiedApplication,
            (LPVOID*) &g_pLobbyApp );
// Initialize the lobbied application object.
hr = g_pLobbyApp->Initialize(NULL, LobbyAppMessageHandler, &
// Determine whether the application was lobby-launched.
if( g_hLobbyHandle != NULL )
  {
    // Attempt to host or connect to a session based on the
    // settings received from the lobby client.
  }
else
         // The application was not lobby-launched.
```

Handling the DPL_MSGID_CONNECT Message

If you were lobby launched, your lobbied application message handler receives a DPL_MSGID_CONNECT message following your call to **IDirectPlay8LobbiedApplication::Initialize**. This message carries with it a variety of information, including the identifier (ID) that you use to send messages to the lobby client. When you process this message, you should call IDirectPlay8Peer::RegisterLobby. Doing so allows applications to automatically propagate game status to the lobby client application.

The following excerpt from the tutorial sample illustrates how to handle **DPL_MSGID_CONNECT**.



Note Your message handler might receive the **DPL_MSGID_CONNECT** message before the **IDirectPlay8LobbiedApplication::Initialize** method returns. Your message handler should be prepared to handle the message appropriately.

Obtaining Connection Settings

If your application was lobby-launched, the lobby client has the option of providing you with connection settings. To determine whether you have been given connection settings, examine the <u>DPL_CONNECTION_SETTINGS</u> structure that accompanies the massage. If no connection settings are specified you must guery the

message. If no connection settings are specified, you must query the user. See **Tutorial 1** for details.

The **DPL_CONNECTION_SETTINGS** structure also provides you with a variety of other information that you will need to begin the session, including:

- A flag indicating whether you are to be the session host.
- A <u>DPN_APPLICATION_DESC</u> structure with a description of the session.
- The address of the host, unless you have been selected as host.

Starting the Session

Once you have selected your service provider, start the session much like you would a non-lobbied peer-to-peer session. You should first create a DirectPlay peer object, and call <u>IDirectPlay8Peer::SetPeerInfo</u> to establish your player's name. You should make this method call synchronous by setting the DPNOP_SYNC flag. Otherwise, there is a risk that the method might not return before you attempt to connect to the session.

If the lobby has selected you as host, you need to start hosting the session by calling <u>IDirectPlay8Peer::Host</u>. The other players typically receive your address through their lobby clients and send you connection requests. If you are not the host, get the address of the session host from the **DPL_CONNECTION_SETTINGS** structure and call the <u>IDirectPlay8Peer::Connect</u> method to connect to the session.

From this point on, the session proceeds much like a non-lobbied session. See the preceding tutorials for further details.

Terminating a Lobbied Session

You should next call <u>IDirectPlay8LobbiedApplication::Close</u> to close the connection to the lobbied application object. If a DirectPlay peer object was successfully initialized you should close the object by calling <u>IDirectPlay8Peer::Close</u>; then release all active objects and terminate the application. See **Tutorial 1** for further discussion.
Registering an Application as Lobbyable

An application must be registered before it can be properly lobby launched. You only need to register the application once, typically during your installation procedure. Registration provides DirectPlay with a variety of information about your application, including:

- The application's unique globally unique identifier (GUID)
- The application's name
- The name and path of the application's executable file
- The name and path of a launcher application (optional)

To register your application, you must create a DirectPlay lobbied application object and call

IDirectPlay8LobbiedApplication::RegisterProgram. Do not attempt to manually enter application information in the registry. Failure to use **IDirectPlay8LobbiedApplication::RegisterProgram** might make your application nonportable and incompatible with later versions of DirectPlay.

The following excerpt from the tutorial sample illustrates how to register a lobbyable application.

DPL_PROGRAM_DESC dplDesc;

ZeroMemory(&dplDesc, sizeof(DPL_PROGRAM_DESC)); dplDesc.dwSize = sizeof(DPL_PROGRAM_DESC); dplDesc.guidApplication = g_guidApp; // The application GUI dplDesc.pwszApplicationName = L"MyApplicationName"; dplDesc.pwszExecutableFilename = L"MyApp.exe"; dplDesc.pwszExecutablePath = L"C:\...\MyAppFolder"; hr = g_pLobbyApp->RegisterProgram(&dplDesc, 0);

When you uninstall a registered lobbyable application, you should unregister it. Your uninstall procedure should create a DirectPlay lobbied application object, and unregister the application by calling the <u>IDirectPlay8LobbiedApplication::UnRegisterProgram</u> method.

The following excerpt from the tutorial sample illustrates how to unregister a lobbyable application.

```
HRESULT UnRegister()
{
    HRESULT hr = S_OK;
    hr = g_pLobbyApp->UnRegisterProgram(&g_guidApp, 0);
    return hr;
}
```

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

Tutorial 8: Direct Play Voice

Microsoft® DirectPlay® Voice is a full-voice communications application programming interface (API) that uses DirectPlay for network session management and network transport. This tutorial extends the preceding tutorials to describe how to add voice communications to a peer-to-peer network application. Much of this information also applies to client/server applications. For further discussion of DirectPlay Voice, see <u>DirectPlay</u> Voice. The complete sample code for this tutorial is included with the Microsoft DirectX® software development kit (SDK) and can be found at (SDK root)\Samples\C++\DirectPlay\Tutorials\Tut08_Voice.

- User's Guide
- Preparing for a DirectPlay Voice Session
- <u>Creating a Voice Session Host</u>
- Starting a Voice Session
- Testing the Voice Setup
- <u>Creating a DirectPlay Voice Client Object</u>
- <u>Connecting to a DirectPlay Voice Client Session</u>
- Setting the Transmission Targets List
- Terminating a Voice Session

Refer to the preceding tutorials for a discussion of the initial steps in the process.

- <u>Tutorial 1: Creating a DirectPlay Object and Enumerating Service</u>
 <u>Providers</u>
- Tutorial 2: Hosting a Session
- <u>Tutorial 3: Enumerating Hosted Sessions</u>
- <u>Tutorial 4: Connecting to a Session</u>

- Tutorial 5: Sending Messages to Other Peers
- Tutorial 6: Handling Host Migration
- Tutorial 7: Creating a Lobbyable Application

Note The error handling code for the examples in this document has been deleted for clarity. See the tutorial sample for a complete version of the code.

User's Guide

When you run this tutorial sample, a window opens and you have the choice to either **Host** or **Connect**.

If you choose **Host**:

- 1. A window will open and you should enter a session name. Select the **Migrate Host** box to allow host migration to take place in this session. Then click **OK**. If you have not used your sound hardware before, DirectPlay may run the Sound Hardware Test Wizard before you can start a session. Once you are hosting, the session status will change to 'Hosting Session "YourSessionName".'
- 2. You can now choose to send a message, disconnect, or speak to other players. If you choose to send a message, enter a text string and click **Send**. To speak to other players, start talking into your microphone, voice recording starts automatically. If you choose to disconnect, click **Disconnect** and the session ends.
- 3. Click **Exit** to end the sample

If you choose Connect:

- The Connect to Session window will open and you should enter an Internet Protocol (IP) address and click Search. If any sessions are found at that address, they will be listed in the Detected Sessions box. Select a session and click Connect. If the address does not exist, a message box opens with an error message.
- Once connected, your session status will change to 'Connected to Session "YourSessionName".' You can now choose to send a message, disconnect, or speak to other players. If you choose to send a message, enter a text string and click **Send**. To speak to other players, start talking into your microphone, voice recording starts automatically. If you choose to disconnect, click **Disconnect**.

3. Click **Exit** to end the sample.

You can run this sample twiceonce to host a session and once to connect. When connecting, enter your computer's IP address. Once you start a session and have at least two players connected (the host and a client), the voice recording starts automatically. Speak into a microphone and your voice will play out on the speakers.

Preparing for a DirectPlay Voice Session

Before you can start up a DirectPlay Voice session, you must have a valid DirectPlay object connected to or hosting a session. For full details on creating, connecting and hosting DirectPlay objects, see previous tutorials.

Creating a Voice Session Host

To enable voice communications, one peer in the session must become the Voice session host. To become the host, you must create a Voice server object and obtain a pointer to its <u>IDirectPlayVoiceServer</u> interface. You use this interface to perform host-specific tasks during the Voice session. You must then call <u>IDirectPlayVoiceServer</u>.:Initialize to initialize the object. As with most DirectPlay objects, the primary purpose of initialization is to provide DirectPlay with a pointer to your Voice server callback message handler.

Note The Voice server host can be a different peer than the session host.

The following excerpt from the tutorial sample illustrates how to create and initialize a Voice server object.

Starting a Voice Session

Before any clients can connect to a Voice session, the Voice session host must start the session by calling <u>IDirectPlayVoiceServer::StartSession</u>. Once the session has been started, Voice clients can connect to the Voice session.

The <u>DVSESSIONDESC</u> structure that you pass to this method contains the information DirectPlay needs to start the session. In particular, you must specify which of several DirectPlay Voice topologies you want to use. This tutorial uses the peer-to-peer topology. For further information about this subject, see <u>DirectPlay Voice Topologies</u>.

The following excerpt from the tutorial sample illustrates how to start a peer-to-peer session using the default CODEC.

```
IDirectPlayVoiceServer* g_pVoiceServer = NULL;
.
.
.
.
ZeroMemory(&dvSessionDesc, sizeof(DVSESSIONDESC));
dvSessionDesc.dwSize = sizeof(DVSESSIONDESC);
dvSessionDesc.dwSessionType = DVSESSIONTYPE_PEER;
dvSessionDesc.dwBufferQuality = DVBUFFERQUALITY_DEFAUI
dvSessionDesc.guidCT = DPVCTGUID_DEFAULT;
dvSessionDesc.dwBufferAggressiveness = DVBUFFERAGGRESSIV
```

hr = g_pVoiceServer->StartSession(&dvSessionDesc, 0);

Testing the Voice Setup

Before connecting a Voice client to a Voice session, you must test the audio configuration. To do so, you must to create an <u>IDirectPlayVoiceTest</u> object and call <u>IDirectPlayVoiceTest::CheckAudioSetup</u>. Call the method first with the DVFLAGS_QUERYONLY set to determine whether the test has already been run.

If the test has not been run, the method returns DVERR_RUNSETUP. You should then call the function again without the DVFLAGS_QUERYONLY flag, and DirectPlay will launch the Sound Hardware Test Wizard. If the method returns a success code, you can continue. Otherwise, you must first handle the error condition. After testing is complete, release the **IDirectPlayVoiceTest** object. When the audio configuration has been tested, your client can connect to the Voice session.

The following excerpt from the tutorial sample illustrates how to test voice setup.

```
guidPlayback = DSDEVID_DefaultVoicePlayback;
```

```
guidCapture = DSDEVID_DefaultVoiceCapture;
hr = pVoiceTest->CheckAudioSetup(&guidPlayback,
                  &guidCapture,
                  NULL, DVFLAGS_QUERYONLY);
if( hr == DVERR_RUNSETUP)
  {
    // The test has not been run yet.
  hr = pVoiceTest->CheckAudioSetup(&guidPlayback,
                     &guidCapture,
                     g_hDlg, DVFLAGS_ALLOWBACK );
  }
else if( FAILED( hr))
  {
    /* Handle Errors */
  }
else
  {
    // The test has been passed, proceed.
  }
```

Creating a DirectPlay Voice Client Object

All clients that want to participate in the voice session must connect to the session, including the host. The first step in connecting to a voice session is to create and initialize a voice client object (CLSID_DirectPlayVoiceClient).

Once you have created the object, initialize it by calling <u>IDirectPlayVoiceClient::Initialize</u>. Pass this method a pointer to your voice callback message handler. This message handler receives voice-related messages from DirectPlay Voice during the voice session. You must also pass the method a pointer to an <u>IDirectPlay8Peer</u> interface. The DirectPlay object that exposes this interface must be either connected to or hosting a session before you call **IDirectPlayVoiceClient::Initialize**.

The following excerpt from the tutorial sample illustrates how to create and initialize a voice client object.

```
// Create the IDirectPlayVoiceClient object.
IDirectPlayVoiceClient* g_pVoiceClient = NULL;
.
.
.
hr = CoCreateInstance(CLSID_DirectPlayVoiceClient, NULL,
CLSCTX_INPROC_SERVER,
IID_IDirectPlayVoiceClient,
(LPVOID*) &g_pVoiceClient );
// Initialize the object.
hr = g_pVoiceClient->Initialize(g_pDP,
DirectVoiceClientMessageHandler,
NULL, 0, 0 );
```

Connecting to a DirectPlay Voice Client Session

Once the IDirectPlayVoiceClient object is created and initialized, you can connect your client to the voice session by calling IDirectPlayVoiceClient::Connect. You need to pass two structures to this method: DVSOUNDDEVICECONFIG and DVCLIENTCONFIG. The structure contains information about the sound device configuration. The structure is used to configure run-time parameters. When you have initialized the structures, connect to the voice session by passing the structures to IDirectPlayVoiceClient::Connect.

The following excerpt from the tutorial sample illustrates how to initialize a structure. In this example, the default voice capture device and default voice playback device are used for audio capture and playback. Additionally, this example enables automatic microphone selection.

ZeroMemory(&dvSoundDeviceConfig, sizeof(DVSOUNDDEVICEC dvSoundDeviceConfig.dwSize = sizeof(DVSOUNDDEVICECONFIC dvSoundDeviceConfig.dwFlags = DVSOUNDCONFIG_AUTOSELE dvSoundDeviceConfig.guidPlaybackDevice = DSDEVID_DefaultVoi dvSoundDeviceConfig.lpdsPlaybackDevice = NULL; dvSoundDeviceConfig.lpdsCaptureDevice = DSDEVID_DefaultVoic dvSoundDeviceConfig.lpdsCaptureDevice = NULL; dvSoundDeviceConfig.lpdsMainBuffer = NULL; dvSoundDeviceConfig.lpdsMainBuffer = NULL; dvSoundDeviceConfig.dwMainBufferFlags = 0; dvSoundDeviceConfig.dwMainBufferFlags = 0;

The following excerpt from the tutorial sample illustrates how to initialize a structure. In this example the system is configured for automatic voice

activation and automatic gain control.

```
dvClientConfig.dwSize = sizeof(DVCLIENTCONFIG);
dvClientConfig.dwFlags = DVCLIENTCONFIG_AUTOVOICEACT
dvClientConfig.lRecordVolume = DVRECORDVOLUME_LAST;
dvClientConfig.lPlaybackVolume = DVPLAYBACKVOLUME_DEF
dvClientConfig.dwThreshold = DVTHRESHOLD_UNUSED;
dvClientConfig.dwBufferQuality = DVBUFFERQUALITY_DEFAUI
dvClientConfig.dwBufferAggressiveness = DVBUFFERAGGRESSIV
dvClientConfig.dwNotifyPeriod = 0;
```

Setting the Transmission Targets List

Before transmitting audio to other voice session clients you must first create a transmission targets list that specifies who should receive audio transmissions. You can send audio to any combination of individual players and/or groups of players. Sending to a group allows you to reach multiple players with a single send. Each player or group is identified by a DVID value, which corresponds to equivalent DirectPlay 8 DPNID value. The transmission targets list may contain up to 64 players and/or groups. If no targets are specified, no audio data is transmitted.

If voice activation is enabled, voice transmission begins when the voice activation module detects speech activity. If voice activation is disabled voice transmission begins when valid set of targets is specified.

The following excerpt from the tutorial sample illustrates how to set a client's transmission targets to all players in the voice session.

DVID dvid = DVID_ALLPLAYERS; hr = g_pVoiceClient->SetTransmitTargets(&dvid, 1, 0);

Terminating a Voice Session

Once the session is over, clients, including the host, must shut down their voice client objects. To do so, the client must disconnect from the DirectPlay Voice session by calling <u>IDirectPlayVoiceClient::Disconnect</u>. They can then release the voice client object.

A voice session host terminates a voice session by calling . The session host must then release the DirectPlay Voice server object.

The following excerpt from the tutorial sample illustrates how to terminate a voice session and shut down the voice server and client objects.

// Shut down the voice client object. hr = g_pVoiceClient->Disconnect(0); g_pVoiceClient->Release();

// Terminate the voice session. hr = g_pVoiceServer->StopSession(0); g_pVoiceServer->Release();

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

Tutorial 9: Creating a DirectPlay Client/Server Session

This tutorial shows you how to create a simple client/server session. Client/server sessions are often used for creating large-scale multiplayer games. One advantage for using a client/server game rather than a peerto-peer game is that the majority of the processing can be done on a separate computerthe serverand therefore you do not need to rely on the power of the client's computer. For more information about client/server sessions, see <u>Client/Server Topology</u> and <u>Client/Server Sessions</u>.

Before beginning this tutorial, you should complete Tutorials 1 through 4, which describe how to create a peer-to-peer session. The steps for creating a client/server session are very similar. Rather than passing information directly from peer to peer however, a client/server session requires clients to pass information to each other indirectly, through the server. No automatic method exists for the server to pass information from one client to another. If you want this feature available to your users, you need to implement it in the server application.

The Microsoft® DirectX® software development kit (SDK) includes the complete sample code for this tutorial. The sample code can be found at (SDK root)\Samples\C++\DirectPlay\Tutorials\Tut09_ClientServer.

This article contains the following sections.

- <u>User's Guide</u>
- <u>Setting Up the Server</u>
- Setting Up a Client
- <u>Server Messages</u>
- <u>Client Messages</u>

• <u>Terminating a Client/Server Session</u>

Note The error handling code for the examples in this article is removed for clarity. For a complete version of the code, see the tutorial sample.

User's Guide

To play this tutorial sample, first run the server application. To host, click the **Host...** button and the **Host New Session** window opens. Enter a session name and click **OK**. the session status will change to 'Hosting Session "YourSessionName".'

Once the server is hosting, you can run the client application. Click **Connect...** and the **Connect to Session** window will open. Enter an Internet Protocol (IP) address in the **Search Address** box and click **Search**. The application prints all the sessions found at the address in the **Detected Sessions** box. The **Search** button will turn grey while the search is taking place. Select one of the **Detected Sessions** and click **Connect**. When connected, your session status will change to 'Connected to Session "YourSessionName".'

The server and the clients can choose to send a message or disconnect. If sending a message, enter a text string and click **Send**. If disconnecting, click **Disconnect**. Click **Exit** to end the sample.

Setting Up the Server

The server application is where the main part of the game's processing will most likely be done. The server is responsible for updating clients about the game state, for example when a player joins or leaves the session. The following topics in this section describe the tasks needed to set up a server application. Each task is described in detail below.

- 1. Create a DirectPlay Server Object
- 2. Create the Server Address Object
- 3. Begin Hosting

Create a DirectPlay Server Object

Creating a Microsoft DirectPlay® server object is similar to <u>Creating a</u> <u>DirectPlay Peer Object</u>. The only difference is that when you call **CoCreateInstance**, you pass the class identifier of a server object (CLSID_DirectPlay8Server), the identifier of the interface (IID_IDirectPlay8Server), and the address of a pointer to an <u>IDirectPlay8Server</u> interface, instead of the equivalent peer parameters. After you've created the DirectPlay server object, you can initialize it by calling the <u>IDirectPlay8Server::Initialize</u> method.

The following excerpt from the tutorial sample illustrates how to create and initialize a DirectPlay server object.

```
IDirectPlay8Server *g_pDPServer = NULL;
.
.
.
.
// Create the IDirectPlay8Server object.
hr = CoCreateInstance(CLSID_DirectPlay8Server, NULL,
```

CLSCTX_INPROC_SERVER, IID_IDirectPlay8Server, (LPVOID*) &g_pDPServer);

// Initialize DirectPlay.
hr = g_pDPServer->Initialize(NULL, DirectPlayMessageHandlerSet)

In the initialization, you pass the pointer to a <u>Implementing a Callback</u> <u>Function in DirectPlay and DirectPlay Voice</u>,

DirectPlayMessageHandlerServer, which handles messages received by the server.

Create the Server Address Object

To host a session, you must specify the address of the host device. Do this by creating an <u>IDirectPlay8Address</u> object and calling the <u>IDirectPlay8Address::SetSP</u> method. This step is identical to the step <u>Creating an Address Object</u> in <u>Tutorial 2: Hosting a Session</u>.

Begin Hosting

To begin hosting a DirectPlay server session, call the <u>IDirectPlay8Server::Host</u> method. This method takes the following parameters.

- *pdnAppDesc*: A pointer to a <u>DPN_APPLICATION_DESC</u>
- prgpDeviceInfo: A pointer to an array of IDirectPlay8Address objects
- cDeviceInfo: The number of objects in the array of IDirectPlay8Address objects
- *pdpSecurity* and *pdpCredentials*: Reserved parameters that must

be set to NULL

- *pvPlayerContext*: A pointer to the <u>player context value</u> which may be set to NULL
- *dwFlags*: An optional flag

For more information about the parameters, see

IDirectPlay8Server::Host.

The following excerpt from the tutorial sample illustrates how to begin hosting a DirectPlay server session.

WCHAR strHost[128] = {0}; DPN_APPLICATION_DESC dpAppDesc;

// Get the session name from dialog

if(IDOK != DialogBox(g_hInst, MAKEINTRESOURCE(IDD_H(
 return S_OK;

DXUtil_ConvertGenericStringToWideCch(strHost, g_strSession, 1

// Set up the Application Description.
ZeroMemory(&dpAppDesc, sizeof(DPN_APPLICATION_DESC))
dpAppDesc.dwSize = sizeof(DPN_APPLICATION_DESC);
dpAppDesc.dwFlags = DPNSESSION_CLIENT_SERVER; // Flag
dpAppDesc.guidApplication = g_guidApp; // GUID for the ap
dpAppDesc.pwszSessionName = strHost; // Session name

// Host the application.

hr = g_pDPServer->Host(&dpAppDesc, // pdnAppDesc &g_pDeviceAddress, 1, // prgpDeviceInfo, cDeviceIn NULL, NULL, // pdpSecurity, pdpCredentials NULL, // pvPlayerContext 0); // dwFlags

Setting Up a Client

The client application is responsible for handling the user interface (UI) and processing messages from the server. The following topics in this section describe the tasks needed to create a DirectPlay client application. Each task is described in detail below.

- 1. Create a DirectPlay Client Object
- 2. Create the Client Address Object
- 3. Create the Server Address Object
- 4. Enumerate the Hosts
- 5. Connect to a Host

Create a DirectPlay Client Object

Setting up a client object is similar to **Creating a DirectPlay Peer Object**. The only difference between setting up a client object and a peer object is that when you call **CoCreateInstance**, instead of passing the equivalent peer parameters, you pass the class identifier of a client object (CLSID_DirectPlay8Client), the identifier of the interface (IID_IDirectPlay8Client), and the address of a pointer to an IDirectPlay8Client interface. After you create the DirectPlay client object, you can initialize it by calling IDirectPlay8Client::Initialize.

The following excerpt from the tutorial sample illustrates how to create and initialize a DirectPlay client object.

(LPVOID*) &g_pDPClient);

// Initialize IDirectPlay8Client object.
hr = g_pDPClient->Initialize(NULL, DirectPlayMessageHandlerCl

In the initialization, you pass the pointer to a callback function, *DirectPlayMessageHandlerClient*, which handles messages the client receives. In this sample, the *pvUserContext* parameter is set to NULL. However, if you use the same message handler function for multiple interfaces, you should specify a value for *pvUserContext*. For more information, see **Using Player Context Values**.

Create the Client Device Address Object

For the client to connect to the server, you must specify the client's device address. Do this by creating an **IDirectPlay8Address** object and calling the **IDirectPlay8Address::SetSP** method. This step is identical to the step **Creating an Address Object** in **Tutorial 2: Hosting a Session**.

Create the Server Address Object

For the client to connect to the server, you must also create an **IDirectPlay8Address** object for the address of the server. This step is identical to the step **Creating an Address Object** in **Tutorial 2: Hosting a Session**.

Enumerate Hosts

If you don't know the address of the server to which you want to connect, you can enumerate all of the available servers. Follow the steps in <u>Tutorial 3: Enumerating Hosted Sessions</u> and replace all instances of the pointer to an <u>IDirectPlay8Peer</u> interface with the pointer to an **IDirectPlay8Client** interface.

Connect to a Server

To connect to a DirectPlay client/server session, follow the same process you use to connect to a peer-to-peer session except that the <u>IDirectPlay8Client::Connect</u> method does not take the *pvPlayerContext* parameter. Therefore, the call to **IDirectPlay8Client::Connect** is as follows:

hr = g_pDPClient->Connect(&dpnAppDesc, // pdnAppDesc pHostAddress, // pHostAddr g_pDeviceAddress, // pDeviceInfo NULL, // pdnSecurity NULL, // pdnCredentials NULL, 0, // pvUserConnectData, Size NULL, // pvAsyncContext NULL, // pvAsyncHandle DPNCONNECT_SYNC); // dwFlags

For more information, see <u>Tutorial 4: Connecting to a Session</u>.

Server Messages

After the server is hosting and a client is connected, the client and server can send messages to each other. If more than one client is connected, the server can send messages to a single player, a group of players, or all the players.

Sending Messages from the Server to Clients

The server can send a message to clients using the <u>IDirectPlay8Server::SendTo</u> method. The following excerpt from the client/server tutorial sample illustrates how to call the **IDirectPlay8Server::SendTo** method.

| hr = g_pDPServer->SendTo(DPNID_ALL_PLAYERS_GROU | | | |
|---|--------------------|--|--|
| &dpnBuffer | , // pBufferDesc | | |
| 1, | // cBufferDesc | | |
| 0, | // dwTimeOut | | |
| NULL, | // pvAsyncContext | | |
| NULL, | // pvAsyncHandle | | |
| DPNSEND_ | _SYNC // dwFlags | | |
| DPNSEND_NOLOOPBACK); | | | |

Setting the *dpnid* parameter to DPNID_ALL_PLAYERS_GROUP sends the message to all players connected to the session. To specify a specific player or group, set *dpnid* to the specific player ID or group ID. The *dpnBuffer* is a pointer to the <u>DPN_BUFFER_DESC</u> structure that contains the data to send. For a description of the message flags and the other parameters, see **IDirectPlay8Server::SendTo**.

Receiving Messages from Clients

Messages received from the clients are processed by the *DirectPlayMessageHandlerServer* function. The message handler function will typically take the following form.

```
HRESULT WINAPI DirectPlayMessageHandlerServer(PVOID pvUs(
    switch (dwMessageId)
    {
        case DPN_MSGID_RECEIVE:
        {
            PDPNMSG_RECEIVE pMsg;
            pMsg = (PDPNMSG_RECEIVE) pMsgBuffer;
            //process data
            break;
        }
        .
        .
        //Other cases
    }
    return hr;
}
```

When a message is received, DirectPlay generates a

DPN_MSGID_RECEIVE message. The

DirectPlayMessageHandlerServer function tells your application what to do with the data it received. In this example, the text in the message data buffer displays on the player's screen.

For other messages that you might want to include, see <u>Handling</u> <u>Client/Server Messages</u>.

Client Messages

A client can send messages to and receive messages from only the server. If a client wants to send a message to another client, the message must first be sent to the server. The server application can implement a method to forward the message to other clients.

Sending Messages to the Server

A client can send a message to the server using the <u>IDirectPlay8Client::Send</u> method. The following example from the client/server tutorial sample illustrates how to call the **IDirectPlay8Client::Send** method.

| hr = g_pDPClient->Send(8 | r = g_pDPClient->Send(&dpnBuffer, | |
|--------------------------|-----------------------------------|------------|
| 1, | // cBufferDesc | |
| 0, | // dwTimeOut | |
| NULL, | // pvAsyncContext | |
| NULL, | // pvAsyncHa | andle |
| DPNSEND_SYNC | | |
| DPNSEND_ | _NOLOOPBACK); | // dwFlags |

The *pBufferDesc* parameter is a pointer to a **DPN_BUFFER_DESC** structure that tells the application what data to send. The *dwTimeOut* parameter is set to 0, which means that the message waits in the queue until it is either sent or the connection ends. You can set *dwTimeOut* to a value so that the message is not sent unless it is sent within the specified number of milliseconds. For a description of the message flags and the other parameters, see **IDirectPlay8Client::Send**.

Receiving Messages from the Server

Messages received from the server are processed by the *DirectPlayMessageHandlerClient* function. The client message handler function takes the same form as the *DirectPlayMessageHandlerServer* function. For more information, see the example of a message handler function in the **Receiving Messages from Clients** section of <u>Server</u> <u>Messages</u>.

Terminating a Client/Server Session

If a DirectPlay client or server object was successfully initialized, you should close the object by calling <u>IDirectPlay8Client::Close</u> or <u>IDirectPlay8Server::Close</u>, and then release all active objects and terminate the application. For further discussion on closing and releasing DirectPlay objects, see <u>Tutorial 1: Creating a DirectPlay Object and</u> <u>Enumerating Service Providers</u>. When a client closes the session, the server receives the <u>DPN_MSGID_DESTROY_PLAYER</u> message but the game will continue if other players are connected. When the server closes the session, the clients receive the

<u>DPN_MSGID_TERMINATE_SESSION</u> message and the session ends. For more information about handling these messages, see **Handling Client/Server Messages**.

 $\ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

Tutorial 10: DirectPlay Thread Pool

This tutorial shows how to use the <u>IDirectPlay8ThreadPool</u> interface and avoid multithreading by taking advantage of the <u>IDirectPlay8ThreadPool::DoWork</u> method.

Some Microsoft® DirectPlay® games choose to use DirectPlay's internal worker threads to send and receive messages. These games must implement synchronization mechanisms to avoid data corruption and deadlocking.

Because multithreading is a complex issue, DirectPlay has implemented the **IDirectPlay8ThreadPool** interface to help users managed their threads. One advantage to using the **IDirectPlay8ThreadPool** interface is that you can use the <u>IDirectPlay8ThreadPool</u>::SetThreadCount method to disable all of DirectPlay's internal threads. All you need to do is call the **IDirectPlay8ThreadPool**::DoWork method regularly in your game loop and DirectPlay will perform networking tasks only during the time specified and only on the calling thread.

This tutorial uses a peer-to-peer, lobbied application, so you should review the following tutorials before beginning this one.

- <u>Tutorial 1: Creating a DirectPlay Object and Enumerating Service</u>
 <u>Providers</u>
- Tutorial 2: Hosting a Session
- <u>Tutorial 3: Enumerating Hosted Sessions</u>
- Tutorial 4: Connecting to a Session
- <u>Tutorial 5: Sending Messages to Other Peers</u>
- Tutorial 6: Handling Host Migration

• Tutorial 7: Creating a Lobbyable Application

The complete sample code for this tutorial is included with the Microsoft DirectX® software development kit (SDK) and can be found at (SDK root)\Samples\C++\DirectPlay\Tutorials\Tut10_ThreadPool.

This article contains the following sections.

- User's Guide
- <u>Creating a DirectPlay ThreadPool Object</u>
- Setting the Thread Count
- <u>Working in DoWork Mode</u>
- Terminating the Application

Note The error handling code for the examples in this document has been deleted for clarity. See the tutorial sample for a complete version of the code.
User's Guide

When you run this tutorial sample, a window opens and you have the choice to either **Host** or **Connect**.

If you choose **Host**:

- 1. A window will open and you should enter a session name. Select the **Migrate Host** box to allow host migration to take place in this session. Then click **OK**. Your session status will change to 'Hosting Session "YourSessionName".'
- 2. You can now draw pictures in **Shared Canvas**. Select a color by clicking on one of the color boxes and then hold the mouse down while you move it inside the canvas window to draw. To disconnect, click **Disconnect** and the session ends.
- 3. Click **Exit** to end the sample

If you choose Connect:

- The Connect to Session window will open and you should enter an Internet Protocol (IP) address and click Search. If any sessions are found at that address, they will be listed in the Detected Sessions box. Select a session and click Connect. If the address does not exist, a message box opens with an error message.
- Once connected, your session status will change to 'Connected to Session "YourSessionName". You can now draw pictures in Shared Canvas. Select a color by clicking on one of the color boxes and then hold the mouse down while you move it inside the canvas window to draw. To disconnect, click Disconnect.
- 3. Click **Exit** to end the sample.

You can run this sample twiceonce to host a session and once to connect. When connecting, enter your computer's IP address. When connected, you can draw pictures on the canvas shared between the host and the client application. To test host migration, close the host application. The client application's session status will change to 'Hosting Session "YourSessionName".'.

Creating a DirectPlay ThreadPool Object

The first step in using the DirectPlay thread pool is to create and initialize an **IDirectPlay8ThreadPool** object. The **IDirectPlay8ThreadPool** object must be initialized before any other DirectPlay object or DirectPlay will create its own **IDirectPlay8ThreadPool** object when <u>IDirectPlay8Peer</u> is initialized. There can be only one **IDirectPlay8ThreadPool** object per process so if DirectPlay has created its own **IDirectPlay8ThreadPool** object then a subsequent call to <u>IDirectPlay8ThreadPool</u>::Initialize will return <u>DPNERR_ALREADYINITIALIZED</u>.

To create an **IDirectPlay8ThreadPool** object, call <u>CoCreateInstance</u> passing the class identifier (CLSID_DirectPlay8ThreadPool), the identifier of the interface (IID_IDirectPlay8ThreadPool), and the address of a pointer to an **IDirectPlay8ThreadPool** object. The following snippet from the tutorial sample shows how to create and initialize an **IDirectPlay8ThreadPool** object.

```
// Initialize ThreadPool
hr = g_pThreadPool->Initialize(NULL, DirectPlayMessageHandler,
```

Setting the Thread Count

After initializing your DirectPlay objects, you should you call IDirectPlay8ThreadPool::SetThreadCount and set the thread count to zero. It is recommended that you do this before making any networking calls.

The following snippet from the tutorial sample shows how to set the thread count to zero.

// Turn off worker DirectPlay worker threads because you'll be using t
// DoWork method to synchronously handle network messages.
hr = g_pThreadPool->SetThreadCount((DWORD) -1, 0, 0);
// dwProcessorNum, dwNumThreads, dwFlags

The *dwProcessorNum* parameter is set to -1 to change the thread count for all processors.

Working in DoWork Mode

After you set the thread count to zero, you must call **IDirectPlay8ThreadPool::DoWork** to perform any DirectPlay tasks. When you call **IDirectPlay8ThreadPool::DoWork**, you specify the amount of time that the application should spend on DirectPlay tasks. DirectPlay will handle starting and closing threads to perform various tasks such as receiving messages. If DirectPlay finishes all the networking tasks before the specified time, the method will return early. If there are any tasks left when the time runs out, **DirectPlay8ThreadPool**.

IDirectPlay8ThreadPool::DoWork will return

DPNSUCCESS_PENDING.

The correct way to use **IDirectPlay8ThreadPool::DoWork** is to call it within the game loop. **IDirectPlay8ThreadPool::DoWork** will block the application until it returns, so you should limit the time your program spends on DirectPlay tasks.

The following snippet from the tutorial sample shows how to use **IDirectPlay8ThreadPool::DoWork**.

- // Here you're setting the allowed timeslice at 100 milliseconds.
- // The program will block while DoWork handles network communica
- // so you don't need to worry about thread synchronization issues as
- // you have on earlier tutorials.
- g_pThreadPool->DoWork(100, 0); // dwAllowedTimeSlice, dwFlag

^{//} Handle incoming network data

Terminating the Application

If an IDirectPlay8ThreadPool object was successfully initialized, you should call IDirectPlay8ThreadPool::Close before terminating the application. You should call IDirectPlay8ThreadPool::Close after terminating other DirectPlay objects. Before IDirectPlay8ThreadPool::Close returns, all existing DirectPlay threads will terminate and you will receive a DPN_MSGID_DESTROY_THREAD message for each thread.

The following snippet from the tutorial sample shows how to use **IDirectPlay8ThreadPool::Close**.

```
if( g_pThreadPool )
    g_pThreadPool->Close(0);
```

 $\ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

DirectPlay C++ Samples

The following sample applications demonstrate the use and capabilities of the Microsoft® DirectPlay® application programming interface for the C++ programming language. Refer to the Readme.txt file in each sample folder for details. All sample folders can be found under the SDK root directory, typically C:\mssdk.

Samples

- <u>AddressOverride</u>
- <u>ChatPeer</u>
- DataRelay
- LobbyClient
- <u>Maze</u>
- <u>NATPeer</u>
- <u>NATResolver</u>
- <u>SimpleClientServer</u>
- <u>SimplePeer</u>
- <u>StagedPeer</u>
- <u>Throttle</u>
- <u>VoiceClientServer</u>
- <u>VoiceConnect</u>
- <u>VoiceGroup</u>
- <u>VoicePosition</u>

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

AddressOverride

AddressOverride demonstrates how to programmatically provide an address to Microsoft® DirectPlay® in order to host or connect to another session on the network. The alternative is to have DirectPlay display a standard dialog box to ask the user the connection settings. Most games will set addresses directly so that they can provide their own user interface.

Path

Source: (SDK root)\Samples\C++\DirectPlay\AddressOverride

Executable: (SDK root)\Samples\C++\DirectPlay\Bin

User's Guide

The AddressOverride sample displays a dialog box that allows you to select address parameters. To use the dialog box:

- 1. Enter the player's name and session.
- 2. If you want to be a session host, select the **Host Session** check box.
- 3. Choose a service provider from the Service Provider list.
- 4. Choose an adapter from the Adapter list.
- 5. The remaining steps depend on which service provider you choose.
 - If you choose the Transmission Control Protocol/Internet Protocol (TCP/IP) service provider:
 - To join a session, select a particular host by entering its Internet Protocol (IP) address and port in the associated edit boxes. You can also have DirectPlay search the local network for available hosts by leaving the IP address and Port boxes blank.
 - To host a session, you can specify a port. If you leave the **Port** box blank, DirectPlay will select a port for you.
 - If you choose the model service provider, enter the phone number.
 - If you choose the Internetwork Packet Exchange (IPX) service provider, enter the port.
 - If you choose the serial service provider, the sample will launch the default DirectPlay dialog box that queries the user for serial settings.

Once you have finished filling in the **Address Override** dialog box, click

OK to start or join a game. This game is similar to SimplePeer. See the <u>SimplePeer</u> user's guide for more details.

Programming Notes

The AddressOverride sample is very similar to the SimplePeer sample. See the Programming Notes section of the SimplePeer sample for details. AddressOverride differs by programmatically specifying an address, rather than having DirectPlay display the default address selection dialog box.

The following list outlines how the AddressOverride sample works. When the **OK** button is pressed, the sample:

- 1. Determines whether the user wants to host or join a session from the **Host Session** check box.
- 2. Determines the service provider from the **Service Provider** list, and selects the appropriate globally unique identifier (GUID).
- 3. Creates an <u>IDirectPlay8Address</u> object called *pDeviceAddress*.
- 4. Calls that object's <u>IDirectPlay8Address::SetSP</u> method to specify the service provider.
- If the user is hosting a session, creates an IDirectPlay8Address object called *pHostAddress*. It then calls that object's IDirectPlay8Address::SetSP method to specify the service provider.
- 6. If an adapter was specified, the sample calls the *pDeviceAddress* object's <u>IDirectPlay8Address::SetDevice</u> to specify the adapter.

The sample then calls <u>IDirectPlay8Address::AddComponent</u> to complete the initialization of the address objects. The details depend on which service provider was selected, and whether the user is joining or hosting a session.

- If the *TCP/IP* service provider was selected and:
 - The user is *hosting* a session, the user can specify the port.

- If a port was specified, set the *pwszName* parameter to DPNA_KEY_PORT and call the *pDeviceAddress* object's **IDirectPlay8Address::AddComponent** to add the user's port to the address.
- The user is *joining* a session, the user can specify the host IP address and the port.
 - If a host IP address was specified, set the *pwszName* parameter to DPNA_KEY_HOSTNAME and call the *pHostAddress* object's
 IDirectPlay8Address::AddComponent to add the host's IP address to the address object.
 - If a port was specified, set the *pwszName* parameter to DPNA_KEY_PORT and call the *pHostAddress* object's **IDirectPlay8Address::AddComponent** method to add the host's port to the address.
- If the *IPX* service provider was selected and:
 - The user is *hosting* a session, the user must specify a port. Set the *pwszName* parameter to DPNA_KEY_PORT and call the *pDeviceAddress* object's
 IDirectPlay8Address::AddComponent method to add the users port to the address.
 - The user is *joining* a session, the user must specify a port. Set the *pwszName* parameter to DPNA_KEY_PORT and call the *pHostAddress* object's
 IDirectPlay8Address::AddComponent method to add the users port to the address.
- If the *modem* service provider was selected and the user is *joining* a session, the user must specify a phone number. Set the *pwszName* parameter to DPNA_KEY_PHONENUMBER and call the *pHostAddress* object's

IDirectPlay8Address::AddComponent method to add the phone number to the address.

Use the device and host address objects to connect to the session, much

like the SimplePeer sample.

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

ChatPeer

The ChatPeer sample is similar in form to <u>SimplePeer</u>. When a player hosts or connects to a session, the players can chat with each other by passing text strings.

Path

Source: (SDK root)\Samples\C++\DirectPlay\ChatPeer

Executable: (SDK root)\Samples\C++\DirectPlay\Bin

User's Guide

Enter the player's name and choose a connection type. You can choose "Wait for Lobby Connection" or choose a service provider. Use the **Multiplayer Games** dialog box to either search for an active game to join or to start a new game. When you join or start a game, you can begin immediately. Other players can join the game at any time. If host migration is on, the host player can also leave at any time, because Microsoft® DirectPlay® will automatically migrate the host session to another player.

Note If you choose the Internetwork Packet Exchange (IPX) service provider and want to have DirectPlay perform a search for the address, select the **use DPNSVR** check box.

Programming Notes

The ChatPeer sample is very similar in form to the SimplePeer sample. The ChatPeer differs by letting clients send text string to all players connected to the session. For detailed programming notes, see the Programming Notes section of the SimplePeer sample.

When **Send** is pressed, the *SendChatMessage* function does the following:

- 1. Retrieves the text string from the dialog box.
- 2. Fills an application-defined structure, GAMEMSG_CHAT. This structure has a message type identifier (ID) as the first **BYTE**. This lets the application figure out what type of application message was received. However, ChatPeer uses only one application-defined message. See <u>StagedPeer</u> for a more complex example of this process.
- 3. Fills out a <u>DPN_BUFFER_DESC</u> structure using the GAMEMSG_CHAT buffer.
- 4. Calls <u>IDirectPlay8Peer::SendTo</u> with the **DPN_BUFFER_DESC**. It passes the DPNID_ALL_PLAYERS_GROUP flag, so this message goes to everyone.

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

DataRelay

The DataRelay sample is similar to <u>SimplePeer</u> but differs by sending a packet of data with options specified in the dialog box's user interface (UI).

Path

Source: (SDK root)\Samples\C++\DirectPlay\DataRelay

Executable: (SDK root)\Samples\C++\DirectPlay\Bin

User's Guide

Host or connect to a session in the same manner as explained in SimplePeer. When the main dialog box appears, select the target, size, rate, and timeout values. Then click **Push to Send**. This will send a packet of data to the targetat the rate specified with the specified size. Using the **Connection Info** drop-down menu, specify a target on which you would like to periodically gather connection information.

Note If you choose the Internetwork Packet Exchange (IPX) service provider and want to have Microsoft® DirectPlay® perform a search for the address, select the **use DPNSVR** check box.

Programming Notes

The DataRelay sample is very similar in form to the SimplePeer sample. For detailed programming notes, see the Programming Notes section of the SimplePeer sample.

When the **Push to Send** button is clicked, a timer is created that goes off every number of milliseconds, as indicated by the UI.

When the timer goes off, the callback function calls the *SendNetworkData* function to do the following tasks.

- 1. Create an application-defined structure.
- 2. Create a GAMEMSG_DATA_NODE, which is handed off to the application worker thread. That thread processes the node and then updates the UI to show that a packet was sent.
- 3. A <u>DPN_BUFFER_DESC</u> structure is filled out, passing in a pointer to the application-defined structure created above.
- 4. The <u>IDirectPlay8Peer::SendTo</u> method is called, passing in the **DPN_BUFFER_DESC** structure.
- 5. The event g_hDPDataAvailEvent is set, telling the worker thread that there is data ready to be processed.

On receipt of the g_hDPDataAvailEvent event, the *ProcessNetDataProc* function calls the *ProcessData* function, which does the following:

- 1. Enters the critical section, g_csDataList.
- 2. Runs through the linked list, processing each node.
- 3. Calls <u>IDirectPlay8Peer::ReturnBuffer</u> so that DirectPlay can free buffer space that it passed in <u>DPN_MSGID_RECEIVE</u>.

The DirectPlayMessageHandler function handles different kinds of

messages, such as **DPN_MSGID_RECEIVE** and <u>DPN_MSGID_SEND_COMPLETE</u>.

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

LobbyClient

LobbyClient is a simple lobby client application. It displays all registered Microsoft® DirectPlay® applications on the local system. It enables the user to launch one or more of these applications using a chosen service provider. A launched lobbied application can be told to either join or host a game.

Path

Source: (SDK root)\Samples\C++\DirectPlay\LobbyClient

Executable: (SDK root)\Samples\C++\DirectPlay\Bin

User's Guide

The LobbyClient sample displays a dialog box that allows you to choose launch parameters before starting the application by clicking **Launch App**. The **Active Connections** list will display the handle to all current lobbied applications. Clicking **Send Message** will send a lobby message to the lobbied application. This is done mainly for demonstration purposes. In a more complex lobby client, you might want to use this functionality in a more meaningful way by passing a message to which the lobbied application responds.

Note If you choose the Internetwork Packet Exchange (IPX) service provider and want to have DirectPlay perform a search for the address, select the **use DPNSVR** check box.

Programming Notes

The *InitDirectPlay* function does the following:

- 1. Initializes Component Object Model (COM) with Colnitialize.
- 2. Creates an <u>IDirectPlay8Peer</u> object with **CoCreateInstance**.
- 3. Calls <u>IDirectPlay8Peer::Initialize</u> to tell the interface about our message handler.
- 4. Creates an <u>IDirectPlay8LobbyClient</u> object with **CoCreateInstance**.
- 5. Calls <u>IDirectPlay8LobbyClient::Initialize</u> to tell the interface about our lobby message handler.

The *OnInitDialog* function does the following:

- 1. Enumerates the registered lobbied applications and display them in the list box. See *EnumRegisteredApplications*.
- 2. Enumerates the DirectPlay service providers and display them in the list box. See *EnumServiceProviders*.
- 3. Whenever a new service provider is selected (and upon initialization), enumerates the service provider's adapters. See *EnumAdapters*.

When **Launch App** is clicked, the *LaunchApp* function does the following:

- 1. Fills out a <u>DPL_CONNECT_INFO</u> structure. This is complex because it contains the host and device addresses as well as the <u>DPN_APPLICATION_DESC</u>. See *LaunchApp* and *AllocAndInitConnectSettings*.
- 2. Calls <u>IDirectPlay8LobbyClient::ConnectApplication</u> passing in the DPL_CONNECT_INFO structure.
- 3. Frees the **DPL_CONNECT_INFO** structure. This is complex

because this structure has a number of DirectPlay addresses. See *FreeConnectSettings*.

The *DirectPlayLobbyMessageHandler* function can handle the following messages.

| DPL_MSGID_DISCONNECT | The lobbied application was disconnected. The <i>pDisconnectMsg>hDisconnectId</i> parameter will contain the handle of the lobbied application that was disconnected and the <i>pDisconnectMsg>hrReason</i> parameter will contain the reason. This sample pops up a message box. |
|-------------------------------|--|
| DPL_MSGID_RECEIVE | The lobbied application sent the client data. This sample does not respond to any message. |
| DPL_MSGID_SESSION_STATUS | A lobbied application has changed its status. The <i>pStatusMsg>hSender</i> parameter will be one of several predefined status codes. This sample updates the user interface (UI), showing that the lobby status has been updated. However, more complex lobby clients might take action. |
| DPL_MSGID_CONNECTION_SETTINGS | A lobbied application has changed its connection settings. This lobby client takes no action. However, more complex clients might take action. |

When **Send Message** is clicked, the *SendMsgToApp* function calls <u>IDirectPlay8LobbyClient::Send</u>. The *hConnection* parameter is set to the handle of the receiving lobbied application and the *pBuffer* parameter is set to the message buffer.

When **Disconnect** is clicked, the *DisconnectFromApp* function calls <u>IDirectPlay8LobbyClient::ReleaseApplication</u>. The *hApplication* parameter is set to the handle of the lobbied application from which to disconnect.

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

Maze

The maze sample is a Microsoft® DirectPlay® client/server application. There are two different client applicationsa console-based version and a Microsoft Direct3D® client. The Direct3D client can optionally be run as screen saver by copying Mazeclient.exe to your \winnt\system32\ folder and renaming it Mazeclient.scr. Doing so will make it a screen saver that can be detected by the display control panel application.
Path

Path: (SDK root)\Samples\C++\DirectPlay\Maze

Executable: (SDK root)\Samples\C++\DirectPlay\Bin

User's Guide

There are three executable maze samplesMazeClient, MazeConsoleClient, and MazeServer. MazeClient is a self-contained executable file but MazeServer and MazeConsoleClient must be run together. Instructions for running each sample are described below.

Note If you choose the Internetwork Packet Exchange (IPX) service provider and want to have DirectPlay perform a search for the address, select the **use DPNSVR** check box.

MazeClient

You can run MazeClient in three ways:

- Start MazeClient from the Bin folder by double-clicking the MazeClient icon. This runs the sample in *settings* mode. When you click the icon, a dialog box appears. Choose the setting you want and click **Launch**. This starts the maze sample. To quit the sample, press any key or click the mouse.
- To run MazeClient in *test* mode, at the command prompt type:

mazeclient.exe /t

In test mode, you can search and join a server game or you can choose to not connect to a server. There are few simple commands you can type when running mazeclient in test mode:

- **a** will turn autopilot on or off, but when moving about. There is no collision detection code, so you can go through walls.
- **r** will turn reflections on or off.
- To run MazeClient in *screen saver* mode, at the command prompt

type the following:

mazeclient.exe /s

This mode will autoconnect to a server using the settings from settings mode. If a server is not found, it will run without connecting to a server. It will also exit upon mouse or keyboard input.

MazeConsoleClient

Start MazeConsoleClient by double-clicking the MazeConsoleClient icon in the Bin folder or by typing at the command prompt:

MazeConsoleClient.exe

MazeConsoleClient should automatically search for a session on the local network. If it does not, at the command prompt type:

setup

When asked if you want to connect to a local server, type:

Yes

If you are already running MazeServer, MazeConsoleClient will automatically connect and start. To search for a session at a specific Internet Protocol (IP) address, type it at the command prompt, for example,

MazeConsoleClient 255.255.255

If a server is not found or the session is lost, it will exit automatically. Press CTRL+C to close the session.

MazeServer

Start MazeServer by double-clicking the MazeServer icon in the Bin folder or typing at the command prompt:

mazeserver.exe

MazeServer will automatically create a host session that clients can join. MazeServer.exe takes an optional command prompt parameter to set the size of the maze. For example, to set the maze to 16 wide and 128 high, type:

mazeserver.exe /size 16 128

The width and height are restricted to these numbers: 16, 32, 64, or 128.

Once started, the server will display a simple command prompt to control the server. Here is the list of commands.

| help | Lists simple commands you can use. |
|-------------------------|--|
| stop | Shuts down the server. |
| stats | Displays how many players are connected. |
| sr or serverreliable | Set what percentage of the packets going from the server to the client have the DPNSEND_GUARANTEED flag. |
| cr or clientreliable | Set what percentage of the packets going from every client to the server have the DPNSEND_GUARANTEED flag. |
| cu or clientupdate | Set how many milliseconds pass between updates from each client. |
| ct or clienttimeout | Sets the timeout value of packets sent by the clients. |
| st or servertimeout | Sets the timeout value of packets sent by the server. |
| ci or connectioninfo | Displays information about the connection from the server to a client. For example, ci 00300003. |
| loglevel | Set how much extra information it provides about what is happening behind the scenes. |
| | |

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

NatPeer

The NATPeer sample allows the user behind a Network Address Translation (NAT) device to specify the address and optional password of an <u>IDirectPlay8NATResolver</u> server to be used for address resolution during calls to **Host**, **EnumHosts**, and **Connect**. For more information about NATs, see <u>Network Address Translation</u>, <u>Firewalls</u>, and <u>Proxies</u>.

Path

Source: (SDK root)\Samples\C++\DirectPlay\NatPeer

Executable: (SDK root)\Samples\C++\DirectPlay\Bin

User's Guide

When you start the program, a dialog box is displayed where you can specify connection information, including an option to enable NAT address resolution. If NAT address resolution is enabled, the **Server Address** and **Password** are used by Microsoft® DirectPlay® for resolving the external addresses of players behind NATs which are not Universal Plug and Play (UPnP) compatible. If the user chooses not to be the session host, a second dialog is displayed that allows users to find and connect to active sessions.

Once in a session, the game play is similar to <u>SimplePeer</u>.

Note If you choose the Internetwork Packet Exchange (IPX) service provider and want to have DirectPlay perform a search for the address, select the **use DPNSVR** check box.

Programming Notes

If the NAT address resolution option is checked, the **Server Address** and **Password** fields are added as components of the local device address under the DPNA_KEY_NAT_RESOLVER and DPNA_KEY_NAT_RESOLVER_USER_STRING keys.

You can also specify multiple comma-delimited servers in the DPNA_KEY_NAT_RESOLVER component. Each server is tried simultaneously for speed, and the first response is used. If no server responds, the **Host**, **Connect**, or **EnumHosts** call still succeeds, but only local and UPnP connectivity information is used.

Because hosting these resolving servers require resources, you might want to prevent arbitrary players from using the server. This can be achieved with the DPNA_KEY_NAT_RESOLVER_USER_STRING address component. This value is passed directly to the resolving server for verification. The server can choose to respond as appropriate. Note that the user string is passed in clear text over the network, so if the text could contain sensitive information, you should encrypt it before sending.

For an implementation of an **IDirectPlay8NATResolver** server and more information about NAT address resolution, check the <u>NATResolver</u> sample included with the software development kit (SDK).

 $[\]ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

NATResolver

The NATResolver sample shows how the <u>IDirectPlay8NATResolver</u> interface can be used to implement address resolution for players behind Network Address Translation (NAT) devices. This interface acts as a simple server, which accepts queries and reflects the perceived address back to the caller.

Path

Source: (SDK root)\Samples\C++\DirectPlay\NATResolver

Executable: (SDK root)\Samples\C++\DirectPlay\NATResolver

User's Guide

When the sample starts, an **IDirectPlay8NATResolver** object is created that waits for queries. The sample always listens on all available Internet Protocol, version 4 (IPv4) devices for incoming queries. If **Require password** is checked, queries are first screened for the plain text password before allowing Microsoft® DirectPlay® to return the resolved address.

The dialog displays the list of addresses currently in use, as well as the number of incoming queries and outgoing responses.

Note If you choose the Internetwork Packet Exchange (IPX) service provider and want to have DirectPlay perform a search for the address, select the **use DPNSVR** check box.

Programming Notes

DirectPlay handles most of the address resolution work, interrupting only to check that incoming queries are from authorized clients. The IDirectPlay8NATResolver::Initialize method sets up the message callback for client authorization, and the IDirectPlay8NATResolver::Start method starts the server on the requested device.

DirectPlay informs your program about client queries with <u>DPN_MSGID_NAT_RESOLVER_QUERY</u> messages, which contain the client address and given password. A successful return value from this callback instructs DirectPlay to handle the address resolution. A failed return value cancels the resolution but continues executing the associated DirectPlay call. Using the password allows you to deny access to your resolution server for clients who are not part of your game.

 $\ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

SimpleClientServer

The SimpleClientServer sample is a simple client/server application. It is similar in form to SimplePeer but uses the client/server interfaces. When the user presses **Wave to other players**, the game passes a single Microsoft® DirectPlay® message to all connected players.

Path

Path: (SDK root)\Samples\C++\DirectPlay\SimpleClientServer

Executable: (SDK root)\Samples\C++\DirectPlay\Bin

User's Guide

To run the game, do the following:

- 1. Start the SimpleServer by double-clicking the SimpleServer icon in the Bin folder.
- 2. Change the port if desired, and click **Start Server**.
- 3. Start the SimpleClient by double-clicking the SimpleClient icon in the Bin folder.
- 4. Enter the player's name.
- 5. Type the Internet Protocol (IP) address, or leave it blank to search the local network.
- 6. Click Start Search.
- 7. Click **Join** when a session appears.

The game begins immediately after it has been created. Other players can join the game at any time.

Note If you choose the Internetwork Packet Exchange (IPX) service provider and want to have DirectPlay perform a search for the address, select the **use DPNSVR** check box.

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

SimplePeer

The SimplePeer sample illustrates how to implement a simple peer-topeer application. After joining or creating a session, the game begins immediately. Other players can join the session at any time.

Path

Source: (SDK root)\Samples\C++\DirectPlay\SimplePeer

Executable: (SDK root)\Samples\C++\DirectPlay\Bin

User's Guide

Enter the player's name and choose a connection type. You can choose Wait for lobby connection or choose a service provider. Use the Multiplayer Games dialog box to search for an active game to join or to start a new game. After the game has been joined or created, the game begins immediately. Other players can join the game at any time. If Host Migration is on, the host player can leave at any time because Microsoft® DirectPlay® will automatically migrate the host session to another player.

Note If you choose the Internetwork Packet Exchange (IPX) service provider and want to have DirectPlay perform a search for the address, select the **use DPNSVR** check box.

Programming Notes

The InitDirectPlay function does the following:

- 1. Initializes the Component Object Model (COM) with Colnitialize.
- 2. Creates an <u>IDirectPlay8Peer</u> object with **CoCreateInstance**.
- 3. Creates an <u>IDirectPlay8LobbiedApplication</u> object with **CoCreateInstance**.
- 4. Calls <u>IDirectPlay8Peer::Initialize</u> and passes its message handler.
- 5. Calls <u>IDirectPlay8LobbiedApplication::Initialize</u> and passes its message handler.
- 6. Checks the return value of the **IDirectPlay8LobbiedApplication::Initialize** method. If it is successful, the application is launched by a lobby client.

If the application is lobby launched, the connection settings can be obtained from the lobby client by the *ConnectUsingLobbySetting* function of the CNetConnectWizard. The CNetConnectWizard class is a helper class. It uses dialog boxes to query the user for information. The *ConnectUsingLobbySettings* function does the following:

- 1. Calls <u>IDirectPlay8LobbiedApplication::GetConnectionSettings</u> to get the connection setting from the client.
- Checks the dwFlags member of the <u>DPL_CONNECTION_SETTINGS</u> structure for the DPLCONNECTIONSETTINGS_HOST flag to see if it should host.
- 3. Calls <u>IDirectPlay8Peer::SetPeerInfo</u>.
- 4. Calls <u>IDirectPlay8Peer::Host</u> if hosting, otherwise <u>IDirectPlay8Peer::Connect</u>.
- 5. Releases the objects in **DPL_CONNECTION_SETTINGS**.

If the connection setting from the lobby client is not provided, the application calls the *DoConnectWizard* function of the CNetConnectWizard. *DoConnectWizard* does the following:

- 1. Calls <u>IDirectPlay8Peer::EnumServiceProviders</u> to enumerate service providers.
- 2. Calls the ConnectionDlgOnOk function, which displays a dialog box where the user can either choose a service provider or choose to use a lobby connection. If Wait for lobby connection is chosen, the function calls IDirectPlay8LobbiedApplication::SetAppAvailable to tell the lobby client that the application is available for connection. If a service provider is selected, the function creates a DirectPlay host and device address objects by calling CoCreateInstance. Then it calls IDirectPlay8Address::SetSP to pass service provider's globally unique identifier (GUID) into the two DirectPlay address objects.
- 3. Calls <u>IDirectPlay8Peer::EnumHosts</u> to enumerate all the games in progress on that service provider.
- 4. Processes the <u>DPN_MSGID_ENUM_HOSTS_RESPONSE</u> that arrives in the callback function.

The wizard displays the list of the current sessions and allows users to choose a game from the list or create a new one. If **Join** is clicked, the *SessionDlgJoinGame* function calls **IDirectPlay8Peer::SetPeerInfo** to set the player name and **IDirectPlay8Peer::Connect** to connect to a game. If **Create** is clicked, the *SessionDlgCreateGame* function calls **IDirectPlay8Peer::SetPeerInfo** to set the player's name and **IDirectPlay8Peer::Host** to begin hosting a game. A <u>DPN_APPLICATION_DESC</u> structure filled with information such as the game name, max player, and the application GUID is passed in the call to

Connect.

Once connected, if **Wave to other players** is clicked, the *WaveToAllPlayers* function calls <u>IDirectPlay8Peer::SendTo</u> with the *dpnid* parameter set to DPNID_ALL_PLAYERS_GROUP and the *pBufferDesc* parameter pointing to a **DWORD** containing GAME_MSGID_WAVE.

When you click **Exit**, all the interfaces are cleaned up.

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

StagedPeer

The StagedPeer sample connects players together with dialog boxes that allow players to chat and to start a new game at the same time. The host can start the game when all the players have joined and everyone is ready. The host player can also reject players or close player slots. The game is identical to the <u>SimplePeer</u> game.

When the game starts, players can press the **Wave to other players** button, which sends a simple Microsoft® DirectPlay® message to all other players.

Path

Path: (SDK root)\Samples\C++\DirectPlay\StagedPeer

Executable: (SDK root)\Samples\C++\DirectPlay\Bin

Programming Notes

The StagedPeer sample is similar in form to the SimplePeer sample. StagedPeer differs only by displaying a multiplayer stage using a dialog box. See Netstage.cpp and Netstage.h for details. For more information, see the Programming Notes section of the SimplePeer sample.

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

Throttle

The Throttle sample demonstrates how to monitor the send queue and scale the rate of network communications.

Path

Source: (SDK root)\Samples\C++\DirectPlay\Throttle

Executable: (SDK root)\Samples\C++\DirectPlay\Bin

User's Guide

Start the Throttle Server by double-clicking on the ThrottleServer.exe in the Bin folder. Wait for a moment while it connects to the network. When the server is ready to accept connection, the dialog user interface (UI) will appear. While the server is running, you can adjust the **Server Load** slider to simulate the processing load on the server. The higher the load setting, the slower the server will handle incoming messages.

After the server is running, launch the Throttle Client by double-clicking on the ThrottleClient.exe in the bin folder. The client will prompt for the host name or Internet Protocol (IP) address where the server is running. The port number is fixed. When the client is connected to the server, the server will indicate the added connection and show the amount of received data.

Note If you choose the Internetwork Packet Exchange (IPX) service provider and want to have Microsoft® DirectPlay® perform a search for the address, select the **use DPNSVR** check box.

You can adjust the **Send Interval** slider on the **Client** window to set the delay between successive calls to <u>IDirectPlay8Client::Send</u>. With the default settings, the server's receive buffer will quickly fill to capacity, and outgoing messages will fill the client's send queue. When the **Regulate Outgoing Rate** box is checked, the program will attempt to scale the number of outgoing messages to keep the queue size below the **Max Queue Size** set by the slider.

Note Adjustments to any of the client or server controls causes a corresponding change in the queue size and wait as reported in the **Outgoing Data** group box. Multiple clients can be connected simultaneously to model a typical multiplayer session.

Programming Notes

To understand why you might need to throttle outgoing data in your application, you need to understand the DirectPlay architecture and <u>DirectPlay Service Providers</u>.

DPN_SP_CAPS contains a list of capabilities and settings for service providers. This sample focuses on **dwNumThreads**, **dwBuffersPerThread**, and **dwSystemBufferSize**. During most Transmission Control Protocol/Internet Protocol (TCP/IP) sessions, DirectPlay immediately delivers messages to the receiver's system queue. The threads take messages from the system buffer and store them in their own message buffers until they can be received by the message handler.

When the thread buffers fill up and the system buffer fills up, DirectPlay won't allow any further messages to be delivered. Any messages destined for that target are then stored in the local send queue until enough space frees up on the remote computer.

You can adjust these parameters to suit your application, but increases in buffer size usually translate to increases in game lag. Therefore, it's best to leave these values for the service provider to decide and concentrate instead on <u>Optimizing Network Usage</u>.

Usually, the send queue is needed only for temporary spikes in network traffic. However, if a player continues to send messages faster than the target can receive them, the send queue will continue to grow. If no precautions are taken, any outgoing messages will take several seconds, possibly minutes, to make their way through all the queues. This will effectively end the game. One easy way to combat this is to place a timeout value on outgoing messages. You can give critical messages a higher timeout value and a different priority. In extreme circumstances, you can still run into a problem where messages consistently time out before reaching the target. For the most flexibility, you should also monitor the send queue and adjust the rate of outgoing data accordingly.

This sample takes the simple approach of blocking a portion of outgoing data, based on the current send queue size. Because the application is responsible for blocking the data, it would be possible to store a running total of blocked data and send an averaged block of data the next time space allows. That way, critical data is never lost and minor update data can be screened or combined to ease the output rate.

© 2003 Microsoft Corporation. All rights reserved.
VoiceClientServer

The VoiceClientServer sample is a simple Microsoft® DirectPlay® voicebased client/server application.

Path

Source: (SDK root)\Samples\C++\DirectPlay\VoiceClientServer

Executable: (SDK root)\Samples\C++\DirectPlay\Bin

User's Guide

Refer to the User's Guide section of the <u>SimpleClientServer</u> sample for basic information about running client/server samples.

In addition to what SimpleClientServer does, the VoiceClientServer allows you to select the voice codec you want to use and to select either a forwarding server or a mixing server. For more information about these server types, see the Microsoft DirectX® documents.

The VoiceClient allows any client to alter the playback or capture settings by clicking **Setup** once the chat session has begun.

Note If you choose the Internetwork Packet Exchange (IPX) service provider and want to have DirectPlay perform a search for the address, select the **use DPNSVR** check box.

VoiceConnect

The VoiceConnect sample shows how to network with other players to start a Microsoft® DirectPlay® Voice chat session. After joining or creating a session, the players can use a microphone to talk to one other. Other players can join the session at any time.

Path

Source: (SDK root)\Samples\C++\DirectPlay\VoiceConnect

Executable: (SDK root)\Samples\C++\DirectPlay\Bin

User's Guide

For information about how to connect, refer to the User's Guide section of the <u>SimplePeer</u> sample. After connecting, the host is presented with a dialog box asking which voice codec to use. Typical voice applications automatically select a voice codec, or present this to the user in some other fashion. When the chat session begins, any client can alter the playback or capture settings by clicking **Setup**.

Note If you choose the Internetwork Packet Exchange (IPX) service provider and want to have DirectPlay perform a search for the address, select the **use DPNSVR** check box.

Programming Notes

The VoiceConnect sample is very similar in form to the SimplePeer sample. The VoiceConnect differs by letting clients use DirectPlay Voice to talk to each other using a computer microphone. For detailed programming notes, see the Programming Notes section of the SimplePeer sample.

To create the voice functionality, the application must first initialize DirectPlay voice. This sample does this by calling CNetVoice::Init when the main dialog is initialized. CNetVoice::Init performs the following tasks.

- Calls the *VoiceSessionTestAudioSetup* function to test the audio setup with DirectPlay Voice. *VoiceSessionTestAudioSetup* does the following:
 - Creates an <u>IDirectPlayVoiceTest</u> object with **CoCreateInstance**.
 - Calls <u>IDirectPlayVoiceTest::CheckAudioSetup</u> with the DVFLAGS_QUERYONLY flag set. This will return DVERR_RUNSETUP if the setup wizard needs to be run. To run the setup wizard, call **IDirectPlayVoiceTest::CheckAudioSetup** again without the DVFLAGS_QUERYONLY flag. If it returns successfully, then the VoiceConnect sample continues, otherwise the sample ends.
 - Releases the IDirectPlayVoiceTest object.
- Calls the *VoiceSessionCreate* function to create the audio setup with DirectPlay Voice if the player is hosting. *VoiceSessionCreate* does the following:

- Creates an <u>IDirectPlayVoiceServer</u> with **CoCreateInstance**.
- Calls <u>IDirectPlayVoiceServer::Initialize</u> to register the voice server message handler. The DirectPlay interface is also passed here.
- Fills out a <u>DVSESSIONDESC</u> structure with the session type and the desired voice codec.
- Calls <u>IDirectPlayVoiceServer::StartSession</u> passing in the **DVSESSIONDESC** structure.
- If the player is either hosting or joining, the application calls the *VoiceSessionConnect* function, which uses the following steps to connect to the session.
 - Create an <u>IDirectPlayVoiceClient</u> with **CoCreateInstance**.
 - Call <u>IDirectPlayVoiceClient::Initialize</u> to register the voice client message handler. The DirectPlay interface is also passed here.
 - Fill out a <u>DVSOUNDDEVICECONFIG</u> structure with the globally unique identifiers (GUIDs) for the Microsoft DirectShow® playback and capture devices.
 - Fill out a <u>DVCLIENTCONFIG</u> structure. In this sample, this structure is filled out when with a dialog box. More complex applications might want to do this a different way. See the *VoiceConfigDlgProc* function in VoiceConnect.cpp.
 - Call <u>IDirectPlayVoiceClient::Connect</u> passing the DVSOUNDDEVICECONFIG and DVCLIENTCONFIG structures.
 - Call <u>IDirectPlayVoiceClient::SetTransmitTargets</u> to the desired default target. This sample sets the target value to DVID_ALLPLAYERS, which targets all players.

• Call <u>IDirectPlayVoiceClient::GetSoundDeviceConfig</u>. Check for the DVSOUNDCONFIG_HALFDUPLEX flag in the **dwflags** member of the **DVSOUNDDEVICECONFIG** structure returned to figure out if this client is in half-duplex mode. In half-duplex mode, the client cannot talk but can listen.

DirectPlayVoiceClientMessageHandler handles different DirectPlay messages such as:

- DVMSGID_CREATEVOICEPLAYER
- DVMSGID_DELETEVOICEPLAYER
- DVMSGID_HOSTMIGRATED
- DVMSGID_GAINFOCUS and DVMSGID_LOSTFOCUS
- DVMSGID_RECORDSTART
- DVMSGID_RECORDSTOP
- DVMSGID_PLAYERVOICESTART
- DVMSGID_PLAYERVOICESTOP

When **Exit** is clicked, the application needs to clean up DirectPlay Voice by calling CNetVoice::Free, which does the following:

- 1. Disconnects the player from the voice session. The *VoiceSessionDisconnect* function calls <u>IDirectPlayVoiceClient::Disconnect</u> and releases the **IDirectPlayVoiceClient** object.
- Destroys the voice session if this player is the host. The VoiceSessionDestroy function calls <u>IDirectPlayVoiceServer::StopSession</u> and releases the **IDirectPlayVoiceServer** object

VoiceGroup

The VoiceGroup sample shows how use Microsoft® DirectPlay® Voice to enable users to talk to a specific group of players.

Path

Source: (SDK root)\Samples\C++\DirectPlay\VoiceGroup

Executable: (SDK root)\Samples\C++\DirectPlay\Bin

User's Guide

Refer to the User's Guide section of the <u>VoiceConnect</u> sample for basic information about starting up the voice samples.

With VoiceGroup, you can click **Cycle Group Assignment** to change the local player's group assignment to **1**, **2**, **3**, **4**, or **5**, or to **unassigned**. You can also click **Cycle Target Assignment** to change the local player's voice target assignment to **1**, **2**, **3**, **4**, or **5**to talk with only one of those groups, or to **Everyone**to talk with everyone in the session.

Note If you choose the Internetwork Packet Exchange (IPX) service provider and want to have DirectPlay perform a search for the address, select the **use DPNSVR** check box.

VoicePosition

VoicePosition is similar in form to VoiceConnect. VoicePosition shows how use 3-D positioning with Microsoft® DirectPlay® Voice. It uses a simple 2-D grid to represent a playing field. Players can move around the playing field to hear the effects of 3-D spatial positioning.

Path

Source: (SDK root)\Samples\C++\DirectPlay\VoicePosition

Executable: (SDK root)\Samples\C++\DirectPlay\Bin

User's Guide

Refer to the User's Guide section of the <u>VoiceConnect</u> sample for basic information about starting up the voice samples.

With VoicePosition, you can click anywhere on the 2-D grid to move your local player on the playing field. The player always faces up, so players to the left of your player will sound off from the left speaker.

Note If you choose the Internetwork Packet Exchange (IPX) service provider and want to have DirectPlay perform a search for the address, select the **use DPNSVR** check box.

DirectPlay C/C++ Reference

Reference material for the Microsoft® DirectPlay® C/C++ application programming interface is divided into the following categories.

- Interfaces
- Functions
- <u>Callback Functions</u>
- <u>System Messages</u>
- <u>Structures</u>
- <u>Return Values</u>

Interfaces

This section contains references for methods of the Microsoft $\mbox{\sc B}$ DirectPlay $\mbox{\sc R}$ interfaces.

Interfaces

- IDirectPlay8Address
- IDirectPlay8AddressIP
- IDirectPlay8Client
- IDirectPlay8LobbiedApplication
- IDirectPlay8LobbyClient
- IDirectPlay8NATResolver
- <u>IDirectPlay8Peer</u>
- IDirectPlay8Server
- IDirectPlay8ThreadPool
- IDirectPlayNATHelp
- IDirectPlayVoiceClient
- IDirectPlayVoiceServer
- IDirectPlayVoiceTest
- IDP8SimControl

IDirectPlay8Address Interface

The **IDirectPlay8Address** interface contains generic addressing methods used to create and manipulate addresses for Microsoft® DirectPlay®. This interface is one of the interfaces available through the CLSID_DirectPlay8Address Component Object Model (COM) object. To create an object that supports this interface, use the **CoCreateInstanceEx** method for the CLSID_CLSID_DirectPlay8Address that specifies the IID_IDirectPlay8Address interface.

IDirectPlay8Address Members

| AddComponent | Adds a component to the address. If the component is part of the address, it is replaced by the new value in this call. | |
|----------------------------|--|--|
| | Values are specified in native formats when making this call. Therefore, the <i>lpvData</i> parameter should be a recast pointer to a variable that holds the data in the native format. For example, if the component is a globally unique identifier (GUID), the <i>lpvData</i> parameter should be a recast pointer to a GUID. This method validates that the predefined component types are the right format. | |
| | | |
| BuildFromDPADDRESS | Sets the current object's internal address to be the DirectPlay 8 equivalent of the specified DirectPlay 4 address. The purpose of this method is to allow lobby developed to launch games with the new DirectPlay interface using the old lobby code. | |
| | This method enumerates the address components in the specified address and adds the corresponding element to the DirectPlay 8 address. | |
| <u>BuildFromURLA</u> | Sets the object equal to the specified DirectPlay URL string. It erases the contents of the object. | |
| <u>BuildFromURLW</u> | Sets the object equal to the specified DirectPlay URL string. It erases the contents of the object. | |
| <u>Clear</u> | Resets the address object to an empty address. | |
| <u>Duplicate</u> | Creates a DirectPlay Address object that duplicates the address in this object. | |
| <u>GetComponentByIndex</u> | Retrieves information about the component at the specified index. Values for the component are retrieved in their native format. If the component key is not found, the method returns DPNERR_DOESNOTEXIST. | |

| | The value of the component is retrieved in its native format. Therefore, if the component's value is a DWORD , a DWORD is retrieved by this call. So buffer size = 4 and <i>pvBuffer</i> should be a recast PDWORD . |
|---------------------------|---|
| <u>GetComponentByName</u> | Retrieves information about the component at the specified key. Values for the component are retrieved in their native format. If the component key is not found, DPNERR_DOESNOTEXIST is returned. The value of the component is retrieved in its native format. Therefore, if the component's value is a DWORD , a DWORD is retrieved by this call. So buffer size = 4 and <i>pvBuffer</i> should be a recast PDWORD . |
| <u>GetDevice</u> | Retrieves the local device GUID in the address object. If no device is specified, this method returns DPNERR_DOESNOTEXIST. |
| <u>GetNumComponents</u> | Retrieves the number of components in the address. |
| <u>GetSP</u> | Retrieves the service provider GUID in the address object. If no service provider is specified, this method returns DPNERR_DOESNOTEXIST. |
| <u>GetURLA</u> | Retrieves the DirectPlay address URL string represented by this object (ANSI version). |
| <u>GetURLW</u> | Retrieves the DirectPlay address URL string represented by this object. |
| <u>GetUserData</u> | Retrieves the user data in the address object. If no user data exists in this address object, this method returns DPNERR_DOESNOTEXIST. |
| <u>IsEqual</u> | Compares two addresses to see if they are equal. |
| <u>SetDevice</u> | Sets the local device GUID in the address object. If a local device is specified for this address, it is overwritten by this call. |
| <u>SetEqual</u> | Sets the contents of the object it is called on to match the contents of the address object passed to the method. |
| <u>SetSP</u> | Sets the service provider GUID in the address object. If a service provider is specified for this address, it is overwritten by this call. |
| <u>SetUserData</u> | Sets the user data in the address object. If there is user data in this address, it is overwritten by this call. |

Remarks

In order to deliver messages, each participant in a multiplayer game must have a unique address. Addresses can refer either to the computer that your application is running on (*device address*), or a computer that your application needs to communicate with (*host address*).

DirectPlay represents addresses as URLs. These URLs are then encapsulated in the address object so that they can be passed to or from the DirectPlay API. In general, address URLs are strings that consist of three basic components in the following order: scheme, scheme separator, and data string.

All DirectPlay addresses use "x-directplay" as the scheme, and ":/" as the scheme separator. Using ":/" as a separator implies that the data that follows is *opaque*. In other words, the data string does not conform to any Internet standard, and should just be passed on to the receiving application without modification. All DirectPlay URLs thus have the following general form:

x-directplay:/[data string]

There are two basic approaches to handling address objects:

- Handle the data string directly, using normal string manipulation techniques.
- Use the methods exposed by **IDirectPlay8Address** to obtain or modify the individual elements of the data string.

For more information about DirectPlay addresses, see <u>DirectPlay</u> <u>Addressing</u>.

Interface Information

| Inherits from | <u>IUnknown</u> |
|---------------|-----------------|
| Header | dpaddr.h |

 $\ensuremath{\textcircled{}^{\circ}}$ 2003 Microsoft Corporation. All rights reserved.

Adds a component to the address. If the component is part of the address, it is replaced by the new value in this call.

Values are specified in native formats when making this call. Therefore, the *lpvData* parameter should be a recast pointer to a variable that holds the data in the native format. For example, if the component is a globally unique identifier (GUID), the *lpvData* parameter should be a recast pointer to a GUID.

This method validates that the predefined component types are the right format.

Syntax

```
HRESULT AddComponent(
    const WCHAR *const pwszName,
    const void *const lpvData,
    const DWORD dwDataSize,
    const DWORD dwDataType
);
```

Parameters

pwszName

[in] NULL-terminated Unicode string that contains the key for the component. You can set this to a valid string or use one of the following predefined values. DPNA_KEY_APPLICATION_INSTANCE DPNA_KEY_BAUD DPNA_KEY_DEVICE DPNA_KEY_FLOWCONTROL DPNA_KEY_HOSTNAME DPNA_KEY_NAMEINFO

DPNA_KEY_NAT_RESOLVER

DPNA_KEY_NAT_RESOLVER_USER_STRING

DPNA_KEY_PARITY

DPNA_KEY_PHONENUMBER

DPNA_KEY_PORT

DPNA_KEY_PROCESSOR

DPNA_KEY_PROGRAM

DPNA_KEY_PROVIDER

DPNA_KEY_SCOPE

DPNA_KEY_STOPBITS

DPNA_KEY_TRAVERSALMODE

lpvData

[in] Pointer to a buffer that contains the value associated with the specified key. Data should be specified in its native format.

dwDataSize

[in] Size, in bytes, of the data in the buffer located at *lpvData*. The size depends on the data type. If the size is not specified correctly, the method returns DPNERR_INVALIDPARAM. DWORD

Size = sizeof(DWORD)

GUID

```
Size = sizeof( GUID )
```

String

Size = size of the string in bytes, including the terminating NULL character.

dwDataType

[in] Data type of the value associated with this key. The data type can be one of the following:

DPNA_DATATYPE_STRING

Data is a NULL-terminated string.

DPNA_DATATYPE_STRING_ANSI

Data is a NULL-terminated ANSI string.

DPNA_DATATYPE_DWORD

Data is a **DWORD**.

DPNA_DATATYPE_GUID

Data is a GUID.

DPNA_DATATYPE_BINARY

Data is in raw binary format.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_INVALIDPOINTERPointer specified as a parameter is invalid.DPNERR_NOTALLOWEDThis function is not allowed on this object.

Remarks

For a discussion of various address components and their keys, see <u>Data Values</u>.

IDirectPlay8Address::BuildFromDPADDRESS Method

Sets the current object's internal address to be the Microsoft® DirectPlay® 8 equivalent of the specified DirectPlay 4 address. The purpose of this method is to allow lobby developers to launch games with the new DirectPlay interface using the old lobby code.

This method enumerates the address components in the specified address and adds the corresponding element to the DirectPlay 8 address.

Syntax

```
HRESULT BuildFromDPADDRESS(
LPVOID pvAddress,
DWORD dwDataSize
);
```

Parameters

pvAddress

[in] Pointer to a DirectPlay 4 address that will be converted to the DirectPlay 8 address format.

dwDataSize

[in] Size of data contained in the *pvAddress* parameter.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDADDRESSFORMATThe address format is invalid.DPNERR_INVALIDOBJECTThe DirectPlay object pointer is invalid.DPNERR_INVALIDPOINTERPointer specified as a parameter is invalid.

Remarks

This method builds a DirectPlay 8 address from a DirectPlay 4 address. This method will clear the current address of all elements before building the new address.

This method has the following limitations.

- The method cannot map the DPAID_Modem address element because DirectPlay 4 used modem names, while DirectPlay 8 uses globally unique identifiers (GUIDs) to identify modems.
- Elements of the DirectPlay 4 address that are not part of the predefined DirectPlay 4 address elements will result in an error and a return value of DPNERR_INVALIDADDRESSFORMAT. See DirectPlay 4 documentation on DirectPlay addresses for a complete list of the DirectPlay 4 address elements.
IDirectPlay8Address::BuildFromURLA Method

Sets the object equal to the specified Microsoft® DirectPlay® URL string. It erases the contents of the object.

Syntax

```
HRESULT BuildFromURLA(
    CHAR *pszSourceURL
);
```

Parameters

pszSourceURL

[in] Pointer to a NULL-terminated ANSI string that contains a properly formatted DirectPlay address.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDPOINTERPointer specified as a parameter is invalid.DPNERR_INVALIDURLSpecified string is not a valid DirectPlayURL.DPNERR_NOTALLOWEDThis function is not allowed on this object.

Remarks

The Dpaddr.h header file defines a number of standard strings that you can use to construct your URL instead using a literal string. All of the string names have the form **DPNA**_*xxx*. For example, DPNA_HEADER can be used in place of L''x-directplay:/'' for the URL header. $\ensuremath{\textcircled{\sc 0}}$ 2003 Microsoft Corporation. All rights reserved.

IDirectPlay8Address::BuildFromURLW Method

Sets the object equal to the specified Microsoft® DirectPlay® URL string. It erases the contents of the object.

Syntax

```
HRESULT BuildFromURLW(
    WCHAR *pwszSourceURL
);
```

Parameters

pwszSourceURL

[in] Pointer to a NULL-terminated Unicode string that contains a properly formatted DirectPlay address.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDPOINTERPointer specified as a parameter is invalid.DPNERR_INVALIDURLSpecified string is not a valid DirectPlayURL.DPNERR_NOTALLOWEDThis function is not allowed on this object.

Remarks

The Dpaddr.h header file defines a number of standard strings that you can use to construct your URL instead using a literal string. All of the string names have the form **DPNA**_*xxx*. For example, DPNA_HEADER can be used in place of L''x-directplay:/'' for the URL header. $\ensuremath{\textcircled{\sc 0}}$ 2003 Microsoft Corporation. All rights reserved.

IDirectPlay8Address::Clear Method

Resets the address object to an empty address.

Syntax

HRESULT Clear(VOID);

Return Value

Returns S_OK if successful, or the following error value.

DPNERR_NOTALLOWED This function is not allowed on this object.

IDirectPlay8Address::Duplicate Method

Creates a Microsoft® DirectPlay® Address object that duplicates the address in this object.

Syntax

```
HRESULT Duplicate(
    PDIRECTPLAY8ADDRESS *ppdpaNewAddress
);
```

Parameters

ppdpaNewAddress

[out] Address of a pointer to receive the <u>IDirectPlay8Address</u> pointer for the duplicate object. DirectPlay increments the reference count for this interface. You must release the interface when you no longer need it.

Return Value

Returns S_OK if successful, or the following error value.

DPNERR_GENERIC An undefined error condition occurred.

IDirectPlay8Address::GetComponentByIndex Method

Retrieves information about the component at the specified index. Values for the component are retrieved in their native format. If the component key is not found, the method returns DPNERR_DOESNOTEXIST.

The value of the component is retrieved in its native format. Therefore, if the component's value is a **DWORD**, a **DWORD** is retrieved by this call. So buffer size = 4 and *pvBuffer* should be a recast **PDWORD**.

Syntax

```
HRESULT GetComponentByIndex(
   const DWORD dwComponentID,
   WCHAR *pwszName,
   PDWORD pdwNameLen,
   void *pvBuffer,
   PDWORD pdwBufferSize,
   PDWORD pdwDataType
);
```

Parameters

dwComponentID

[in] Index of the component to retrieve. This value is zero-based and should be in the range of [0..GetNumComponents()-1]. *pwszName*

[out] Buffer to retrieve the name of the component on a successful call. To retrieve the size required, specify NULL for this parameter and 0 for the **DWORD** pointed to by *pdwNameLen*. The method returns

DPNERR_BUFFERTOOSMAL in this case.

pdwNameLen

[in, out] On input, a pointer to a **DWORD** that contains the size of the buffer, in characters including the terminating NULL

character, pointed to by *pwszName*. On output, a pointer to a **DWORD** that contains the number of characters written to the buffer, including the terminating NULL character, on success and on failure, the number of characters required, including the terminating NULL character, to store this value.

pvBuffer

[out] Buffer to retrieve the data stored in the value of the component. To retrieve the size required, specify NULL for this parameter and 0 for the **DWORD** pointed to by *pdwBufferSize*. The method returns DPNERR_BUFFERTOOSMALL in this case.

pdwBufferSize

[in, out] On input, a pointer to a **DWORD** containing the size of the buffer, in bytes, pointed to by *pvBuffer*. On output, a pointer to a **DWORD** that contains the number of bytes written to the buffer on success and on failure, the number of bytes required to store the data.

pdwDataType

[out] **DWORD** pointed to by this parameter that is set to the type of data that is stored in this component. This can be one of the following:

DPNA_DATATYPE_STRING

Data is a NULL-terminated string.

DPNA_DATATYPE_DWORD

Data is a **DWORD**.

DPNA_DATATYPE_GUID

Data is a globally unique identifier (GUID).

DPNA_DATATYPE_BINARY

Data is raw binary.

Return Value

Returns S_OK if successful, or one of the following error values.

| DPNERR_BUFFERTOOSMALL | The supplied buffer is not large enough to contain the requested data. |
|-----------------------|--|
| DPNERR_DOESNOTEXIST | Requested element is not part of the address. |
| DPNERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |

 $\ensuremath{\textcircled{\sc 0}}$ 2003 Microsoft Corporation. All rights reserved.

IDirectPlay8Address::GetComponentByName Method

Retrieves information about the component at the specified key. Values for the component are retrieved in their native format. If the component key is not found, DPNERR_DOESNOTEXIST is returned.

The value of the component is retrieved in its native format. Therefore, if the component's value is a **DWORD**, a **DWORD** is retrieved by this call. So buffer size = 4 and *pvBuffer* should be a recast **PDWORD**.

Syntax

```
HRESULT GetComponentByName(
   const WCHAR *const pwszName,
   void *pvBuffer,
   PDWORD pdwBufferSize,
   PDWORD pdwDataType
);
```

Parameters

pwszName

[in] String specifying the name of the component you want to retrieve.

pvBuffer

[out] Buffer to retrieve the data stored in the value of the component. To retrieve the size required, specify NULL for this parameter and 0 for the **DWORD** pointed to by *pdwBufferSize*. The method returns DPNERR_BUFFERTOOSMALL in this case.

pdwBufferSize

[in, out] On input, a pointer to a **DWORD** that contains the size of the buffer, in bytes, pointed to by *pvBuffer*. On output, a pointer to a **DWORD** that contains the number of bytes written to the buffer on success and on failure, the number of bytes required to store the data.

pdwDataType

[out] **DWORD** pointed to by this parameter that is set to the type of data that is stored in this component. This can be one of the following:

DPNA_DATATYPE_STRING

Data is a NULL-terminated string.

DPNA_DATATYPE_DWORD

Data is a **DWORD**.

DPNA_DATATYPE_GUID

Data is a globally unique identifier (GUID).

DPNA_DATATYPE_BINARY

Data is raw binary.

Return Value

Returns S_OK if successful, or one of the following error values.

| | The supplied buffer is not large enough to contain the requested data |
|------------------------|--|
| DPINERR_DUFFERIOUSMALL | The supplied buller is not large enough to contain the requested data. |
| DPNERR_DOESNOTEXIST | Requested element is not part of the address. |
| DPNERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |

IDirectPlay8Address::GetDevice Method

Retrieves the local device globally unique identifier (GUID) in the address object. If no device is specified, this method returns DPNERR_DOESNOTEXIST.

Syntax

```
HRESULT GetDevice(
    GUID *pguidDevice
);
```

Parameters

pguidDevice

[out] Pointer to a GUID to receive the device in the address object.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_DOESNOTEXISTRequested element is not part of the address.DPNERR_INVALIDPOINTERPointer specified as a parameter is invalid.

IDirectPlay8Address::GetNumComponents Method

Retrieves the number of components in the address.

Syntax

```
HRESULT GetNumComponents(
    PDWORD pdwNumComponents
);
```

Parameters

```
pdwNumComponents
[out] Pointer to a DWORD to receive the number of components
in this address object.
```

Return Value

Returns S_OK if successful, or the following error value.

DPNERR_INVALIDPOINTER Pointer specified as a parameter is invalid.

IDirectPlay8Address::GetSP Method

Retrieves the service provider globally unique identifier (GUID) in the address object. If no service provider is specified, this method returns DPNERR_DOESNOTEXIST.

Syntax

```
HRESULT GetSP(
    GUID *pguidSP
);
```

Parameters

pguidSP

[out] Pointer to a GUID to receive the service provider in the address object.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_DOESNOTEXISTRequested element is not part of the address.DPNERR_INVALIDPOINTERPointer specified as a parameter is invalid.

IDirectPlay8Address::GetURLA Method

Retrieves the Microsoft® DirectPlay® address URL string represented by this object (ANSI version).

Syntax

```
HRESULT GetURLA(
CHAR *pszURL,
PDWORD pdwNumChars
);
```

Parameters

pszURL

[out] Address of a pointer to receive the URL represented by this object. This parameter can be NULL if *pdwNumChars* points to a **DWORD** containing 0.

pdwNumChars

[in, out] Pointer to a **DWORD** that contains the number of characters the specified buffer can hold, including the terminating NULL character. On success this value contains the number of characters written to the specified buffer, including the terminating NULL character. On failure this value contains the number of characters, including the terminating NULL character, required to hold the URL and the method returns DPNERR BUFFERTOOSMALL.

Return Value

Returns S_OK if successful, or one of the following error values.

| DPNERR_BUFFERTOOSMALL | The supplied buffer is not large enough to contain the requested data. |
|-----------------------|--|
| DPNERR_GENERIC | An undefined error condition occurred. |
| DPNERR_INVALIDURL | Specified string is not a valid DirectPlayURL. |
| DPNERR_OUTOFMEMORY | There is insufficient memory to perform the requested operation. |

 $\ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.

IDirectPlay8Address::GetURLW Method

Retrieves the Microsoft® DirectPlay® address URL string represented by this object.

Syntax

```
HRESULT GetURLW(
WCHAR *pwszURL,
PDWORD pdwNumChars
);
```

Parameters

pwszURL

[out] Address of a pointer to receive the URL represented by this object. This parameter can be NULL if *pdwNumChars* points to a **DWORD** containing 0.

pdwNumChars

[in, out] Pointer to a **DWORD** that contains the number of characters the specified buffer can hold, including the terminating NULL character. On success this value contains the number of characters written to the specified buffer, including the terminating NULL character. On failure this value contains the number of characters, including the terminating NULL character, required to hold the URL and the method returns DPNERR_BUFFERTOOSMALL.

Return Value

Returns S_OK if successful, or one of the following error values.

| DPNERR_BUFFERTOOSMALL | The supplied buffer is not large enough to contain the requested data. |
|-----------------------|--|
| DPNERR_GENERIC | An undefined error condition occurred. |
| DPNERR_INVALIDURL | Specified string is not a valid DirectPlayURL. |
| DPNERR_OUTOFMEMORY | There is insufficient memory to perform the requested operation. |

IDirectPlay8Address::GetUserData Method

Retrieves the user data in the address object. If no user data exists in this address object, this method returns DPNERR_DOESNOTEXIST.

Syntax

```
HRESULT GetUserData(
    void *pvUserData,
    PDWORD pdwBufferSize
);
```

Parameters

pvUserData

[out] Pointer to a buffer to receive the user data from this address. To retrieve the required size, set this parameter to NULL and the **DWORD** in *pdwBufferSize*to 0.

pdwBufferSize

[in, out] Size, in bytes, of the buffer pointed to by *pvUserData*. If *pvUserData* is NULL, this parameter must point to a **DWORD** containing 0. On output, the contained **DWORD** is set to the number of bytes written to the buffer. On failure, this contains the number of bytes required to retrieve the user data and the method returns DPNERR_BUFFERTOOSMALL.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_DOESNOTEXISTRequested element is not part of the address.DPNERR_INVALIDPOINTERPointer specified as a parameter is invalid.DPNERR_BUFFERTOOSMALLThe supplied buffer is not large enough to contain the requested data.

IDirectPlay8Address::IsEqual Method

Compares two addresses to see if they are equal.

Syntax

```
HRESULT IsEqual(
    PDIRECTPLAY8ADDRESS pdpaAddress
);
```

Parameters

pdpaAddress [in] Address to compare to the address contained within the object.

Return Value

If the addresses are equal DPNSUCCESS_EQUAL is returned. If the addresses are not equal DPNSUCCESS_NOTEQUAL is returned.

If the method fails, one of the following error values may be returned.

DPNERR_INVALIDADDRESSFORMATThe address format is invalid.DPNERR_INVALIDOBJECTThe Microsoft® DirectPlay® object pointer is invalid.DPNERR_INVALIDPOINTERPointer specified as a parameter is invalid.

Remarks

This method checks the contents of the address specified by the *pdpaAddress* parameter and compares it to the address contained within the object this method was called on. This method does not affect the contents of either address.

 $\ensuremath{\textcircled{\sc 0}}$ 2003 Microsoft Corporation. All rights reserved.
IDirectPlay8Address::SetDevice Method

Sets the local device globally unique identifier (GUID) in the address object. If a local device is specified for this address, it is overwritten by this call.

Syntax

```
HRESULT SetDevice(
    const GUID *const pguidDevice
);
```

Parameters

pguidDevice [in] Pointer to a GUID of the local device.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDPOINTERPointer specified as a parameter is invalid.DPNERR_NOTALLOWEDThis function is not allowed on this object.

IDirectPlay8Address::SetEqual Method

Sets the contents of the object it is called on to match the contents of the address object passed to the method.

Syntax

```
HRESULT SetEqual(
    PDIRECTPLAY8ADDRESS pdpaAddress
);
```

Parameters

pdpaAddress

[in] Pointer to a <u>IDirectPlay8Address</u> object that this object will be set to.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDADDRESSFORMATThe address format is invalid.DPNERR_INVALIDOBJECTThe Microsoft® DirectPlay® object pointer is invalid.DPNERR_INVALIDPOINTERPointer specified as a parameter is invalid.

IDirectPlay8Address::SetSP Method

Sets the service provider globally unique identifier (GUID) in the address object. If a service provider is specified for this address, it is overwritten by this call.

Syntax

```
HRESULT SetSP(
    const GUID *const pguidSP
);
```

Parameters

pguidSP [in] Pointer to the service provider GUID. This can be one of the following predefined values. CLSID_DP8SP_TCPIP Internet Protocol (IP) service provider CLSID_NETWORKSIMULATOR_DP8SP_TCPIP Network simulator IP service providers CLSID_DP8SP_SERIAL Serial service provider CLSID_DP8SP_MODEM Modem service provider CLSID_DP8SP_IPX IPX service provider CLSID_DP8SP_BLUETOOTH Bluetooth Service Provider

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDPOINTERPointer specified as a parameter is invalid.DPNERR_NOTALLOWEDThis function is not allowed on this object.

IDirectPlay8Address::SetUserData Method

Sets the user data in the address object. If there is user data in this address, it is overwritten by this call.

Syntax

```
HRESULT SetUserData(
    const void *const pvUserData,
    const DWORD dwDataSize
);
```

Parameters

pvUserData

[in] Pointer to a buffer that contains the data to place in the user data section of the address. Set to NULL to clear the user data. *dwDataSize*

[in] Size, in bytes, of the data in *pvUserData*. If *pvUserData* is NULL, this must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDPOINTERPointer specified as a parameter is invalid.DPNERR_NOTALLOWEDThis function is not allowed on this object.

IDirectPlay8AddressIP Interface

The **IDirectPlay8AddressIP** interface is available through the CLSID_DirectPlay8Address Component Object Model (COM) object. This interface is used for Internet Protocol (IP) provider-specific addressing services.

IDirectPlay8AddressIP Members

| BuildAddress | Builds a remote Microsoft® DirectPlay® IP address from a host name and a port. |
|--------------------------|---|
| BuildFromSockAddr | Builds a remote DirectPlay IP address from a valid SOCKADDR structure. |
| <u>BuildLocalAddress</u> | Builds a local DirectPlay IP address from a device and port. |
| <u>GetAddress</u> | Retrieves the remote address information from a remote DirectPlay IP address. |
| <u>GetLocalAddress</u> | Retrieves the local address information from a DirectPlay IP address. |
| <u>GetSockAddress</u> | Retrieves a list of SOCKADDR structures describing the addresses represented by this object. |

Interface Information

| Inherits from | <u>IUnknown</u> |
|---------------------------|----------------------------|
| Header | dpaddr.h |
| Minimum operating systems | Windows 98, Pocket PC 2002 |

IDirectPlay8AddressIP::BuildAddress Method

Builds a remote Microsoft® DirectPlay® Internet Protocol (IP) address from a host name and a port.

Syntax

```
HRESULT BuildAddress(
    const WCHAR *const wszAddress,
    const USHORT usPort
);
```

Parameters

wszAddress

[in] Remote host address can be a dotted Internet addressfor example, 127.0.0.1or a valid host namefor example, example.microsoft.com.

usPort

[in] Port on the remote host to which to connect.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_INVALIDPOINTERPointer specified as a parameter is invalid.DPNERR_NOTALLOWEDThis function is not allowed on this object.

Remarks

The result of a successful call is a valid remote address with the following elements.

• DPNA_KEY_PROVIDER = CLSID_DP8SP_TCPIP

- DPNA_KEY_HOSTNAME = specified host name
- DPNA_KEY_PORT = specified port

All addressing information contained in the object before the call is erased.

Note The DPNSVR is a DirectPlay feature that allows multiple processes to share a single port for enumeration. Do not use the DPNA_DPNSVR_PORT flag when constructing a device address, or when making a connection. This flag should only be used for enumerations. If you do not add a port element to the enumeration address, the port represented by the flag will be automatically added to that address. See <u>Using the DirectPlay DPNSVR Application</u> for a further discussion of DPNSVR.

IDirectPlay8AddressIP::BuildFromSockAddr Method

Builds a remote Microsoft® DirectPlay® Internet Protocol (IP) address from a valid **SOCKADDR** structure.

Syntax

```
HRESULT BuildFromSockAddr(
    const SOCKADDR *const pSockAddr
);
```

Parameters

pSockAddr

[in] Valid User Datagram Protocol (UDP) address specified in **SOCKADDR** form.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_INVALIDPOINTERPointer specified as a parameter is invalid.DPNERR_NOTALLOWEDThis function is not allowed on this object.

Remarks

The **SOCKADDR** structure must specify an IP address. If the address is not in the correct format, DPNERR_INVALIDPARAM is returned.

The result of a successful call is a valid remote address with the following elements.

- DPNA_KEY_PROVIDER = CLSID_DP8SP_TCPIP
- DPNA_KEY_HOSTNAME = specified host name
- DPNA_KEY_PORT = specified port

All addressing information contained in the object before the call is erased.

IDirectPlay8AddressIP::BuildLocalAddress Method

Builds a local Microsoft® DirectPlay® Internet Protocol (IP) address from a device and port.

Syntax

```
HRESULT BuildLocalAddress(
    const GUID *const pguidAdapter,
    const USHORT usPort
);
```

Parameters

pguidAdapter

[in] Local device identifier to host on. *usPort*

[in] Port on the local device to host on. This value can be set to 0 to allow DirectPlay to automatically select the port.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_INVALIDPOINTERPointer specified as a parameter is invalid.DPNERR_NOTALLOWEDThis function is not allowed on this object.

Remarks

The result of a successful call is a valid remote address with the following elements.

• DPNA_KEY_PROVIDER = CLSID_DP8SP_TCPIP

- DPNA_KEY_DEVICE= specified device
- DPNA_KEY_PORT = specified port

All addressing information contained in the object before the call is erased.

Note The DPNSVR is a DirectPlay feature that allows multiple processes to share a single port for enumeration. Do not use the DPNA_DPNSVR_PORT flag when constructing a device address, or when making a connection. This flag should only be used for enumerations. If you do not add a port element to the enumeration address, the port represented by the flag will be automatically added to that address. See <u>Using the DirectPlay DPNSVR Application</u> for a further discussion of DPNSVR.

IDirectPlay8AddressIP::GetAddress Method

Retrieves the remote address information from a remote Microsoft® DirectPlay® Internet Protocol (IP) address.

Syntax

```
HRESULT GetAddress(
WCHAR *wszAddress,
PDWORD pdwAddressLength,
USHORT *psPort
);
```

Parameters

wszAddress

[out] Pointer to a buffer to receive the host name. This parameter can be NULL to retrieve the required size. *pdwAddressLength*

[in, out] Size, in characters, of the buffer specified in *wszAddress*, including the terminating NULL character. On success, this parameter contains the number of characters, including the terminating NULL character, written to the specified buffer. On failure, this parameter contains the number of characters, including the terminating NULL character, required to retrieve the host name.

psPort

[out] Pointer to a **USHORT** to contain the port specified in this local address.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_BUFFERTOOSMALLThe supplied buffer is not large enough to contain the requested data.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.

Remarks

To succeed, the contained address must have at least the following elements.

- DPNA_KEY_PROVIDER
- DPNA_KEY_HOSTNAME
- DPNA_KEY_PORT = specified port

IDirectPlay8AddressIP::GetLocalAddress Method

Retrieves the local address information from a Microsoft® DirectPlay® Internet Protocol (IP) address.

Syntax

```
HRESULT GetLocalAddress(
GUID *pguidAdapter,
USHORT *pusPort
);
```

Parameters

pguidAdapter

[out] Pointer to a globally unique identifier (GUID) to retrieve the GUID of the local device specified in this address.

pusPort

[out] Pointer to a **USHORT** to contain the port specified in this local address.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_INVALIDPOINTERPointer specified as a parameter is invalid.

Remarks

To succeed, the contained address must have at least the following elements.

• DPNA_KEY_PROVIDER

- DPNA_KEY_DEVICE
- DPNA_KEY_PORT

 $\ensuremath{\textcircled{}^{\circ}}$ 2003 Microsoft Corporation. All rights reserved.

IDirectPlay8AddressIP::GetSockAddress Method

Retrieves a list of **SOCKADDR** structures describing the addresses represented by this object.

Syntax

```
HRESULT GetSockAddress(
    SOCKADDR *psockAddress,
    PDWORD pdwAddressBufferSize
);
```

Parameters

psockAddress

[out] Pointer to buffer to retrieve the array of **SOCKADDR** structures. There is one **SOCKADDR** structure for each address the host resolves to.

pdwAddressBufferSize

[in, out] Size, in bytes, of the buffer specified in *psockAddress*. On success, this parameter contains the number of bytes written to the specified buffer. On failure, this parameter contains the number of bytes required to retrieve the array of **SOCKADDR** structures. You can divide the value of this parameter by the size of the **SOCKADDR** structure to determine the number of items present in the returned array.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_BUFFERTOOSMALLThe supplied buffer is not large enough to contain the requested data.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_INVALIDPOINTERPointer specified as a parameter is invalid.

Remarks

If the host name specified in the object requires a Domain Name System (DNS) lookup, it is performed. Therefore, this method may block while the DNS is queried. It is also possible for a host name to resolve to multiple addresses.

To succeed, the contained address must have at least the following elements.

- DPNA_KEY_PROVIDER
- DPNA_KEY_HOSTNAME
- DPNA_KEY_PORT = specified port

IDirectPlay8Client Interface

Applications use the methods of the **IDirectPlay8Client** interface to create and manage client applications for client/server sessions.

IDirectPlay8Client Members

| CancelAsyncOperation | Cancels asynchronous requests. |
|---------------------------|---|
| <u>Close</u> | Closes the open connection to a session and uninitializes the IDirectPlay8Client object. This method must be called on any object that is successfully initialized with a call to the <u>IDirectPlay8Client::Initialize</u> method. |
| <u>Connect</u> | Establishes the connection to the server. |
| EnumHosts | Enumerates applications that host Microsoft® DirectPlay® games. |
| EnumServiceProviders | Enumerates the registered service providers available to the application. |
| <u>GetApplicationDesc</u> | Retrieves the full application description for the connected application. |
| <u>GetCaps</u> | Retrieves the <u>DPN_CAPS</u> or <u>DPN_CAPS_EX</u> structure for the current interface. |
| <u>GetConnectionInfo</u> | Retrieves statistical information about the connection between the local client and the server. |
| <u>GetSendQueueInfo</u> | Used by the application to monitor the size of the send queue. DirectPlay does not send messages faster than the receiving computer can process them. As a result, if the sending computer is sending faster than the receiver can receive, messages accumulate in the sender's queue. If the application registers that the send queue is growing too large, it should decrease the rate that messages are sent. |
| <u>GetServerAddress</u> | Retrieves the address of the server for the session. |
| <u>GetServerInfo</u> | Retrieves the data set for the server set by the call to the IDirectPlay8Server::SetServerInfo method. |
| <u>GetSPCaps</u> | Retrieves the <u>DPN_SP_CAPS</u> structure for the specified service provider. |
| <u>Initialize</u> | Registers an entry point in the client's code that receives the messages from the IDirectPlay8Client interface and from the server. This method must be called before calling any other methods of this interface. |

| RegisterLobby | Allows launched applications to automatically propagate game status to the lobby. |
|----------------------|--|
| <u>ReturnBuffer</u> | Retrieves message buffers provided to the application through the pReceiveData member of the <u>DPN_MSGID_RECEIVE</u> system message. If the user's message handler returns <u>DPNSUCCESS_PENDING</u> to the RECEIVE callback, DirectPlay assumes ownership of the buffer has been transferred to the application, and neither frees nor modifies it until ownership is returned to DirectPlay through this call. |
| <u>Send</u> | Transmits data to the server. The message can be sent synchronously or asynchronously. |
| <u>SetCaps</u> | Sets the DPN_CAPS or DPN_CAPS_EX structure for the current interface. |
| <u>SetClientInfo</u> | Sets the static settings of a client with an application. Call this method before connecting to relay basic player information to the application. When the client successfully connects with the application, the server can retrieve information obtained through this method by calling the <u>IDirectPlay8Server::GetClientInfo</u> method. |
| <u>SetSPCaps</u> | Sets the DPN_SP_CAPS structure for the specified service provider. |

Interface Information

Inherits fromIUnknownMinimum operating systemsWindows 98, Pocket PC 2002

 $\ensuremath{\textcircled{}^{\odot}}$ 2003 Microsoft Corporation. All rights reserved.

IDirectPlay8Client::CancelAsyncOperation Method

Cancels asynchronous requests.

Syntax

```
HRESULT CancelAsyncOperation(
    const DPNHANDLE hAsyncHandle,
    const DWORD dwFlags
);
```

Parameters

hAsyncHandle

[in] Handle of the asynchronous operation to stop. You receive this handle when you call one of several methods that support asynchronous operations. This value can be set to NULL to stop all requests or a particular type of asynchronous request. If a particular handle is specified, the *dwFlags* parameter must be 0. If one of the DPNCANCEL_PLAYER_SENDS flags is specified in the *dwFlags* parameter, *hAsyncHandle* must be 0 to cancel all pending calls to the server.

dwFlags

[in] Flag that specifies which asynchronous request to canceled. You can set one of the following flags.

DPNCANCEL_ENUM

Cancel all asynchronous <u>IDirectPlay8Client::EnumHosts</u> requests. A single **IDirectPlay8Client::EnumHosts** request can be canceled by specifying the handle returned from the **IDirectPlay8Client::EnumHosts** method.

DPNCANCEL_CONNECT

Cancel an asynchronous <u>IDirectPlay8Client::Connect</u> request.

DPNCANCEL_SEND

Cancel an asynchronous <u>IDirectPlay8Client::Send</u> request. DPNCANCEL_PLAYER_SENDS Cancel all asynchronous **IDirectPlay8Client::Send** requests.

DPNCANCEL_PLAYER_SENDS_PRIORITY_LOW Cancel low-priority asynchronous IDirectPlay8Client::Send requests. DPNCANCEL_PLAYER_SENDS_PRIORITY_NORMAL Cancel normal-priority asynchronous IDirectPlay8Client::Send requests. DPNCANCEL_PLAYER_SENDS_PRIORITY_HIGH Cancel high-priority asynchronous IDirectPlay8Client::Send requests. DPNCANCEL_ALL_OPERATIONS Cancel all asynchronous requests.

Return Value

Returns S_OK if successful, or one of the following error values.

| DPNERR_CANNOTCANCEL | The operation could not be canceled. |
|----------------------|--|
| DPNERR_INVALIDFLAGS | The flags passed to this method are invalid. |
| DPNERR_INVALIDHANDLE | The handle specified is invalid. |
| DPNSUCCESS_PENDING | An asynchronous operation has reached the point where it is successfully queued. |

Remarks

Many methods of the <u>IDirectPlay8Client</u> interface run asynchronously by default. Depending on the situation, you might want to cancel requests before they are processed. All the methods of this interface that can be run asynchronously return a *hAsyncHandle* parameter.

Specific requests are canceled by passing the *hAsyncHandle* of the request in this method's *hAsyncHandle* parameter. You can cancel all pending asynchronous operations by calling this method, specifying NULL in the *hAsyncHandle* parameter, and specifying

DPNCANCEL_ALL_OPERATIONS in the *dwFlags* parameter. If a specific handle is provided to this method, no flags should be set.

You can use this method to cancel an asynchronous operation for the IDirectPlay8Client::Connect, IDirectPlay8Client::Send, and IDirectPlay8Client::EnumHosts methods. Microsoft® DirectPlay® does not support cancellation of other asynchronous operations.

You can cancel a send by providing the handle returned from **IDirectPlay8Client::Send** method. A

DPN_MSGID_SEND_COMPLETE system message is still posted to the applications message handler for each asynchronous send that is sent without the DPNSEND_NOCOMPLETE flag set. Sends that are canceled by this method return DPNERR_USERCANCEL in their hResultCode member of the

DPN_MSGID_SEND_COMPLETE message.

If you set the DPNCANCEL_ALL_OPERATIONS, DPNCANCEL_CONNECT, DPNCANCEL_SEND, or DPNCANCEL_ENUM flags in *dwFlags*, DirectPlay will attempt to cancel all matching operations. This method will return an error if any attempted cancellation fails, even though some cancellations may have been successful.

Note The completion message might not arrive until after this method returns. Do not assume that the operation has been terminated until you have received a DPN_MSGID_SEND_COMPLETE, DPN_MSGID_CONNECT_COMPLETE, or DPN_MSGID_ASYNC_OP_COMPLETE message.
IDirectPlay8Client::Close Method

Closes the open connection to a session and uninitializes the <u>IDirectPlay8Client</u> object. This method must be called on any object that is successfully initialized with a call to the <u>IDirectPlay8Client::Initialize</u> method.

Syntax

```
HRESULT Close(
const DWORD dwFlags
);
```

Parameters

dwFlags [in] The following flag can be specified. DPNCLOSE_IMMEDIATE Close immediately. Do not wait for outstanding calls to complete.

Return Value

Returns S_OK if successful, or the following error value.

DPNERR_UNINITIALIZED The requested object has not been initialized.

Remarks

Calling **IDirectPlay8Client::Close** will cancel all outstanding operations, including guaranteed messages that are in the queue waiting to be sent. Messages that have already been sent as guaranteed will continue to be retried until acknowledgement of their delivery has been received. To make sure all messages are sent, wait for all outstanding <u>IDirectPlay8Client::Send</u> calls to complete before calling **IDirectPlay8Client::Close**.

If you do not want the application to wait, the application should call IDirectPlay8Client::CancelAsyncOperation to cancel all outstanding sends prior to calling IDirectPlay8Client::Close or doing a final release call on the IDirectPlay8Client interface. Failing to do so causes unpredictable results.

Calling IDirectPlay8Client::Close will invalidate any DPN_CAPS, DPN_CAPS_EX, and DPN_SP_CAPS associated with the IDirectPlay8Client object.

IDirectPlay8Client::Connect Method

Establishes the connection to the server.

Syntax

```
HRESULT Connect(
const DPN_APPLICATION_DESC *const pdnAppDesc,
IDirectPlay8Address *const pHostAddr,
IDirectPlay8Address *const pDeviceInfo,
const DPN_SECURITY_DESC *const pdnSecurity,
const DPN_SECURITY_CREDENTIALS *const pdnCredentials,
const void *const pvUserConnectData,
const DWORD dwUserConnectDataSize,
void *const pvAsyncContext,
DPNHANDLE *const phAsyncHandle,
const DWORD dwFlags
);
```

Parameters

pdnAppDesc

[in] Pointer to a <u>DPN_APPLICATION_DESC</u> structure that describes the application. Only some of the members of this structure are used by this method. The only member of this structure that you must set is the **guidApplication** member. You can also set **guidInstance**, **pwszPassword**, **dwFlags**, and **dwSize**.

pHostAddr

[in] Pointer to an <u>IDirectPlay8Address</u> interface that specifies the addressing information to use to connect to the computer that is hosting. The user can be queried for any missing address information if you set the

DPNENUMHOSTS_OKTOQUERYFORADDRESSING flag in the *dwFlags* parameter.

pDeviceInfo

[in] Pointer to an **IDirectPlay8Address** object that specifies what network adapter (for example, network interface card, modem, and so on) to use to connect to the server. Some service providers allow this parameter to be NULL or be an address object containing only the service provider component. In this case, they will use the most appropriate device to reach the designated host. If you set the

DPNCONNECT_OKTOQUERYFORADDRESSING flag in *dwFlags*, the user can be queried for any missing address information.

pdnSecurity

[in] Reserved. Must be NULL.

pdnCredentials

[in] Reserved. Must be NULL.

pvUserConnectData

[in] Pointer to application-specific data provided to the host or server to further validate the connection. Microsoft® DirectPlay® will make a copy of this data when the method is called and therefore you can modify or destroy this data once the connection is complete. This data is sent to the DPN_MSGID_INDICATE_CONNECT message in the *pvUserConnectData* member. This parameter is optional and you can pass NULL to bypass the connection validation provided by the user code.

dwUserConnectDataSize

[in] Variable of type **DWORD** that specifies the size of the data contained in *pvUserConnectData*.

pvAsyncContext

[in] Pointer to the user-supplied context, which is returned in the **pvUserContext** member of the

<u>DPN_MSGID_CONNECT_COMPLETE</u> system message. This parameter is optional and can be set to NULL.

phAsyncHandle

[out] A **DPNHANDLE**. When the method returns,

phAsyncHandle will point to a handle that you can pass to <u>IDirectPlay8Client::CancelAsyncOperation</u> to cancel the operation. This parameter must be set to NULL if you set the DPNCONNECT_SYNC flag in *dwFlags*.

dwFlags

[in] Flag that describes the connection mode. You can set the following flag.

DPNCONNECT_OKTOQUERYFORADDRESSING

Setting this flag will display a standard DirectPlay dialog box, which queries the user for more information if not enough information is passed in this method.

DPNCONNECT_SYNC

Process the connection request synchronously. Setting this flag does not generate a

DPN_MSGID_CONNECT_COMPLETE system message.

Return Value

Returns S_OK if this method is processed synchronously and is successful. If the request is processed asynchronously, S_OK will be returned if the method is instantly processed. By default, this method is run asynchronously and generally returns

<u>DPNSUCCESS_PENDING</u> or one of the following error values.

| DPNERR_HOSTREJECTEDCONNECTION | The connection request was rejected. Check the ReplyData member of the DPN_MSGID_CONNECT_COMPLETE type for details. |
|-------------------------------|--|
| DPNERR_INVALIDAPPLICATION | The globally unique identifier (GUID) supplied for the application is invalid. |
| DPNERR_INVALIDDEVICEADDRESS | The address for the local computer or adapter is invalid. |
| DPNERR_INVALIDFLAGS | The flags passed to this method are invalid. |
| DPNERR_INVALIDHOSTADDRESS | The specified remote address is invalid. |
| DPNERR_INVALIDINSTANCE | The GUID for the application instance is invalid. |
| DPNERR_INVALIDINTERFACE | The interface parameter is invalid. This value will be returned in a connect request if the connecting player was not a client in a client/server game or a peer in a peer-to-peer game. |
| DPNERR_INVALIDPASSWORD | An invalid password was supplied when attempting to join a session that requires a password. |
| DPNERR_NOCONNECTION | No communication link was established. |
| DPNERR_NOTHOST | The client attempted to connect to a nonhost computer. Additionally, this error value may be returned by a nonhost that tried to set the application description. |
| DPNERR_SESSIONFULL | The request to connect to the host or server failed because the maximum number of players allotted for the session has been reached. |
| DPNERR_ALREADYCONNECTED | The object is already connected to the session. |
| DPNERR_ALREADYINITIALIZED | The object has already been initialized. |

Remarks

It is not required to enumerate hosts before calling IDirectPlay8Client::Connect if you know the appropriate host and device information.

If you do call the <u>IDirectPlay8Client::EnumHosts</u> method and you want to ensure better Network Address Translation (NAT) and proxy support when using the Transmission Control Protocol/Internet Protocol (TCP/IP) service provider or to prevent redialing with the modem service provider, keep the enumeration active when calling the **IDirectPlay8Client::Connect** method. To prevent the enumeration from completing, set the *dwEnumCount* parameter to INFINITE and do not use the

IDirectPlay8Client::CancelAsyncOperation to terminate the enumeration before the connect operation has completed. You should also pass the pAddressSender and pAddressDevice address objects in the DPNMSG_ENUM_HOSTS_RESPONSE message without modification into the *pHostAddr* and *pDeviceInfo* parameters of the IDirectPlay8Client::Connect method. To pass the address objects to IDirectPlay8Client::Connect outside of the callback function, use IDirectPlay8Address::Duplicate or IDirectPlay8Address::AddRef to prevent the object from being destroyed and store the pointers using thread-safe code. DirectPlay will automatically cancel the enumeration when the connect completes with DPN_OK or when IDirectPlay8Client::Close is called.

Before this method is called, you can obtain an application description by calling IDirectPlay8Client::EnumHosts. When you call IDirectPlay8Client::EnumHosts,

DPN_MSGID_ENUM_HOSTS_RESPONSE messages are sent to

your message handler with the **IDirectPlay8Address** objects and the **DPN_APPLICATION_DESC** structure for each host found. This information can be passed without modification to the **IDirectPlay8Client::Connect** method.

After a connection is established, the communication channel on the interface is open and the application should expect messages to arrive immediately. No messages can be sent by means of the IDirectPlay8Client::Send method until the connection has completed.

Although multiple enumerations can be run concurrently, and can be run across the duration of a connection, only one connection is allowed per interface. To establish a connection to more than one application, you must create another interface.

When this method is called, a **DPN_MSGID_INDICATE_CONNECT** message is posted to the server's message handler. On retrieval of this message, the host can pass back connection reply data to the **IDirectPlay8Client::Connect** method. Connection reply data can send a message indicating that the host does not approve the connection. The calling application can then handle this reply appropriately.

If **IDirectPlay8Client::Connect** is called synchronously, the following outcomes are possible.

- Connection Successful. The application will receive a DPN_MSGID_CONNECT_COMPLETE message containing the success code and the IDirectPlay8Client::Connect method will return with DPN_OK.
- Connection fails because the server rejects the connection. The application will receive a

DPN_MSGID_CONNECT_COMPLETE message containing the DPNERR_HOSTREJECTEDCONNECTION failure code. The **IDirectPlay8Client::Connect** method will also return with the error code DPNERR_HOSTREJECTEDCONNECTION. The **DPN_MSGID_CONNECT_COMPLETE** message provides an opportunity for the client application to inspect any data the server returns with the rejection.

• Connection fails for any other reason. The application will not receive a **DPN_MSGID_CONNECT_COMPLETE** message, and the **IDirectPlay8Client::Connect** method will return with the appropriate error code.

If **IDirectPlay8Client::Connect** is called asynchronously, the method returns immediately with DPNSUCCESS_PENDING. A **DPN_MSGID_CONNECT_COMPLETE** message will follow after the connection completes, containing the result of the connection. The only time the method does not return DPNSUCCESS_PENDING is when validation of the supplied parameters fails, in which case the appropriate error code is returned.

When the connection request completes, all outstanding enumerations are canceled with the return of DPNERR_USERCANCEL.

The hResultCode on the completion will indicate S_OK if the Connect() attempt was successful, or an error otherwise. If the Host player returned anything other than S_OK from the **DPN_MSGID_INDICATE_CONNECT** message, the likely error code in the completion will be DPNERR_HOSTREJECTEDCONNECTION.

When the connection completes, a **DPN_MSGID_CONNECT_COMPLETE** message is sent to the

application's message handler. All outstanding enumerations are canceled with the return of DPNERR_USERCANCEL.

To close the connection established with this method, call the **IDirectPlay8Client::Close** method.

Data Value Summary specifies the required addressing information for each service provider.

Note If you set the

DPNCONNECT_OKTOQUERYFORADDRESSING flag in *dwFlags*, the service provider might attempt to display a dialog box to ask the user to complete the address information. You must have a visible window present when the service provider tries to display the dialog box, or your application will lock.

IDirectPlay8Client::EnumHosts Method

Enumerates applications that host Microsoft® DirectPlay® games.

Syntax

HRESULT EnumHosts(PDPN_APPLICATION_DESC const pApplicationDesc, IDirectPlay8Address *const pdpaddrHost, IDirectPlay8Address *const pdpaddrDeviceInfo, PVOID const pvUserEnumData, const DWORD dwUserEnumDataSize, const DWORD dwEnumCount, const DWORD dwRetryInterval, const DWORD dwRetryInterval, const DWORD dwTimeOut, PVOID const pvUserContext, HANDLE *const pAsyncHandle, const DWORD dwFlags);

Parameters

pApplicationDesc

[in] Pointer to a <u>DPN_APPLICATION_DESC</u> structure that specifies which application hosts to enumerate. You must set the *pApplicationDesc*.dwSize member to the appropriate value. To reduce the number of responses, set

pApplicationDesc.guidApplication to the globally unique identifier (GUID) of the application to be found. If this member is not set, the search will include all applications.

pdpaddrHost

[in] Pointer to an <u>IDirectPlay8Address</u> object that specifies the address of the computer that is hosting the application. Some service providers allow this parameter to be NULL or be an address object containing only the service provider component. In this case, DirectPlay will get the information by using a

broadcast mechanism or from the *pdpaddrDeviceInfo* parameter. If you set the

DPNENUMHOSTS_OKTOQUERYFORADDRESSING flag in *dwFlags*, the user can be queried for any missing address information.

pdpaddrDeviceInfo

[in] Pointer to an **IDirectPlay8Address** object that specifies the service provider and local device settings to use when enumerating. The user can be queried for any missing address information if you set the

DPNENUMHOSTS_OKTOQUERYFORADDRESSING flag in the *dwFlags* parameter.

pvUserEnumData

[in] Pointer to a block of data that is sent in the enumeration request to the host. The size of the data is limited depending on the network type. Call <u>IDirectPlay8Client::GetSPCaps</u> to obtain the exact value.

dwUserEnumDataSize

[in] Variable of type **DWORD** that specifies the size of the data pointed at in the *pvUserEnumData* parameter.

dwEnumCount

[in] Value specifying how many times the enumeration data will be sent. Set this parameter to zero to use the default value. You can obtain the default value for *dwEnumCount* by calling **IDirectPlay8Client::GetSPCaps**. If *dwEnumCount* is set to INFINITE, the enumeration will continue until canceled.

dwRetryInterval

[in] Value specifying how many milliseconds between enumeration retries. Set this parameter to zero to use the default value. You can obtain the default value for

dwRetryInterval by calling **IDirectPlay8Client::GetSPCaps**. *dwTimeOut*

[in] Variable of type **DWORD** that specifies the number of milliseconds that DirectPlay will wait for replies after the last enumeration is sent. Set this parameter to zero to use the default value. You can obtain the default value for *dwTimeOut* by calling **IDirectPlay8Client::GetSPCaps**. If INFINITE is specified, the enumeration continues until it is canceled.

pvUserContext

[in] Context that is provided in the client's message handler when it is called with responses to the enumeration. This can be useful to differentiate replies from concurrent enumerations.

pAsyncHandle

[out] A **DPNHANDLE**. When the method returns, *pAsyncHandle* will point to a handle that you can pass to

IDirectPlay8Client::CancelAsyncOperation to cancel the operation. This parameter must be set to NULL if you set the DPNENUMHOSTS_SYNC flag in *dwFlags*.

dwFlags

[in] The following flags can be set.

DPNENUMHOSTS_SYNC

Causes the method to process synchronously.

DPNENUMHOSTS_OKTOQUERYFORADDRESSING Setting this flag will display a standard DirectPlay dialog box, which queries the user for more information if not enough information is passed in this method.

DPNENUMHOSTS_NOBROADCASTFALLBACK

If the service provider supports broadcasting, setting this flag will disable the broadcast capabilities. Check to see if broadcasting is supported by examining the <u>DPN_SP_CAPS</u> structure before setting this flag.

Return Value

Returns S_OK if this method is processed synchronously and is successful. By default, this method is run asynchronously and normally returns <u>DPNSUCCESS_PENDING</u>. It can also return one of the following error values.

| DPNERR_INVALIDDEVICEADDRESS | The address for the local computer or adapter is invalid. |
|-----------------------------|---|
| DPNERR_INVALIDFLAGS | The flags passed to this method are invalid. |
| DPNERR_INVALIDHOSTADDRESS | The specified remote address is invalid. |
| DPNERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
| DPNERR_ENUMQUERYTOOLARGE | The query data specified is too large. |
| DPNERR_USERCANCEL | The user canceled the operation. |
| DPNERR_ALREADYINITIALIZED | The object has already been initialized. |

Remarks

When an application is found that meets the enumeration criteria, the application's message handler is called with a <u>DPN_MSGID_ENUM_HOSTS_RESPONSE</u> system message. The message contains a **DPN_APPLICATION_DESC** structure describing the applications found and **IDirectPlay8Address** objects identifying the location of the hosts.

When an application is found that meets the enumeration criteria, the application's message handler is called with a **DPN_MSGID_ENUM_HOSTS_RESPONSE** system message. The message contains a **DPN_APPLICATION_DESC** structure describing the applications found and **IDirectPlay8Address** objects identifying the location of the hosts.

To ensure better Network Address Translation (NAT) and proxy support when using the Transmission Control Protocol/Internet Protocol (TCP/IP) service provider or to prevent redialing with the modem service provider, keep the enumeration active when calling the IDirectPlay8Client::Connect method. To prevent the enumeration from completing, set the *dwEnumCount* parameter to INFINITE and do not use the IDirectPlay8Client::CancelAsyncOperation to terminate the enumeration before the connect operation has completed. You should also pass the pAddressSender and pAddressDevice address objects in the DPNMSG_ENUM_HOSTS_RESPONSE message without modification into the *pHostAddr* and *pDeviceInfo* parameters of the IDirectPlay8Client::Connect outside of the callback function, use IDirectPlay8Address::Duplicate or IDirectPlay8Address::AddRef to prevent the object from being destroyed and store the pointers using thread-safe code.

Any number of enumerations can be run concurrently. The *pvUserContext* value is provided in the message handler to help differentiate replies to different enumerations.

Because of the variation in the number of ways enumeration can happen, it is not recommended that an application attempt to specify *dwEnumCount*, *dwRetryInterval*, or *dwTimeOut* unless the application has some specific media knowledge. The only exception is if you want to have the enumeration continue until explicitly cancelled, then set *dwEnumCount* to INFINITE.

The default enumeration count and timeout values will cause **IDirectPlay8Client::EnumHosts** to complete within a reasonable amount of time. These values are set by the service provider, and can be obtained by calling **IDirectPlay8Client::GetSPCaps**. Asynchronous enumerations can be stopped at any time by calling **IDirectPlay8Client::CancelAsyncOperation** and either passing the handle returned in the *pAsyncHandle* parameter or setting the DPNCANCEL_ENUM flag in the *dwFlags* parameter. An enumeration can also be stopped by returning anything other than S_OK from the message handler when processing a **DPN MSGID ENUM HOSTS RESPONSE** message.

You might receive multiple

DPN_MSGID_ENUM_HOSTS_RESPONSE messages from the same host during one enumeration session. The **guidInstance** member of the associated DPN_APPLICATION_DESC structure can be used to correlate these duplicate responses.

If you set the DPNENUMHOSTS_OKTOQUERYFORADDRESSING flag in *dwFlags*, the service provider might attempt to display a dialog box to ask the user to complete the address information. You must have a visible window present when the service provider tries to display the dialog box, or your application will lock.

Data Value Summary specifies the required addressing information for each service provider.

DPNERR_USERCANCEL will be returned if the enumeration is canceled by calling the IDirectPlay8Client::CancelAsyncOperation method or if DPN_OK is not returned when processing a DPN_MSGID_ENUM_HOSTS_RESPONSE message.

IDirectPlay8Client::EnumServiceProviders Method

Enumerates the registered service providers available to the application.

Syntax

```
HRESULT EnumServiceProviders(
const GUID *const pguidServiceProvider,
const GUID *const pguidApplication,
DPN_SERVICE_PROVIDER_INFO *const pSPInfoBuffer,
PDWORD const pcbEnumData,
PDWORD const pcReturned,
const DWORD dwFlags
);
```

Parameters

```
pguidServiceProvider
```

[in] Pointer to a variable of type **GUID** that specifies a service provider. This optional parameter forces the enumeration of subdevices for the specified service provider. You should normally set this value to NULL, to enumerate all available service providers. Otherwise, set *pguidServiceProvider* to one of the following predefined values.

CLSID_DP8SP_TCPIP

Internet Protocol (IP) service providers

```
CLSID_NETWORKSIMULATOR_DP8SP_TCPIP
```

DP8Sim service providers

```
CLSID_DP8SP_SERIAL
```

Serial service providers

```
CLSID_DP8SP_MODEM
```

Modem service providers

```
CLSID_DP8SP_IPX
```

IPX service providers

pguidApplication

[in] Pointer to a variable of type **GUID** that specifies an

application. If a pointer is passed in this parameter, only service providers who can be connected to the application are enumerated. You can also pass NULL to enumerate the registered service providers for the system.

pSPInfoBuffer

[out] Pointer to an array of <u>DPN_SERVICE_PROVIDER_INFO</u> structures that will be filled with service provider information.

pcbEnumData

[out] Pointer to **DWORD**, which is filled with the size of the *pSPInfoBuffer* array if the buffer is too small.

pcReturned

[out] Pointer to a variable of type **DWORD** that specifies the number of **DPN_SERVICE_PROVIDER_INFO** structures returned in the *pSPInfoBuffer* array.

dwFlags

[in] The following flag can be specified.

DPNENUMSERVICEPROVIDERS_ALL

Enumerates all the registered service providers for the system, including those that are not available to the application or do not have devices installed.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_BUFFERTOOSMALLThe supplied buffer is not large enough to contain the requested data.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.

Remarks

Call this method initially by specifying NULL in the *pguidServiceProvider* parameter to determine the base service providers available to the system. Specific devices for a service provider can be obtained by passing a pointer to a service provider globally unique identifier (GUID) in the *pguidServiceProvider*. This is useful, for example, when using the Modem Connection for

Microsoft® DirectPlay® service provider. You can choose among different modems for dialing out and select specific modems for hosting.

If the *pEnumData*buffer is not big enough to hold the requested service provider information, the method returns DPNERR_BUFFERTOOSMALL and the *cbEnumData* parameter contains the required buffer size. Typically, the best strategy is to call the method once with a zero-length buffer to determine the required size. Then call the method again with the appropriate-sized buffer.

Normally, this method will return only those service providers that can be used by the application. For example, if the Internetwork Packet Exchange (IPX) networking protocol is not installed, DirectPlay will not return the IPX service provider. To have DirectPlay return all service providers, even those that cannot be used by the application, set the DPNENUMSERVICEPROVIDERS_ALL flag in *dwFlags*.

IDirectPlay8Client::GetApplicationDesc Method

Retrieves the full application description for the connected application.

Syntax

```
HRESULT GetApplicationDesc(
    DPN_APPLICATION_DESC *const pAppDescBuffer,
    DWORD *const pcbDataSize,
    const DWORD dwFlags
);
```

Parameters

pAppDescBuffer

[out] Pointer to a <u>DPN_APPLICATION_DESC</u> structure where the application description data is written. Set this parameter to NULL to request only the size of data. If *pAppDescBuffer* is not set to NULL, you must set the *pAppDescBuffer*.dwSize member to an appropriate value. The *pcbDataSize* parameter is set to the size required to hold the data.

pcbDataSize

[in, out] Pointer to a variable of type **DWORD** that is initialized to the size of the buffer before calling this method. After the method returns, this parameter is set to the size, in bytes, of the session data. If the buffer is too small, this method returns the DPNERR_BUFFERTOOSMALL error value, and this parameter is set to the buffer size required. If this parameter is NULL, the method returns DPNERR_INVALIDPARAM.

dwFlags

[in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_BUFFERTOOSMALLThe supplied buffer is not large enough to contain the requested data.DPNERR_INVALIDFLAGSThe flags passed to this method are invalid.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_NOCONNECTIONNo communication link was established.

Remarks

Call this method initially by passing NULL in the *pAppDescBuffer* parameter to obtain the size of the required buffer. When you call the method a second time to fill the buffer, be sure to set the structures **dwSize** member to the appropriate value.

The returned DPN_APPLICATION_DESC structure will have the **guidInstance**, **guidApplication**, and **pwszSessionName** members set. It will not contain information about other clients that are connected to the session. That information, if available, can be obtained only from the server application. In particular, the **dwCurrentPlayers** member will always be set to 0.

IDirectPlay8Client::GetCaps Method

Retrieves the <u>DPN_CAPS</u> or <u>DPN_CAPS_EX</u> structure for the current interface.

Syntax

```
HRESULT GetCaps(
    DPN_CAPS *const pdpnCaps,
    const DWORD dwFlags
);
```

Parameters

pdpnCaps

[out] Pointer to a **DPN_CAPS** or **DPN_CAPS_EX** structure to receive caps information. You must set the **dwSize** member of this structure to an appropriate value.

dwFlags

[in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDOBJECTThe Microsoft® DirectPlay® object pointer is invalid.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_INVALIDPOINTERPointer specified as a parameter is invalid.DPNERR_UNINITIALIZEDThe requested object has not been initialized.

Remarks

A successful call to <u>IDirectPlay8Client::Initialize</u> must be made before this method can be called.

DirectPlay will determine whether **DPN_CAPS** or **DPN_CAPS_EX** is being used, based on the size of the structure referenced by *pdpnCaps*.

IDirectPlay8Client::GetConnectionInfo Method

Retrieves statistical information about the connection between the local client and the server.

Syntax

```
HRESULT GetConnectionInfo(
    DPN_CONNECTION_INFO *const pdnConnectInfo,
    const DWORD dwFlags
);
```

Parameters

pdnConnectInfo

[out] Pointer to a <u>DPN_CONNECTION_INFO</u> structure to retrieve information about the specified connection. You must set the **dwSize** member of this structure to an appropriate value. *dwFlags*

[in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDOBJECTThe Microsoft® DirectPlay® object pointer is invalid.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_INVALIDPOINTERPointer specified as a parameter is invalid.DPNERR_UNINITIALIZEDThe requested object has not been initialized.

Remarks

This method can be called only after a successful <u>IDirectPlay8Client::Connect</u> call has completed. $\ensuremath{\textcircled{\sc 0}}$ 2003 Microsoft Corporation. All rights reserved.

IDirectPlay8Client::GetSendQueueInfo Method

Used by the application to monitor the size of the send queue. Microsoft® DirectPlay® does not send messages faster than the receiving computer can process them. As a result, if the sending computer is sending faster than the receiver can receive, messages accumulate in the sender's queue. If the application registers that the send queue is growing too large, it should decrease the rate that messages are sent.

Syntax

HRESULT GetSendQueueInfo(DWORD *const pdwNumMsgs, DWORD *const pdwNumBytes, const DWORD dwFlags);

Parameters

pdwNumMsgs

[out] Pointer to a variable of type **DWORD** that contains the number of messages currently queued. This value is optional, and may be set to NULL.

pdwNumBytes

[out] Pointer to a variable of type **DWORD** that specifies the total number of bytes of data of the messages currently queued. This value is optional, and may be set to NULL.

dwFlags

[in] You may specify the DPNGETSENDQUEUEINFO_PRIORITY_NORMAL, DPNGETSENDQUEUEINFO_PRIORITY_HIGH, or DPNGETSENDQUEUEINFO_PRIORITY_LOW flag to inquire about specific messages of that priority.

Return Value

Returns S_OK if successful, or the following error value.

DPNERR_INVALIDPARAM One or more of the parameters passed to the method are invalid.

Remarks

You cannot set both *pdwNumMsgs* and *pdwNumBytes* to NULL. At least one of them must be set to a valid pointer.

IDirectPlay8Client::GetServerAddress Method

Retrieves the address of the server for the session.

Syntax

```
HRESULT GetServerAddress(
    IDirectPlay8Address **const ppAddress,
    const DWORD dwFlags
);
```

Parameters

ppAddress

[out] Address of a pointer to an <u>IDirectPlay8Address</u> object that specifies the address of the server. You must release this object when you no longer need it.

dwFlags

[in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDOBJECTThe Microsoft® DirectPlay® object pointer is invalid.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_INVALIDPOINTERPointer specified as a parameter is invalid.DPNERR_UNINITIALIZEDThe requested object has not been initialized.
IDirectPlay8Client::GetServerInfo Method

Retrieves the data set for the server set by the call to the IDirectPlay8Server::SetServerInfo method.

Syntax

```
HRESULT GetServerInfo(
    DPN_PLAYER_INFO *const pdpnPlayerInfo,
    DWORD *const pdwSize,
    const DWORD dwFlags
);
```

Parameters

pdpnPlayerInfo

[out] Pointer to a <u>DPN_PLAYER_INFO</u> structure to be filled with the server's information. If *pdwSize* is not set to NULL, you must set *pdpnPlayerInfo*.dwSize to the size of a **DPN_PLAYER_INFO** structure.

pdwSize

[out] Pointer to a variable of type **DWORD** that contains the size of the data returned in the *pdpnPlayerInfo* parameter. If this value is too small, the method returns

DPNERR_BUFFERTOOSMALL, and this parameter is set to the required size of the buffer.

dwFlags

[in] Reserved. Must be set to 0.

Return Value

Returns S_OK if successful, or one of the following error values.

```
DPNERR_BUFFERTOOSMALLThe supplied buffer is not large enough to contain the requested data.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.
```

Remarks

Call this method after the client receives a DPN_MSGID_SERVER_INFO message, indicating that the server has updated its information.

Microsoft® DirectPlay® returns the **DPN_PLAYER_INFO** structure and the pointers assigned to the structure's **pwszName** and **pvData** members in a contiguous buffer. If the two pointers were set, you must have allocated enough memory for the structure, plus the two pointers. The most robust way to use this method is to first call it with *pdwSize* set to NULL. When the method returns, *pdwSize* will point to the correct value. Use that value to allocate memory for your structure and call the method a second time to retrieve the information.

When the method returns, the **dwinfoFlags** member of the **DPN_PLAYER_INFO** structure will always have the DPNINFO_DATA and DPNINFO_NAME flags set, even if the corresponding pointers are set to NULL. These flags are used when calling **IDirectPlay8Server::SetServerInfo**, to notify DirectPlay of which values have changed.

Transmission of nonstatic information should be handled with the <u>IDirectPlay8Server::SendTo</u> method because of the high cost of using the **IDirectPlay8Server::SetServerInfo** method.

IDirectPlay8Client::GetSPCaps Method

Retrieves the <u>DPN_SP_CAPS</u> structure for the specified service provider.

Syntax

```
HRESULT GetSPCaps(
    const GUID *const pguidSP,
    DPN_SP_CAPS *const pdpnSPCaps,
    const DWORD dwFlags
);
```

Parameters

pguidSP

[in] Pointer to a globally unique identifier (GUID) specifying the service provider you want to get information about.

pdpnSPCaps

[out] Pointer to a **DPN_SP_CAPS** structure to receive the information about the specified service provider. You must set the *pdpnSPCaps*.dwSize member to the size of a **DPN SP CAPS** structure.

dwFlags

[in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDOBJECTThe Microsoft® DirectPlay® object pointer is invalid.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_INVALIDPOINTERPointer specified as a parameter is invalid.DPNERR_UNINITIALIZEDThe requested object has not been initialized.

Remarks

This method retrieves information about the specified service provider. A successful call to <u>IDirectPlay8Client::Initialize</u> must be made before this method can be called.

IDirectPlay8Client::Initialize Method

Registers an entry point in the client's code that receives the messages from the <u>IDirectPlay8Client</u> interface and from the server. This method must be called before calling any other methods of this interface.

Syntax

```
HRESULT Initialize(
    PVOID const pvUserContext,
    const PFNDPNMESSAGEHANDLER pfn,
    const DWORD dwFlags
);
```

Parameters

pvUserContext

[in] Pointer to the user-provided context value in calls to the message handler. Providing a user-context value can be useful to differentiate messages coming from multiple interfaces to a common message handler.

pfn

[in] Pointer to a <u>PFNDPNMESSAGEHANDLER</u> callback function that receives all messages from the server, and receives indications of session changes from the **IDirectPlay8Client** interface.

dwFlags

[in] You can specify the following flags.

DPNINITIALIZE_DISABLEPARAMVAL

Disable parameter validation for the current object.

DPNINITIALIZE_HINT_LANSESSION

Opens a larger send window for games running on a local area network (LAN).

DPNINITIALIZE_DISABLELINKTUNING

Disable any attempts by Microsoft DirectPlay® to tune the rate it sends at to the observed network conditions.

Messages will be pushed out onto the network at the first available opportunity.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDFLAGSThe flags passed to this method are invalid.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.

Remarks

This is the first method you should call after using **CoCreateInstance** to obtain the **IDirectPlay8Client** interface.

Specify the DPNINITIALIZE_HINT_LANSESSION flag for sessions where all players will be on the same LAN.

Applications might want to specify the

DPNINITIALIZE_DISABLELINKTUNING flag when they send at a fixed rate and do not alter the rate based on the network conditions. With this flag specified, DirectPlay will always assume the network has the capacity to carry all the application data and will therefore not attempt to tune its send rate to the network bandwidth. Specifying this flag and then sending at a rate that exceeds the capacity of the network will lead to unpredictable network behavior such as higher latency and increased packet drop rates. Applications that monitor the send queues and dynamically adjust their send rate to make best use of the available bandwidth should not specify this flag.

If the DPNINITIALIZE_DISABLELINKTUNING flag is specified,

DirectPlay features such as message prioritization, coalescence, and timeout are not useful because messages always go directly to the network and are not queued.

IDirectPlay8Client::RegisterLobby Method

Allows launched applications to automatically propagate game status to the lobby.

Syntax

```
HRESULT RegisterLobby(
    const DPNHANDLE dpnHandle,
    IDirectPlay8LobbiedApplication *const pIDP8LobbiedApplication,
    const DWORD dwFlags
);
```

Parameters

dpnHandle

[in] Connection handle used when making the calls to IDirectPlay8LobbiedApplication::UpdateStatus.

pIDP8LobbiedApplication

[in] Pointer to the <u>IDirectPlay8LobbiedApplication</u> object that specifies the application.

dwFlags

[in] One of the following flags:

DPNLOBBY_REGISTER

Registers the lobby with the application.

DPNLOBBY_UNREGISTER

Unregisters the lobby with the application.

Return Value

Returns S_OK if successful, or the following error value.

DPNERR_INVALIDPARAM One or more of the parameters passed to the method are invalid.

 $\ensuremath{\textcircled{\sc 0}}$ 2003 Microsoft Corporation. All rights reserved.

IDirectPlay8Client::ReturnBuffer Method

Retrieves message buffers provided to the application through the **pReceiveData** member of the <u>DPN_MSGID_RECEIVE</u> system message. If the user's message handler returns DPNSUCCESS_PENDING to the RECEIVE callback, Microsoft® DirectPlay® assumes ownership of the buffer has been transferred to the application, and neither frees nor modifies it until ownership is returned to DirectPlay through this call.

Syntax

```
HRESULT ReturnBuffer(
const DPNHANDLE hBufferHandle,
const DWORD dwFlags
);
```

Parameters

hBufferHandle

[in] Variable of type **DPNHANDLE** that specifies the buffer handle to the message. This is obtained in the *hBufferHandle* member of the **DPN_MSGID_RECEIVE** system message. *dwFlags*

[in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDHANDLEThe handle specified is invalid.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.

IDirectPlay8Client::Send Method

Transmits data to the server. The message can be sent synchronously or asynchronously.

Syntax

```
HRESULT Send(
const DPN_BUFFER_DESC *const pBufferDesc,
const DWORD cBufferDesc,
const DWORD dwTimeOut,
void *const pvAsyncContext,
DPNHANDLE *const phAsyncHandle,
const DWORD dwFlags
```

);

Parameters

pBufferDesc

[in] Pointer to a <u>DPN_BUFFER_DESC</u> structure that describes the data to send.

cBufferDesc

[in] Number of **DPN_BUFFER_DESC** structures pointed to by *pBufferDesc*. There can be up to eight buffers in this version of Microsoft® DirectPlay®.

dwTimeOut

[in] Number of milliseconds to wait for the message to send. If the message has not been sent by the *dwTimeOut* value, it is deleted from the send queue. If you set this parameter to 0, the message remains in the send queue until it is sent or until the link is dropped.

pvAsyncContext

[in] Pointer to the user-supplied context, which is returned in the **pvUserContext** member of the

<u>DPN_MSGID_SEND_COMPLETE</u> system message. phAsyncHandle [in, out] A DPNHANDLE. When the method returns,

phAsyncHandle will point to a handle that you can pass to <u>IDirectPlay8Client::CancelAsyncOperation</u> to cancel the operation. This parameter must be set to NULL if you set the

DPNSEND_SYNC flag in *dwFlags*.

dwFlags

[in] Flags that describe send behavior. You can set one or more of the following flags.

DPNSEND_SYNC

Process the **IDirectPlay8Client::Send** request synchronously.

DPNSEND_NOCOPY

Use the data in the **DPN_BUFFER_DESC** structure and do not make an internal copy. This can be a more efficient method of sending data to the server. However, it is less robust because modifying or deleting the data before receiving the **DPN_MSGID_SEND_COMPLETE** message can cause erroneous data to be sent. This flag cannot be combined with DPNSEND_NOCOMPLETE.

DPNSEND_NOCOMPLETE

Does not send **DPN_MSGID_SEND_COMPLETE** to the message handler. This flag cannot be used with DPNSEND_NOCOPY or DPNSEND_GUARANTEED. Additionally, when using this flag *pvAsyncContext* must be NULL.

DPNSEND_COMPLETEONPROCESS

Send **DPN_MSGID_SEND_COMPLETE** to the message handler when this message has been delivered to the target and the target's message handler returns from indicating its reception. There is additional internal message overhead when this flag is set, and the message transmission process might become significantly slower. If you set this flag, DPNSEND GUARANTEED must also be set.

DPNSEND_GUARANTEED

Send the message by a guaranteed method of delivery. DPNSEND_PRIORITY_HIGH

Sets the priority of the message to high. This flag cannot be used with DPNSEND_PRIORITY_LOW.

DPNSEND_PRIORITY_LOW

Sets the priority of the message to low. This flag cannot be used with DPNSEND_PRIORITY_HIGH.

DPNSEND_NOLOOPBACK

Suppress the <u>DPN_MSGID_RECEIVE</u> system message to your message handler if you are sending to yourself. DPNSEND_NONSEQUENTIAL

If the flag is not set, messages are delivered to the target application in the order that they are sent, which can necessitate buffering out of sequence messages until the missing messages arrive. Messages are delivered to the target application in the order that they are received.

DPNSEND_COALESCE

Allows DirectPlay to combine packets when sending.

Return Value

Returns S_OK if this method is processed synchronously and is successful. By default, this method is run asynchronously and generally returns <u>DPNSUCCESS_PENDING</u> or one of the following error values.

DPNERR_INVALIDFLAGSThe flags passed to this method are invalid.DPNERR_TIMEDOUTThe operation could not complete because it has timed out.

Remarks

This method generates a **DPN_MSGID_RECEIVE** system message in the server's message handler. The data buffer is contained in the **pReceiveData** member of the associated structure.

Messages can have one of three priorities: low, normal, and high. To specify a low or high priority for the message, set the appropriate flag in *dwFlags*. If neither of the priority flags is set, the message will have normal priority. See <u>Basic Networking</u> for a discussion of send

priorities.

When the **IDirectPlay8Client::Send** request is completed, a **DPN_MSGID_SEND_COMPLETE** system message is posted to the sender's message handler. The success or failure of the request is contained in the **hResultCode** member of the associate structure. You can suppress the send completion by setting the DPN_NOCOMPLETE flag in *dwFlags*.

If a player joins a game and needs to send multiple messages immediately, the player should first send a message with the DPNSEND_COMPLETEONPROCESS flag set. When the **DPN_MSGID_SEND_COMPLETE** message is returned, the application can begin sending messages. If the player does not do this, some of the messages might need to be queued on the receiver and, if too much data arrives, the queue can grow faster than the receiver can handle the messages. This might result in lost data. After a player is established in the game, however, throttling in DirectPlay will control the data flow by using message timeouts or the GetSendQueueInfo method. For more information, see <u>Optimizing Network Usage</u>.

Send completions are typically posted on the source computer as soon as the message is sent. In other words, a send completion does not necessarily mean that the message has been processed on the target. It might still be in a queue. If you want to be certain that the message has been processed by the target, set the DPN_COMPLETEONPROCESS flag in *dwFlags*. This flag ensures that the send completion will not be sent until the target's message handler has processed the message and returned.

If the DPNSEND_COALESCE flag is set in *dwFlags*, DirectPlay will try to coalesce up to 32 packets waiting in the queue into the outgoing frame. DirectPlay does not guarantee coalescence, even if the DPNSEND_COALESCE flag is set. Packets will only be coalesced if there is more than one message in the queue and the player receiving is running Microsoft DirectX® 9.0 or later. All voice packets can be coalesced. Both guaranteed and non-guaranteed packets will be coalesced into the same frame. If the frame is dropped before it reaches its destination, only the guaranteed parts of the frame will be resent and no other data will be coalesced into the frame.

Note Do not assume that resources such as the data buffer will remain valid until the method has returned. If you call this method asynchronously, the **DPN_MSGID_SEND_COMPLETE** message can be received and processed by your message handler before the call has returned. If your message handler deallocates or otherwise invalidates a resource such as the data buffer, that resource can become invalid at any time after the method has been called.

IDirectPlay8Client::SetCaps Method

Sets the <u>DPN_CAPS</u> or <u>DPN_CAPS_EX</u> structure for the current interface.

Syntax

```
HRESULT SetCaps(
    const DPN_CAPS *const pdpCaps,
    const DWORD dwFlags
);
```

Parameters

pdpCaps

[in] Pointer to a DPN_CAPS or DPN_CAPS_EX structure used to set the information about the current interface.

dwFlags

[in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDOBJECTThe Microsoft® DirectPlay® object pointer is invalid.DPNERR_INVALIDPOINTERPointer specified as a parameter is invalid.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_UNINITIALIZEDThe requested object has not been initialized.

Remarks

A successful call to <u>IDirectPlay8Client::Initialize</u> must be made before this method can be called.

DirectPlay will determine whether **DPN_CAPS** or **DPN_CAPS_EX** is being used based on the size of the structure referenced by *pdpCaps*.

IDirectPlay8Client::SetClientInfo Method

Sets the static settings of a client with an application. Call this method before connecting to relay basic player information to the application. When the client successfully connects with the application, the server can retrieve information obtained through this method by calling the IDirectPlay8Server::GetClientInfo method.

Syntax

HRESULT SetClientInfo(const DPN_PLAYER_INFO *const pdpnPlayerInfo, PVOID const pvAsyncContext, DPNHANDLE *const phAsyncHandle, const DWORD dwFlags

);

Parameters

pdpnPlayerInfo

[in] Pointer to a <u>DPN_PLAYER_INFO</u> structure that contains the client information to set.

pvAsyncContext

[in] Pointer to the user-supplied context, which is returned in the **pvUserContext** member of the

DPN_MSGID_ASYNC_OP_COMPLETE system message.

phAsyncHandle

[in, out] A **DPNHANDLE**. A value will be returned. However, Microsoft® DirectPlay® does not permit cancellation of this operation, so the value cannot be used.

dwFlags

[in] Flag that controls how this method is processed. The following flag can be set for this method.

DPNSETCLIENTINFO_SYNC

Causes the method to process synchronously.

Return Value

Returns S_OK if this method is processed synchronously and is successful. If the request is processed asynchronously, S_OK can return if the method is instantly processed. By default, this method is run asynchronously and generally returns

<u>DPNSUCCESS_PENDING</u> or one of the following error values.

DPNERR_NOCONNECTIONNo communication link was established.DPNERR_INVALIDFLAGSThe flags passed to this method are invalid.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.

Remarks

This method can be called at any time during the session.

The **DPN_PLAYER_INFO** structure's **dwPlayerFlags** member must be set to zero.

Transmission of nonstatic information should be handled with the <u>IDirectPlay8Client::Send</u> method because of the high cost of using the **IDirectPlay8Client::SetClientInfo** method.

You can modify the client information with this method after connecting to the application. Calling this method after connection generates a <u>DPN_MSGID_CLIENT_INFO</u> system message to all players, informing them that data has been updated.

When calling this method asynchronously, the contents of the *pdpnPlayerInfo* and *pvAsyncContext* buffers will be copied by DirectPlay so that the calling application can clean up the buffers before the method returns.

This method is guaranteed as long as the player is connected to the session. DirectPlay will ensure that this method completes and that the information is propagated to all players.

IDirectPlay8Client::SetSPCaps Method

Sets the <u>DPN_SP_CAPS</u> structure for the specified service provider.

Syntax

```
HRESULT SetSPCaps(
    const GUID *const pguidSP,
    const DPN_SP_CAPS *const pdpnSPCaps,
    const DWORD dwFlags
);
```

Parameters

pguidSP

[in] Pointer to a globally unique identifier (GUID) that specifies the service provider to set information about.

pdpnSPCaps

[in] Pointer to a **DPN_SP_CAPS** structure to set the information about the specified service provider.

dwFlags

[in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDOBJECTThe Microsoft® DirectPlay® object pointer is invalid.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_INVALIDPOINTERPointer specified as a parameter is invalid.DPNERR_UNINITIALIZEDThe requested object has not been initialized.

Remarks

This method sets parameters for the specified service provider. A

successful call to <u>IDirectPlay8Client::Initialize</u> must be made before this method can be called. Currently, only the **dwSystemBufferSize** member can be set by this call. The **dwNumThreads** member is for legacy support. Microsoft DirectX® 9.0 applications should use the <u>IDirectPlay8ThreadPool::SetThreadCount</u> method to set the number of threads. The other members of the **DPN_SP_CAPS** structure are get-only or ignored.

IDirectPlay8LobbiedApplication Interface

The **IDirectPlay8LobbiedApplication** interface is used by an application that supports lobbying. This interface allows the application to register with the system so that it can be lobby launched. Additionally, it also lets the application get the connection information necessary to launch a game without querying the user. Lastly, this interface allows the lobbied application to send messages and notifications to the lobby client that launched the application.

| <u>Close</u> | Deletes the lobbied application. |
|------------------------------|--|
| <u>GetConnectionSettings</u> | Retrieves the set of connection settings for the specified connection. These settings can be set through a call to the <u>IDirectPlay8LobbyClient::ConnectApplication</u> , <u>IDirectPlay8LobbyClient::SetConnectionSettings</u> , or <u>IDirectPlay8LobbiedApplication::SetConnectionSettings</u> method. When you get connection settings, a reference will be added for each address object that is returned to the user. Therefore, users must be sure to call Release on each address object when they are done with the structure. |
| <u>Initialize</u> | Registers a message handler function that receives notifications about changes in the state of the lobby client and receives messages from the lobby client. |
| <u>RegisterProgram</u> | Registers a lobby-aware application with Microsoft® DirectPlay®. Applications must be registered to enable lobby launching. |
| Send | Sends a message from the lobbied application to the lobby client. |
| <u>SetAppAvailable</u> | Makes an application available or unavailable for a lobby client to connect to. This method is typically called if a lobbied application is independently launched, that is, not launched by a lobby client. Additionally, this method should be called if a game has ended and the lobbied application needs to be available to connect to a lobby client at the start of another game. |
| SetConnectionSettings | Sets the connection settings to be associated with the specified connection. Calling this method generates a DPL_MSGID_CONNECTION_SETTINGS message to be sent to the client specified by <i>hConnection</i> . When you set connection settings, the lobby application will add a reference to each of |

IDirectPlay8LobbiedApplication Members

| | the address objects specified in the call. |
|--------------------------|---|
| <u>UnRegisterProgram</u> | Unregisters a lobby-aware application that was registered through the IDirectPlay8LobbiedApplication::RegisterProgram method. |
| <u>UpdateStatus</u> | Updates the status of a connected lobby client. |

Interface Information

| Inherits from | <u>IUnknown</u> |
|---------------------------|----------------------------|
| Header | dplobby8.h |
| Minimum operating systems | Windows 98, Pocket PC 2002 |

 $\ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.

IDirectPlay8LobbiedApplication::Close Method

Deletes the lobbied application.

Syntax

```
HRESULT Close(
const DWORD dwFlags);
```

Parameters

```
dwFlags
[in] Reserved, must be 0.
```

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_UNINITIALIZEDThe requested object has not been initialized.DPNERR_INVALIDOBJECTThe Microsoft® DirectPlay® object pointer is invalid.DPNERR_OUTOFMEMORYThere is insufficient memory to perform the requested operation.
IDirectPlay8LobbiedApplication::GetConnectionSettings Method

Retrieves the set of connection settings for the specified connection. These settings can be set through a call to the <u>IDirectPlay8LobbyClient::ConnectApplication</u>, <u>IDirectPlay8LobbyClient::SetConnectionSettings</u>, or <u>IDirectPlay8LobbiedApplication::SetConnectionSettings</u> method.

When you get connection settings, a reference will be added for each address object that is returned to the user. Therefore, users must be sure to call **Release** on each address object when they are done with the structure.

Syntax

```
HRESULT GetConnectionSettings(
    const DPNHANDLE hLobbyClient,
    DPL_CONNECTION_SETTINGS *const pdplSessionInfo,
    DWORD *pdwInfoSize,
    const DWORD dwFlags
);
```

Parameters

hLobbyClient

[in] Handle to the connection for which to retrieve the settings. *pdplSessionInfo*

[out] Pointer to a <u>DPL_CONNECTION_SETTINGS</u> structure to receive the connection settings for the specified connection. pdwInfoSize

[in, out] Pointer to a **DWORD** containing the size, in bytes, of the buffer specified in the *pdplSessionInfo* structure. If the buffer is not large enough to hold the connection settings,

DPNERR_BUFFERTOOSMALL is returned and this value will be set to the required buffer size. On success, this value will contain the number of bytes written to the specified buffer.

dwFlags

[in] Reserved, must be 0.

Return Value

Returns S_OK if successful, or the following error value.

| DPNERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
|-----------------------|--|
| DPNERR_BUFFERTOOSMALL | The supplied buffer is not large enough to contain the requested data. |
| DPNERR_INVALIDOBJECT | The Microsoft® DirectPlay® object pointer is invalid. |
| DPNERR_INVALIDFLAGS | The flags passed to this method are invalid. |

IDirectPlay8LobbiedApplication::Initialize Method

Registers a message handler function that receives notifications about changes in the state of the lobby client and receives messages from the lobby client.

Syntax

```
HRESULT Initialize(
const PVOID pvUserContext,
const PFNDPNMESSAGEHANDLER pfn,
DPNHANDLE *const pdpnhConnection,
const DWORD dwFlags
);
```

Parameters

pvUserContext

[in] Pointer to the user-provided context value in calls to the message handler. Providing a user-context value is useful to differentiate messages from multiple interfaces to a common message handler.

pfn

[in] Pointer to a <u>PFNDPNMESSAGEHANDLER</u> callback function that receives all messages from the <u>IDirectPlay8LobbyClient</u> interface and indications of session changes from the <u>IDirectPlay8LobbiedApplication</u> interface.

pdpnhConnection

[out] Value used to detect if your application was lobby launched. If your application was lobby launched, this parameter will be set to the connection handle to the lobby client. If your process was not lobby launched, this parameter is set to NULL.

dwFlags

[in] The following flag can be specified.

DPLINITIALIZE_DISABLEPARAMVAL

Disables parameter validation.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_ALREADYINITIALIZEDThe object has already been initialized.DPNERR_INVALIDFLAGSThe flags passed to this method are invalid.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_NOTALLOWEDThis function is not allowed on this object.

Remarks

Call this method first after using **CoCreateInstance** to obtain the **IDirectPlay8LobbiedApplication** interface.

This method automatically establishes a connection to the lobby client if you were lobby launched. If you call

IDirectPlay8LobbiedApplication::Initialize and you were lobby launched and the lobbied application interface is unable to contact the lobby client process,

IDirectPlay8LobbiedApplication::Initialize will time out after four seconds. In this case, IDirectPlay8LobbiedApplication::Initialize will return DPNERR_TIMEDOUT but will still succeed.

Note Only one instance of **IDirectPlay8LobbyClient** and **IDirectPlay8LobbiedApplication** is allowed to be running for each process.

IDirectPlay8LobbiedApplication::RegisterProgram Method

Registers a lobby-aware application with Microsoft® DirectPlay®. Applications must be registered to enable lobby launching.

Syntax

```
HRESULT RegisterProgram(
    PDPL_PROGRAM_DESC pdplProgramDesc,
    const DWORD dwFlags
);
```

Parameters

pdplProgramDesc
[in] Pointer to the DPL_PROGRAM_DESC structure that
 describes the lobby-aware application to register.
dwFlags
[in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDFLAGS The flags passed to this method are invalid. DPNERR_INVALIDPARAM One or more of the parameters passed to the method are invalid.

Remarks

The application needs to register only once. It should be unregistered with a call to the <u>IDirectPlay8LobbiedApplication::UnRegisterProgram</u> method when it is uninstalled.

IDirectPlay8LobbiedApplication::RegisterProgram must be used. You cannot manually enter application information in the registry. Failure to use this interface might make your application nonportable and incompatible with later versions of DirectPlay.

If your application is running on Microsoft Windows® Powered Pocket PC 2002, do not set the **pwszExecutablePath** member of the **DPL_PROGRAM_DESC** structure to NULL unless your application's executable is located in the Windows directory (\Windows). If it is not in that directory, set **pwszExecutablePath** to the correct path.

IDirectPlay8LobbiedApplication::Send Method

Sends a message from the lobbied application to the lobby client.

Syntax

```
HRESULT Send(
const DPNHANDLE hConnection,
BYTE *const pBuffer,
const DWORD pBufferSize,
const DWORD dwFlags
);
```

Parameters

hConnection

[in] Variable of type **DPNHANDLE** that specifies the lobby client that the message is sent to. You may also specify the following flag.

DPLHANDLE_ALLCONNECTIONS

The message you have specified will be sent to all lobby clients to which you are connected.

pBuffer

[in] Pointer to a variable of type **BYTE** that contains the message buffer.

pBufferSize

[in] Variable of type **DWORD** that specifies the size of the message buffer in the *pBuffer* parameter, in bytes. This parameter must be at least 1 byte and no more than 64 KB. *dwFlags*

[in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDFLAGSThe flags passed to this method are invalid.DPNERR_INVALIDHANDLEThe handle specified is invalid.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_SENDTOOLARGEThe buffer was too large.

Remarks

If the buffer size is larger than 64 KB, the method returns DPNERR_SENDTOOLARGE. If the buffer size is set to 0, the method returns DPNERR_INVALIDPARAM.

IDirectPlay8LobbiedApplication::SetAppAvailable Method

Makes an application available or unavailable for a lobby client to connect to. This method is typically called if a lobbied application is independently launched, that is, not launched by a lobby client. Additionally, this method should be called if a game has ended and the lobbied application needs to be available to connect to a lobby client at the start of another game.

Syntax

```
HRESULT SetAppAvailable(
const BOOL fAvailable,
const DWORD dwFlags
);
```

Parameters

fAvailable

[in] Boolean value that sets the availability of the application. Set this value to TRUE to indicate that your application is available, or to FALSE to indicate that it is not available.

dwFlags

[in] The following flag can be set for this method.

DPLAVAILABLE_ALLOWMULTIPLECONNECT

The default behavior for this method is to automatically mark the interface as Unavailable when the first connection is established. By specifying this flag, the interface is not automatically marked unavailable after the first connection is established, thereby allowing multiple connections.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDOBJECTThe Microsoft® DirectPlay® object pointer is invalid.DPNERR_UNINITIALIZEDThe requested object has not been initialized.

IDirectPlay8LobbiedApplication::SetConnectionSettings Method

Sets the connection settings to be associated with the specified connection. Calling this method generates a DPL_MSGID_CONNECTION_SETTINGS message to be sent to the client specified by *hConnection*.

When you set connection settings, the lobby application will add a reference to each of the address objects specified in the call.

Syntax

```
HRESULT SetConnectionSettings(
    const DPNHANDLE hConnection,
    const DPL_CONNECTION_SETTINGS *const pdplConnectSettings
    const DWORD dwFlags
);
```

Parameters

hConnection

[in] Handle to the connection to set the settings for. You may also specify the following flag.

DPLHANDLE_ALLCONNECTIONS

The connection settings will be updated for all the lobby clients to which you are connected.

pdplConnectSettings

[in] Pointer to a <u>DPL_CONNECTION_SETTINGS</u> structure containing the settings associated with the specified connection.

```
dwFlags
```

[in] Reserved, must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_INVALIDOBJECTThe Microsoft® DirectPlay® object pointer is invalid.DPNERR_INVALIDFLAGSThe flags passed to this method are invalid.

IDirectPlay8LobbiedApplication::UnRegisterProgram Method

Unregisters a lobby-aware application that was registered through the <u>IDirectPlay8LobbiedApplication::RegisterProgram</u> method.

Syntax

```
HRESULT UnRegisterProgram(
GUID *pguidApplication,
const DWORD dwFlags
);
```

Parameters

pguidApplication [in] Pointer to the globally unique identifier (GUID) of the application to unregister. dwFlags [in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDFLAGS The flags passed to this method are invalid. DPNERR_INVALIDPARAM One or more of the parameters passed to the method are invalid.

IDirectPlay8LobbiedApplication::UpdateStatus Method

Updates the status of a connected lobby client.

Syntax

```
HRESULT UpdateStatus(
const DPNHANDLE hConnection,
const DWORD dwStatus,
const DWORD dwFlags
);
```

Parameters

hConnection

[in] Variable of type **DPNHANDLE** that specifies the lobby client. You may also specify the following flag.

DPLHANDLE_ALLCONNECTIONS

The status update will be sent to all lobby clients to which you are connected.

dwStatus

[in] Variable of type **DWORD** that is filled with one of the following values that indicate the status between the lobby client and the lobbied application.

DPLSESSION_CONNECTED

The lobby client and lobbied application are currently connected.

DPLSESSION_COULDNOTCONNECT

The lobby client was not able to connect to the lobbied application.

DPLSESSION_DISCONNECTED

The lobby client and lobbied application are currently disconnected.

DPLSESSION_TERMINATED

The connection between the lobby client and lobbied application has been terminated.

DPLSESSION_HOSTMIGRATED

The peer object associated with the connection is involved in a session where a host migration takes place and the local client is not the new host.

DPLSESSION_HOSTMIGRATEDHERE

The peer object associated with the connection is involved in a session where a host migration takes place and the local client becomes the new host.

dwFlags

[in] Reserved, must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDHANDLEThe handle specified is invalid.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.

IDirectPlay8LobbyClient Interface

Applications use methods of the **IDirectPlay8LobbyClient** interface to enumerate and launch lobby-enabled games on a local computer, and communicate with the games when they are running. The lobby client must register a message-handler routine to process messages from the lobby and the lobbied game application.

IDirectPlay8LobbyClient Members

| <u>Close</u> | Deletes the lobby client. |
|------------------------------|--|
| <u>ConnectApplication</u> | Connects a lobby-enabled application to the session specified in the <u>DPL_CONNECT_INFO</u> structure. If the application is not running, this method can be used to launch the application. When the connection is successfully established, the lobbied application generates a <u>DPL_MSGID_CONNECT</u> system message to the message handler. |
| EnumLocalPrograms | Enumerates the lobbied applications that are registered on the system. |
| <u>GetConnectionSettings</u> | Retrieves the set of connection settings for the specified connection. These settings can be set through a call to the <u>IDirectPlay8LobbyClient::ConnectApplication</u> , <u>IDirectPlay8LobbyClient::SetConnectionSettings</u> , or <u>IDirectPlay8LobbiedApplication::SetConnectionSettings</u> method. When you get connection settings, a reference will be added for each address object that is returned to the user. Therefore, users must be sure to call Release on each address object when they are finished with the structure. |
| <u>Initialize</u> | Registers an entry point in the lobby client's code that receives notifications on changes of state for any launched applications. The message handler also receives messages from the lobbied application. This method must be called before calling any other methods of this interface. |
| <u>ReleaseApplication</u> | Releases a lobbied application and closes the connection between the lobby client and the application. This method should be called whenever a lobby client has finished its session with an application. |
| <u>Send</u> | Sends a message to a lobbied application that was launched by this lobby client or was connected by this lobby client. This method sends a <u>DPL_MSGID_RECEIVE</u> system message to the target's message handler. |

<u>SetConnectionSettings</u>

Sets the connection settings to be associated with the specified connection. Calling this method will generate a DPL_MSGID_CONNECTION_SETTINGS message to be sent to the client specified by *hConnection*.

When you set connection settings, the lobby application will add a reference to each of the address objects specified in the call.

Interface Information

| Inherits from | <u>IUnknown</u> |
|---------------------------|----------------------------|
| Header | dplobby8.h |
| Minimum operating systems | Windows 98, Pocket PC 2002 |

IDirectPlay8LobbyClient::Close Method

Deletes the lobby client.

Syntax

```
HRESULT Close(
const DWORD dwFlags
);
```

Parameters

dwFlags [in] Reserved, must be 0.

Return Value

Returns S_OK if successful, or the following error value.

DPNERR_UNINITIALIZED The requested object has not been initialized.

IDirectPlay8LobbyClient::ConnectApplication Method

Connects a lobby-enabled application to the session specified in the <u>DPL_CONNECT_INFO</u> structure. If the application is not running, this method can be used to launch the application.

When the connection is successfully established, the lobbied application generates a <u>DPL_MSGID_CONNECT</u> system message to the message handler.

Syntax

```
HRESULT ConnectApplication(
DPL_CONNECT_INFO *const pdplConnectionInfo,
const PVOID pvUserApplicationContext,
DPNHANDLE *const phApplication,
const DWORD dwTimeOut,
const DWORD dwFlags
);
```

Parameters

pdplConnectionInfo

[in] Pointer to a **DPL_CONNECT_INFO** structure, which describes the connection parameters, including the globally unique identifier (GUID) of the application to connect to. *pvUserApplicationContext*

[in] Pointer to a context value defined for the lobby client that is passed in calls to the lobby client's message handler.

phApplication

[out] Pointer to a **DPNHANDLE** that specifies the application connect handle that is set if this method succeeds. This handle is used for further communication with the application. Additionally, this handle is used in the *phApplication* parameter in the <u>IDirectPlay8LobbyClient::ReleaseApplication</u> method. *dwTimeOut*

[in] Variable of type **DWORD** that specifies the number of milliseconds to wait for the connection to process.

dwFlags

[in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

| DPNERR_CANTLAUNCHAPPLICATION | The lobby cannot launch the specified application. |
|------------------------------|---|
| DPNERR_INVALIDFLAGS | The flags passed to this method are invalid. |
| DPNERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
| DPNERR_TIMEDOUT | The operation could not complete because it has timed out. |
| DPNERR_NOCONNECTION | No communication link was established. |
| DPNERR_DOESNOTEXIST | Requested element is not part of the address. |

Remarks

When this method returns DPNERR_NOCONNECTION, the reason is usually that the application described in the *pdplConnectionInfo* parameter has not called

IDirectPlay8LobbiedApplication::SetAppAvailable.

IDirectPlay8LobbyClient::EnumLocalPrograms Method

Enumerates the lobbied applications that are registered on the system.

Syntax

```
HRESULT EnumLocalPrograms(
GUID *const pGuidApplication,
BYTE *const pEnumData,
DWORD *const pdwEnumData,
DWORD *const pdwItems,
const DWORD dwFlags
);
```

Parameters

pGuidApplication

[in] Pointer to a variable of type globally unique identifier (GUID) that specifies the lobbied application to enumerate. This parameter is optional, and passing NULL enumerates all available lobbied applications.

pEnumData

[out] Pointer to a variable of type **BYTE**, which is filled with a description of the lobbied application.

pdwEnumData

[in] Pointer to variable of type **DWORD** that specifies the number of bytes contained in the *pEnumData* buffer. If the buffer in *pEnumData* is too small, this method returns

<u>DPNERR_BUFFERTOOSMALL</u> and sets this parameter to the size of the required buffer.

pdwltems

[out] Pointer to a variable of type **DWORD** that contains the number of <u>DPL_APPLICATION_INFO</u> structures in the *pEnumData* buffer. This parameter is filled only if the method succeeds.

dwFlags

[in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

| DPNERR_BUFFERTOOSMALL | The supplied buffer is not large enough to contain the requested data. |
|-----------------------|--|
| DPNERR_DOESNOTEXIST | Requested element is not part of the address. |
| DPNERR_INVALIDFLAGS | The flags passed to this method are invalid. |
| DPNERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |

Remarks

This method is generally called twiceonce to obtain the size of the required buffer, and then with the correct buffer size.

IDirectPlay8LobbyClient::GetConnectionSettings Method

Retrieves the set of connection settings for the specified connection. These settings can be set through a call to the <u>IDirectPlay8LobbyClient::ConnectApplication</u>, <u>IDirectPlay8LobbyClient::SetConnectionSettings</u>, or <u>IDirectPlay8LobbiedApplication::SetConnectionSettings</u> method.

When you get connection settings, a reference will be added for each address object that is returned to the user. Therefore, users must be sure to call **Release** on each address object when they are finished with the structure.

Syntax

```
HRESULT GetConnectionSettings(
    const DPNHANDLE hConnection,
    DPL_CONNECTION_SETTINGS *const pdplConnectSettings,
    DWORD *pdwDataSize,
    const DWORD dwFlags
);
```

Parameters

hConnection

[in] Handle to the connection for which to retrieve the settings. *pdplConnectSettings*

[out] Pointer to a buffer to receive the connection settings for the specified connection.

pdwDataSize

[in, out] Pointer to a **DWORD** containing the size, in bytes, of the buffer specified in the *pdplConnectSettings* structure. If the buffer is not large enough to hold the connection settings,
DPNERR_BUFFERTOOSMALL is returned and this value is set to the required buffer size. On success, this value contains the number of bytes written to the specified buffer.

dwFlags

[in] Reserved, must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

| DPNERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
|-----------------------|--|
| DPNERR_BUFFERTOOSMALL | The supplied buffer is not large enough to contain the requested data. |
| DPNERR_INVALIDOBJECT | The Microsoft® DirectPlay® object pointer is invalid. |
| DPNERR_INVALIDFLAGS | The flags passed to this method are invalid. |

IDirectPlay8LobbyClient::Initialize Method

Registers an entry point in the lobby client's code that receives notifications on changes of state for any launched applications. The message handler also receives messages from the lobbied application. This method must be called before calling any other methods of this interface.

Syntax

```
HRESULT Initialize(
    const PVOID pvUserContext,
    const PFNDPNMESSAGEHANDLER pfn,
    const DWORD dwFlags
);
```

Parameters

pvUserContext

[in] Pointer to the user-provided context value provided in calls to the message handler. Providing a user-context value is useful to differentiate messages from multiple interfaces to a common message handler.

pfn

[in] Pointer to a <u>PFNDPNMESSAGEHANDLER</u> callback function that receives all messages from the <u>IDirectPlay8LobbyClient</u> interface and indications of session changes from the <u>IDirectPlay8LobbiedApplication</u> interface.

dwFlags

[in] The following flag can be specified. DPLINITIALIZE_DISABLEPARAMVAL Disables parameter validation.

Return Value

Returns S_OK if successful, or one of the following error values.

| DPNERR_ALREADYINITIALIZED | The object has already been initialized. |
|---------------------------|---|
| DPNERR_INVALIDFLAGS | The flags passed to this method are invalid. |
| DPNERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
| DPNERR_NOTALLOWED | This function is not allowed on this object. |

Remarks

Call this is method first after using **CoCreateInstance** to obtain the **IDirectPlay8LobbyClient** interface.

Note Only one instance of **IDirectPlay8LobbyClient** and **IDirectPlay8LobbiedApplication** is allowed to be running for each process.

IDirectPlay8LobbyClient::ReleaseApplication Method

Releases a lobbied application and closes the connection between the lobby client and the application. This method should be called whenever a lobby client has finished its session with an application.

Syntax

```
HRESULT ReleaseApplication(
const DPNHANDLE hApplication,
const DWORD dwFlags
);
```

Parameters

hApplication

[in] The DPNHANDLE of the lobbied application to release. This value is set in the *phApplication* parameter of the
 IDirectPlay8LobbyClient::ConnectApplication method. You can
 also specify the following flag.
 DPLHANDLE_ALLCONNECTIONS
 All application connections will be released.

dwFlags

[in] Reserved, must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDFLAGSThe flags passed to this method are invalid.DPNERR_INVALIDHANDLEThe handle specified is invalid.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.

IDirectPlay8LobbyClient::Send Method

Sends a message to a lobbied application that was launched by this lobby client or was connected by this lobby client.

This method sends a <u>DPL_MSGID_RECEIVE</u> system message to the target's message handler.

Syntax

HRESULT Send(const DPNHANDLE *hConnection*, BYTE *const *pBuffer*, const DWORD *pBufferSize*, const DWORD *dwFlags*

);

Parameters

hConnection

[in] Variable of type **DPNHANDLE** that specifies the target for the message transmission. You may also specify the following flag.

DPLHANDLE_ALLCONNECTIONS

The message you have specified will be sent to all lobbied applications that are connected to your lobby client application.

pBuffer

[in] Pointer to an array of bytes that contains the message. *pBufferSize*

[in] Variable of type **DWORD** that specifies the size of the message buffer in the *pBuffer* parameter, in bytes. This

parameter must be at least 1 byte and no more than 64 KB. *dwFlags*

[in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDFLAGSThe flags passed to this method are invalid.DPNERR_INVALIDHANDLEThe handle specified is invalid.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_SENDTOOLARGEThe buffer was too large.

Remarks

If the buffer size is larger than 64 KB, the method returns DPNERR_SENDTOOLARGE. If the buffer size is set to 0, the method returns DPNERR_INVALIDPARAM.

IDirectPlay8LobbyClient::SetConnectionSettings Method

Sets the connection settings to be associated with the specified connection. Calling this method will generate a DPL_MSGID_CONNECTION_SETTINGS message to be sent to the client specified by *hConnection*.

When you set connection settings, the lobby application will add a reference to each of the address objects specified in the call.

Syntax

```
HRESULT SetConnectionSettings(
    const DPNHANDLE hConnection,
    const DPL_CONNECTION_SETTINGS *const pdplConnectSettings
    const DWORD dwFlags
);
```

Parameters

hConnection

[in] Handle to the connection to set the settings for. You may also specify the following flag.

DPLHANDLE_ALLCONNECTIONS

The connection settings will be updated for all the lobbied applications you are connected to.

pdplConnectSettings

[in] Pointer to a DPL_CONNECTION_SETTINGS structure containing the settings associated with the specified connection. *dwFlags*

[in] Reserved, must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_INVALIDOBJECTThe Microsoft® DirectPlay® object pointer is invalid.DPNERR_INVALIDFLAGSThe flags passed to this method are invalid.

 $\ensuremath{\textcircled{}^{\circ}}$ 2003 Microsoft Corporation. All rights reserved.

IDirectPlay8NATResolver Interface

Applications use the methods of the **IDirectPlay8NATResolver** interface to improve connectivity for players behind Network Address Translation (NAT) devices that are not Universal Plug and Play (UPnP).

Note This interface is only for players using the Internet Protocol, version 4 (IPv4) service provider.

IDirectPlay8NATResolver Members

| <u>Close</u> | Closes the IDirectPlay8NATResolver interface. |
|---------------------|--|
| EnumDevices | Enumerates the list of available devices on which the NAT Resolver can be started. |
| <u>GetAddresses</u> | Retrieves the list of IDirectPlay8Address objects on which the NAT Resolver has been started. |
| <u>Initialize</u> | Initializes the IDirectPlay8NATResolver interface. This method must be called before calling any other methods of this interface. |
| <u>Start</u> | Launches the NAT Resolver server using the specified device addresses. |

Interface Information

| Inherits from | IUnknown |
|---------------------------|----------------------------|
| Header | dplay8.h |
| Minimum operating systems | Windows 98, Pocket PC 2002 |

IDirectPlay8NATResolver::Close Method

Closes the IDirectPlay8NATResolver interface.

Syntax

```
HRESULT Close(
const DWORD dwFlags);
```

Parameters

dwFlags [in] Reserved. Set to NULL.

Return Value

Returns DPN_OK if successful, or one of the following error values.

DPNERR_INVALIDFLAGSThe flags passed to this method are invalid.DPNERR_INVALIDOBJECTThe Microsoft® DirectPlay® object pointer is invalid.DPNERR_UNINITIALIZEDThe requested object has not been initialized.

IDirectPlay8NATResolver::EnumDevices Method

Enumerates the list of available devices on which the Network Address Translation (NAT) Resolver can be started.

Syntax

```
HRESULT EnumDevices(
    DPN_SERVICE_PROVIDER_INFO *const pSPInfoBuffer,
    PDWORD const pdwBufferSize,
    PDWORD const pdwNumDevices,
    const DWORD dwFlags
);
```

Parameters

pSPInfoBuffer

[out] Pointer to an array of <u>DPN_SERVICE_PROVIDER_INFO</u> structures that will be filled with service provider information. *pdwBufferSize*

[out] Pointer to **DWORD**, which is filled with the size of the *pSPInfoBuffer* buffer.

pdwNumDevices

Pointer to **DWORD** which is filled with the number of **DPN_SERVICE_PROVIDER_INFO** structures returned in *pSPInfoBuffer*.

dwFlags

[in] Reserved. Set to NULL.

Return Value

Returns DPN_OK if successful, or one of the following error values.

| DPNERR_BUFFERTOOSMALL | The supplied buffer is not large enough to contain the requested data. |
|-----------------------|--|
| DPNERR_INVALIDFLAGS | The flags passed to this method are invalid. |
| DPNERR_INVALIDOBJECT | The Microsoft® DirectPlay® object pointer is invalid. |
| | |

Remarks

If the value at *pdwBufferSize* indicates the buffer is too small to hold the list of devices, the size required is placed in *pdwBufferSize* and DPNERR_BUFFERTOOSMALL is returned. Otherwise the size written is placed in *pdwBufferSize*

IDirectPlay8NATResolver::GetAddresses Method

Retrieves the list of <u>IDirectPlay8Address</u> objects on which the Network Address Translation (NAT) Resolver has been started.

Syntax

```
HRESULT GetAddresses(
    IDirectPlay8Address **const ppAddresses,
    DWORD *const pdwNumAddresses,
    const DWORD dwFlags
);
```

Parameters

ppAddresses

[out] Receives an array of **IDirectPlay8Address** address objects.

pdwNumAddresses

[in] Number of address objects returned in the *ppAddresses*. *dwFlags*

[in] Reserved. Set to NULL.

Return Value

Returns DPN_OK if successful, or one of the following error values.

| DPNERR_BUFFERTOOSMALL | The supplied buffer is not large enough to contain the requested data. |
|-----------------------|--|
| DPNERR_INVALIDFLAGS | The flags passed to this method are invalid. |
| DPNERR_INVALIDOBJECT | The Microsoft® DirectPlay® object pointer is invalid. |
| DPNERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
| DPNERR_NOCONNECTION | No communication link was established. |
| DPNERR_UNINITIALIZED | The requested object has not been initialized. |

Remarks

IDirectPlay8Address::Release must be called on each of the address interface pointers returned by IDirectPlay8NATResolver::GetAddresses when you are finished

with the objects.

IDirectPlay8NATResolver::Initialize Method

Initializes the <u>IDirectPlay8NATResolver</u> interface. This method must be called before calling any other methods of this interface.

Syntax

```
HRESULT Initialize(
const PVOID pvUserContext,
const PFNDPNMESSAGEHANDLER pfn,
const DWORD dwFlags
);
```

Parameters

pvUserContext

[in] Pointer to the user-provided context value in calls to the message handler. A user-provided context value can be used to differentiate messages coming from multiple interfaces to a common message handler.

pfn

[in] Pointer to a <u>PFNDPNMESSAGEHANDLER</u> callback function that is used to receive all messages.

dwFlags

[in] The following flag can be set.

DPNINITIALIZE_DISABLEPARAMVAL

Passing this flag will disable parameter validation for the current object.

Return Value

Returns DPN_OK if successful, or one of the following error values.

| DPNERR_ALREADYINITIALIZED | The object has already been initialized. |
|---------------------------|---|
| DPNERR_INVALIDFLAGS | The flags passed to this method are invalid. |
| DPNERR_INVALIDOBJECT | The Microsoft® DirectPlay® object pointer is invalid. |
| | |

 $\ensuremath{\textcircled{}^{\circ}}$ 2003 Microsoft Corporation. All rights reserved.

IDirectPlay8NATResolver::Start Method

Launches the Network Address Translation (NAT) Resolver server using the specified device addresses.

Syntax

```
HRESULT Start(
    IDirectPlay8Address **const ppDevices,
    const DWORD dwNumDevices,
    const DWORD dwFlags
);
```

Parameters

ppDevices

[in] An array of <u>IDirectPlay8Address</u> device address objects. Set to NULL to use all Internet Protocol (IP) devices. See Remarks. *dwNumDevices*

[in] Number of device addresses in the *ppDevices* array. *dwFlags*

[in] Reserved. Set to NULL.

Return Value

Returns DPN_OK if successful, or one of the following error values.

| DPNERR_INVALIDDEVICEADDRESS | The address for the local computer or adapter is invalid. |
|-----------------------------|---|
| DPNERR_INVALIDFLAGS | The flags passed to this method are invalid. |
| DPNERR_INVALIDOBJECT | The Microsoft® DirectPlay® object pointer is invalid. |
| DPNERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
| DPNERR_UNINITIALIZED | The requested object has not been initialized. |

Remarks

The *ppDevices* parameter must contain address objects which have called <u>IDirectPlay8Address::SetSP</u> with the *pGuidSP* parameter set to CLSID_DP8SP_TCPIP. If <u>IDirectPlay8Address::SetDevice</u> has not been called to set the device globally unique identifier (GUID), all adapters will be used. Only eight addresses can be specified simultaneously.

IDirectPlay8Peer Interface

Applications use the methods of the **IDirectPlay8Peer** interface to create a peer-to-peer Microsoft® DirectPlay® session.

IDirectPlay8Peer Members

| AddPlayerToGroup | Adds a peer to a group. |
|-----------------------------|---|
| | When this method is called, all peers connected to the application receive a <u>DPN_MSGID_ADD_PLAYER_TO_GROUP</u> message. |
| <u>CancelAsyncOperation</u> | Cancels asynchronous requests. For instance, several methods of the IDirectPlay8Peer interface run asynchronously by default. Depending on the situation, you might want to cancel requests before they are processed. All the methods of this interface that can run asynchronously return an <i>hAsyncHandle</i> parameter. |
| | Specific requests are canceled by passing the <i>hAsyncHandle</i> of the request in this method's <i>hAsyncHandle</i> parameter. You can cancel all pending asynchronous operations by calling this method, specifying NULL in the <i>hAsyncHandle</i> parameter, and specifying DPNCANCEL_ALL_OPERATIONS in the <i>dwFlags</i> parameter. If a specific handle is provided to this method, no flags should be set. |
| <u>Close</u> | Closes the open connection to a session and uninitializes the IDirectPlay8Peer object. This method must be called on any object successfully initialized with <u>IDirectPlay8Peer::Initialize</u> . |
| <u>Connect</u> | Establishes the connection to all the peers in a peer-to-peer session. When a connection is established, the communication channel on the interface is open and the application should expect messages to arrive immediately. No messages can be sent by way of the <u>IDirectPlay8Peer::SendTo</u> method until the connection has completed. |
| <u>CreateGroup</u> | Creates a group in the current session. A group is a logical collection of players. |
| <u>DestroyGroup</u> | Deletes a group created by the <u>IDirectPlay8Peer::CreateGroup</u> method. This method can be called by any peer in the session. |
| <u>DestroyPeer</u> | Deletes a peer from the session. |
| EnumGroupMembers | Retrieves a list of all players in a group. |
| EnumHosts | Enumerates applications that host DirectPlay games. |

| EnumPlayersAndGroups | Retrieves a list of all the player and/or group identifiers for the session. |
|------------------------------|--|
| EnumServiceProviders | Enumerates all the registered service providers available to the application. |
| GetApplicationDesc | Retrieves the full application description for the connected application. |
| <u>GetCaps</u> | Retrieves the <u>DPN_CAPS</u> or <u>DPN_CAPS_EX</u> structure for the current interface. |
| <u>GetConnectionInfo</u> | Retrieves statistical information about the connection between the local application and the specified remote player. |
| <u>GetGroupContext</u> | Retrieves the group context value for the specified group. |
| <u>GetGroupInfo</u> | Retrieves a block of data associated with a group, including the group name. This method is typically called after a <u>DPN_MSGID_GROUP_INFO</u> system message is received indicating that the group data has been modified. |
| <u>GetLocalHostAddresses</u> | Retrieves the local addresses being used to host the session. |
| <u>GetPeerAddress</u> | Retrieves the address for the specified remote player in the session. |
| <u>GetPeerInfo</u> | Retrieves peer information set for the specified peer. |
| <u>GetPlayerContext</u> | Retrieves the player context value for the specified peer. |
| <u>GetSendQueueInfo</u> | Used by the application to monitor the size of the send queue. |
| <u>GetSPCaps</u> | Retrieves the <u>DPN_SP_CAPS</u> structure for the specified service provider. |
| Host | Creates a new peer-to-peer session, hosted by the local computer. |
| <u>Initialize</u> | Registers an entry point in the peer's code that receives all the messages from the IDirectPlay8Peer interface and from remote peers. This method must be called before calling any other methods of this interface. |
| RegisterLobby | Allows launched applications to automatically propagate game status to the lobby. |
| RemovePlayerFromGroup | Removes a peer from a group. When this method is called all peers connected to the application receive a <u>DPN_MSGID_REMOVE_PLAYER_FROM_GROUP</u> message. |
| <u>ReturnBuffer</u> | Retrieves message buffers provided to the application through the pReceiveData member of the <u>DPN_MSGID_RECEIVE</u> system message. If the user's message handler returns <u>DPNSUCCESS_PENDING</u> to the RECEIVE callback, DirectPlay assumes that ownership of the buffer is transferred to the application, and neither |

| | frees nor modifies it until ownership is returned to DirectPlay through this call. |
|-------------------------|--|
| <u>SendTo</u> | Transmits data to another peer or group within the session by sending a message to the appropriate message handlers. The message can be sent synchronously or asynchronously. |
| SetApplicationDesc | Changes the settings for the application that is being hosted. Only some settings can be changed. |
| <u>SetCaps</u> | Sets the DPN_CAPS or DPN_CAPS_EX structure for the current interface. |
| <u>SetGroupInfo</u> | Sets a block of data associated with a group, including the name of the group. Calling this method generates a DPN_MSGID_GROUP_INFO message, which is sent to all the peers connected to the application. |
| <u>SetPeerInfo</u> | Sets the static settings of the local peer. Call this method before connecting to relay basic player information with the application. When the peer successfully connects with the application, information set through this method can be retrieved by other players by calling the <u>IDirectPlay8Peer::GetPeerInfo</u> method. |
| <u>SetSPCaps</u> | Sets the DPN_SP_CAPS structure for the specified service provider. |
| <u>TerminateSession</u> | Terminates the current DirectPlay session. |

Interface Information

| Inherits from | <u>IUnknown</u> |
|---------------------------|----------------------------|
| Header | dplay8.h |
| Minimum operating systems | Windows 98, Pocket PC 2002 |

 $\ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.

IDirectPlay8Peer::AddPlayerToGroup Method

Adds a peer to a group.

When this method is called, all peers connected to the application receive a <u>DPN_MSGID_ADD_PLAYER_TO_GROUP</u> message.

Syntax

```
HRESULT AddPlayerToGroup(
const DPNID idGroup,
const DPNID idClient,
PVOID const pvAsyncContext,
DPNHANDLE *const phAsyncHandle,
const DWORD dwFlags
```

);

Parameters

idGroup

[in] Variable of type **DPNID** that specifies the identifier of the group to add the peer to.

idClient

[in] Variable of type **DPNID** that specifies the identifier of the peer that is added to the group.

pvAsyncContext

[in] Pointer to the user-supplied context, which is returned in the **pvUserContext** member of the

DPN_MSGID_ASYNC_OP_COMPLETE system message. This parameter is optional and may be set to NULL.

phAsyncHandle

[out] A **DPNHANDLE**. A value will be returned. However, Microsoft® DirectPlay® does not permit cancellation of this operation, so the value cannot be used.

dwFlags

[in] Flag that controls how this method is processed. The

following flag can be set for this method. DPNADDPLAYERTOGROUP_SYNC Causes the method to process synchronously.

Return Value

Returns S_OK if this method is processed synchronously and is successful. By default, this method runs asynchronously and usually returns <u>DPNSUCCESS_PENDING</u>. It may also return one of the following error values.

| DPNERR_INVALIDFLAGS | The flags passed to this method are invalid. |
|-----------------------------|---|
| DPNERR_INVALIDGROUP | The group ID is not recognized as a valid group ID for this game session. |
| DPNERR_INVALIDPLAYER | The player ID is not recognized as a valid player ID for this game session. |
| DPNERR_PLAYERALREADYINGROUP | The player ID is already included in the group. |

Remarks

Any peer can add itself or another peer to an existing group. When the peer is successfully added to the group, all messages sent to the group are also sent to the peer.
IDirectPlay8Peer::CancelAsyncOperation Method

Cancels asynchronous requests. For instance, several methods of the IDirectPlay8Peer interface run asynchronously by default. Depending on the situation, you might want to cancel requests before they are processed. All the methods of this interface that can run asynchronously return an *hAsyncHandle* parameter.

Specific requests are canceled by passing the *hAsyncHandle* of the request in this method's *hAsyncHandle* parameter. You can cancel all pending asynchronous operations by calling this method, specifying NULL in the *hAsyncHandle* parameter, and specifying DPNCANCEL_ALL_OPERATIONS in the *dwFlags* parameter. If a specific handle is provided to this method, no flags should be set.

Syntax

```
HRESULT CancelAsyncOperation(
    const DPNHANDLE hAsyncHandle,
    const DWORD dwFlags
);
```

Parameters

hAsyncHandle

[in] Handle of the asynchronous operation to stop. You receive this handle when you call one of several methods that support asynchronous operations. This value can be set to NULL to stop all operations or a particular type of asynchronous request. If a particular handle is specified, the *dwFlags* parameter must be 0. If one of the DPNCANCEL_PLAYER_SENDS flags is specified in the *dwFlags* parameter, *hAsyncHandle* must be set to a player's DPNID. dwFlags

[in] Flag that specifies which asynchronous request is to be canceled. One of the following flags can be set.

DPNCANCEL_ENUM

Cancel all asynchronous <u>IDirectPlay8Peer::EnumHosts</u> requests. A single **IDirectPlay8Peer::EnumHosts** request can be canceled by specifying the handle returned from the **IDirectPlay8Peer::EnumHosts** method.

DPNCANCEL_CONNECT

Cancel an asynchronous <u>IDirectPlay8Peer::Connect</u> request.

DPNCANCEL_SEND

Cancel an asynchronous <u>IDirectPlay8Peer::SendTo</u> request.

DPNCANCEL_PLAYER_SENDS

Cancel all asynchronous **IDirectPlay8Peer::SendTo** requests for the player specified in the *hAsyncHandle* parameter.

DPNCANCEL_PLAYER_SENDS_PRIORITY_LOW

Cancel low-priority asynchronous

IDirectPlay8Peer::SendTo requests for the player specified in the *hAsyncHandle* parameter.

DPNCANCEL_PLAYER_SENDS_PRIORITY_NORMAL Cancel normal-priority asynchronous

IDirectPlay8Peer::SendTo requests for the player specified in the *hAsyncHandle* parameter.

DPNCANCEL_PLAYER_SENDS_PRIORITY_HIGH Cancel high-priority asynchronous

IDirectPlay8Peer::SendTo requests for the player specified in the *hAsyncHandle* parameter.

DPNCANCEL_ALL_OPERATIONS

Cancel all asynchronous IDirectPlay8Peer::Connect, IDirectPlay8Peer::SendTo, and IDirectPlay8Peer::EnumHosts operations.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_CANNOTCANCELThe operation could not be canceled.DPNERR_INVALIDFLAGSThe flags passed to this method are invalid.DPNERR_INVALIDHANDLEThe handle specified is invalid.

Remarks

You can use this method to cancel an asynchronous operation for the **IDirectPlay8Peer::Connect**, **IDirectPlay8Peer::SendTo**, and **IDirectPlay8Peer::EnumHosts** methods. Microsoft® DirectPlay® does not support cancellation of other asynchronous operations.

You can cancel a send request by providing the handle returned from the IDirectPlay8Peer::SendTo method. A DPN_MSGID_SEND_COMPLETE, or DPN_MSGID_CONNECT_COMPLETE system message is still posted to the application's message handler for each asynchronous send request that is sent without the DPNSEND_NOCOMPLETE flag set. Send requests that are canceled by this method return DPNERR_USERCANCEL in the hResultCode member of the DPN_MSGID_SEND_COMPLETE message.

If you set the DPNCANCEL_ALL_OPERATIONS, DPNCANCEL_CONNECT, DPNCANCEL_SEND, or DPNCANCEL_ENUM flags in *dwFlags*, DirectPlay will attempt to cancel all matching operations. This method will return an error if any attempted cancellation fails, even though some cancellations may have been successful.

If you set one of the DPNCANCEL_PLAYER_SENDS flags in *dwFlags*, you must specify a player's DPNID in *hAsyncHandle*. This will cancel all pending **IDirectPlay8Peer::SendTo** requests where

the DPNID specified in the *dpnid* parameter matches the value set in the *hAsyncHandle* parameter.

Note The completion message might not arrive until after this method returns. Do not assume that the operation has been terminated until you have received a DPN_MSGID_SEND_COMPLETE, DPN_MSGID_CONNECT_COMPLETE, or DPN_MSGID_ASYNC_OP_COMPLETE message.

IDirectPlay8Peer::Close Method

Closes the open connection to a session and uninitializes the <u>IDirectPlay8Peer</u> object. This method must be called on any object successfully initialized with <u>IDirectPlay8Peer::Initialize</u>.

Syntax

```
HRESULT Close(
const DWORD dwFlags
);
```

Parameters

dwFlags [in] The following flag can be specified. DPNCLOSE_IMMEDIATE Close immediately. Do not wait for outstanding calls to complete.

Return Value

Returns S_OK if successful, or the following error value.

DPNERR_UNINITIALIZED The requested object has not been initialized.

Remarks

This method will cancel any operations still outstanding, including guaranteed messages that are in the queue waiting to be sent. Messages that have already been sent as guaranteed will continue to be retried until acknowledgement of their delivery has been received. To make sure all messages are sent, wait for all outstanding <u>IDirectPlay8Peer::SendTo</u> calls to complete before calling **IDirectPlay8Peer::Close**.

If this method is called by the host player and host migration has been enabled, the host will migrate and the session will continue for other players. If host migration is not enabled, the session will terminate. If the host player wants to terminate the session without host migration, IDirectPlay8Peer::TerminateSession should be called before calling IDirectPlay8Peer::Close. See Host Migration for more information.

To start a new session or connect to another session after calling IDirectPlay8Peer::Close, you must first call IDirectPlay8Peer::Initialize on the IDirectPlay8Peer object before calling IDirectPlay8Peer::Host or IDirectPlay8Peer::Connect.

Calling **IDirectPlay8Peer::Close** will invalidate any <u>DPN_CAPS</u>, <u>DPN_CAPS_EX</u>, and <u>DPN_SP_CAPS</u> associated with the **IDirectPlay8Peer** object.

See Also

Leaving a Peer-to-Peer Session, Terminating a Peer-to-Peer Session

IDirectPlay8Peer::Connect Method

Establishes the connection to all the peers in a peer-to-peer session. When a connection is established, the communication channel on the interface is open and the application should expect messages to arrive immediately. No messages can be sent by way of the <u>IDirectPlay8Peer::SendTo</u> method until the connection has completed.

Syntax

HRESULT Connect(const DPN_APPLICATION_DESC *const pdnAppDesc, IDirectPlay8Address *const pHostAddr, IDirectPlay8Address *const pDeviceInfo, const DPN_SECURITY_DESC *const pdnSecurity, const DPN_SECURITY_CREDENTIALS *const pdnCredentials, const void *const pvUserConnectData, const DWORD dwUserConnectDataSize, void *const pvPlayerContext, void *const pvAsyncContext, DPNHANDLE *const phAsyncHandle, const DWORD dwFlags

);

Parameters

pdnAppDesc

[in] Pointer to a <u>DPN_APPLICATION_DESC</u> structure that describes the application. Only some of the members of this structure are used by this method. The only members that you must set are **dwSize** and **guidApplication**. You can also set **guidInstance**, **pwszPassword**, and **dwFlags**.

pHostAddr

[in] Pointer to an <u>IDirectPlay8Address</u> interface that specifies the addressing information to use to connect to the computer that is

hosting. The user can be queried for any missing address information if you set the

DPNENUMHOSTS_OKTOQUERYFORADDRESSING flag in the *dwFlags* parameter.

pDeviceInfo

[in] Pointer to an **IDirectPlay8Address** interface that specifies the network adapter (for example, network interface card, modem, and so on) to use to connect to the server. Some service providers allow this parameter to be NULL or be an address object containing only the service provider component. In this case, they will use the most appropriate device to reach the designated host. If you set the

DPNCONNECT_OKTOQUERYFORADDRESSING flag in *dwFlags*, the user can be queried for any missing address information.

pdnSecurity

[in] Reserved. Must be NULL.

pdnCredentials

[in] Reserved. Must be NULL.

pvUserConnectData

[in] Pointer to application-specific data provided to the host or server to further validate the connection. Microsoft® DirectPlay® will make a copy of this data when the method is called and therefore you can modify or destroy this data once the connection is complete. This data is sent to the

DPN_MSGID_INDICATE_CONNECT message in the

pvUserConnectData member. This parameter is optional and can be set to NULL if no additional connection validation is provided by the user code.

dwUserConnectDataSize

[in] Variable of type **DWORD** that specifies the size of the data contained in the *pvUserConnectData* parameter.

pvPlayerContext

[in] Pointer to the context value of the local player. This value is preset when the local computer handles the

DPN_MSGID_CREATE_PLAYER message. This parameter is optional and can be set to NULL.

pvAsyncContext

[in] Pointer to the user-supplied context, which is returned in the

pvUserContext member of the

<u>DPN_MSGID_CONNECT_COMPLETE</u> system message. This parameter is optional and can be set to NULL.

phAsyncHandle

[out] A **DPNHANDLE**. When the method returns, phAsyncHandle will point to a handle that you can pass to <u>IDirectPlay8Peer::CancelAsyncOperation</u> to cancel the operation. This parameter must be set to NULL if you set the DPNCONNECT_SYNC flag in *dwFlags*.

dwFlags

[in] Flag that describes the connection mode. You can set the following flags.

DPNCONNECT_OKTOQUERYFORADDRESSING

Setting this flag will display a standard DirectPlay dialog box, which queries the user for more information if not enough information is passed in this method.

DPNCONNECT_SYNC

Process the connection request synchronously. Your message handler still receives a

DPN_MSGID_CONNECT_COMPLETE message, so that you can process any connection reply data from the host. You will receive this message before

IDirectPlay8Peer::Connect returns.

Return Value

Returns S_OK if this method is processed synchronously and is successful. By default, this method runs asynchronously and normally returns <u>DPNSUCCESS_PENDING</u>. It might also return one of the following error values.

| DPNERR_HOSTREJECTEDCONNECTION | The connection request was rejected. Check the ReplyData member of the DPN_MSGID_CONNECT_COMPLETE type for details. |
|-------------------------------|--|
| DPNERR_INVALIDAPPLICATION | The globally unique identifier (GUID) supplied for the application is invalid. |
| DPNERR_INVALIDDEVICEADDRESS | The address for the local computer or adapter is invalid. |
| DPNERR_INVALIDFLAGS | The flags passed to this method are invalid. |
| DPNERR_INVALIDHOSTADDRESS | The specified remote address is invalid. |
| DPNERR_INVALIDINSTANCE | The GUID for the application instance is invalid. |

| DPNERR_INVALIDINTERFACE | The interface parameter is invalid. This value will be returned in a connect request if the connecting player was not a client in a client/server game or a peer in a peer-to-peer game. |
|---------------------------|--|
| DPNERR_INVALIDPASSWORD | An invalid password was supplied when attempting to join a session that requires a password. |
| DPNERR_NOCONNECTION | No communication link was established. |
| DPNERR_NOTHOST | The client attempted to connect to a nonhost computer. Additionally, this error value may be returned by a nonhost that tried to set the application description. |
| DPNERR_SESSIONFULL | The request to connect to the host or server failed because the maximum number of players allotted for the session has been reached. |
| DPNERR_ALREADYINITIALIZED | The object has already been initialized. |

Remarks

Before this method is called, you can obtain application descriptions and the addresses of the associated hosts by calling <u>IDirectPlay8Peer::EnumHosts</u>. When you call

IDirectPlay8Peer::EnumHosts,

DPN_MSGID_ENUM_HOSTS_RESPONSE messages are sent to your message handler with the **IDirectPlay8Address** objects and the **DPN_APPLICATION_DESC** structure for each host found. This information can be passed without modification to the **IDirectPlay8Peer::Connect** method.

It is not required to enumerate hosts before calling IDirectPlay8Peer::Connect if you know the appropriate host and device information.

If you do call the **IDirectPlay8Peer::EnumHosts** method and you want to ensure better Network Address Translation (NAT) and proxy support when using the Transmission Control Protocol/Internet Protocol (TCP/IP) service provider or to prevent redialing with the modem service provider, keep the enumeration active when calling the **IDirectPlay8Peer::Connect** method. To prevent the

enumeration from completing, set the *dwEnumCount* parameter to INFINITE and do not use the

IDirectPlay8Peer::CancelAsyncOperation to terminate the enumeration before the connect operation has completed. You should also pass the pAddressSender and pAddressDevice address objects in the DPNMSG_ENUM_HOSTS_RESPONSE message without modification into the *pHostAddr* and *pDeviceInfo* parameters of the IDirectPlay8Peer::Connect method. To pass the address objects to IDirectPlay8Peer::Connect outside of the callback function, use IDirectPlay8Address::Duplicate or IDirectPlay8Address::AddRef to prevent the object from being destroyed and store the pointers using thread-safe code. DirectPlay will automatically cancel the enumeration when the connect completes with DPN_OK or when IDirectPlay8Peer::Close is called.

Although multiple enumerations can be run concurrently and can be run across the duration of a connection, only one connection is allowed per interface. To establish a connection to more than one application, you must create another interface.

When this method is called, a **DPN_MSGID_INDICATE_CONNECT** message is posted to the host's message handler. When the host handles this message, it can specify connection reply data that the player will receive with the **DPN_MSGID_CONNECT_COMPLETE** message. If the host accepts the connection, the connection reply data might contain custom startup information. If the connection was rejected, the connection reply data might contain of the rejection.

If <u>IDirectPlay8Client::Connect</u> is called synchronously, the following outcomes are possible.

- Connection Successful. The application will receive a **DPN_MSGID_CONNECT_COMPLETE** message containing the success code and the **IDirectPlay8Client::Connect** method will return with DPN_OK.
- Connection fails because the host rejects the connection. The application will receive a
 DPN_MSGID_CONNECT_COMPLETE message containing the DPNERR_HOSTREJECTEDCONNECTION failure code. The IDirectPlay8Client::Connect method will also return with the error code DPNERR_HOSTREJECTEDCONNECTION. The DPN_MSGID_CONNECT_COMPLETE message provides an opportunity for the client application to inspect any data the host returns with the rejection.
- Connection fails for any other reason. The application will not receive a **DPN_MSGID_CONNECT_COMPLETE** message, and the **IDirectPlay8Client::Connect** method will return with the appropriate error code.

If **IDirectPlay8Client::Connect** is called asynchronously, the method returns immediately with DPNSUCCESS_PENDING. A **DPN_MSGID_CONNECT_COMPLETE** message will follow after the connection completes, containing the result of the connection. The only time the method does not return DPNSUCCESS_PENDING is when validation of the supplied parameters fails, in which case the appropriate error code is returned.

When the connection request completes, all outstanding enumerations are canceled with the return of DPNERR_USERCANCEL.

You must call **IDirectPlay8Peer::Close** to end the connection to the host.

Data Value Summary specifies the required addressing information for each service provider.

Note If you set the

DPNCONNECT_OKTOQUERYFORADDRESSING flag in *dwFlags*, the service provider might attempt to display a dialog box to ask the user to complete the address information. You must have a visible window present when the service provider tries to display the dialog box, or your application will lock.

IDirectPlay8Peer::CreateGroup Method

Creates a group in the current session. A group is a logical collection of players.

Syntax

```
HRESULT CreateGroup(
const DPN_GROUP_INFO *const pdpnGroupInfo,
VOID *const pvGroupContext,
VOID *const pvAsyncContext,
DPNHANDLE *const phAsyncHandle,
const DWORD dwFlags
```

```
);
```

Parameters

pdpnGroupInfo

[in] Pointer to a <u>DPN_GROUP_INFO</u> structure that contains the group description.

pvGroupContext

[in] Pointer to the group's context value. This value is preset when the local application's message handler receives the associated DPN_MSGID_CREATE_GROUP message. This parameter is optional and may be set to NULL.

pvAsyncContext

[in] Pointer to the user-supplied context, which is returned in the **pvUserContext** member of the

DPN_MSGID_ASYNC_OP_COMPLETE system message. This parameter is optional and may be set to NULL.

phAsyncHandle

[out] A **DPNHANDLE**. A value will be returned. However, Microsoft® DirectPlay® does not permit cancellation of this operation, so the value cannot be used.

dwFlags

[in] Flag that controls how this method is processed. The

following flag can be set for this method. DPNCREATEGROUP_SYNC Causes the method to process synchronously.

Return Value

Returns S_OK if this method is processed synchronously and is successful. By default, this method is run asynchronously and normally returns <u>DPNSUCCESS_PENDING</u>. It can also return the following error value.

DPNERR_INVALIDFLAGS The flags passed to this method are invalid.

Remarks

DirectPlay does not maintain hierarchical groups because these can easily be implemented with flat groups and expeditious use of the group data.

All peers receive a <u>DPN_MSGID_CREATE_GROUP</u> message when this method is called.

Note Multicasting is not supported for this release.

IDirectPlay8Peer::DestroyGroup Method

Deletes a group created by the <u>IDirectPlay8Peer::CreateGroup</u> method. This method can be called by any peer in the session.

Syntax

```
HRESULT DestroyGroup(
const DPNID idGroup,
PVOID const pvAsyncContext,
DPNHANDLE *const phAsyncHandle,
const DWORD dwFlags
);
```

Parameters

idGroup

[in] Variable of type **DPNID** that should be set to the identifier of the group to be deleted.

pvAsyncContext

[in] Pointer to the user-supplied context, which is returned in the **pvUserContext** member of the

<u>DPN_MSGID_ASYNC_OP_COMPLETE</u> system message. This parameter is optional and may be set to NULL.

phAsyncHandle

[out] A **DPNHANDLE**. A value will be returned. However, Microsoft® DirectPlay® does not permit cancellation of this operation, so the value cannot be used.

dwFlags

[in] Flag that controls how this method is processed. The following flag can be set for this method.

DPNDESTROYGROUP_SYNC

Causes the method to process synchronously.

Return Value

Returns S_OK if this method is processed synchronously and is successful. By default, this method is run asynchronously and normally returns <u>DPNSUCCESS_PENDING</u>. It can also return one of the following error values.

DPNERR_INVALIDFLAGSThe flags passed to this method are invalid.DPNERR_INVALIDGROUPThe group ID is not recognized as a valid group ID for this game session.

IDirectPlay8Peer::DestroyPeer Method

Deletes a peer from the session.

Syntax

```
HRESULT DestroyPeer(
   const DPNID dpnidClient,
   void *const pDestroyInfo,
   const DWORD dwDestroyInfoSize,
   const DWORD dwFlags
);
```

Parameters

dpnidClient

[in] Variable of type **DPNID** that specifies the identifier of the peer to delete.

pDestroyInfo

[in] Pointer to a value that describes additional delete data information.

dwDestroyInfoSize

[in] Variable of type **DWORD** that specifies the size of the data contained in the *pDestroyInfo* parameter.

dwFlags

[in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

```
DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_INVALIDPLAYERThe player ID is not recognized as a valid player ID for this game session.DPNERR_NOTHOSTThe client attempted to connect to a nonhost computer. Additionally, this error value<br/>may be returned by a nonhost that tried to set the application description.
```

Remarks

A player can only be deleted by the session host. The deleted player will be notified through a <u>DPN_MSGID_TERMINATE_SESSION</u> message. The structure associated with the message will contain the data passed through the *pDestroyInfo* parameter. If any other session member calls this method, it will fail, and return DPNERR_NOTHOST.

IDirectPlay8Peer::EnumGroupMembers Method

Retrieves a list of all players in a group.

Syntax

```
HRESULT EnumGroupMembers(
const DPNID dpnid,
DPNID *const prgdpnid,
DWORD *const pcdpnid,
const DWORD dwFlags
);
```

Parameters

dpnid

[in] Variable of type **DPNID** that specifies the group that contains the players to enumerate.

prgdpnid

[out] Pointer to an array that will contain the identifiers of the group's players.

pcdpnid

[in] Pointer to a variable of type **DWORD** that specifies the number of identifiers that can be contained in the buffer pointed to by *dpnid*. If the buffer is too small, this method returns DPNERR_BUFFERTOOSMALL and this parameter contains the number of entries that are required.

dwFlags

[in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

| DPNERR_BUFFERTOOSMALL | The supplied buffer is not large enough to contain the requested data. |
|-----------------------|---|
| DPNERR_INVALIDFLAGS | The flags passed to this method are invalid. |
| DPNERR_INVALIDGROUP | The group ID is not recognized as a valid group ID for this game session. |

Remarks

Because player information changes frequently, the required buffer size returned may change between subsequent calls. Check and reallocate the buffer until the method succeeds.

IDirectPlay8Peer::EnumHosts Method

Enumerates applications that host Microsoft® DirectPlay® games.

Syntax

HRESULT EnumHosts(PDPN_APPLICATION_DESC const pApplicationDesc, IDirectPlay8Address *const pdpaddrHost, IDirectPlay8Address *const pdpaddrDeviceInfo, PVOID const pvUserEnumData, const DWORD dwUserEnumDataSize, const DWORD dwEnumCount, const DWORD dwRetryInterval, const DWORD dwRetryInterval, const DWORD dwTimeOut, PVOID const pvUserContext, HANDLE *const pAsyncHandle, const DWORD dwFlags);

Parameters

pApplicationDesc

[in] Pointer to a <u>DPN_APPLICATION_DESC</u> structure that specifies which application hosts to enumerate. You must set the *pApplicationDesc*.dwSize member to the appropriate value. To reduce the number of responses, set

pApplicationDesc.guidApplication to the globally unique identifier (GUID) of the application to be found. If this member is not set, the search will include all applications.

pdpaddrHost

[in] Pointer to an <u>IDirectPlay8Address</u> object that specifies the address of the computer that is hosting the application. Some service providers allow this parameter to be NULL or be an address object containing only the service provider component. In this case, DirectPlay will get the information by using a

broadcast mechanism or from the *pdpaddrDeviceInfo* parameter. If you set the

DPNENUMHOSTS_OKTOQUERYFORADDRESSING flag in *dwFlags*, the user can be queried for any missing address information.

pdpaddrDeviceInfo

[in] Pointer to an **IDirectPlay8Address** object that specifies the service provider and local device settings to use when enumerating. The user can be queried for any missing address information if you set the

DPNENUMHOSTS_OKTOQUERYFORADDRESSING flag in the *dwFlags* parameter.

pvUserEnumData

[in] Pointer to a block of data that is sent in the enumeration request to the host. The size of the data is limited depending on the network type. Call <u>IDirectPlay8Peer::GetSPCaps</u> to obtain the exact value.

dwUserEnumDataSize

[in] Variable of type **DWORD** that specifies the size of the data pointed at in the *pvUserEnumData* parameter.

dwEnumCount

[in] Value specifying how many times that the enumeration data will be sent. Set this parameter to zero to use the default value. You can obtain the default value for *dwEnumCount* by calling

IDirectPlay8Peer::GetSPCaps. If *dwEnumCount* is set to INFINITE, the enumeration will continue until canceled.

dwRetryInterval

[in] Value specifying how many milliseconds between enumeration retries. Set this parameter to zero to use the default value. You can obtain the default value for

dwRetryInterval by calling **IDirectPlay8Peer::GetSPCaps**. *dwTimeOut*

[in] Variable of type **DWORD** that specifies the number of milliseconds that DirectPlay will wait for replies after the last enumeration is sent. Set this parameter to zero to use the default value. You can obtain the default value for *dwTimeOut* by calling **IDirectPlay8Peer::GetSPCaps**. If INFINITE is specified, the enumeration continues until it is canceled.

pvUserContext

[in] Context that is provided in the peer's message handler when it is called with responses to the enumeration. This can be

useful to differentiate replies from concurrent enumerations. *pAsyncHandle*

[out] A **DPNHANDLE**. When the method returns, *pAsyncHandle* will point to a handle that you can pass to

IDirectPlay8Peer::CancelAsyncOperation to cancel the operation. This parameter must be set to NULL if you set the DPNENUMHOSTS_SYNC flag in *dwFlags*.

dwFlags

[in] The following flags can be set.

DPNENUMHOSTS_SYNC

Causes the method to process synchronously.

DPNENUMHOSTS_OKTOQUERYFORADDRESSING Setting this flag will display a standard DirectPlay dialog box, which queries the user for more information if not enough information is passed in this method.

DPNENUMHOSTS_NOBROADCASTFALLBACK

If the service provider supports broadcasting, setting this flag will disable the broadcast capabilities. Check to see if broadcasting is supported by examining the <u>DPN_SP_CAPS</u> structure before setting this flag.

Return Value

Returns S_OK if this method is processed synchronously and is successful. By default, this method is run asynchronously and normally returns <u>DPNSUCCESS_PENDING</u>. It can also return one of the following error values.

| DPNERR_INVALIDDEVICEADDRESS | The address for the local computer or adapter is invalid. |
|-----------------------------|---|
| DPNERR_INVALIDFLAGS | The flags passed to this method are invalid. |
| DPNERR_INVALIDHOSTADDRESS | The specified remote address is invalid. |
| DPNERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
| DPNERR_ENUMQUERYTOOLARGE | The query data specified is too large. |
| DPNERR_USERCANCEL | The user canceled the operation. |
| DPNERR_ALREADYINITIALIZED | The object has already been initialized. |

Remarks

When an application is found that meets the enumeration criteria, the application's message handler is called with a <u>DPN_MSGID_ENUM_HOSTS_RESPONSE</u> system message. The message contains a **DPN_APPLICATION_DESC** structure describing the applications found and **IDirectPlay8Address** objects identifying the location of the hosts.

To ensure better Network Address Translation (NAT) and proxy support when using the Transmission Control Protocol/Internet Protocol (TCP/IP) service provider or to prevent redialing with the modem service provider, keep the enumeration active when calling the <u>IDirectPlay8Peer::Connect</u> method. To prevent the enumeration from completing, set the *dwEnumCount* parameter to INFINITE and do not use the IDirectPlay8Peer::CancelAsyncOperation to terminate the enumeration before the connect operation has completed. You should also pass the **pAddressSender** and **pAddressDevice** address objects in the DPNMSG ENUM HOSTS RESPONSE message without modification into the *pHostAddr* and *pDeviceInfo* parameters of the IDirectPlay8Peer::Connect method. To pass the address objects to IDirectPlay8Peer::Connect outside of the callback function, use <u>IDirectPlay8Address::Duplicate</u> or <u>IDirectPlay8Address::AddRef</u> to prevent the object from being destroyed and store the pointers using thread-safe code.

Any number of enumerations can be run concurrently. The *pvUserContext* value is provided in the message handler to help differentiate replies to different enumerations.

Because of the variation in the number of ways enumeration can happen, an application should not attempt to specify *dwEnumCount*, *dwRetryInterval*, or *dwTimeOut* unless the application has some specific media knowledge. The only exception is if you want to have the enumeration continue until explicitly cancelled, then set *dwEnumCount* to INFINITE.

The default enumeration count and timeout values will cause **IDirectPlay8Peer::EnumHosts** to complete within a reasonable amount of time. These values are set by the service provider, and can be obtained by calling **IDirectPlay8Peer::GetSPCaps**. Asynchronous enumerations can be stopped at any time by calling **IDirectPlay8Peer::CancelAsyncOperation** and either passing the handle returned in the *pAsyncHandle* parameter or setting the DPNCANCEL_ENUM flag in the *dwFlags* parameter. An enumeration can also be stopped by returning anything other than S_OK from the message handler when processing a DPN_MSGID_ENUM_HOSTS_RESPONSE message.

You might receive multiple

DPN_MSGID_ENUM_HOSTS_RESPONSE messages from the same host during one enumeration session. The **guidInstance** member of the associated DPN_APPLICATION_DESC structure can be used to correlate these duplicate responses.

If you set the DPNENUMHOSTS_OKTOQUERYFORADDRESSING flag in *dwFlags*, the service provider might attempt to display a dialog box to ask the user to complete the address information. You must have a visible window present when the service provider tries to display the dialog box or your application will lock.

Data Value Summary specifies the required addressing information for each service provider.

DPNERR_USERCANCEL will be returned if the enumeration is canceled by calling the IDirectPlay8Peer::CancelAsyncOperation method or if DPN_OK is not returned when processing a DPN_MSGID_ENUM_HOSTS_RESPONSE message.
IDirectPlay8Peer::EnumPlayersAndGroups Method

Retrieves a list of all the player and/or group identifiers for the session.

Syntax

```
HRESULT EnumPlayersAndGroups(
DPNID *const prgdpnid,
DWORD *const pcdpnid,
const DWORD dwFlags
```

);

Parameters

prgdpnid

[out] Pointer to an array that will be filled with the session's group and/or player identifiers.

pcdpnid

[in] Pointer to a variable of type **DWORD** that specifies the number of identifiers that can be contained in the buffer pointed to by *prgdpnid*. If the buffer is too small, this method returns DPNERR_BUFFERTOOSMALL and this parameter contains the number of entries that are required.

dwFlags

[in] Flag that describes enumeration behavior. You can set one or both of the following flags.

DPNENUM_PLAYERS

Return a list of player identifiers.

```
DPNENUM_GROUPS
```

Return a list of group identifiers.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_BUFFERTOOSMALL The supplied buffer is not large enough to contain the requested data.

Remarks

Because group and player information changes frequently, the required buffer size returned may change between subsequent calls. Check and reallocate the buffer until the method succeeds.

IDirectPlay8Peer::EnumServiceProviders Method

Enumerates all the registered service providers available to the application.

Syntax

```
HRESULT EnumServiceProviders(
const GUID *const pguidServiceProvider,
const GUID *const pguidApplication,
const DPN_SERVICE_PROVIDER_INFO *const pSPInfoBuffer,
DWORD *const pcbEnumData,
DWORD *const pcReturned,
const DWORD dwFlags
```

```
);
```

Parameters

pguidServiceProvider

[in] Pointer to a variable of type globally unique identifier (GUID) that specifies a service provider. This optional parameter forces the enumeration of subdevices for the specified service provider. You should normally set this value to NULL, to enumerate all available service providers. Otherwise, set pguidServiceProvider to one of the following predefined values. CLSID_DP8SP_TCPIP Internet Protocol (IP) service providers CLSID NETWORKSIMULATOR DP8SP TCPIP DP8Sim service providers CLSID_DP8SP_SERIAL Serial service providers CLSID_DP8SP_MODEM Modem service providers CLSID DP8SP IPX **IPX** service providers pguidApplication

[in] Pointer to a variable of type GUID that specifies an application. If a pointer is passed in this parameter, only service providers who can be connected by the application are enumerated. You can also pass NULL to enumerate all the registered service providers for the system.

pSPInfoBuffer

[out] Pointer to an array of <u>DPN_SERVICE_PROVIDER_INFO</u> structures that will be filled with service provider information.

pcbEnumData

[out] Pointer to **DWORD** that is filled with the size of the *pSPInfoBuffer* buffer if the buffer is too small.

pcReturned

[out] Pointer to a variable of type **DWORD** that specifies the number of **DPN_SERVICE_PROVIDER_INFO** structures returned in the *pcbEnumData* array.

dwFlags

[in] The following flag can be specified.

DPNENUMSERVICEPROVIDERS_ALL

Enumerates all the registered service providers for the system, including those that are not available to the application or do not have devices installed.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_BUFFERTOOSMALLThe supplied buffer is not large enough to contain the requested data.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.

Remarks

Call this method initially by specifying NULL in the *pguidServiceProvider* parameter to determine the base service providers available to the system. Specific devices for a service provider can then be obtained by passing a pointer to a service provider GUID in the *pguidServiceProvider*. This is useful, for

example, when using the Modem Connection for Microsoft® DirectPlay® service provider. You can choose different modems for dialing out and specific modems for hosting.

If the *pcbEnumData* buffer is not big enough to hold the requested service provider information, the method returns DPNERR_BUFFERTOOSMALL and the *pcbEnumData* parameter contains the required buffer size. Typically, the best strategy is to call the method once with a zero-length buffer to determine the required size. Then call it again with the appropriate-sized buffer.

Typically, this method will return only those service providers that can be used by the application. For example, if the Internetwork Packet Exchange (IPX) networking protocol is not installed, DirectPlay will not return the IPX service provider. To have DirectPlay return all service providers, even those that cannot be used by the application, set the DPNENUMSERVICEPROVIDERS_ALL flag in *dwFlags*.

IDirectPlay8Peer::GetApplicationDesc Method

Retrieves the full application description for the connected application.

Syntax

```
HRESULT GetApplicationDesc(
    DPN_APPLICATION_DESC *const pAppDescBuffer,
    DWORD *const pcbDataSize,
    const DWORD dwFlags
);
```

Parameters

pAppDescBuffer

[out] Pointer to a <u>DPN_APPLICATION_DESC</u> structure where the application description data is to be written. Set this parameter to NULL to request only the size of data. If *pAppDescBuffer* is not set to NULL, you must set the *pAppDescBuffer*.dwSize member to an appropriate value. The *pcbDataSize* parameter is set to the size required to hold the data.

pcbDataSize

[in, out] Pointer to a variable of type **DWORD** that is initialized to the size of the buffer before calling this method. After the method returns, this parameter is set to the size, in bytes, of the session data. If the buffer is too small, this method returns the DPNERR_BUFFERTOOSMALL error value, and this parameter is set to the buffer size required. If this parameter is NULL, the method returns DPNERR INVALIDPARAM.

```
dwFlags
```

[in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

| DPNERR_BUFFERTOOSMALL | The supplied buffer is not large enough to contain the requested data. |
|-----------------------|--|
| DPNERR_INVALIDFLAGS | The flags passed to this method are invalid. |
| DPNERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
| DPNERR_NOCONNECTION | No communication link was established. |

Remarks

Call this method initially by passing NULL in the *pAppDescBuffer* parameter to obtain the size of the required buffer. When you call the method a second time to fill the buffer, be sure to set the structures **dwSize** member to the appropriate value.

The returned DPN_APPLICATION_DESC structure will have the guidInstance, guidApplication, and pwszSessionName members set. It will not contain information about other clients that are connected to the session. That information, if available, can be obtained only from the server application. In particular, the dwCurrentPlayers member will always be set to 0.

To avoid accidentally overwriting the application description, applications should call **IDirectPlay8Peer::GetApplicationDesc** and fill in the **DPN_APPLICATION_DESC** structure before calling <u>IDirectPlay8Peer::SetApplicationDesc</u>.

IDirectPlay8Peer::GetCaps Method

Retrieves the <u>DPN_CAPS</u> or <u>DPN_CAPS_EX</u> structure for the current interface.

Syntax

```
HRESULT GetCaps(
    DPN_CAPS *const pdpnCaps,
    const DWORD dwFlags
);
```

Parameters

pdpnCaps

[out] Pointer to a **DPN_CAPS** or **DPN_CAPS_EX** structure to receive caps information. You must set the **dwSize** member of this structure to an appropriate value.

dwFlags

[in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDOBJECTThe Microsoft® DirectPlay® object pointer is invalid.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_INVALIDPOINTERPointer specified as a parameter is invalid.DPNERR_UNINITIALIZEDThe requested object has not been initialized.

Remarks

A successful call to <u>IDirectPlay8Peer::Initialize</u> must be made before this method can be called.

DirectPlay will determine whether **DPN_CAPS** or **DPN_CAPS_EX** is being used based on the size of the structure referenced by *pdpnCaps*.

IDirectPlay8Peer::GetConnectionInfo Method

Retrieves statistical information about the connection between the local application and the specified remote player.

Syntax

```
HRESULT GetConnectionInfo(
    const DPNID dpnidEndPoint,
    DPN_CONNECTION_INFO *const pdnConnectInfo,
    const DWORD dwFlags
);
```

Parameters

dpnidEndPoint

[in] The DPNID of the remote player whose connection information will be retrieved.

pdnConnectInfo

[out] Pointer to a <u>DPN_CONNECTION_INFO</u> structure to retrieve information about the specified connection. The **dwSize** member of this structure must be set to the size of a **DPN_CONNECTION_INFO** structure.

dwFlags

[in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

```
DPNERR_INVALIDOBJECTThe Microsoft® DirectPlay® object pointer is invalid.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_INVALIDPOINTERPointer specified as a parameter is invalid.DPNERR_UNINITIALIZEDThe requested object has not been initialized.DPNERR_INVALIDPLAYERThe player ID is not recognized as a valid player ID for this game session.
```

Remarks

This method can be called only after a successful <u>IDirectPlay8Peer::Host</u> or <u>IDirectPlay8Peer::Connect</u> call has completed.

IDirectPlay8Peer::GetGroupContext Method

Retrieves the group context value for the specified group.

Syntax

```
HRESULT GetGroupContext(
const DPNID dpnid,
PVOID *const ppvGroupContext,
const DWORD dwFlags
);
```

Parameters

dpnid [in] Variable of type **DPNID** that specifies the identifier of the group to retrieve context data for. *ppvGroupContext* [out] Pointer to the context value of the group. *dwFlags* [in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDGROUPThe group ID is not recognized as a valid group ID for this game session.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_NOTREADYThe object is not ready for use.

Remarks

Group context values are set by pointing the **pvGroupContext** member of the <u>DPN_MSGID_CREATE_GROUP</u> system message to the context value data.

This method returns <u>DPNERR_NOTREADY</u> when it is called before a **DPN_MSGID_CREATE_GROUP** message is received by Microsoft® DirectPlay® for the group specified in *dpnid*. Call **IDirectPlay8Peer::GetGroupContext** again, allowing task switching so that the thread carrying the message can return.

IDirectPlay8Peer::GetGroupInfo Method

Retrieves a block of data associated with a group, including the group name.

This method is typically called after a <u>DPN_MSGID_GROUP_INFO</u> system message is received indicating that the group data has been modified.

Syntax

```
HRESULT GetGroupInfo(
    const DPNID dpnid,
    DPN_GROUP_INFO *const pdpnGroupInfo,
    DWORD *const pdwSize,
    const DWORD dwFlags
);
```

Parameters

dpnid

[in] Variable of type **DPNID** that specifies the identifier of the group whose data block will be retrieved.

pdpnGroupInfo

[out] Pointer to a <u>DPN_GROUP_INFO</u> structure that describes the group data. If *pdwSize* is not set to NULL, you must set *pdpnGroupInfo*.dwSize to the size of a **DPN_GROUP_INFO** structure.

pdwSize

[out] Pointer to a variable of type **DWORD** that returns the size of the data in the *pdpnGroupInfo* parameter. If the buffer is too small, this method returns DPNERR_BUFFERTOOSMALL and this parameter contains the required size.

dwFlags

[in] Reserved. Set to 0.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_BUFFERTOOSMALLThe supplied buffer is not large enough to contain the requested data.DPNERR_INVALIDFLAGSThe flags passed to this method are invalid.DPNERR_INVALIDGROUPThe group ID is not recognized as a valid group ID for this game session.

Remarks

Microsoft® DirectPlay® returns the **DPN_GROUP_INFO** structure, and the pointers assigned to the structure's **pwszName** and **pvData** members in a contiguous buffer. If the two pointers were set, you must have allocated enough memory for the structure, plus the two pointers. The most robust way to use this method is to first call it with *pdwSize* set to NULL. When the method returns, *pdwSize* will point to the correct value. Use that value to allocate memory for your structure and call the method a second time to retrieve the information.

When the method returns, the **dwInfoFlags** member of the **DPN_GROUP_INFO** structure will always have the DPNINFO_DATA and DPNINFO_NAME flags set, even if the corresponding pointers are set to NULL. These flags are used when calling <u>IDirectPlay8Peer::SetGroupInfo</u>, to notify DirectPlay which values have changed.

Transmission of nonstatic information should be handled with the <u>IDirectPlay8Peer::SendTo</u> method because of the high cost of using the **IDirectPlay8Peer::SetGroupInfo** method.

 $\ensuremath{\textcircled{\sc 0}}$ 2003 Microsoft Corporation. All rights reserved.

IDirectPlay8Peer::GetLocalHostAddresses Method

Retrieves the local addresses being used to host the session.

Syntax

```
HRESULT GetLocalHostAddresses(
IDirectPlay8Address **const prgpAddress,
DWORD *const pcAddress,
const DWORD dwFlags
```

);

Parameters

prgpAddress

[out] Pointer to an array of <u>IDirectPlay8Address</u> objects that specify the local host addresses. You must release these objects when you no longer need them, or you will create memory leaks.

pcAddress

[in, out] Maximum number of address objects that can be returned in the array pointed to by *prgpAddress*. If the buffer is too small, this method returns DPNERR_BUFFERTOOSMALL and this parameter contains the required size.

dwFlags

[in] The following flag can be specified when using the Transmission Control Protocol/Internet Protocol (TCP/IP) service provider. This method will return

DPNERR_UNSUPPORTED if this flag is used with any other service provider.

DPNGETLOCALHOSTADDRESSES_COMBINED Return all listening addresses combined into one IDirectPlay8Address object.

Return Value

Returns S_OK if successful, or one of the following error values.

| DPNERR_BUFFERTOOSMALL | The supplied buffer is not large enough to contain the requested data. |
|-----------------------|---|
| DPNERR_INVALIDOBJECT | The Microsoft® DirectPlay® object pointer is invalid. |
| DPNERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
| DPNERR_INVALIDPOINTER | Pointer specified as a parameter is invalid. |
| DPNERR_UNINITIALIZED | The requested object has not been initialized. |
| DPNERR_UNSUPPORTED | The function or feature is not available in this implementation or on this service provider. |
| DPNERR_NOTHOST | The client attempted to connect to a nonhost computer. Additionally, this error value may be returned by a nonhost that tried to set the application description. |

Remarks

The most robust way to use this method is to call it first with *pcAddress* set to 0. When the method returns, *pcAddress* will point to the correct value, and you can use that value to call the method a second time to retrieve the information.

If the calling application is not the session host, the method returns DPNERR_NOTHOST. Use <u>IDirectPlay8Peer::GetPeerAddress</u> to retrieve the address of a remote player.

If DPNGETLOCALHOSTADDRESSES_COMBINED is specified, the address object returned will contain all listening host addresses. For example, the host might have multiple addresses if it is behind a Network Address Translation (NAT) device or if it has multiple network cards. In this case, players can connect to the host faster if they can try all of the addresses simultaneously. The application must provide its own mechanism for passing the combined address object to the connecting players. One way to do this is by using IDirectPlay8Address::GetURLA or IDirectPlay8Address::GetURLW and IDirectPlay8Address::BuildFromURLA or IDirectPlay8Address::BuildFromURLA or using a Web page or lobby mechanism.

IDirectPlay8Peer::GetPeerAddress Method

Retrieves the address for the specified remote player in the session.

Syntax

```
HRESULT GetPeerAddress(
const DPNID dpnid,
IDirectPlay8Address **const pAddress,
const DWORD dwFlags
);
```

Parameters

dpnid

[in] Variable of type **DPNID** specifying the identification of the player.

pAddress

[out] Address of a pointer to an <u>IDirectPlay8Address</u> object that specifies the address of the peer. You must release this object when you no longer need it.

dwFlags

[in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDOBJECTThe Microsoft® DirectPlay® object pointer is invalid.DPNERR_INVALIDPLAYERThe player ID is not recognized as a valid player ID for this game session.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_INVALIDPOINTERPointer specified as a parameter is invalid.DPNERR_UNINITIALIZEDThe requested object has not been initialized.

Remarks

This method cannot be used to retrieve the address of the local player. If the player calls this method on his or her own identifier (ID), it returns a DPNERR_INVALIDPLAYER error message.

Use <u>IDirectPlay8Peer::GetLocalHostAddresses</u> to retrieve addresses that can be used to connect to the session.

IDirectPlay8Peer::GetPeerInfo Method

Retrieves peer information set for the specified peer.

Syntax

```
HRESULT GetPeerInfo(
const DPNID dpnid,
DPN_PLAYER_INFO *const pdpnPlayerInfo,
DWORD *const pdwSize,
const DWORD dwFlags
```

);

Parameters

dpnid

[in] Variable of type **DPNID** that specifies the identifier of the peer whose information will be retrieved.

pdpnPlayerInfo

[out] Pointer to a <u>DPN_PLAYER_INFO</u> structure to fill with peer information. If *pdwSize* is not set to NULL, you must set *pdpnPlayerInfo*.dwSize to the size of a **DPN_PLAYER_INFO** structure.

pdwSize

[in, out] Pointer to a variable of type **DWORD** that contains the size of the peer data returned in the *pdpnPlayerInfo* parameter. If the buffer is too small this method returns

DPNERR_BUFFERTOOSMALL and this parameter contains the size of the required buffer.

dwFlags

[in] Reserved. Set to 0.

Return Value

Returns S_OK if successful, or one of the following error values.

| DPNERR_BUFFERTOOSMALL | The supplied buffer is not large enough to contain the requested data. |
|-----------------------|---|
| DPNERR_INVALIDFLAGS | The flags passed to this method are invalid. |
| DPNERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
| DPNERR_INVALIDPLAYER | The player ID is not recognized as a valid player ID for this game session. |
| DPNERR_CONNECTING | The method is in the process of connecting to the network. |

Remarks

Call this method after the peer receives a <u>DPN_MSGID_PEER_INFO</u> message from the application, which indicates a peer has updated their information.

Calling IDirectPlay8Peer::GetPeerInfo before the host has returned from IDirectPlay8Peer::Host will cause this method to fail with DPNERR_CONNECTING. If this happens, try calling IDirectPlay8Peer::GetPeerInfo again.

Microsoft® DirectPlay® returns the **DPN_PLAYER_INFO** structure, and the pointers assigned to the structure's **pwszName** and **pvData** members in a contiguous buffer. If the two pointers were set, you must have allocated enough memory for the structure, plus the two pointers. The most robust way to use this method is to first call it with *pdwSize* set to NULL. When the method returns, *pdwSize* will point to the correct value. Use that value to allocate memory for the structure and call the method a second time to retrieve the information.

When the method returns, the **dwinfoFlags** member of the **DPN_PLAYER_INFO** structure will always have the DPNINFO_DATA and DPNINFO_NAME flags set, even if the corresponding pointers are set to NULL. These flags are used when calling <u>IDirectPlay8Peer::SetPeerInfo</u>, to notify DirectPlay of which

values have changed.

Transmission of nonstatic information should be handled with the <u>IDirectPlay8Peer::SendTo</u> method because of the high cost of using the **IDirectPlay8Peer::SetPeerInfo** method.

IDirectPlay8Peer::GetPlayerContext Method

Retrieves the player context value for the specified peer.

Syntax

```
HRESULT GetPlayerContext(
const DPNID dpnid,
PVOID *const ppvPlayerContext,
const DWORD dwFlags
);
```

Parameters

dpnid [in] Variable of type **DPNID** that specifies the identifier of the player to get context data for. *ppvPlayerContext* [out] Pointer to the context data of the peer. *dwFlags* [in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

```
DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_INVALIDPLAYERThe player ID is not recognized as a valid player ID for this game session.DPNERR_NOTREADYThe object is not ready for use.
```

Remarks

Player context values are set by pointing the **pvPlayerContext** member of the <u>DPN_MSGID_CREATE_PLAYER</u> system message to the context value data.

This method returns <u>DPNERR_NOTREADY</u> when it is called before a **DPN_MSGID_CREATE_PLAYER** message is received by Microsoft® DirectPlay® for the player specified in *dpnid*. Call **IDirectPlay8Peer::GetPlayerContext** again allowing task switching so that the thread carrying the message can return.
IDirectPlay8Peer::GetSendQueueInfo Method

Used by the application to monitor the size of the send queue.

Syntax

```
HRESULT GetSendQueueInfo(
const DPNID dpnid,
DWORD *const pdwNumMsgs,
DWORD *const pdwNumBytes,
const DWORD dwFlags
);
```

Parameters

dpnid

[in] **DPNID** of the player to get send queue information for. *pdwNumMsgs*

[out] Pointer to a variable of type **DWORD** that contains the number of messages currently queued. This value is optional, and can be set to NULL.

pdwNumBytes

[out] Pointer to a variable of type **DWORD** that specifies the total number of bytes of data of the messages currently queued. This value is optional, and can be set to NULL.

dwFlags

[in] You can specify the DPNGETSENDQUEUEINFO_PRIORITY_NORMAL, DPNGETSENDQUEUEINFO_PRIORITY_HIGH, or DPNGETSENDQUEUEINFO_PRIORITY_LOW flag to inquire about specific messages of that priority.

Return Value

Returns S_OK if successful, or the following error value.

Remarks

Microsoft® DirectPlay® will not send messages faster than the receiving computer can process them. As a result, if the sending computer is sending faster than the receiver can receive, messages accumulate in the sender's queue. If the application registers that the send queue is growing too large, it should slow the rate that messages are sent.

You cannot set both *pdwNumMsgs* and *pdwNumBytes* to NULL. At least one of them must be set to a valid pointer.

IDirectPlay8Peer::GetSPCaps Method

Retrieves the <u>DPN_SP_CAPS</u> structure for the specified service provider.

Syntax

```
HRESULT GetSPCaps(
    const GUID *const pguidSP,
    DPN_SP_CAPS *const pdpnSPCaps,
    const DWORD dwFlags
);
```

Parameters

pguidSP

[in] Pointer to a globally unique identifier (GUID) specifying the service provider you want to get information about.

pdpnSPCaps

[out] Pointer to a **DPN_SP_CAPS** structure to receive the information about the specified service provider. You must set the *pdpnSPCaps*.dwSize member of the structure.

dwFlags

[in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDOBJECTThe Microsoft® DirectPlay® object pointer is invalid.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_INVALIDPOINTERPointer specified as a parameter is invalid.DPNERR_UNINITIALIZEDThe requested object has not been initialized.

Remarks

This method retrieves information about the specified service provider. A successful call to <u>IDirectPlay8Peer::Initialize</u> must be made before this method can be called.

IDirectPlay8Peer::Host Method

Creates a new peer-to-peer session, hosted by the local computer.

Syntax

```
HRESULT Host(
const DPN_APPLICATION_DESC *const pdnAppDesc,
IDirectPlay8Address **const prgpDeviceInfo,
const DWORD cDeviceInfo,
const DPN_SECURITY_DESC *const pdpSecurity,
const DPN_SECURITY_CREDENTIALS *const pdpCredentials,
VOID *const pvPlayerContext,
const DWORD dwFlags
```

);

Parameters

pdnAppDesc

[in] Pointer to a <u>DPN_APPLICATION_DESC</u> structure that describes the application.

prgpDeviceInfo

[in] Pointer to an array of <u>IDirectPlay8Address</u> objects containing the device addresses that should be used to host the application. You must release these objects when you no longer need them.

cDeviceInfo

[in] Variable of type **DWORD** that specifies the number of device address objects in the array pointed to by *prgpDeviceInfo*.

pdpSecurity

[in] Reserved. Must be NULL.

pdpCredentials

[in] Reserved. Must be NULL.

pvPlayerContext

[in] Pointer to the context value of the local player. This value is preset when the local computer handles the

DPN_MSGID_CREATE_PLAYER message. This parameter is

optional and may be set to NULL.

dwFlags

[in] The following flag can be specified.

DPNHOST_OKTOQUERYFORADDRESSING Setting this flag will display a standard Microsoft® DirectPlay® dialog box, which queries the user for more information if not enough information is passed in this

method.

Return Value

Returns S_OK if successful, or the following error value.

| DPNERR_DATATOOLARGE | The application data is too large for the service provider's Maximum Transmission Unit. |
|-----------------------------|---|
| DPNERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
| DPNERR_INVALIDDEVICEADDRESS | The address for the local computer or adapter is invalid. |
| DPNERR_DPNSVRNOTAVAILABLE | Port 6073 is already in use. |
| DPNERR_ALREADYINITIALIZED | The object has already been initialized. |

Remarks

Do not set the **guidInstance** member of the **DPN_APPLICATION_DESC** structure when calling **IDirectPlay8Peer::Host** because DirectPlay will ignore any value passed in and determine its own globally unique identifier (GUID). The only way to retrieve the **guidInstance** is by calling <u>IDirectPlay8Peer::GetApplicationDesc</u>.

If you set the DPNHOST_OKTOQUERYFORADDRESSING flag in *dwFlags*, the service provider might attempt to display a dialog box to ask the user to complete the address information. You must have a visible window present when the service provider tries to display the dialog box, or your application will lock.

Data Value Summary specifies the required addressing information for each service provider.

The maximum size of the application data that you assign to the **pvApplicationReservedData** member of the **DPN_APPLICATION_DESC** structure is limited by the service provider's Maximum Transmission Unit. If your application data is too large, the method will fail and return DPNERR_DATATOOLARGE.

IDirectPlay8Peer::Initialize Method

Registers an entry point in the peer's code that receives all the messages from the <u>IDirectPlay8Peer</u> interface and from remote peers. This method must be called before calling any other methods of this interface.

Syntax

```
HRESULT Initialize(
    PVOID const pvUserContext,
    const PFNDPNMESSAGEHANDLER pfn,
    const DWORD dwFlags
);
```

Parameters

pvUserContext

[in] Pointer to the user-provided context value in calls to the message handler. A user-provided context value can be used to differentiate messages coming from multiple interfaces to a common message handler.

pfn

[in] Pointer to a <u>PFNDPNMESSAGEHANDLER</u> callback function that is used to receive all messages from remote peers and indications of session changes from the **IDirectPlay8Peer** interface.

dwFlags

[in] You can specify the following flags.

DPNINITIALIZE_DISABLEPARAMVAL

Passing this flag will disable parameter validation for the current object.

DPNINITIALIZE_HINT_LANSESSION

Opens a larger send window for games running on a local area network (LAN).

DPNINITIALIZE_DISABLELINKTUNING

Disable any attempts by Microsoft® DirectPlay® to tune the

rate it sends at to the observed network conditions. Messages will be pushed out onto the network at the first available opportunity.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDFLAGS The flags passed to this method are invalid. DPNERR INVALIDPARAM One or more of the parameters passed to the method are invalid.

Remarks

Call this method first after using **CoCreateInstance** to obtain the **IDirectPlay8Peer** interface.

Specify the DPNINITIALIZE_HINT_LANSESSION flag for sessions where all players will be on the same LAN.

Applications might want to specify the

DPNINITIALIZE_DISABLELINKTUNING flag when they send at a fixed rate and do not alter the rate based on the network conditions. With this flag specified, DirectPlay will always assume the network has the capacity to carry all the application data and will therefore not attempt to tune its send rate to the network bandwidth. Specifying this flag and then sending at a rate that exceeds the capacity of the network will lead to unpredictable network behavior such as higher latency and increased packet drop rates. Applications that monitor the send queues and dynamically adjust their send rate to make best use of the available bandwidth should not specify this flag.

If the DPNINITIALIZE_DISABLELINKTUNING flag is specified,

DirectPlay features such as message prioritization, coalescence, and timeout are not useful because messages always go directly to the network and are not queued.

IDirectPlay8Peer::RegisterLobby Method

Allows launched applications to automatically propagate game status to the lobby.

Syntax

```
HRESULT RegisterLobby(
    const DPNHANDLE dpnHandle,
    IDirectPlay8LobbiedApplication *const pIDP8LobbiedApplication,
    const DWORD dwFlags
);
```

Parameters

dpnHandle

[in] The connection handle used when making the calls to IDirectPlay8LobbiedApplication::UpdateStatus.

pIDP8LobbiedApplication

[in] Pointer to the <u>IDirectPlay8LobbiedApplication</u> object that specifies the application.

dwFlags

[in] One of the following flags:

DPNLOBBY_REGISTER

Registers the lobby with the application.

DPNLOBBY_UNREGISTER

Unregisters the lobby with the application.

Return Value

Returns S_OK if successful, or the following error value.

DPNERR_INVALIDPARAM One or more of the parameters passed to the method are invalid.

 $\ensuremath{\textcircled{\sc 0}}$ 2003 Microsoft Corporation. All rights reserved.

IDirectPlay8Peer::RemovePlayerFromGroup Method

Removes a peer from a group.

When this method is called all peers connected to the application receive a <u>DPN_MSGID_REMOVE_PLAYER_FROM_GROUP</u> message.

Syntax

HRESULT RemovePlayerFromGroup(const DPNID *idGroup*, const DPNID *idClient*, PVOID const *pvAsyncContext*, DPNHANDLE *const *phAsyncHandle*, const DWORD *dwFlags*

);

Parameters

idGroup

[in] Variable of type **DPNID** that specifies the identifier of the group that the peer will be removed from.

idClient

[in] Variable of type **DPNID** that specifies the identifier of the peer that will be removed from the group.

pvAsyncContext

[in] Pointer to the user-supplied context, which is returned in the **pvUserContext** member of the

<u>DPN_MSGID_ASYNC_OP_COMPLETE</u> system message. phAsyncHandle

[out] A **DPNHANDLE**. A value will be returned. However, Microsoft® DirectPlay® does not permit cancellation of this operation, so the value cannot be used.

dwFlags

[in] Flag that controls how this method is processed. The following flag can be set for this method.

DPNREMOVEPLAYERFROMGROUP_SYNC Causes the method to process synchronously.

Return Value

Returns S_OK if this method is processed synchronously and is successful. By default, this method is run asynchronously and normally returns <u>DPNSUCCESS_PENDING</u>. It can also return one of the following error values.

| DPNERR_INVALIDFLAGS | The flags passed to this method are invalid. |
|-------------------------|---|
| DPNERR_INVALIDGROUP | The group ID is not recognized as a valid group ID for this game session. |
| DPNERR_INVALIDPLAYER | The player ID is not recognized as a valid player ID for this game session. |
| DPNERR_PLAYERNOTINGROUP | The player ID is not included in the group. |

IDirectPlay8Peer::ReturnBuffer Method

Retrieves message buffers provided to the application through the **pReceiveData** member of the <u>DPN_MSGID_RECEIVE</u> system message. If the user's message handler returns DPNSUCCESS_PENDING to the RECEIVE callback, Microsoft® DirectPlay® assumes that ownership of the buffer is transferred to the application, and neither frees nor modifies it until ownership is returned to DirectPlay through this call.

Syntax

```
HRESULT ReturnBuffer(
const DPNHANDLE hBufferHandle,
const DWORD dwFlags
);
```

Parameters

hBufferHandle

[in] Variable of type **DPNHANDLE** that specifies the buffer handle to the message. This is obtained in the *hBufferHandle* member of the **DPN_MSGID_RECEIVE** system message. *dwFlags*

[in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDHANDLEThe handle specified is invalid.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.

IDirectPlay8Peer::SendTo Method

Transmits data to another peer or group within the session by sending a message to the appropriate message handlers. The message can be sent synchronously or asynchronously.

Syntax

```
HRESULT SendTo(
    const DPNID dpnid,
    const DPN_BUFFER_DESC *const pBufferDesc,
    const DWORD cBufferDesc,
    const DWORD dwTimeOut,
    void *const pvAsyncContext,
    DPNHANDLE *const phAsyncHandle,
    const DWORD dwFlags
);
```

Parameters

dpnid

[in] Identifier of the peer or group that receives data. Set this parameter to DPNID_ALL_PLAYERS_GROUP to send a message to all players in the session.

pBufferDesc

[in] Pointer to a <u>DPN_BUFFER_DESC</u> structure that contains the data to be sent.

cBufferDesc

[in] Number of **DPN_BUFFER_DESC** structures pointed to by *pBufferDesc*. There can be up to eight buffers in this version of Microsoft® DirectPlay®.

dwTimeOut

[in] Number of milliseconds to wait for the message to send. If the message has not been sent by the *dwTimeOut* value, it is deleted from the send queue. If you set this parameter to 0, the message remains in the send queue until it is sent or until the

link is dropped.

pvAsyncContext

[in] Pointer to the user-supplied context, which is returned in the **pvUserContext** member of the

DPN_MSGID_SEND_COMPLETE system message. This parameter is optional and can be set to NULL.

phAsyncHandle

[out] A **DPNHANDLE**. When the method returns, *phAsyncHandle* will point to a handle that you can pass to

IDirectPlay8Peer::CancelAsyncOperation to cancel the

operation. This parameter must be set to NULL if you set the DPNSEND_SYNC flag in *dwFlags*.

dwFlags

[in] Flags that describe send behavior. You can set one or more of the following flags.

DPNSEND_SYNC

Process the **IDirectPlay8Peer::SendTo** request synchronously.

DPNSEND_NOCOPY

Use the data in the **DPN_BUFFER_DESC** structure and do not make an internal copy. This can be a more efficient method of sending data. However, it is less robust because modifying or deleting the data before receiving the **DPN_MSGID_SEND_COMPLETE** message can cause erroneous data to be sent. This flag cannot be combined with DPNSEND_NOCOMPLETE.

DPNSEND_NOCOMPLETE

Does not send the **DPN_MSGID_SEND_COMPLETE** to the message handler. This flag cannot be used with DPNSEND_NOCOPY or DPNSEND_GUARANTEED. Additionally, when using this flag the *pvAsyncContext* must be NULL.

DPNSEND_COMPLETEONPROCESS

Sends the **DPN_MSGID_SEND_COMPLETE** to the message handler when this message has been delivered to the target and the target's message handler returns from indicating its reception. There is additional internal message overhead when this flag is set, and the message transmission process might become significantly slower. If

you set this flag, DPNSEND_GUARANTEED must also be set.

DPNSEND_GUARANTEED

Sends the message by a guaranteed method of delivery. DPNSEND_PRIORITY_HIGH

Sets the priority of the message to high. This flag cannot be used with DPNSEND_PRIORITY_LOW.

DPNSEND_PRIORITY_LOW

Sets the priority of the message to low. This flag cannot be used with DPNSEND_PRIORITY_HIGH.

DPNSEND_NONSEQUENTIAL

If this flag is set, the target application will receive the messages in the order that they arrive at the user's computer. If this flag is not set, messages are delivered sequentially, and will be received by the target application in the order that they were sent. Doing so might require buffering incoming messages until missing messages arrive.

DPNSEND_NOLOOPBACK

Suppresses the <u>DPN_MSGID_RECEIVE</u> system message to your message handler when you are sending to a group that includes the local player. For example, this flag is useful if you are broadcasting to the entire session.

DPNSEND_COALESCE

Allows DirectPlay to combine packets when sending.

Return Value

Returns S_OK if this method is processed synchronously and is successful. By default, this method is run asynchronously and normally returns <u>DPNSUCCESS_PENDING</u>. It can also return one of the following error values.

| The service provider connection was reset while data was being sent. |
|---|
| An undefined error condition occurred. |
| The flags passed to this method are invalid. |
| One or more of the parameters passed to the method are invalid. |
| The player ID is not recognized as a valid player ID for this game session. |
| |

Remarks

This method generates a **DPN_MSGID_RECEIVE** system message in the receiver's message handler. The data buffer is contained in the **pReceiveData** member of the associated structure.

Messages can have one of three priorities: low, normal, and high. To specify a low or high priority for the message, set the appropriate flag in *dwFlags*. If neither of the priority flags is set, the message will have normal priority. See <u>Basic Networking</u> for a discussion of send priorities.

When the **IDirectPlay8Peer::SendTo** request is completed, a **DPN_MSGID_SEND_COMPLETE** system message is normally posted to the sender's message handler. The success or failure of the request is contained in the **hResultCode** member of the associated structure. You can suppress send completions by setting the DPNSEND_NOCOMPLETE flag in *dwFlags*.

Send completions are typically posted on the source computer as soon as the message is sent. In other words, a send completion does not necessarily mean that the message has been processed on the target. It might still be in a queue. If you want to be certain that the message has been processed by the target, set the DPNSEND_COMPLETEONPROCESS flag in *dwFlags*. This flag ensures that the send completion will not be sent until the target's message handler has processed the message and returned.

If the DPNSEND_COALESCE flag is set in *dwFlags*, DirectPlay will

try to coalesce up to 32 packets waiting in the queue into the outgoing frame. DirectPlay does not guarantee coalescence, even if the DPNSEND_COALESCE flag is set. Packets will only be coalesced if there is more than one message in the queue and the player receiving is running Microsoft DirectX® 9.0 or later. All voice packets can be coalesced. Both guaranteed and non-guaranteed packets will be coalesced into the same frame. If the frame is dropped before it reaches its destination, only the guaranteed parts of the frame will be resent and no other data will be coalesced into the frame.

Note Do not assume that resources such as the data buffer will remain valid until the method has returned. If you call this method asynchronously, the **DPN_MSGID_SEND_COMPLETE** message can be received and processed by your message handler before the call has returned. If your message handler deallocates or otherwise invalidates a resource such as the data buffer, that resource can become invalid at any time after the method has been called.

This method returns a DPNERR_GENERIC error value if the send computer is host with no other player connected and DPNSEND SYNC and DPNSEND NOLOOPBACK flags are set.

IDirectPlay8Peer::SetApplicationDesc Method

Changes the settings for the application that is being hosted. Only some settings can be changed.

Syntax

```
HRESULT SetApplicationDesc(
    const DPN_APPLICATION_DESC *const pad,
    const DWORD dwFlags
);
```

Parameters

pad

[in] Pointer to a <u>DPN_APPLICATION_DESC</u> structure that describes the application settings to modify.

dwFlags

[in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

| DPNERR_DATATOOLARGE | The application data is too large for the service provider's Maximum Transmission Unit. |
|---------------------|---|
| DPNERR_INVALIDFLAGS | The flags passed to this method are invalid. |
| DPNERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
| DPNERR_NOTHOST | The client attempted to connect to a nonhost computer. Additionally, this error value may be returned by a nonhost that tried to set the application description. |

Remarks

You can use this method to modify only the following members of the **DPN_APPLICATION_DESC** structure.

- dwMaxPlayers
- pwszSessionName
- pwszPassword
- pvApplicationReservedData
- dwApplicationReservedDataSize

When IDirectPlay8Peer::SetApplicationDesc is called, Microsoft® DirectPlay® makes a copy of the data *pad* points to. You do not need to save the DPN_APPLICATION_DESC structure once IDirectPlay8Peer::SetApplicationDesc returns.

You cannot set the **dwMaxPlayers** member to a smaller value than the current number of players in the session.

The maximum size of the application data that you assign to the **pvApplicationReservedData** member of the **DPN_APPLICATION_DESC** structure is limited by the service provider's Maximum Transmission Unit. If your application data is too large, the method will fail and return DPNERR_DATATOOLARGE.

To avoid accidentally overwriting the application description, applications should call <u>IDirectPlay8Peer::GetApplicationDesc</u> and fill in the **DPN_APPLICATION_DESC** structure before calling **IDirectPlay8Peer::SetApplicationDesc**.

 $\ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.

IDirectPlay8Peer::SetCaps Method

Sets the <u>DPN_CAPS</u> or <u>DPN_CAPS_EX</u> structure for the current interface.

Syntax

```
HRESULT SetCaps(
    const DPN_CAPS *const pdpCaps,
    const DWORD dwFlags
);
```

Parameters

pdpCaps

[in] Pointer to a **DPN_CAPS** or **DPN_CAPS_EX** structure used to set the information about the current interface.

dwFlags

[in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDOBJECTThe Microsoft® DirectPlay® object pointer is invalid.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_INVALIDPOINTERPointer specified as a parameter is invalid.DPNERR_UNINITIALIZEDThe requested object has not been initialized.

Remarks

A successful call to <u>IDirectPlay8Peer::Initialize</u> must be made before this method can be called.

DirectPlay will determine whether **DPN_CAPS** or **DPN_CAPS_EX** is being used based on the size of the structure referenced by *pdpCaps*.
IDirectPlay8Peer::SetGroupInfo Method

Sets a block of data associated with a group, including the name of the group.

Calling this method generates a <u>DPN_MSGID_GROUP_INFO</u> message, which is sent to all the peers connected to the application.

Syntax

```
HRESULT SetGroupInfo(
const DPNID dpnid,
DPN_GROUP_INFO *const pdpnGroupInfo,
PVOID const pvAsyncContext,
DPNHANDLE *const phAsyncHandle,
const DWORD dwFlags
);
```

Parameters

dpnid

[in] Variable of type **DPNID** that specifies the identifier of the group whose data block will be modified.

pdpnGroupInfo

[in] Pointer to a <u>DPN_GROUP_INFO</u> structure that describes the group data to set. To change the values of the **pwszName** or **pvData** members, you must set the corresponding DPNINFO_NAME or DPNINFO_DATA flags in the **dwinfoFlags** member.

pvAsyncContext

[in] Pointer to the user-supplied context, which is returned in the **pvUserContext** member of the

DPN_MSGID_ASYNC_OP_COMPLETE system message.

phAsyncHandle

[out] A **DPNHANDLE**. A value will be returned. However, Microsoft® DirectPlay® does not permit cancellation of this operation, so the value cannot be used. *dwFlags* [in] Flag that controls how this method is processed. The following flag can be set for this method: DPNSETGROUPINFO_SYNC Causes the method to process synchronously.

Return Value

Returns S_OK if this method is processed synchronously and is successful. By default, this method is run asynchronously and normally returns <u>DPNSUCCESS_PENDING</u>. It can also return one of the following error values.

DPNERR_INVALIDFLAGSThe flags passed to this method are invalid.DPNERR_INVALIDGROUPThe group ID is not recognized as a valid group ID for this game session.

Remarks

Transmission of nonstatic information should be handled with the <u>IDirectPlay8Peer::SendTo</u> method because of the high cost of using the **IDirectPlay8Peer::SetGroupInfo** method.

IDirectPlay8Peer::SetPeerInfo Method

Sets the static settings of the local peer. Call this method before connecting to relay basic player information with the application. When the peer successfully connects with the application, information set through this method can be retrieved by other players by calling the <u>IDirectPlay8Peer::GetPeerInfo</u> method.

Syntax

HRESULT SetPeerInfo(const DPN_PLAYER_INFO *const pdpnPlayerInfo, PVOID const pvAsyncContext, DPNHANDLE *const phAsyncHandle, const DWORD dwFlags);

Parameters

pdpnPlayerInfo

[in] Pointer to a <u>DPN_PLAYER_INFO</u> structure that contains the peer information to set.

pvAsyncContext

[in] Pointer to the user-supplied context, which is returned in the **pvUserContext** member of the

DPN_MSGID_ASYNC_OP_COMPLETE system message.

phAsyncHandle

[out] A **DPNHANDLE**. A value will be returned. However, Microsoft® DirectPlay® does not permit cancellation of this operation, so the value cannot be used.

dwFlags

[in] Flag that controls how this method is processed. The following flag can be set for this method.

DPNSETPEERINFO_SYNC

Causes the method to process synchronously.

Return Value

Returns S_OK if this method is processed synchronously and is successful. By default, this method is run asynchronously and normally returns <u>DPNSUCCESS_PENDING</u>. It can also return one of the following error values.

DPNERR_INVALIDFLAGSThe flags passed to this method are invalid.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_NOCONNECTIONNo communication link was established.

Remarks

This method can be called at any time during the session.

Transmission of nonstatic information should be handled with the <u>IDirectPlay8Peer::SendTo</u> method because of the high cost of using the **IDirectPlay8Peer::SetPeerInfo** method.

The **DPN_PLAYER_INFO** structure's **dwPlayerFlags** member must be set to zero.

You can modify the peer information with this method after connecting to the application. Calling this method after connection generates a <u>DPN_MSGID_PEER_INFO</u> system message to all players, informing them that data has been updated. The dwPlayerFlags method in the DPN_PLAYER_INFO structure must be set to 0 when making this call.

When calling this method asynchronously, the contents of the *pdpnPlayerInfo* and *pvAsyncContext* buffers will be copied by DirectPlay so that the calling application can clean up the buffers

before the method returns.

This method is guaranteed as long as the player is connected to the session. Even if this method is called during a host migration, DirectPlay will ensure that this method completes and that the information is propagated to all players.

IDirectPlay8Peer::SetSPCaps Method

Sets the <u>DPN_SP_CAPS</u> structure for the specified service provider.

Syntax

```
HRESULT SetSPCaps(
    const GUID *const pguidSP,
    const DPN_SP_CAPS *const pdpSPCaps,
    const DWORD dwFlags
);
```

Parameters

pguidSP

[in] Pointer to a globally unique identifier (GUID) that specifies the service provider to set information about.

pdpSPCaps

[in] Pointer to a **DPN_SP_CAPS** structure to set the information about the specified service provider.

dwFlags

[in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDOBJECTThe Microsoft® DirectPlay® object pointer is invalid.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_INVALIDPOINTERPointer specified as a parameter is invalid.DPNERR_UNINITIALIZEDThe requested object has not been initialized.

Remarks

A successful call to IDirectPlay8Peer::Initialize must be made before

this method can be called. Currently, only the **dwSystemBufferSize** member can be set by this call. The **dwNumThreads** member is for legacy support. Microsoft DirectX® 9.0 applications should use the <u>IDirectPlay8ThreadPool::SetThreadCount</u> method to set the number of threads. The other members of the **DPN_SP_CAPS** structure are get-only or ignored.

IDirectPlay8Peer::TerminateSession Method

Terminates the current Microsoft® DirectPlay® session.

Syntax

```
HRESULT TerminateSession(
   void *const pvTerminateData,
   const DWORD dwTerminateDataSize,
   const DWORD dwFlags
);
```

Parameters

pvTerminateData [in] Pointer to termination data. This data is also sent in the pvTerminateData member of the <u>DPN_MSGID_TERMINATE_SESSION</u> system message. dwTerminateDataSize [in] Size of data contained in the pvTerminateData parameter. dwFlags [in] Posonvod_Must bo 0

[in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or an error value otherwise.

Remarks

This method may be called only by the host player.

This method terminates the session even if host migration is enabled. To close the host player's connection to the session and allow host migration to take place, call <u>IDirectPlay8Peer::Close</u> instead.

When this method is called, the

DPN_MSGID_TERMINATE_SESSION will be sent to the message handler of each player in the session.

Players should call **IDirectPlay8Peer::Close** after receiving the **DPN_MSGID_TERMINATE_SESSION** message to uninitialize the **IDirectPlay8Peer** object.

See Also

Host Migration, Leaving a Peer-to-Peer Session, Terminating a Peerto-Peer Session

IDirectPlay8Server Interface

Applications use the methods of the **IDirectPlay8Server** interface to create and manage the server for a Microsoft® DirectPlay® client/server transport session.

IDirectPlay8Server Members

| <u>AddPlayerToGroup</u> | Adds a client to a group. After the client is successfully added to the group, all messages sent to the group are sent to the client. |
|-----------------------------|--|
| <u>CancelAsyncOperation</u> | Cancels asynchronous requests. Many methods of the IDirectPlay8Server interface run asynchronously by default. Depending on the situation, you might want to cancel requests before they are processed. All the methods of this interface that can be run asynchronously return an <i>hAsyncHandle</i> parameter. Specific requests are canceled by passing the <i>hAsyncHandle</i> of the request in this method's <i>hAsyncHandle</i> parameter. You can cancel all pending asynchronous operations by calling this method, specifying NULL in the <i>hAsyncHandle</i> parameter, and specifying DPNCANCEL_ALL_OPERATIONS in the <i>dwFlags</i> parameter. If a specific handle is provided to this method, no flags should be set. |
| <u>Close</u> | Closes the open connection to a session and uninitializes the IDirectPlay8Server object. |
| <u>CreateGroup</u> | Creates a group in the current session. When this method is called, the server's message handler receives a DPN_MSGID_CREATE_GROUP message. |
| <u>DestroyClient</u> | Deletes a client from the session. |
| <u>DestroyGroup</u> | Deletes a group created by the <u>IDirectPlay8Server::CreateGroup</u> method. |
| EnumGroupMembers | Retrieves a list of all players in a group. |
| EnumPlayersAndGroups | Retrieves a list of all the player and/or group identifiers for the application. |
| EnumServiceProviders | Enumerates the registered service providers available to the application. |
| <u>GetApplicationDesc</u> | Retrieves the full application description for the connected application. |
| <u>GetCaps</u> | Retrieves the <u>DPN_CAPS</u> or <u>DPN_CAPS_EX</u> structure for the current interface. |
| <u>GetClientAddress</u> | Retrieves the address for the specified player in the session. |
| <u>GetClientInfo</u> | |

| | Retrieves the client information set for the specified client. |
|------------------------------|---|
| GetConnectionInfo | Retrieves statistical information about the connection between the local server and the specified remote client. |
| <u>GetGroupContext</u> | Retrieves the group context value for a group. |
| <u>GetGroupInfo</u> | Retrieves a block of data associated with a group, including the group name. This method is typically called after a <u>DPN_MSGID_GROUP_INFO</u> system message is received, indicating that the group data has been modified. |
| <u>GetLocalHostAddresses</u> | Retrieves the local addresses being used to host the session. |
| <u>GetPlayerContext</u> | Retrieves the player context value for a client. |
| <u>GetSendQueueInfo</u> | Used by the application to monitor the size of the send queue. DirectPlay does not send messages faster than the receiving computer can process them. As a result, if the sending computer is sending faster than the receiver can receive, messages accumulate in the sender's queue. If the application registers that the send queue is growing too large, it should slow the rate that messages are sent. |
| <u>GetSPCaps</u> | Retrieves the <u>DPN_SP_CAPS</u> structure for the specified service provider. |
| <u>Host</u> | Creates a new client/server session, hosted by the local computer. |
| Initialize | Registers an entry point in the server's code that receives the messages from the IDirectPlay8Server interface and from remote clients. This method must be called before calling any other methods of this interface. |
| RegisterLobby | Allows launched applications to automatically propagate game status to the lobby. |
| RemovePlayerFromGroup | Removes a client from a group. |
| <u>ReturnBuffer</u> | Retrieves message buffers provided to the application through the pReceiveData member of the <u>DPN_MSGID_RECEIVE</u> system message. If the user's message handler returns <u>DPNSUCCESS_PENDING</u> to the RECEIVE callback, DirectPlay assumes ownership of the buffer has been transferred to the application, and neither frees nor modifies it until ownership is returned to DirectPlay through this call. |
| <u>SendTo</u> | Transmits data to a client or group within the session. The message can be sent synchronously or asynchronously. |
| SetApplicationDesc | Changes the settings for the application that is being hosted. Only some settings can be changed. |
| <u>SetCaps</u> | Sets the DPN_CAPS or DPN_CAPS_EX structure for the current interface. |

| <u>SetGroupInfo</u> | Sets a block of data associated with a group, including the name of the group. |
|----------------------|--|
| <u>SetServerInfo</u> | Sets the static settings of a server with an application. After clients successfully connect to the server, they can retrieve the information set by this method by calling the <u>IDirectPlay8Client::GetServerInfo</u> method. |
| <u>SetSPCaps</u> | Sets the DPN_SP_CAPS structure for the specified service provider. |

Interface Information

| Inherits from | <u>IUnknown</u> |
|---------------------------|----------------------------|
| Header | dplay8.h |
| Minimum operating systems | Windows 98, Pocket PC 2002 |

 $\ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.

IDirectPlay8Server::AddPlayerToGroup Method

Adds a client to a group. After the client is successfully added to the group, all messages sent to the group are sent to the client.

Syntax

```
HRESULT AddPlayerToGroup(
const DPNID idGroup,
const DPNID idClient,
PVOID const pvAsyncContext,
DPNHANDLE *const phAsyncHandle,
const DWORD dwFlags
```

);

Parameters

idGroup

[in] Variable of type **DPNID** that specifies the identifier of the group to add the client to.

idClient

[in] Variable of type **DPNID** that specifies the identifier of the client to add to the group.

pvAsyncContext

[in] Pointer to the user-supplied context, which is returned in the **pvUserContext** member of the

DPN_MSGID_ASYNC_OP_COMPLETE system message. This parameter is optional and can be set to NULL.

phAsyncHandle

[out] A **DPNHANDLE**. A value will be returned. However, Microsoft® DirectPlay® does not permit cancellation of this operation, so the value cannot be used.

dwFlags

[in] Flag that controls how this method is processed. The following flag can be set for this method. DPNADDPLAYERTOGROUP_SYNC

Causes this method to process synchronously.

Return Value

Returns S_OK if this method is processed synchronously and is successful. By default, this method is run asynchronously and generally returns <u>DPNSUCCESS_PENDING</u> or one of the following error values.

| DPNERR_INVALIDFLAGS | The flags passed to this method are invalid. |
|-----------------------------|---|
| DPNERR_INVALIDGROUP | The group ID is not recognized as a valid group ID for this game session. |
| DPNERR_INVALIDPLAYER | The player ID is not recognized as a valid player ID for this game session. |
| DPNERR PLAYERALREADYINGROUP | The player ID is already included in the group. |

Remarks

The server can add itself or a client to an existing group. After a player is successfully added to a group, all messages sent to the group will be received by the player.

IDirectPlay8Server::CancelAsyncOperation Method

Cancels asynchronous requests. Many methods of the <u>IDirectPlay8Server</u> interface run asynchronously by default. Depending on the situation, you might want to cancel requests before they are processed. All the methods of this interface that can be run asynchronously return an *hAsyncHandle* parameter.

Specific requests are canceled by passing the *hAsyncHandle* of the request in this method's *hAsyncHandle* parameter. You can cancel all pending asynchronous operations by calling this method, specifying NULL in the *hAsyncHandle* parameter, and specifying DPNCANCEL_ALL_OPERATIONS in the *dwFlags* parameter. If a specific handle is provided to this method, no flags should be set.

Syntax

```
HRESULT CancelAsyncOperation(
    const DPNHANDLE hAsyncHandle,
    const DWORD dwFlags
);
```

Parameters

hAsyncHandle

[in] Handle of the asynchronous operation to stop. This value can be NULL to stop all requests or a particular type of asynchronous request. If a specific handle is specified, the *dwFlags* parameter must be 0. You will receive this handle when you call one of several methods that support asynchronous operations. If one of the DPNCANCEL_PLAYER_SENDS flags is specified in the *dwFlags* parameter, *hAsyncHandle* must be set to a player's DPNID. dwFlags

[in] Flag that specifies which asynchronous request to cancel. You can set one of the following flags.

DPNCANCEL_SEND

Cancel an asynchronous <u>IDirectPlay8Server::SendTo</u> request.

DPNCANCEL_PLAYER_SENDS

Cancel all asynchronous **IDirectPlay8Server::SendTo** requests for the player specified in the *hAsyncHandle* parameter.

DPNCANCEL_PLAYER_SENDS_PRIORITY_LOW

Cancel low-priority asynchronous IDirectPlay8Server::SendTo requests for the player specified in the *hAsyncHandle* parameter.

DPNCANCEL_PLAYER_SENDS_PRIORITY_NORMAL Cancel normal-priority asynchronous IDirectPlay8Server::SendTo requests for the player specified in the hAsyncHandle parameter.

DPNCANCEL PLAYER SENDS PRIORITY HIGH

Cancel high-priority asynchronous

IDirectPlay8Server::SendTo requests for the player specified in the *hAsyncHandle* parameter.

DPNCANCEL_ALL_OPERATIONS

Cancel all asynchronous requests.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_CANNOTCANCELThe operation could not be canceled.DPNERR_INVALIDFLAGSThe flags passed to this method are invalid.DPNERR_INVALIDHANDLEThe handle specified is invalid.DPNSUCCESS_PENDINGAn asynchronous operation has reached the point where it is successfully queued.

Remarks

You can use this method to cancel an asynchronous operation for

the **IDirectPlay8Server::SendTo** method. Microsoft® DirectPlay® does not support cancellation of other asynchronous operations.

You can cancel a **IDirectPlay8Server::SendTo** request by providing the handle returned from **IDirectPlay8Server::SendTo** method. A <u>DPN_MSGID_SEND_COMPLETE</u> system message is still posted to the applications message handler for each asynchronous **IDirectPlay8Server::SendTo** request that is sent without the DPNSEND_NOCOMPLETE flag set. Send requests that are canceled by this method return DPNERR_USERCANCEL in their **hResultCode** member of the **DPN_MSGID_SEND_COMPLETE** message.

If you set the DPNCANCEL_ALL_OPERATIONS or DPNCANCEL_SEND flags in *dwFlags*, DirectPlay will attempt to cancel all matching operations. This method will return an error if any attempted cancellation fails, even though some cancellations may have been successful.

If you set one of the DPNCANCEL_PLAYER_SENDS flags in *dwFlags*, you must specify a player's DPNID in *hAsyncHandle*. This will cancel all pending **IDirectPlay8Server::SendTo** requests where the DPNID specified in the *dpnid* parameter matches the value set in the *hAsyncHandle* parameter.

Note The completion message might not arrive until after this method returns. Do not assume that the operation has been terminated until you have received a DPN_MSGID_SEND_COMPLETE, DPN_MSGID_CONNECT_COMPLETE, or DPN_MSGID_ASYNC_OP_COMPLETE message.

 $\ensuremath{\textcircled{\sc 0}}$ 2003 Microsoft Corporation. All rights reserved.

IDirectPlay8Server::Close Method

Closes the open connection to a session and uninitializes the <u>IDirectPlay8Server</u> object.

Syntax

HRESULT Close(const DWORD *dwFlags*);

Parameters

dwFlags

[in] The following flag can be specified. DPNCLOSE_IMMEDIATE Close immediately. Do not wait for outstanding calls to complete.

Return Value

Returns S_OK if successful, or the following error value.

DPNERR_UNINITIALIZED The requested object has not been initialized.

Remarks

This method must be called on any object successfully initialized with <u>IDirectPlay8Server::Initialize</u>.

This method is a counterpart to <u>IDirectPlay8Server::Host</u>. It closes all active network connections hosted by the server. This method is synchronous, and will not return until the server has processed all

DPN_MSGID_DESTROY_PLAYER messages. This feature guarantees that when **IDirectPlay8Server::Close** returns, you can safely shut down the server application.

Calling **IDirectPlay8Server::Close** will cancel all outstanding operations, including guaranteed messages that are in the queue waiting to be sent. Messages that have already been sent as guaranteed will continue to be retried until acknowledgement of their delivery has been received. To make sure all messages are sent, wait for all outstanding <u>IDirectPlay8Server::SendTo</u> calls to complete before calling **IDirectPlay8Server::Close**.

Calling IDirectPlay8Server::Close will invalidate any DPN_CAPS, DPN_CAPS_EX, and DPN_SP_CAPS associated with the IDirectPlay8Server object.

IDirectPlay8Server::CreateGroup Method

Creates a group in the current session. When this method is called, the server's message handler receives a DPN_MSGID_CREATE_GROUP message.

Syntax

```
HRESULT CreateGroup(
   const DPN_GROUP_INFO const *const pdpnGroupInfo,
   VOID *const pvGroupContext,
   VOID *const pvAsyncContext,
   DPNHANDLE *const phAsyncHandle,
   const DWORD dwFlags
);
```

Parameters

pdpnGroupInfo

[in] Pointer to a <u>DPN_GROUP_INFO</u> structure that contains the group description.

pvGroupContext

[in] Pointer to the context value for the group. This value is preset when the local application's message handler processes the DPN_MSGID_CREATE_GROUP message. This parameter is optional and may be set to NULL.

pvAsyncContext

[in] Pointer to the user-supplied context, which is returned in the **pvUserContext** member of the

<u>DPN_MSGID_ASYNC_OP_COMPLETE</u> system message. *phAsyncHandle*

[out] A **DPNHANDLE**. A value will be returned. However, Microsoft® DirectPlay® does not permit cancellation of this operation, so the value cannot be used.

dwFlags

[in] Flag that controls how this method is processed. The

following flag can be set for this method. DPNCREATEGROUP_SYNC Causes this method to process synchronously.

Return Value

Returns S_OK if this method is processed synchronously and is successful. By default, this method is run asynchronously and generally returns <u>DPNSUCCESS_PENDING</u> or the following error value.

DPNERR_INVALIDFLAGS The flags passed to this method are invalid.

Remarks

DirectPlay does not maintain hierarchical groups because these can easily be implemented with flat groups and expeditious use of the group data.

Note Multicasting is not supported for this release.

IDirectPlay8Server::DestroyClient Method

Deletes a client from the session.

Syntax

```
HRESULT DestroyClient(
    const DPNID dpnidClient,
    const VOID *const pDestroyInfo,
    const DWORD dwDestroyInfoSize,
    const DWORD dwFlags
);
```

Parameters

dpnidClient

[in] Variable of type **DPNID** that specifies the identifier of the client to delete.

pDestroyInfo

[in] Pointer that describes additional delete data information. *dwDestroyInfoSize*

[in] Variable of type **DWORD** that specifies the size of the data in the *pDestroyInfo* parameter.

dwFlags

[in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

```
DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_INVALIDPLAYERThe player ID is not recognized as a valid player ID for this game session.DPNERR_NOTHOSTThe client attempted to connect to a nonhost computer. Additionally, this error value<br/>may be returned by a nonhost that tried to set the application description.
```

 $\ensuremath{\textcircled{\sc 0}}$ 2003 Microsoft Corporation. All rights reserved.

IDirectPlay8Server::DestroyGroup Method

Deletes a group created by the <u>IDirectPlay8Server::CreateGroup</u> method.

Syntax

```
HRESULT DestroyGroup(
const DPNID idGroup,
PVOID const pvAsyncContext,
DPNHANDLE *const phAsyncHandle,
const DWORD dwFlags
```

);

Parameters

idGroup

[in] **DPNID** of the group to delete.

pvAsyncContext

[in] Pointer to the user-supplied context, which is returned in the **pvUserContext** member of the

<u>DPN_MSGID_ASYNC_OP_COMPLETE</u> system message. This parameter is optional and may be set to NULL.

phAsyncHandle

[out] A **DPNHANDLE**. A value will be returned. However, Microsoft® DirectPlay® does not permit cancellation of this operation, so the value cannot be used.

dwFlags

[in] Flag that controls how this method is processed. The following flag can be set for this method.

DPNDESTROYGROUP_SYNC

Causes the method to process synchronously.

Return Value

Returns S_OK if this method is processed synchronously and is successful. By default, this method is run asynchronously and

generally returns <u>DPNSUCCESS_PENDING</u> or one of the following error values.

DPNERR_INVALIDFLAGSThe flags passed to this method are invalid.DPNERR_INVALIDGROUPThe group ID is not recognized as a valid group ID for this game session.
IDirectPlay8Server::EnumGroupMembers Method

Retrieves a list of all players in a group.

Syntax

```
HRESULT EnumGroupMembers(
const DPNID dpnid,
DPNID *const prgdpnid,
DWORD *const pcdpnid,
const DWORD dwFlags
);
```

Parameters

dpnid

[in] Variable of type **DPNID** that specifies the group that contains the players to enumerate.

prgdpnid

[out] Pointer to an array that contains the identifiers of the group's players.

pcdpnid

[in, out] Pointer to a variable of type **DWORD** that contains the number of player identifiers in the *prgdpnid* parameter. If the buffer is too small, this method returns

DPNERR_BUFFERTOOSMALL and this parameter is set to the number of entries that are required.

dwFlags

[in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

| DPNERR_BUFFERTOOSMALL | The supplied buffer is not large enough to contain the requested data. |
|-----------------------|---|
| DPNERR_INVALIDFLAGS | The flags passed to this method are invalid. |
| DPNERR_INVALIDGROUP | The group ID is not recognized as a valid group ID for this game session. |

IDirectPlay8Server::EnumPlayersAndGroups Method

Retrieves a list of all the player and/or group identifiers for the application.

Syntax

```
HRESULT EnumPlayersAndGroups(
DPNID *const prgdpnid,
DWORD *const pcdpnid,
const DWORD dwFlags
);
```

Parameters

prgdpnid

[out] Pointer to an array that will be filled with the session's group and/or player identifiers.

pcdpnid

[in, out] Pointer to a variable of type **DWORD** that specifies the number of identifiers in the *prgdpnid* parameter. If the buffer is too small, this method returns DPNERR_BUFFERTOOSMALL and this parameter contains the number of entries that are required.

dwFlags

[in] Flag that describes enumeration behavior. You can set one or both of the following flags.

DPNENUM_PLAYERS

Return a list of player identifiers.

DPNENUM_GROUPS

Return a list of group identifiers.

Return Value

Returns S_OK if successful, or one of the following error values.

Remarks

Because group and player information changes frequently, the required buffer size returned may change between subsequent calls. Check and reallocate the buffer until the method succeeds.

IDirectPlay8Server::EnumServiceProviders Method

Enumerates the registered service providers available to the application.

Syntax

```
HRESULT EnumServiceProviders(
const GUID *const pguidServiceProvider,
const GUID *const pguidApplication,
DPN_SERVICE_PROVIDER_INFO *const pSPInfoBuffer,
PDWORD const pcbEnumData,
PDWORD const pcReturned,
const DWORD dwFlags
);
```

Parameters

```
pguidServiceProvider
```

```
[in] Pointer to a variable of type globally unique identifier (GUID) that specifies a service provider. This optional parameter forces the enumeration of subdevices for the specified service provider. You should normally set this value to NULL, to enumerate all available service providers. Otherwise, set pguidServiceProvider to one of the following predefined values.
```

```
CLSID_DP8SP_TCPIP
```

```
Internet Protocol (IP) service providers
```

```
CLSID_NETWORKSIMULATOR_DP8SP_TCPIP
```

```
DP8Sim service providers
```

```
CLSID_DP8SP_SERIAL
```

Serial service providers

```
CLSID_DP8SP_MODEM
```

Modem service providers

```
CLSID_DP8SP_IPX
```

IPX service providers

pguidApplication

[in] Pointer to a variable of type GUID that specifies an

application. If a pointer is passed in this parameter, only service providers who can be connected to the application are enumerated. You can also pass NULL to enumerate the registered service providers for the system.

pSPInfoBuffer

[out] Pointer to an array of <u>DPN_SERVICE_PROVIDER_INFO</u> structures that will be filled with service provider information.

pcbEnumData

[out] Pointer to **DWORD**, which is filled with the size of the *pSPInfoBuffer* buffer if the buffer is too small.

pcReturned

[out] Pointer to a variable of type **DWORD** that specifies the number of **DPN_SERVICE_PROVIDER_INFO** structures returned in the *pcbEnumData* array.

dwFlags

[in] The following flag can be specified.

DPNENUMSERVICEPROVIDERS_ALL

Enumerates all the registered service providers for the system including those that are not available to the application or do not have devices installed.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_BUFFERTOOSMALLThe supplied buffer is not large enough to contain the requested data.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.

Remarks

Call this method initially by specifying NULL in the *pguidServiceProvider* parameter to determine the base service providers available to the system. Specific devices for a service provider can be obtained by passing a pointer to a specific service provider GUID in the *pguidServiceProvider*. This is useful, for example, when using the Modem Connection for Microsoft®

DirectPlay® service provider. You can choose between different modems for dialing out and select specific modems for hosting.

If the *pcbEnumData* buffer is not big enough to hold the requested service provider information, the method returns DPNERR_BUFFERTOOSMALL and the *pcbEnumData* parameter contains the required buffer size. Typically, the best strategy is to call the method once with a zero-length buffer to determine the required size. Then call the method again with the appropriate sized buffer.

Normally, this method will return only those service providers that can be used by the application. For example, if the Internetwork Packet Exchange (IPX) networking protocol is not installed, DirectPlay will not return the IPX service provider. To have DirectPlay return all service providers, even those that cannot be used by the application, set the DPNENUMSERVICEPROVIDERS_ALL flag in *dwFlags*.

IDirectPlay8Server::GetApplicationDesc Method

Retrieves the full application description for the connected application.

Syntax

```
HRESULT GetApplicationDesc(
    DPN_APPLICATION_DESC *const pAppDescBuffer,
    DWORD *const pcbDataSize,
    const DWORD dwFlags
);
```

Parameters

pAppDescBuffer

[out] Pointer to a <u>DPN_APPLICATION_DESC</u> structure where the application description data is to be written. Set this parameter to NULL to request only the size of data. If *pAppDescBuffer* is not set to NULL, you must set the *pAppDescBuffer*.dwSize member to an appropriate value. The *pcbDataSize* parameter is set to the size required to hold the data.

pcbDataSize

[in, out] Pointer to a variable of type **DWORD** that is initialized to the size of the buffer before calling this method. After the method returns, this parameter is set to the size, in bytes, of the session data. If the buffer is too small, this method returns the DPNERR_BUFFERTOOSMALL error value, and this parameter is set to the buffer size required. If this parameter is NULL, the method returns DPNERR INVALIDPARAM.

```
dwFlags
```

[in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

| DPNERR_BUFFERTOOSMALL | The supplied buffer is not large enough to contain the requested data. |
|-----------------------|--|
| DPNERR_INVALIDFLAGS | The flags passed to this method are invalid. |
| DPNERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
| DPNERR_NOCONNECTION | No communication link was established. |

Remarks

Call this method initially by passing NULL in the *pcbDataSize* parameter to obtain the size of the required buffer. When you call the method a second time to fill the buffer, be sure to set the structures **dwSize** member to the appropriate value.

To avoid accidentally overwriting the application description, applications should call **IDirectPlay8Server::GetApplicationDesc** and fill in the **DPN_APPLICATION_DESC** structure before calling <u>IDirectPlay8Server::SetApplicationDesc</u>.

IDirectPlay8Server::GetCaps Method

Retrieves the <u>DPN_CAPS</u> or <u>DPN_CAPS_EX</u> structure for the current interface.

Syntax

```
HRESULT GetCaps(
    DPNCAPS *const pdpnCaps,
    const DWORD dwFlags
);
```

Parameters

pdpnCaps

[out] Pointer to a **DPN_CAPS** or **DPN_CAPS_EX** structure to receive caps information. You must set the **dwSize** member of this structure to an appropriate value.

dwFlags

[in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDOBJECTThe Microsoft® DirectPlay® object pointer is invalid.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_INVALIDPOINTERPointer specified as a parameter is invalid.DPNERR_UNINITIALIZEDThe requested object has not been initialized.

Remarks

A successful call to <u>IDirectPlay8Server::Initialize</u> must be made before this method can be called.

DirectPlay will determine whether **DPN_CAPS** or **DPN_CAPS_EX** is being used, based on the size of the structure referenced by *pdpnCaps*.

IDirectPlay8Server::GetClientAddress Method

Retrieves the address for the specified player in the session.

Syntax

```
HRESULT GetClientAddress(
    const DPNID dpnid,
    IDirectPlay8Address **const pAddress,
    const DWORD dwFlags
);
```

Parameters

dpnid

[in] Variable of type **DPNID** specifying the identification of the player.

pAddress

[out] Address of a pointer to an <u>IDirectPlay8Address</u> object that specifies the address of the client. You must release this object when you no longer need it.

```
dwFlags
```

[in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDOBJECTThe Microsoft® DirectPlay® object pointer is invalid.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_INVALIDPLAYERThe player ID is not recognized as a valid player ID for this game session.DPNERR_INVALIDPOINTERPointer specified as a parameter is invalid.DPNERR_UNINITIALIZEDThe requested object has not been initialized.

Remarks

Use the <u>IDirectPlay8Server::GetLocalHostAddresses</u> method to retrieve addresses that can be used to connect to the session.

IDirectPlay8Server::GetClientInfo Method

Retrieves the client information set for the specified client.

Syntax

```
HRESULT GetClientInfo(
const DPNID dpnid,
DPN_PLAYER_INFO *const pdpnPlayerInfo,
DWORD *const pdwSize,
const DWORD dwFlags
```

);

Parameters

dpnid

[in] Variable of type **DPNID** that specifies the identifier of the client to retrieve the information for.

pdpnPlayerInfo

[out] Pointer to a <u>DPN_PLAYER_INFO</u> structure that is filled with client information. If *pdwSize* is not set to NULL, you must set *pdpnPlayerInfo*.dwSize to an appropriate value.

pdwSize

[in, out] Pointer to a variable of type **DWORD** that contains the size of the client data returned in the *pdpnPlayerInfo* parameter. If the buffer is too small, this method returns

DPNERR_BUFFERTOOSMALL and this parameter contains the size of the required buffer.

dwFlags

[in] Flags describing the information returned for the client.

Currently, both of the following flags are returned.

DPNINFO_NAME

The **DPN_PLAYER_INFO** structure contains the name set for the client.

DPNINFO_DATA

The **DPN_PLAYER_INFO** structure contains the data set

for the client.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_BUFFERTOOSMALLThe supplied buffer is not large enough to contain the requested data.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_INVALIDPLAYERThe player ID is not recognized as a valid player ID for this game session.

Remarks

Call this method after the server receives a <u>DPN_MSGID_CLIENT_INFO</u> message from the application. This message indicates that a client has updated its information.

Microsoft® DirectPlay® returns the **DPN_PLAYER_INFO** structure, and the pointers assigned to the structure's **pwszName** and **pvData** members in a contiguous buffer. If the two pointers were set, you must have allocated enough memory for the structure, plus the two pointers. The most robust way to use this method is to first call it with *pdwSize* set to NULL. When the method returns, *pdwSize* will point to the correct value. Use that value to allocate memory for the structure and call the method a second time to retrieve the information.

When the method returns, the **dwInfoFlags** member of the **DPN_PLAYER_INFO** structure will always have the DPNINFO_DATA and DPNINFO_NAME flags set, even if the corresponding pointers are set to NULL. These flags are used when calling <u>IDirectPlay8Client::SetClientInfo</u>, to notify DirectPlay of which values have changed.

Transmission of nonstatic information should be handled with the <u>IDirectPlay8Client::Send</u> method because of the high cost of using the <u>IDirectPlay8Peer::SetPeerInfo</u> method.

The player sets the information by calling **IDirectPlay8Client::SetClientInfo**.

IDirectPlay8Server::GetConnectionInfo Method

Retrieves statistical information about the connection between the local server and the specified remote client.

Syntax

```
HRESULT GetConnectionInfo(
    const DPNID dpnidEndPoint,
    DPN_CONNECTION_INFO *const pdnConnectInfo,
    const DWORD dwFlags
);
```

Parameters

dpnidEndPoint

[in] **DPNID** of the player whose connection information will be retrieved.

pdnConnectInfo

[out] Pointer to a <u>DPN_CONNECTION_INFO</u> structure to retrieve information about the specified connection. The dwSize member of this structure must be set to the size of a **DPN_CONNECTION_INFO** structure.

dwFlags

[in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDOBJECTThe Microsoft® DirectPlay® object pointer is invalid.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_INVALIDPOINTERPointer specified as a parameter is invalid.DPNERR_UNINITIALIZEDThe requested object has not been initialized.

Remarks

This method can be called only after a successful <u>IDirectPlay8Server::Host</u> call has completed.

IDirectPlay8Server::GetGroupContext Method

Retrieves the group context value for a group.

Syntax

```
HRESULT GetGroupContext(
const DPNID dpnid,
PVOID *const ppvGroupContext,
const DWORD dwFlags
);
```

Parameters

dpnid [in] Variable of type **DPNID** that specifies the identifier of the group to get context data for. *ppvGroupContext* [out] Pointer to the context value of the group. *dwFlags* [in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDGROUPThe group ID is not recognized as a valid group ID for this game session.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_NOTREADYThe object is not ready for use.

Remarks

Group context values are set by pointing the **pvGroupContext** member of the <u>DPN_MSGID_CREATE_GROUP</u> system message to the context value data.

This method returns <u>DPNERR_NOTREADY</u> when it is called before a **DPN_MSGID_CREATE_GROUP** message is received by Microsoft® DirectPlay® for the group specified in *dpnid*. Call **IDirectPlay8Server::GetGroupContext** again allowing task switching so that the thread carrying the message can return.

IDirectPlay8Server::GetGroupInfo Method

Retrieves a block of data associated with a group, including the group name.

This method is typically called after a <u>DPN_MSGID_GROUP_INFO</u> system message is received, indicating that the group data has been modified.

Syntax

```
HRESULT GetGroupInfo(
    const DPNID dpnid,
    DPN_GROUP_INFO *const pdpnGroupInfo,
    DWORD *const pdwSize,
    const DWORD dwFlags
);
```

Parameters

dpnid

[in] Variable of type **DPNID** that specifies the identifier of the group whose data block will be retrieved.

pdpnGroupInfo

[out] Pointer to a <u>DPN_GROUP_INFO</u> structure that describes the group data. If pdwSize is not set to NULL, you must set *pdpnGroupInfo*.dwSize to the size of a **DPN_GROUP_INFO** structure.

pdwSize

[in, out] Pointer to a variable of type **DWORD** that returns the size of the data in the *pdpnGroupInfo* parameter. If the buffer is too small, this method returns DPNERR_BUFFERTOOSMALL and this parameter contains the required size.

dwFlags

[in] Flags describing the information returned for the group.

Currently, both of the following flags are returned. DPNINFO_NAME The DPN_PLAYER_INFO structure contains the name set for the client. DPNINFO_DATA The DPN_PLAYER_INFO structure contains the data set for the client.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_BUFFERTOOSMALLThe supplied buffer is not large enough to contain the requested data.DPNERR_INVALIDFLAGSThe flags passed to this method are invalid.DPNERR_INVALIDGROUPThe group ID is not recognized as a valid group ID for this game session.

Remarks

Microsoft® DirectPlay® returns the **DPN_GROUP_INFO** structure, and the pointers assigned to the structure's **pwszName** and **pvData** members in a contiguous buffer. If the two pointers were set, you must have allocated enough memory for the structure, plus the two pointers. The most robust way to use this method is to first call it with *pdwSize* set to NULL. When the method returns, *pdwSize* will point to the correct value. Use that value to allocate memory for the structure and call the method a second time to retrieve the information.

IDirectPlay8Server::GetLocalHostAddresses Method

Retrieves the local addresses being used to host the session.

Syntax

```
HRESULT GetLocalHostAddresses(
IDirectPlay8Address **const prgpAddress,
DWORD *const pcAddress,
const DWORD dwFlags
);
```

Parameters

prgpAddress

[out] Address of a pointer to an array of <u>IDirectPlay8Address</u> objects that specify the local host addresses. You must release these objects when you no longer need them or you will create memory leaks.

pcAddress

[in, out] Maximum number of address objects that can be contained in the array pointed to by *prgpAddress*. If the buffer is too small, the method returns DPNERR_BUFFERTOOSMALL, and *pcAddress* will be set to the required value.

dwFlags

[in] The following flag can be specified when using the Transmission Control Protocol/Internet Protocol (TCP/IP) service provider. This method will return

DPNERR_UNSUPPORTED if this flag is used with any other service provider.

DPNGETLOCALHOSTADDRESSES_COMBINED Return all listening addresses combined into one IDirectPlay8Address object.

Return Value

Returns S_OK if successful, or one of the following error values.

| DPNERR_INVALIDOBJECT | The Microsoft® DirectPlay® object pointer is invalid. |
|-----------------------|--|
| DPNERR_BUFFERTOOSMALL | The supplied buffer is not large enough to contain the requested data. |
| DPNERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
| DPNERR_INVALIDPOINTER | Pointer specified as a parameter is invalid. |
| DPNERR_UNINITIALIZED | The requested object has not been initialized. |
| DPNERR_UNSUPPORTED | The function or feature is not available in this implementation or on this service provider. |

Remarks

The most robust way to use this method is to first call it with *pcAddress* set to 0. When the method returns, *pcAddress* will point to the required value. You can use that value when you call the method for a second time to retrieve the information.

If DPNGETLOCALHOSTADDRESSES_COMBINED is specified, the address object returned will contain all listening server addresses. For example, the server might have multiple addresses if it is behind a Network Address Translation (NAT) device or if it has multiple network cards. In this case, players can connect to the server faster if they can try all of the addresses simultaneously. The application must provide its own mechanism for passing the combined address object to the connecting players. One way to do this is by using IDirectPlay8Address::GetURLA or IDirectPlay8Address::GetURLW and IDirectPlay8Address::BuildFromURLA or IDirectPlay8Address::BuildFromURLA or IDirectPlay8Address::BuildFromURLW to create a string to pass using a Web page or lobby mechanism.
IDirectPlay8Server::GetPlayerContext Method

Retrieves the player context value for a client.

Syntax

```
HRESULT GetPlayerContext(
const DPNID dpnid,
PVOID *const ppvPlayerContext,
const DWORD dwFlags
);
```

Parameters

dpnid [in] Variable of type **DPNID** that specifies the identifier of the player to get context data for. *ppvPlayerContext* [out] Pointer to the context data of the client. *dwFlags* [in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

```
DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_INVALIDPLAYERThe player ID is not recognized as a valid player ID for this game session.DPNERR_NOTREADYThe object is not ready for use.
```

Remarks

Player context values are set by pointing the **pvPlayerContext** member of the <u>DPN_MSGID_CREATE_PLAYER</u> system message to the context value data.

This method returns <u>DPNERR_NOTREADY</u> when it is called before a **DPN_MSGID_CREATE_PLAYER** message is received by Microsoft® DirectPlay® for the player specified in *dpnid*. Call **IDirectPlay8Server::GetPlayerContext** again allowing task switching so that the thread carrying the message can return.

IDirectPlay8Server::GetSendQueueInfo Method

Used by the application to monitor the size of the send queue. Microsoft® DirectPlay® does not send messages faster than the receiving computer can process them. As a result, if the sending computer is sending faster than the receiver can receive, messages accumulate in the sender's queue. If the application registers that the send queue is growing too large, it should slow the rate that messages are sent.

Syntax

HRESULT GetSendQueueInfo(const DPNID dpnid, DWORD *const pdwNumMsgs, DWORD *const pdwNumBytes, const DWORD dwFlags

);

Parameters

dpnid

[in] Variable of type **DPNID** that specifies the identifier of the player to get the send-queue information for.

pdwNumMsgs

[out] Pointer to a variable of type **DWORD** that contains the number of messages currently queued. This value is optional, and may be set to NULL.

pdwNumBytes

[out] Pointer to a variable of type **DWORD** that specifies the total number of bytes of data of the messages currently queued. This value is optional, and may be set to NULL.

dwFlags

[in] You may specify the DPNGETSENDQUEUEINFO_PRIORITY_NORMAL, DPNGETSENDQUEUEINFO_PRIORITY_HIGH, or DPNGETSENDQUEUEINFO_PRIORITY_LOW flag to inquire about specific messages of that priority.

Return Value

Returns S_OK if successful, or the following error value.

DPNERR_INVALIDPARAM One or more of the parameters passed to the method are invalid.

Remarks

You cannot set both *pdwNumMsgs* and *pdwNumBytes* to NULL. At least one of them must be set to a valid pointer.

IDirectPlay8Server::GetSPCaps Method

Retrieves the <u>DPN_SP_CAPS</u> structure for the specified service provider.

Syntax

```
HRESULT GetSPCaps(
    const GUID *const pguidSP,
    DPN_SP_CAPS *const pdpnSPCaps,
    const DWORD dwFlags
);
```

Parameters

pguidSP

[in] Pointer to a globally unique identifier (GUID) specifying the service provider you want to get information about.

pdpnSPCaps

[out] Pointer to a **DPN_SP_CAPS** structure to receive the information about the specified service provider. You must set the *pdpnSPCaps*.dwSize member of this structure to an appropriate value.

```
dwFlags
```

[in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDOBJECTThe Microsoft® DirectPlay® object pointer is invalid.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_INVALIDPOINTERPointer specified as a parameter is invalid.DPNERR_UNINITIALIZEDThe requested object has not been initialized.

Remarks

This method retrieves information about the specified service provider. A successful call to <u>IDirectPlay8Server::Initialize</u> must be made before this method can be called.

IDirectPlay8Server::Host Method

Creates a new client/server session, hosted by the local computer.

Syntax

```
HRESULT Host(
const DPN_APPLICATION_DESC *const pdnAppDesc,
IDirectPlay8Address **const prgpDeviceInfo,
const DWORD cDeviceInfo,
const DPN_SECURITY_DESC *const pdpSecurity,
const DPN_SECURITY_CREDENTIALS *const pdpCredentials,
VOID *const pvPlayerContext,
const DWORD dwFlags
```

);

Parameters

pdnAppDesc

[in] Pointer to a <u>DPN_APPLICATION_DESC</u> structure that describes the application.

prgpDeviceInfo

[in] Pointer to an array of <u>IDirectPlay8Address</u> objects containing device addresses that should be used to host the application.

cDeviceInfo

[in] Variable of type **DWORD** that specifies the number of device address objects in the array pointed to by *prgpDeviceInfo*.

pdpSecurity

[in] Reserved. Must be set to NULL.

pdpCredentials

[in] Reserved. Must be set to NULL.

pvPlayerContext

[in] Pointer to the context value of the player. This value is preset when the local computer handles the

DPN_MSGID_CREATE_PLAYER message. This parameter is

optional, and may be set to NULL. *dwFlags* [in] The following flag can be specified. DPNHOST_OKTOQUERYFORADDRESSING Setting this flag will display a standard Microsoft® DirectPlay® dialog box, which queries the user for more information if not enough information is passed in this method.

Return Value

Returns S_OK if successful, or the following error value.

| DPNERR_DATATOOLARGE | The application data is too large for the service provider's Maximum Transmission Unit. |
|---------------------------|---|
| DPNERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
| DPNERR_DPNSVRNOTAVAILABLE | Port 6073 is already in use. |
| DPNERR_ALREADYINITIALIZED | The object has already been initialized. |

Remarks

Do not set the **guidInstance** member of the **DPN_APPLICATION_DESC** structure when calling **IDirectPlay8Server::Host** because DirectPlay will ignore any value passed in and determine its own globally unique identifier (GUID). The only way to retrieve the **guidInstance** is by calling <u>IDirectPlay8Server::GetApplicationDesc</u>.

If you set the DPNHOST_OKTOQUERYFORADDRESSING flag in *dwFlags*, the service provider might attempt to display a dialog box to ask the user to complete the address information. You must have a visible window present when the service provider tries to display the dialog box, or your application will lock.

Data Value Summary specifies the required addressing information

for each service provider.

The maximum size of the application data that you assign to the **pvApplicationReservedData** member of the **DPN_APPLICATION_DESC** structure is limited by the service provider's Maximum Transmission Unit. If your application data is too large, the method will fail and return DPNERR_DATATOOLARGE.

IDirectPlay8Server::Initialize Method

Registers an entry point in the server's code that receives the messages from the <u>IDirectPlay8Server</u> interface and from remote clients. This method must be called before calling any other methods of this interface.

Syntax

```
HRESULT Initialize(
    PVOID const pvUserContext,
    const PFNDPNMESSAGEHANDLER pfn,
    const DWORD dwFlags
);
```

Parameters

pvUserContext

[in] Pointer to the user-provided context value in calls to the message handler. Providing a user-context value is useful to differentiate messages from multiple interfaces to a common message handler.

pfn

[in] Pointer to a <u>PFNDPNMESSAGEHANDLER</u> callback function that receives all messages from remote clients and indications of session changes from the **IDirectPlay8Server** interface.

dwFlags

[in] You can specify the following flags.

DPNINITIALIZE_DISABLEPARAMVAL

Passing this flag will disable parameter validation for the current object.

DPNINITIALIZE_HINT_LANSESSION

Opens a larger send window for games running on a local area network (LAN).

DPNINITIALIZE_DISABLELINKTUNING

Disable any attempts by Microsoft® DirectPlay® to tune the rate it sends at to the observed network conditions.

Messages will be pushed out onto the network at the first available opportunity.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDFLAGSThe flags passed to this method are invalid.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.

Remarks

Call this method first after using **CoCreateInstance** to obtain the **IDirectPlay8Server** interface.

Applications might want to specify the

DPNINITIALIZE_DISABLELINKTUNING flag when they send at a fixed rate and do not alter the rate based on the network conditions. With this flag specified, DirectPlay will always assume the network has the capacity to carry all the application data and will therefore not attempt to tune its send rate to the network bandwidth. Specifying this flag and then sending at a rate that exceeds the capacity of the network will lead to unpredictable network behavior such as higher latency and increased packet drop rates. Applications that monitor the send queues and dynamically adjust their send rate to make best use of the available bandwidth should not specify this flag.

If the DPNINITIALIZE_DISABLELINKTUNING flag is specified, DirectPlay features such as message prioritization, coalescence, and timeout are not useful because messages always go directly to the network and are not queued. $\ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.

IDirectPlay8Server::RegisterLobby Method

Allows launched applications to automatically propagate game status to the lobby.

Syntax

```
HRESULT RegisterLobby(
    const DPNHANDLE dpnHandle,
    IDirectPlay8LobbiedApplication *const pIDP8LobbiedApplication,
    const DWORD dwFlags
);
```

Parameters

dpnHandle

[in] Connection handle used when making the calls to IDirectPlay8LobbiedApplication::UpdateStatus.

pIDP8LobbiedApplication

[in] Pointer to the <u>IDirectPlay8LobbiedApplication</u> object that specifies the application.

dwFlags

[in] One of the following flags.

DPNLOBBY_REGISTER

Registers the lobby with the application.

DPNLOBBY_UNREGISTER

Unregisters the lobby with the application.

Return Value

Returns S_OK if successful, or the following error value.

DPNERR_INVALIDPARAM One or more of the parameters passed to the method are invalid.

 $\ensuremath{\textcircled{\sc 0}}$ 2003 Microsoft Corporation. All rights reserved.

IDirectPlay8Server::RemovePlayerFromGroup Method

Removes a client from a group.

Syntax

```
HRESULT RemovePlayerFromGroup(
const DPNID idGroup,
const DPNID idClient,
PVOID const pvAsyncContext,
DPNHANDLE *const phAsyncHandle,
const DWORD dwFlags
```

);

Parameters

idGroup

[in] Variable of type **DPNID** that specifies the identifier of the group to remove the client from.

idClient

[in] Variable of type **DPNID** that specifies the identifier of the client to remove from the group.

pvAsyncContext

[in] Pointer to the user-supplied context, which is returned in the **pvUserContext** member of the

DPN_MSGID_ASYNC_OP_COMPLETE system message.

phAsyncHandle

[out] A **DPNHANDLE**. A value will be returned. However, Microsoft® DirectPlay® does not permit cancellation of this operation, so the value cannot be used.

dwFlags

[in] Flag that controls how this method is processed. The following flag can be set for this method.

DPNREMOVEPLAYERFROMGROUP_SYNC

Causes this method to process synchronously.

Return Value

Returns S_OK if this method is processed synchronously and is successful. By default, this method is run asynchronously and generally returns <u>DPNSUCCESS_PENDING</u> or one of the following error values.

| DPNERR_INVALIDFLAGS | The flags passed to this method are invalid. |
|-------------------------|---|
| DPNERR_INVALIDGROUP | The group ID is not recognized as a valid group ID for this game session. |
| DPNERR_INVALIDPLAYER | The player ID is not recognized as a valid player ID for this game session. |
| DPNERR PLAYERNOTINGROUP | The player ID is not included in the group. |

Remarks

When this method is called, the server's message handler receives a <u>DPN_MSGID_REMOVE_PLAYER_FROM_GROUP</u> message.

IDirectPlay8Server::ReturnBuffer Method

Retrieves message buffers provided to the application through the **pReceiveData** member of the <u>DPN_MSGID_RECEIVE</u> system message. If the user's message handler returns DPNSUCCESS_PENDING to the RECEIVE callback, Microsoft® DirectPlay® assumes ownership of the buffer has been transferred to the application, and neither frees nor modifies it until ownership is returned to DirectPlay through this call.

Syntax

```
HRESULT ReturnBuffer(
const DPNHANDLE hBufferHandle,
const DWORD dwFlags
);
```

Parameters

hBufferHandle

[in] Variable of type **DPNHANDLE** that specifies the buffer handle to the message. This is obtained in the *hBufferHandle* member of the **DPN_MSGID_RECEIVE** system message. *dwFlags*

[in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDHANDLEThe handle specified is invalid.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.

IDirectPlay8Server::SendTo Method

Transmits data to a client or group within the session. The message can be sent synchronously or asynchronously.

Syntax

```
HRESULT SendTo(
const DPNID dpnid,
const DPN_BUFFER_DESC *const pBufferDesc,
const DWORD cBufferDesc,
const DWORD dwTimeOut,
void *const pvAsyncContext,
DPNHANDLE *const phAsyncHandle,
const DWORD dwFlags
```

```
);
```

Parameters

dpnid

[in] Identifier of the client or group to receive data. Set this parameter to DPNID_ALL_PLAYERS_GROUP to send a message to all players in the session.

pBufferDesc

[in] Pointer to a <u>DPN_BUFFER_DESC</u> structure that describes the data to send.

cBufferDesc

[in] Number of **DPN_BUFFER_DESC** structures pointed to by *pBufferDesc*. There can be up to eight buffers in this version of Microsoft® DirectPlay®.

dwTimeOut

[in] Number of milliseconds to wait for the message to send. If the message has not been sent by the *dwTimeOut* value, it is deleted from the send queue. If you set this parameter to 0, the message remains in the send queue until it is sent or until the link is dropped. *pvAsyncContext*

[in] Pointer to the user-supplied context, which is returned in the **pvUserContext** member of the

DPN_MSGID_SEND_COMPLETE system message.

phAsyncHandle

[out] A **DPNHANDLE**. When the method returns,

phAsyncHandle will point to a handle that you can pass to <u>IDirectPlay8Server::CancelAsyncOperation</u> to cancel the operation. This parameter must be set to NULL if you set the DPNSEND SYNC flag in *dwFlags*.

dwFlags

[in] Flags that describe send behavior. You can set one or more of the following flags.

DPNSEND_SYNC

Process the **IDirectPlay8Server::SendTo** request synchronously.

DPNSEND_NOCOPY

Use the data in the **DPN_BUFFER_DESC** structure and do not make an internal copy. This can be a more efficient method of sending data. However, it is less robust because modifying or deleting the data before receiving the **DPN_MSGID_SEND_COMPLETE** message can cause erroneous data to be sent. This flag cannot be used with DPNSEND_NOCOMPLETE.

DPNSEND_NOCOMPLETE

Do not send the **DPN_MSGID_SEND_COMPLETE** structure to the message handler. This flag cannot be used with DPNSEND_NOCOPY or DPNSEND_GUARANTEED. Additionally, when using this flag *pvAsyncContext* must be NULL.

DPNSEND_COMPLETEONPROCESS

Send the **DPN_MSGID_SEND_COMPLETE** to the message handler when this message has been delivered to the target and the target's message handler returns from indicating its reception. There is additional internal message overhead when this flag is set, and the message transmission process might become significantly slower. If you set this flag, DPNSEND_GUARANTEED must also be set.

DPNSEND_GUARANTEED

Send the message by a guaranteed method of delivery. DPNSEND_PRIORITY_HIGH

Sets the priority of the message to high. This flag cannot be used with DPNSEND_PRIORITY_LOW.

DPNSEND_PRIORITY_LOW

Sets the priority of the message to low. This flag cannot be used with DPNSEND_PRIORITY_HIGH.

DPNSEND_NOLOOPBACK

Suppress the <u>DPN_MSGID_RECEIVE</u> system message to your message handler when you are sending to a group that includes the local player. For example, this flag is useful if you are broadcasting to the entire session.

DPNSEND_NONSEQUENTIAL

If this flag is set, the target application will receive the messages in the order that they arrive at the user's computer. If this flag is not set, messages are delivered sequentially, and will be received by the target application in the order that they were sent. Doing so might require buffering incoming messages until missing messages arrive.

DPNSEND_COALESCE

Allows DirectPlay to combine packets when sending.

Return Value

Returns S_OK if this method is processed synchronously and is successful. By default, this method is run asynchronously and generally returns <u>DPNSUCCESS_PENDING</u> or one of the following error values.

| DPNERR_CONNECTIONLOST | The service provider connection was reset while data was being sent. |
|-----------------------|---|
| DPNERR_INVALIDFLAGS | The flags passed to this method are invalid. |
| DPNERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
| DPNERR_INVALIDPLAYER | The player ID is not recognized as a valid player ID for this game session. |
| DPNERR_TIMEDOUT | The operation could not complete because it has timed out. |

Remarks

This method generates a **DPN_MSGID_RECEIVE** system message in the receiver's message handler. The data is contained in the **pReceiveData** member of the associated structure.

Messages can have one of three priorities: low, normal, and high. To specify a low or high priority for the message set the appropriate flag in *dwFlags*. If neither of the priority flags is set, the message will have normal priority. For a discussion of send priorities, see <u>Basic</u> <u>Networking</u>.

When the **IDirectPlay8Server::SendTo** request is completed, a **DPN_MSGID_SEND_COMPLETE** system message is posted to the sender's message handler. The success or failure of the request is contained in the **hResultCode** member of the associated structure. You can suppress the send completion by setting the DPNSEND_NOCOMPLETE flag in *dwFlags*.

Send completions are typically posted on the source computer as soon as the message is sent. In other words, a send completion does not necessarily mean that the message has been processed on the target. It might still be in a queue. If you want to be certain that the message has been processed by the target, set the DPNSEND_COMPLETEONPROCESS flag in *dwFlags*. This flag ensures that the send completion will not be sent until the target's message handler has processed the message and returned.

If the DPNSEND_COALESCE flag is set in *dwFlags*, DirectPlay will try to coalesce up to 32 packets waiting in the queue into the outgoing frame. DirectPlay does not guarantee coalescence, even if

the DPNSEND_COALESCE flag is set. Packets will only be coalesced if there is more than one message in the queue and the player receiving is running Microsoft DirectX® 9.0 or later. All voice packets can be coalesced. Both guaranteed and non-guaranteed packets will be coalesced into the same frame. If the frame is dropped before it reaches its destination, only the guaranteed parts of the frame will be resent and no other data will be coalesced into the frame.

Note Do not assume that resources such as the data buffer will remain valid until the method has returned. If you call this method asynchronously, the **DPN_MSGID_SEND_COMPLETE** message can be received and processed by your message handler before the call has returned. If your message handler deallocates or otherwise invalidates a resource such as the data buffer, that resource can become invalid at any time after the method has been called.

IDirectPlay8Server::SetApplicationDesc Method

Changes the settings for the application that is being hosted. Only some settings can be changed.

Syntax

```
HRESULT SetApplicationDesc(
    const DPN_APPLICATION_DESC *const pad,
    const DWORD dwFlags
);
```

Parameters

pad

[in] Pointer to a <u>DPN_APPLICATION_DESC</u> structure that describes the application settings to modify. *dwFlags*

[in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

| DPNERR_DATATOOLARGE | The application data is too large for the service provider's Maximum Transmission Unit. |
|---------------------|---|
| DPNERR_INVALIDFLAGS | The flags passed to this method are invalid. |
| DPNERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |

Remarks

You can use this method to modify only the following members of the **DPN_APPLICATION_DESC** structure.

• dwMaxPlayers

- pwszSessionName
- pwszPassword
- pvApplicationReservedData
- dwApplicationReservedDataSize

When IDirectPlay8Server::SetApplicationDesc is called, Microsoft® DirectPlay® makes a copy of the data *pad* points to. You do not need to save the DPN_APPLICATION_DESC structure once IDirectPlay8Server::SetApplicationDesc returns.

You cannot set the **dwMaxPlayers** member to a smaller value than the current number of players in the session.

The maximum size of the application data that you assign to the **pvApplicationReservedData** member of the **DPN_APPLICATION_DESC** structure is limited by the service provider's Maximum Transmission Unit. If your application data is too large, the method will fail and return DPNERR_DATATOOLARGE.

To avoid accidentally overwriting the application description, applications should call <u>IDirectPlay8Server::GetApplicationDesc</u> and fill in the **DPN_APPLICATION_DESC** structure before calling **IDirectPlay8Server::SetApplicationDesc**.

IDirectPlay8Server::SetCaps Method

Sets the <u>DPN_CAPS</u> or <u>DPN_CAPS_EX</u> structure for the current interface.

Syntax

```
HRESULT SetCaps(
    const DPNCAPS *const pdpCaps,
    const DWORD dwFlags
);
```

Parameters

pdpCaps

[in] Pointer to a DPN_CAPS or DPN_CAPS_EX structure used to set the information about the current interface.

dwFlags

[in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDOBJECTThe Microsoft® DirectPlay® object pointer is invalid.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_INVALIDPOINTERPointer specified as a parameter is invalid.DPNERR_UNINITIALIZEDThe requested object has not been initialized.

Remarks

A successful call to <u>IDirectPlay8Server::Initialize</u> must be made before this method can be called.
DirectPlay will determine whether **DPN_CAPS** or **DPN_CAPS_EX** is being used, based on the size of the structure referenced by *pdpCaps*.

IDirectPlay8Server::SetGroupInfo Method

Sets a block of data associated with a group, including the name of the group.

Syntax

```
HRESULT SetGroupInfo(
const DPNID dpnid,
DPN_GROUP_INFO *const pdpnGroupInfo,
PVOID const pvAsyncContext,
DPNHANDLE *const phAsyncHandle,
const DWORD dwFlags
```

```
);
```

Parameters

dpnid

[in] Variable of type **DPNID** that specifies the identifier of the group whose data block will be modified.

pdpnGroupInfo

[in] Pointer to a <u>DPN_GROUP_INFO</u> structure that describes the group data to set. To change the values of the **pwszName** or **pvData** members of this structure, you must set the corresponding DPNINFO_NAME OR DPNINFO_DATA flag in the **dwInfoFlags** member.

pvAsyncContext

[in] Pointer to the user-supplied context, which is returned in the **pvUserContext** member of the

<u>DPN_MSGID_ASYNC_OP_COMPLETE</u> system message. *phAsyncHandle*

[out] A **DPNHANDLE**. A value will be returned. However, Microsoft® DirectPlay® does not permit cancellation of this operation, so the value cannot be used.

dwFlags

[in] Flag that controls how this method is processed. The

following flag can be set for this method. DPNSETGROUPINFO_SYNC Causes this method to process synchronously.

Return Value

Returns S_OK if this method is processed synchronously and is successful. By default, this method is run asynchronously and generally returns <u>DPNSUCCESS_PENDING</u> or one of the following error values.

DPNERR_INVALIDFLAGSThe flags passed to this method are invalid.DPNERR_INVALIDGROUPThe group ID is not recognized as a valid group ID for this game session.

IDirectPlay8Server::SetServerInfo Method

Sets the static settings of a server with an application. After clients successfully connect to the server, they can retrieve the information set by this method by calling the<u>IDirectPlay8Client::GetServerInfo</u> method.

Syntax

```
HRESULT SetServerInfo(
const DPN_PLAYER_INFO *const pdpnPlayerInfo,
PVOID const pvAsyncContext,
DPNHANDLE *const phAsyncHandle,
const DWORD dwFlags
);
```

Parameters

pdpnPlayerInfo

[in] Pointer to a <u>DPN_PLAYER_INFO</u> structure that contains the server information to set.

pvAsyncContext

[in] Pointer to the user-supplied context, which is returned in the **pvUserContext** member of the

<u>DPN_MSGID_ASYNC_OP_COMPLETE</u> system message. *phAsyncHandle*

[out] A **DPNHANDLE**. A value will be returned. However, Microsoft® DirectPlay® does not permit cancellation of this operation, so the value cannot be used.

dwFlags

[in] Flag that controls how this method is processed. The following flag can be set for this method.

DPNSETSERVERINFO_SYNC

Causes this method to process synchronously.

Return Value

Returns S_OK if this method is processed synchronously and is successful. If the request is processed asynchronously, S_OK can return if the method is instantly processed. By default, this method is run asynchronously and generally returns

<u>DPNSUCCESS_PENDING</u> or one of the following error values.

DPNERR_INVALIDFLAGSThe flags passed to this method are invalid.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_NOCONNECTIONNo communication link was established.

Remarks

This method may be called before calling <u>IDirectPlay8Server::Host</u>, and at any time during the session.

The **DPN_PLAYER_INFO** structure's **dwPlayerFlags** member must be set to zero.

Handle transmission of nonstatic information with the <u>IDirectPlay8Server::SendTo</u> method because of the high cost of using the **IDirectPlay8Server::SetServerInfo** method.

You can modify the server information with this method after clients have connected to the application. Calling this method after connection generates a <u>DPN_MSGID_SERVER_INFO</u> system message to all players, informing them that data has been updated.

When calling this method asynchronously, the contents of the *pdpnPlayerInfo* and *pvAsyncContext* buffers will be copied by DirectPlay so that the calling application can clean up the buffers before the method returns.

This method is guaranteed as long as the player is connected to the session. DirectPlay will ensure that this method completes and that the information is propagated to all players.

IDirectPlay8Server::SetSPCaps Method

Sets the <u>DPN_SP_CAPS</u> structure for the specified service provider.

Syntax

```
HRESULT SetSPCaps(
    const GUID *const pguidSP,
    const DPN_SP_CAPS *const pdpnSPCaps,
    const DWORD dwFlags
);
```

Parameters

pguidSP

[in] Pointer to a globally unique identifier (GUID) specifying the service provider you want to set information about.

pdpnSPCaps

[in] Pointer to a **DPN_SP_CAPS** structure to set the information about the specified service provider.

dwFlags

[in] Reserved. Must be 0.

Return Value

Returns S_OK if successful, or one of the following error values.

DPNERR_INVALIDOBJECTThe Microsoft® DirectPlay® object pointer is invalid.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_INVALIDPOINTERPointer specified as a parameter is invalid.DPNERR_UNINITIALIZEDThe requested object has not been initialized.

Remarks

This method sets parameters for the specified service provider. A

successful call to <u>IDirectPlay8Server</u>::Initialize must be made before this method can be called. Currently, only the **dwSystemBufferSize** member can be set by this call. The **dwNumThreads** member is for legacy support. Microsoft DirectX® 9.0 applications should use the <u>IDirectPlay8ThreadPool</u>::SetThreadCount method to set the number of threads. The other members of the **DPN_SP_CAPS** structure are get-only or ignored.

IDirectPlay8ThreadPool Interface

Applications use the methods of the **IDirectPlay8ThreadPool** interface to manage threads in a Microsoft® DirectPlay® application.

IDirectPlay8ThreadPool Members

| <u>Close</u> | Closes the IDirectPlay8ThreadPool object. |
|-----------------------|--|
| <u>DoWork</u> | Performs work that is currently scheduled. |
| <u>GetThreadCount</u> | Retrieves the number of threads for the specified processor or all processors. |
| <u>Initialize</u> | Initializes the thread pool interface for the process. |
| <u>SetThreadCount</u> | Changes the number of threads for a specified processor or all processors. |

Remarks

The serial and modem service providers do not support the **IDirectPlay8ThreadPool** interface. Therefore, if you use these service providers, you must handle multithreaded callbacks from threads that did not generate a <u>DPN_MSGID_CREATE_THREAD</u> message when created.

DirectPlay Voice uses a different set of threads to perform audio capture, playback, and message notification. If you use the IDirectPlayVoiceClient or IDirectPlayVoiceServer interface, you must handle multithreaded callbacks from threads that did not generate a DPN_MSGID_CREATE_THREAD message when created.

Interface Information

| Inherits from | <u>IUnknown</u> |
|---------------------------|-----------------|
| Header | dplay8.h |
| Minimum operating systems | Windows 98 |

IDirectPlay8ThreadPool::Close Method

Closes the IDirectPlay8ThreadPool object.

Syntax

```
HRESULT Close(
const DWORD dwFlags
);
```

Parameters

dwFlags [in] Reserved. Must be 0.

Return Value

Returns DPN_OK if successful. Otherwise, returns one of the following errors.

DPNERR_UNINITIALIZEDThe requested object has not been initialized.DPNERR_INVALIDFLAGSThe flags passed to this method are invalid.DPNERR_NOTALLOWEDThis function is not allowed on this object.

Remarks

Any threads that exist will send a

<u>DPN_MSGID_DESTROY_THREAD</u> message before this method will return.

DPNERR_NOTALLOWED is returned if this method is called when there is an outstanding call to <u>IDirectPlay8ThreadPool</u>::DoWork or if this method is called from an **IDirectPlay8ThreadPool** thread. Always close the IDirectPlay8ThreadPool interface *after* closing all of the other Microsoft® DirectPlay® objects used by the process. Closing an IDirectPlay8ThreadPool interface that was in a IDirectPlay8ThreadPool::DoWork call before closing an IDirectPlay8Peer, IDirectPlay8Client, or IDirectPlay8Server interface can cause the IDirectPlay8ThreadPool interface to appear to hang.

IDirectPlay8ThreadPool::DoWork Method

Performs work that is currently scheduled.

Syntax

```
HRESULT DoWork(
    const DWORD dwAllowedTimeSlice,
    const DWORD dwFlags
);
```

Parameters

dwAllowedTimeSlice

[in] Specifies the time allowed for the work to complete. Set to INFINITE to allow all immediately available items to be run.

dwFlags_

[in] Reserved. Must be 0.

Return Value

Returns DPN_OK if no additional work is immediately available. If *dwAllowedTimeSlice* is not set to INFINITE and the time specified has expired, leaving outstanding work items,

DPNSUCCESS_PENDING will be returned. Otherwise, returns one of the following errors.

DPNERR_UNINITIALIZEDThe requested object has not been initialized.DPNERR_INVALIDFLAGSThe flags passed to this method are invalid.DPNERR_NOTREADYThe object is not ready for use.DPNERR_NOTALLOWEDThis function is not allowed on this object.

Remarks

This method allows Microsoft® DirectPlay® to operate without any threads of its own. It is expected that this method will be called at regular intervals so that time critical operations can be performed with reasonable accuracy.

The *dwAllowedTimeSlice* parameter must be between 0 and 60,000 milliseconds (1 minute) or it can be set to INFINITE. If it is set to 0, at most the first work item will be performed.

This method cannot be called unless the thread count has been set to 0. It will return **DPNERR_NOTREADY** if there are threads currently active.

If an attempt is made to call this method by more than one thread simultaneously, recursively, or within a DirectPlay callback, **DPNERR_NOTALLOWED** is returned.

IDirectPlay8ThreadPool::GetThreadCount Method

Retrieves the number of threads for the specified processor or all processors.

Syntax

```
HRESULT GetThreadCount(
    const DWORD dwProcessorNum,
    DWORD *const pdwNumThreads,
    const DWORD dwFlags
);
```

Parameters

dwProcessorNum

[in] Specifies the processor number. Set to -1 to retrieve the thread count for all processors.

pdwNumThreads

[out] Receives the current number of threads for the processor specified in *dwProcessorNum*.

dwFlags

[in] Reserved. Must be 0.

Return Value

Returns DPN_OK if successful. Otherwise, returns one of the following errors.

DPNERR_UNINITIALIZEDThe requested object has not been initialized.DPNERR_INVALIDFLAGSThe flags passed to this method are invalid.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.

IDirectPlay8ThreadPool::Initialize Method

Initializes the thread pool interface for the process.

Syntax

```
HRESULT Initialize(

PVOID const pvUserContext,

const PFNDPNMESSAGEHANDLER pfn,

const DWORD dwFlags

);
```

Parameters

pvUserContext

[in] User context for all message callbacks.

pfn

[in] Pointer to a <u>PFNDPNMESSAGEHANDLER</u> function to handle thread pool messages.

dwFlags

[in] The following flag can be specified. DPNINITIALIZE_DISABLEPARAMVAL Disables parameter validation.

Return Value

Returns DPN_OK if successful. Otherwise, returns one of the following errors.

| DPNERR_ALREADYINITIALIZED | The object has already been initialized. |
|---------------------------|---|
| DPNERR_INVALIDFLAGS | The flags passed to this method are invalid. |
| DPNERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
| DPNERR_NOTALLOWED | This function is not allowed on this object. |

Remarks

Only one IDirectPlay8ThreadPool object is allowed in a process. DPNERR_ALREADYINITALIZED is returned if IDirectPlay8ThreadPool::Initialize is called on a second IDirectPlay8ThreadPool object.

If a Microsoft® DirectPlay® object has already created threads, the **IDirectPlay8ThreadPool** object cannot be initialized. If the **IDirectPlay8ThreadPool** object has not been initialized, **DPNERR_NOTALLOWED** will be returned.

IDirectPlay8ThreadPool::SetThreadCount Method

Changes the number of threads for a specified processor or all processors.

Syntax

```
HRESULT SetThreadCount(
const DWORD dwProcessorNum,
const DWORD dwNumThreads,
const DWORD dwFlags
);
```

Parameters

dwProcessorNum

[in] Specifies the processor number. Set to -1 to change the total thread count for all processors.

dwNumThreads

[in] Specifies the new thread count.

dwFlags

[in] Reserved. Must be 0.

Return Value

Returns DPN_OK if successful. Otherwise, returns one of the following errors.

DPNERR_UNINITIALIZEDThe requested object has not been initialized.DPNERR_INVALIDFLAGSThe flags passed to this method are invalid.DPNERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DPNERR_NOTALLOWEDThis function is not allowed on this object.

Remarks

If the value in the *dwNumThreads* parameter is larger than the current thread count, the new threads will be started, generating a <u>DPN_MSGID_CREATE_THREAD</u> message for each new thread before this method returns.

If the value in the *dwNumThreads* parameter is smaller than the current thread count, the excess threads will be shut down, generating a <u>DPN_MSGID_DESTROY_THREAD</u> message for each closed thread before this method returns.

If the thread count is set to 0, Microsoft® DirectPlay® will not create any threads in the application. Therefore, to make anything happen in the application, you'll need to call <u>IDirectPlay8ThreadPool::DoWork</u> regularly.

DirectPlay performs tasks differently when the thread count set to 0 than when you are using DirectPlay threads. Therefore, it is recommended that you do not switch between zero thread count mode and multithread count mode once a session has been created.

DPNERR_NOTALLOWED is returned if this method is called when there is an outstanding call to **IDirectPlay8ThreadPool::DoWork** on a thread. In this case, the thread count will not change.

DPNERR_NOTALLOWED is returned if *dwNumThreads* parameter is smaller than the current thread count and this method is called from an <u>IDirectPlay8ThreadPool</u> thread. In this case, the thread count will not change.

IDirectPlayNATHelp Interface

Applications use the methods of the **IDirectPlayNATHelp** interface to simplify traversal through the Internet Connection Sharing (ICS) features available in Microsoft® Windows® Millennium Edition (Windows Me) and Windows XP.

| <u>Close</u> | Closes and unregisters this application with any Internet gateway servers. This method must be called on any object successfully initialized with IDirectPlayNATHelp . |
|------------------------|---|
| <u>GetCaps</u> | Retrieves the capabilities of the Internet gateway server(s) and information about leased ports. |
| GetRegisteredAddresses | Returns the current public address mappings for the specified registered port group. |
| <u>Initialize</u> | Initializes an IDirectPlayNATHelp object. This method must be called before using any other functions. |
| <u>QueryAddress</u> | Determine a private alias for a given public address. |
| <u>RegisterPorts</u> | Asks for public realm port(s) that are aliases for the local port(s) on this private realm node. |
| <u>SetAlertEvent</u> | Specifies an event that will be set when maintenance needs to be performed. |

IDirectPlayNATHelp Members

Interface Information

| Inherits from | IUnknown |
|---------------------------|----------------------------|
| Header | dpnathlp.h |
| Minimum operating systems | Windows 98, Pocket PC 2002 |

IDirectPlayNATHelp::Close Method

Closes and unregisters this application with any Internet gateway servers. This method must be called on any object successfully initialized with <u>IDirectPlayNATHelp</u>.

Syntax

```
HRESULT Close(
const DWORD dwFlags
);
```

Parameters

dwFlags [in] Reserved. Must be 0.

Return Value

Returns DPNH_OK if successful, or one of the following error values.

| DPNHERR_GENERIC | An error occurred while closing. |
|------------------------|---|
| DPNHERR_INVALIDFLAGS | Invalid flags were specified. |
| DPNHERR_INVALIDOBJECT | The interface object is invalid. |
| DPNHERR_INVALIDPARAM | An invalid parameter was specified. |
| DPNHERR_NOTINITIALIZED | The object has not been initialized. |
| DPNHERR_OUTOFMEMORY | There is not enough memory to perform this operation. |
| DPNHERR_REENTRANT | The interface has been re-entered on the same thread. |

IDirectPlayNATHelp::GetCaps Method

Retrieves the capabilities of the Internet gateway server(s) and information about leased ports.

Syntax

```
HRESULT GetCaps(

PPDPNHCAPS *const pdpnhcaps,

const DWORD dwFlags

);
```

Parameters

pdpnhcaps

[in] Pointer to a <u>DPNHCAPS</u> structure to be filled with Network Address Translation (NAT) helper's capabilities. The **dwSize** member of the structure must be set.

dwFlags

[in] May be the following value.

DPNHGETCAPS_UPDATESERVERSTATUS

Automatically extend expiring leases and detect changes in the server status.

Return Value

Returns DPNH_OK if successful, or one of the following error values.

| DPNHERR_GENERIC | An error occurred while closing. |
|------------------------|---|
| DPNHERR_INVALIDFLAGS | Invalid flags were specified. |
| DPNHERR_INVALIDOBJECT | The interface object is invalid. |
| DPNHERR_INVALIDPARAM | An invalid parameter was specified. |
| DPNHERR_INVALIDPOINTER | An invalid pointer was specified. |
| DPNHERR_NOTINITIALIZED | The object has not been initialized. |
| DPNHERR_OUTOFMEMORY | There is not enough memory to perform this operation. |
| DPNHERR_REENTRANT | The interface has been re-entered on the same thread. |

Remarks

This method should be called periodically with the DPNHGETCATS_UPDATESERVERSTATUS flag set to automatically extend port leases that are about to expire. The DPNHGETCATS_UPDATESERVERSTATUS flag also causes detection of changes in the status of the servers since the last call to **IDirectPlayNATHelp::GetCaps**. If a new server becomes available, an existing server becomes unavailable, or a server's public address changes in a way that affects an existing registered port mapping, then DPNHSUCCESS_ADDRESSESCHANGED is returned instead of DPNH_OK.

If DPNHSUCCESS_ADDRESSESCHANGED is returned, the user should call <u>IDirectPlayNATHelp::GetRegisteredAddresses</u> to update port binding information.

When the DPNHGETCATS_UPDATESERVERSTATUS flag is set, this method may stall while attempts are made to communicate with the server.

This method must be called with the DPNHGETCATS_UPDATESERVERSTATUS flag set at least once before calling **IDirectPlayNATHelp::GetRegisteredAddresses** or <u>IDirectPlayNATHelp::QueryAddress</u>.

IDirectPlayNATHelp::GetRegisteredAddresses Method

Returns the current public address mappings for the specified registered port group.

Syntax

```
HRESULT GetRegisteredAddresses(
const DPNHANDLE hRegisteredPorts,
SOCKADDR *const paPublicAddresses,
DWORD *const pdwPublicAddressSize,
DWORD *const pdwAddressTypeFlags,
DWORD *const pdwLeaseTimeRemaining,
const DWORD dwFlags
```

);

Parameters

hRegisteredPorts

[in] Handle to a specific binding returned by <u>IDirectPlayNATHelp::RegisterPorts</u>.

paPublicAddresses

[out] Buffer to receive assigned public realm addresses. Set to NULL if not desired.

pdwPublicAddressSize

[in, out] Pointer to a size of *paPublicAddresses* buffer on input. On output, receives the size written to *paPublicAddresses* or the size required if the buffer is too small. Can be NULL only if *paPublicAddresses* is NULL.

pdwAddressTypeFlags

[out] Receives flags describing the address types returned. Set to NULL if not desired. The following values can be returned. DPNHANDLEADDRESSTYPE_TCP

The mappings are for TCP ports instead of User Datagram Protocol (UDP) ports.

DPNHANDLEADDRESSTYPE_FIXEDPORTS
The mappings are for ports which are the same on the Internet gateway.

DPNHANDLEADDRESSTYPE_SHAREDPORTS The mappings are for shared UDP fixed ports. DPNHANDLEADDRESSTYPE_LOCALFIREWALL The addresses are opened on a local firewall. DPNHANDLEADDRESSTYPE_GATEWAY The addresses are registered with an Internet gateway. DPNHANDLEADDRESSTYPE_GATEWAYISLOCAL The Internet gateway is local. pdwLeaseTimeRemaining [out] Receives time remaining in the port lease in milliseconds. Set to NULL if not desired. Call <u>IDirectPlayNATHelp::GetCaps</u> to

automatically extend leases.

dwFlags

[in] Can be the following value.

DPNHGETREGISTEREDADDRESSES_LOCALFIREWALLREMAPO Retrieve the public address for the local firewall only, even if mapped on remote Internet gateway.

Return Value

Returns DPNH_OK if successful, or one of the following error values.

| DPNHERR_GENERIC | An error occurred while closing. |
|----------------------------|--|
| DPNHERR_BUFFERTOOSMALL | The supplied buffer is not large enough to contain the requested data. |
| DPNHERR_INVALIDFLAGS | Invalid flags were specified. |
| DPNHERR_INVALIDOBJECT | The interface object is invalid. |
| DPNHERR_INVALIDPARAM | An invalid parameter was specified. |
| DPNHERR_NOMAPPING | The server does not have valid public interfaces. |
| DPNHERR_NOTINITIALIZED | The object has not been initialized. |
| DPNHERR_OUTOFMEMORY | There is not enough memory to perform this operation. |
| DPNHERR_REENTRANT | The interface has been re-entered on the same thread. |
| DPNHERR_SERVERUNAVAILABLE | No servers are currently present. |
| DPNHERR_UPDATESERVERSTATUS | IDirectPlayNATHelp::GetCaps has not been called with the DPNHGETCAPS_UPDATESERVERSTATUS flag set. |
| | |

IDirectPlayNATHelp::Initialize Method

Initializes an <u>IDirectPlayNATHelp</u> object. This method must be called before using any other functions.

Syntax

HRESULT Initialize(const DWORD *dwFlags*);

Parameters

dwFlags [in] Can be one of the following values. DPNHINITIALIZE_DISABLEGATEWAYSUPPORT Do not attempt to traverse the Internet gateway. DPNHINITIALIZE_DISABLELOCALFIREWALLSUPPORT Do not attempt to traverse a local firewall.

Return Value

Returns DPNH_OK if successful, or one of the following error values.

| DPNHERR_ALREADYINITIALIZED | This object has already been initialized. |
|----------------------------|---|
| DPNHERR_GENERIC | An error occurred while closing. |
| DPNHERR_INVALIDFLAGS | Invalid flags were specified. |
| DPNHERR_INVALIDOBJECT | The interface object is invalid. |
| DPNHERR_INVALIDPARAM | An invalid parameter was specified. |
| DPNHERR_OUTOFMEMORY | There is not enough memory to perform this operation. |
| DPNHERR_REENTRANT | The interface has been re-entered on the same thread |

Remarks

This method does not attempt to contact any Internet gateway

servers.

Call <u>IDirectPlayNATHelp::GetCaps</u> with the DPNHGETCAPS_UPDATESERVERSTATUS flag set to search for a server.

IDirectPlayNATHelp::QueryAddress Method

Determine a private alias for a given public address.

Syntax

```
HRESULT QueryAddress(
const SOCKADDR *const pSourceAddress,
const SOCKADDR *const pQueryAddress,
SOCKADDR *const pResponseAddress,
const int iAddressSize,
const DWORD dwFlags
```

);

Parameters

pSourceAddress

[in] Address for the network interface that is using the desired address. This can be set to INADDR_ANY in which case the best server will be used.

pQueryAddress

[in] Address to look up. Do not set this value to INADDR_ANY or INADDR_BROADCAST.

pResponseAddress

[out] Receives the private alias to the public address specified. *iAddressSize*

[in] Size of the SOCKADDR structure used for the *pSourceAddress*, *pQueryAddress*, and *pResponseAddress* buffers.

dwFlags

[in] Set to 0 to query for a User Datagram Protocol (UDP) port. Otherwise, set to one of the following values.

DPNHQUERYADRESS_TCP

Query for a TCP port.

DPNHQUERYADRESS_CACHEFOUND

Cache the address if a mapping is found.

DPNHQUERYADRESS_CACHENOTFOUND Cache the address if a mapping is not found. DPNHQUERYADRESS_CHECKFORPRIVATEBUTUNMAPPED Determine if an address is private when no specific mapping is found.

Return Value

Returns DPNH_OK if successful, or one of the following error values.

| DPNHERR_GENERIC | An error occurred while closing. |
|-----------------------------|---|
| DPNHERR_INVALIDFLAGS | Invalid flags were specified. |
| DPNHERR_INVALIDOBJECT | The interface object is invalid. |
| DPNHERR_INVALIDPARAM | An invalid parameter was specified. |
| DPNHERR_INVALIDPOINTER | An invalid pointer was specified. |
| DPNHERR_NOMAPPING | The server does not have valid public interfaces. |
| DPNHERR_NOMAPPINGBUTPRIVATE | The server indicated that no mapping was found, but it is a private address. |
| DPNHERR_NOTINITIALIZED | The object has not been initialized. |
| DPNHERR_OUTOFMEMORY | There is not enough memory to perform this operation. |
| DPNHERR_REENTRANT | The interface has been re-entered on the same thread. |
| DPNHERR_SERVERUNAVAILABLE | No servers are currently present. |
| DPNHERR_UPDATESERVERSTATUS | IDirectPlayNATHelp::GetCaps has not been called with the DPNHGETCAPS_UPDATESERVERSTATUS flag set. |

IDirectPlayNATHelp::RegisterPorts Method

Asks for public realm port(s) that are aliases for the local port(s) on this private realm node.

Syntax

```
HRESULT RegisterPorts(
const SOCKADDR *const aLocalAddress,
const DWORD dwAddressesSize,
const DWORD dwNumAddresses,
const DWORD dwLeaseTime,
DPNHANDLE *const phRegisteredPorts,
const DWORD dwFlags
```

);

Parameters

aLocalAddress

[in] Array of local address and port tuples for which remote ports are selected.

dwAddressesSize

[in] Size of entire local addresses array.

dwNumAddresses

[in] Number of SOCKADDR structures in local addresses array. *dwLeaseTime*

[in] Requested time to lease the ports, in milliseconds. If <u>IDirectPlayNATHelp::GetCaps</u> is called before this time expires, the lease will automatically be renewed.

phRegisteredPorts

[in] Place to store an identifier for this binding which can be used to query or release the binding.

dwFlags

[in] Set to 0 for User Datagram Protocol (UDP) ports or one of the following values.

DPNHREGISTERPORTS_TCP

TCP ports. DPNHREGISTERPORTS_FIXEDPORTS Asks the server to use the same port on the public interface. DPNHREGISTERPORTS_SHAREDPORTS Allow the UDP fixed port to be shared.

Return Value

Returns DPNH_OK if successful, or one of the following error values.

| DPNHERR_GENERIC | An error occurred while closing. |
|-------------------------------|--|
| DPNHERR_INVALIDFLAGS | Invalid flags were specified. |
| DPNHERR_INVALIDOBJECT | The interface object is invalid. |
| DPNHERR_INVALIDPARAM | An invalid parameter was specified. |
| DPNHERR_NOTINITIALIZED | The object has not been initialized. |
| DPNHERR_OUTOFMEMORY | There is not enough memory to perform this operation. |
| DPNHERR_PORTALREADYREGISTERED | At least one of the ports has already been registered in a different address array or order. |
| DPNHERR_REENTRANT | The interface has been re-entered on the same thread. |
| | |

IDirectPlayNATHelp::SetAlertEvent Method

Specifies an event that will be set when maintenance needs to be performed.

Syntax

```
HRESULT SetAlertEvent(
const HANDLE hEvent,
const DWORD dwFlags
);
```

Parameters

hEvent

[in] Handle to the event to signal when <u>IDirectPlayNATHelp::GetCaps</u> is to be called.

```
dwFlags
```

[in] Reserved. Must be 0.

Return Value

Returns DPNH_OK if successful, or one of the following error values.

| DPNHERR_GENERIC | An error occurred while closing. |
|------------------------|---|
| DPNHERR_INVALIDFLAGS | Invalid flags were specified. |
| DPNHERR_INVALIDOBJECT | The interface object is invalid. |
| DPNHERR_INVALIDPARAM | An invalid parameter was specified. |
| DPNHERR_NOTINITIALIZED | The object has not been initialized. |
| DPNHERR_OUTOFMEMORY | There is not enough memory to perform this operation. |
| DPNHERR_REENTRANT | The interface has been re-entered on the same thread. |

Remarks

Call IDirectPlayNATHelp::GetCaps using the

DPNHGETCAPS_UPDATESERVERSTATUS flag when the event is signalled.

This method is used in addition to the regular polling by **IDirectPlayNATHelp::GetCaps**. It allows the polling to be less frequent.

IDirectPlayVoiceClient Interface

Applications use the methods of the **IDirectPlayVoiceClient** interface to manage clients in a voice session.

IDirectPlayVoiceClient Members

| <u>Connect</u> | Connects the client to a Microsoft® DirectPlay® Voice session. |
|---------------------------|---|
| Create3DSoundBuffer | Retrieves a 3-D sound buffer for a player or group. You can use the methods of the 3-D sound buffer object to change the virtual 3-D position of incoming voice transmissions from the specified group or player. |
| Delete3DSoundBuffer | Returns exclusive control of the 3-D sound buffer object to the DirectPlay voice client object. |
| <u>Disconnect</u> | Disconnects the DirectPlay Voice client from the existing DirectPlay Voice session. |
| <u>GetCaps</u> | Retrieves the DirectPlay Voice capabilities. |
| <u>GetClientConfig</u> | Retrieves the client configuration. |
| GetCompressionTypes | Retrieves the available compression types on the system. |
| GetSessionDesc | Retrieves the session properties. |
| GetSoundDeviceConfig | Retrieves the sound device configuration of the session. |
| <u>GetTransmitTargets</u> | Retrieves the transmit targets, if any, of the voice stream from this client. |
| <u>Initialize</u> | Initializes the DirectPlayVoiceClient object by associating it with a DirectPlay object. Additionally, this method registers a message handler with the DirectPlayVoiceClient object. |
| SetClientConfig | Sets the client configuration. |
| <u>SetNotifyMask</u> | Specifies which messages are sent to the message handler. |
| <u>SetTransmitTargets</u> | Specifies which players and/or groups receive audio transmissions from the local client. |

Interface Information

Inherits fromIUnknownHeaderdvoice.hMinimum operating systemsWindows 98

 $\ensuremath{\textcircled{}^{\circ}}$ 2003 Microsoft Corporation. All rights reserved.

IDirectPlayVoiceClient::Connect Method

Connects the client to a Microsoft® DirectPlay® Voice session.

Syntax

```
HRESULT Connect(
    PDVSOUNDDEVICECONFIG pSoundDeviceConfig,
    PDVCLIENTCONFIG pdvClientConfig,
    DWORD dwFlags
);
```

Parameters

pSoundDeviceConfig[in] Pointer to a <u>DVSOUNDDEVICECONFIG</u> structure that describes the sound device configuration. *pdvClientConfig*[in] Pointer to a <u>DVCLIENTCONFIG</u> structure that describes the general configuration of the client. *dwFlags*[in] Flag. You can specify the following flag.
DVFLAGS_SYNC
The method does not return until the operation is completed.

Return Value

If the method is processed synchronously and is successful, it returns DV_OK. By default, this method is run asynchronously and returns DV_PENDING. On error, this method will return one of the following values.

```
DVERR_ALREADYPENDINGAn asynchronous call of this type is already pending.DVERR_COMPRESSIONNOTSUPPORTEDThe specified compression type is not supported on the local<br/>computer.
```

| DVERR_INCOMPATIBLEVERSION | The client connected to a voice session that is incompatible with the host. |
|---------------------------|--|
| DVERR_INVALIDDEVICE | The specified device is invalid. |
| DVERR_INVALIDFLAGS | The flags passed to this method are invalid. |
| DVERR_INVALIDOBJECT | The DirectPlay object pointer is invalid. |
| DVERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
| DVERR_INVALIDPOINTER | The pointer specified is invalid. |
| DVERR_NOTINITIALIZED | The <u>IDirectPlayVoiceClient::Initialize</u> or <u>IDirectPlayVoiceServer::Initialize</u> method must be called before calling this method. |
| DVERR_OUTOFMEMORY | There is insufficient memory to perform the requested operation. |
| DVERR_RUNSETUP | The specified audio configuration has not been tested. Call the IDirectPlayVoiceTest::CheckAudioSetup method. |
| DVERR_SENDERROR | An error occurred while sending data. |
| DVERR_SOUNDINITFAILURE | A failure was encountered initializing the sound card. |
| DVERR_TIMEOUT | The operation could not be performed in the specified time. |
| DVERR_TRANSPORTNOPLAYER | The specified transport is connected/hosting but no local player exists. |
| DVERR_TRANSPORTNOSESSION | The specified transport is valid but is not connected/hosting. |
| DVERR_CONNECTED | The DirectPlay Voice object is connected. |
| DVERR_NOVOICESESSION | The session specified is not a voice session. |

Remarks

If you call this method asynchronously and it returns DVERR_PENDING, you will receive a DVMSGID_CONNECTRESULT message with the result of the connection attempt. This message can arrive before or after the method returns DVERR_PENDING. If you call this method

asynchronously, you will not receive a

DVMSGID_CONNECTRESULT message.

You must test the sound devices selected for playback and capture by invoking the setup wizard before connecting the client to the DirectPlay Voice session. On application startup, check the audio configuration by using **IDirectPlayVoiceTest::CheckAudioSetup**. If this method returns DVERR_RUNSETUP, the sound configuration specified has not been tested. The setup wizard needs to be run only once for any configuration.

If you specify a buffer that is not the right format, the method will return DVERR_INVALIDBUFFER.

If the buffer or a portion of the buffer is locked when DirectPlay Voice attempts to write to it, the method will return DVERR_INVALIDBUFFER, and DirectPlay Voice will disconnect from the session. You will also receive a DVMSGID_SESSIONLOST message. The **hResult** member of the associated structure will be set to DVERR_LOCKEDBUFFER. Subsequent method calls will return a DVERR_NOTCONNECTED error code.

If full duplex operation is not supported, DirectPlay Voice falls back to half duplex (listen only) mode. To determine if you are in halfduplex mode, call <u>IDirectPlayVoiceClient::GetSoundDeviceConfig</u> after you have completed the connection. If you are in half-duplex mode, the **DVSOUNDDEVICECONFIG** structure's **dwFlags** member will have the DVSOUNDCONFIG_HALFDUPLEX flag set.

Regardless of how the interfaces are obtained, the DirectPlayVoiceClient object maintains a reference, through a call to AddRef, to the IDirectSound and IDirectSoundCapture interfaces it uses until IDirectPlayVoiceClient::Disconnect is called. When IDirectPlayVoiceClient::Disconnect is called, the DirectPlayVoiceClient object calls Release on both interfaces.

Any calls to **IDirectPlayVoiceClient::Connect** while a connection is pending return DVERR_ALREADYPENDING. Additionally, only one connection can be pending at a time.

A transport session must be started on the specified DirectPlay

object before calling this method. A successful call to IDirectPlayVoiceClient::Initialize must be made before calling the Connect method.

IDirectPlayVoiceClient::Create3DSoundBuffer Method

Retrieves a 3-D sound buffer for a player or group. You can use the methods of the 3-D sound buffer object to change the virtual 3-D position of incoming voice transmissions from the specified group or player.

Syntax

```
HRESULT Create3DSoundBuffer(
DVID dvID,
LPDIRECTSOUNDBUFFER lpdsSourceBuffer,
DWORD dwPriority,
DWORD dwFlags,
LPDIRECTSOUND3DBUFFER *lpUserBuffer
);
```

Parameters

dvID

[in] Variable of type **DVID** that specifies the identification of the player or group that the user wants to reserve a buffer for. You can also specify DVID_REMAINING to create a 3-D user buffer for all players or groups that do not have a user buffer. If DVID_REMAINING is specified, the *lpdsSourceBuffer* parameter must be set to NULL and the *dwPriority* and *dwFlags* parameters must be set to 0.

IpdsSourceBuffer

[in] Pointer to an **IDirectSoundBuffer** interface, which is used to create the Microsoft® DirectPlay® Voice main buffer. This can be either NULL or a user-created Microsoft DirectSound® buffer. If this member is set to NULL, then DirectPlay Voice creates a buffer for you.

dwPriority

[in] Direct pass-through. This value is passed in the *dwPriority* parameter when the call to **IDirectSoundBuffer::Play** is made. For more information, see **IDirectSoundBuffer8::Play**. This

parameter must be 0 if *lpdsSourceBuffer* is NULL. *dwFlags* [in] Direct pass-through. This value is passed to the *dwFlags*parameter when the call to **IDirectSoundBuffer::Play** is made. For more information, see **IDirectSoundBuffer8::Play**. This parameter must be 0 if *lpdsSourceBuffer* is NULL.

IpUserBuffer

[out] Pointer to memory where the reserved buffer is placed.

Return Value

Returns DV_OK if successful, or one of the following error values.

| DVERR_ALREADYBUFFERED | There is already a user buffer for the specified ID. |
|-----------------------|--|
| DVERR_INVALIDOBJECT | The DirectPlay object pointer is invalid. |
| DVERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
| DVERR_INVALIDPOINTER | The pointer specified is invalid. |
| DVERR_NOTALLOWED | The object does not have the permission to perform this operation. |
| DVERR_NOTCONNECTED | The DirectPlay Voice object is not connected. |
| DVERR_NOTINITIALIZED | The <u>IDirectPlayVoiceClient::Initialize</u> or <u>IDirectPlayVoiceServer::Initialize</u> method must be called before calling this method. |
| DVERR_OUTOFMEMORY | There is insufficient memory to perform the requested operation. |
| DVERR_SESSIONLOST | The transport has lost the connection to the session. |

Remarks

If the DirectPlay voice session is a mixing server session, this method fails and returns DVERR_NOTALLOWED.

Although you can access all the member functions of the 3-D sound buffer object, because the DirectPlay voice client uses the buffer to stream incoming audio, do not use the **Lock**, **UnLock**, or **Play** methods of the DirectSound3DBuffer object.

If the user specifies a buffer, DirectPlay uses that buffer for the player's or group's buffer. User-created buffers have the following

restrictions.

- The buffer must be 22 kilohertz, 16-bit, Mono format.
- The buffer must be at least 1 second in length.
- The buffer must have been created with the DSBCAPS_GETCURRENTPOSITION2 and DSBCAPS_CTRL3D flags.
- The buffer must not be a primary buffer.
- The buffer must not be playing when it is passed to DirectPlay.

If the buffer is not the right format, the method will return DVERR_INVALIDBUFFER.

The buffer must not be locked when you pass it to DirectPlay. When the buffer for the individual user is no longer required or when a player leaves the voice session, it is important to call **IDirectPlayVoiceClient::Delete3DSoundBuffer** to free up resources.

If the buffer or a portion of the buffer is locked when DirectPlay Voice attempts to write to it, the method will return DVERR_INVALIDBUFFER. If you lock the buffer after the method has returned, you will receive a <u>DVMSGID_SESSIONLOST</u> message. The **hResult** member of the associated structure will be set to DVERR_LOCKEDBUFFER. Subsequent method calls will return a <u>DVERR_NOTCONNECTED</u> error code.

IDirectPlayVoiceClient::Delete3DSoundBuffer Method

Returns exclusive control of the 3-D sound buffer object to the Microsoft® DirectPlay® voice client object.

Syntax

```
HRESULT Delete3DSoundBuffer(
    DVID dvID,
    LPDIRECTSOUND3DBUFFER *lpUserBuffer
);
```

Parameters

dvID

[in] DVID of the player or group that the user wants to delete a buffer for.

lpUserBuffer

[in] Pointer to the user buffer to delete. This must be a user buffer obtained through the

IDirectPlayVoiceClient::Create3DSoundBuffer method.

Return Value

Returns DV_OK if successful, or one of the following error values.

| DVERR_ALREADYBUFFERED | There is already a user buffer for the specified ID. |
|-----------------------|--|
| DVERR_INVALIDOBJECT | The DirectPlay object pointer is invalid. |
| DVERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
| DVERR_INVALIDPOINTER | The pointer specified is invalid. |
| DVERR_NOTALLOWED | The object does not have the permission to perform this operation. |
| DVERR_NOTBUFFERED | There is no user buffer for the specified ID. |
| DVERR_NOTCONNECTED | The DirectPlay Voice object is not connected. |
| DVERR_NOTINITIALIZED | The <u>IDirectPlayVoiceClient::Initialize</u> or <u>IDirectPlayVoiceServer::Initialize</u> method must be called before calling this method. |
| DVERR_SESSIONLOST | The transport has lost the connection to the session. |

Remarks

If the DirectPlay Voice session is a mixing server session, this method fails and returns DVERR_NOTALLOWED.

IDirectPlayVoiceClient::Disconnect Method

Disconnects the Microsoft® DirectPlay® Voice client from the existing DirectPlay Voice session.

Syntax

HRESULT Disconnect(DWORD dwFlags);

Parameters

dwFlags

[in] Flag. You can specify the following flag. DVFLAGS_SYNC Do not return until the operation is completed.

Return Value

Returns DV_OK if successful, or one of the following error values.

| DVERR_ALREADYPENDING | An asynchronous call of this type is already pending. |
|-----------------------|--|
| DVERR_CONNECTABORTING | The connection is being disconnected. |
| DVERR_INVALIDFLAGS | The flags passed to this method are invalid. |
| DVERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
| DVERR_NOTCONNECTED | The DirectPlay Voice object is not connected. |
| DVERR_NOTINITIALIZED | The <u>IDirectPlayVoiceClient::Initialize</u> or <u>IDirectPlayVoiceServer::Initialize</u> method must be called before calling this method. |
| DVERR_PENDING | Not an error, this return indicates that an asynchronous operation has reached the point where it is successfully queued. |
| DVERR_SESSIONLOST | The transport has lost the connection to the session. |
| DVERR_TIMEOUT | The operation could not be performed in the specified time. |

Remarks

On calling this method, all recording and playback is stopped. If a connection is being processed, it is canceled by this call.

Unless the DVFLAGS_SYNC is specified, calling this method immediately returns a DVERR_PENDING error value and proceeds to process the disconnection request in the background. The status of the disconnection is not known until the DirectPlay Voice client generates a <u>DVMSGID_DISCONNECTRESULT</u> message that contains the disconnection result. Only one disconnection can be pending at a time. If you call **IDirectPlayVoiceClient::Disconnect** while a disconnect is pending, DirectPlay will return a DVERR_ALREADYPENDING error value.

If this method is called synchronously by setting the DVFLAGS_SYNC flag, the method does not return until the **IDirectPlayVoiceClient::Disconnect** method completes. The result of the disconnection is the return value from this method. No **DVMSGID_DISCONNECTRESULT** message is generated.

IDirectPlayVoiceClient::GetCaps Method

Retrieves the Microsoft® DirectPlay® Voice capabilities.

Syntax

```
HRESULT GetCaps(
PDVCAPS pCaps
);
```

Parameters

pCaps

[out] Pointer to the <u>DVCAPS</u> structure that contains the capabilities of the DirectPlayVoiceClient object.

Return Value

Returns DV_OK if successful, or one of the following error values.

DVERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DVERR_INVALIDPOINTERThe pointer specified is invalid.

IDirectPlayVoiceClient::GetClientConfig Method

Retrieves the client configuration.

Syntax

```
HRESULT GetClientConfig(
    PDVCLIENTCONFIG pClientConfig
);
```

Parameters

```
pClientConfig
```

[out] Pointer to a <u>DVCLIENTCONFIG</u> structure that contains the configuration of the local client.

Return Value

Returns DV_OK if successful, or one of the following error values.

| DVERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
|----------------------|--|
| DVERR_INVALIDPOINTER | The pointer specified is invalid. |
| DVERR_NOTCONNECTED | The Microsoft® DirectPlay® Voice object is not connected. |
| DVERR_NOTINITIALIZED | The IDirectPlayVoiceClient::Initialize or IDirectPlayVoiceServer::Initialize method must be called before calling this method. |
| DVERR_SESSIONLOST | The transport has lost the connection to the session. |

Remarks

Before calling this method, you must set the **DVCLIENTCONFIG** structure's **dwSize** member.

You can call this method only after a connection is successfully established with a DirectPlay Voice session.

 $\ensuremath{\textcircled{\sc 0}}$ 2003 Microsoft Corporation. All rights reserved.
IDirectPlayVoiceClient::GetCompressionTypes Method

Retrieves the available compression types on the system.

Syntax

```
HRESULT GetCompressionTypes(

PVOID pData,

PDWORD pdwDataSize,

PDWORD pdwNumElements,

DWORD dwFlags

);
```

Parameters

pData

[out] Pointer to buffer that receives an array of

DVCOMPRESSIONINFO structures, one structure for every compression type supported through this object.

pdwDataSize

[in] Pointer to a **DWORD** that contains the size of the buffer, in bytes, passed in the *pData* parameter.

pdwNumElements

[out] Pointer to a **DWORD** where the method writes the number of elements returned in the array of **DVCOMPRESSIONINFO** structures. This contains the number of structures only if the buffer specified in the *pData* is large enough to hold the information.

dwFlags

[in] Reserved. Must be 0.

Return Value

Returns DP_OK if successful, or one of the following error values.

DVERR_BUFFERTOOSMALL The supplied buffer is not large enough to contain the requested data.

Remarks

If the buffer passed is not large enough to store the list of compression types, the method returns DVERR_BUFFERTOOSMALL and the *pdwDataSize* parameter is set to the minimum required size.

IDirectPlayVoiceClient::GetSessionDesc Method

Retrieves the session properties.

Syntax

```
HRESULT GetSessionDesc(
    PDVSESSIONDESC pvSessionDesc
);
```

Parameters

```
pvSessionDesc
[out] Pointer to a <u>DVSESSIONDESC</u> structure to receive the session description.
```

Return Value

Returns DV_OK if successful, or one of the following error values.

| DVERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
|----------------------|--|
| DVERR_INVALIDPOINTER | The pointer specified is invalid. |
| DVERR_NOTCONNECTED | The Microsoft® DirectPlay® Voice object is not connected. |
| DVERR_NOTINITIALIZED | The <u>IDirectPlayVoiceClient::Initialize</u> or <u>IDirectPlayVoiceServer::Initialize</u> method must be called before calling this method. |
| DVERR SESSIONLOST | The transport has lost the connection to the session. |

Remarks

Before calling this method, make sure to set the **DVSESSIONDESC** structure's **dwSize** member.

This method may be called only after a connection is successfully established with a DirectPlay Voice session.

 $\ensuremath{\textcircled{\sc 0}}$ 2003 Microsoft Corporation. All rights reserved.

IDirectPlayVoiceClient::GetSoundDeviceConfig Method

Retrieves the sound device configuration of the session.

Syntax

```
HRESULT GetSoundDeviceConfig(
    PDVSOUNDDEVICECONFIG pSoundDeviceConfig,
    PDWORD pdwSize
);
```

Parameters

pSoundDeviceConfig

[out] Pointer to a <u>DVSOUNDDEVICECONFIG</u> structure that is filled with the configuration of the sound device.

pdwSize

[in, out] Pointer to a **DWORD** that specifies the size of the buffer in *pSoundDeviceConfig* parameter. If the buffer is too small, the method returns DVERR_BUFFERTOOSMALL and this parameter contains the size of the required buffer.

Return Value

Returns DV_OK if successful, or one of the following error values.

| DVERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
|----------------------|--|
| DVERR_INVALIDPOINTER | The pointer specified is invalid. |
| DVERR_NOTCONNECTED | The Microsoft® DirectPlay® Voice object is not connected. |
| DVERR_NOTINITIALIZED | The <u>IDirectPlayVoiceClient::Initialize</u> or <u>IDirectPlayVoiceServer::Initialize</u> method must be called before calling this method. |
| DVERR_SESSIONLOST | The transport has lost the connection to the session. |

Remarks

You can call this method only after a connection is successfully

established with a DirectPlay Voice session.

IDirectPlayVoiceClient::GetTransmitTargets Method

Retrieves the transmit targets, if any, of the voice stream from this client.

Syntax

HRESULT GetTransmitTargets(PDVID pdvIDTargets, PDWORD pdwNumTargets, DWORD dwFlags);

Parameters

pdvIDTargets

[out] Member to fill with an array of DVIDs that specify the targets that were set by the

IDirectPlayVoiceClient::SetTransmitTargets or

<u>IDirectPlayVoiceServer::SetTransmitTargets</u> method. You can retrieve the number of targets by specifying NULL for this parameter.

pdwNumTargets

[in, out] Number of DVIDs in the *pdvIDTargets* array. If the call is successful, when the method return, this parameter will be set to the number of elements in the *pdvIDTargets* array. If the array is too small, the method returns DVERR_BUFFERTOOSMALL, and *pdwNumTargets* will be set to the required number of elements. If *pdvIDTargets* is NULL, this must be 0.

dwFlags

[in] Reserved. Must be 0.

Return Value

Returns DV_OK if successful, or one of the following error values.

DVERR_BUFFERTOOSMALL The supplied buffer is not large enough to contain the requested data.

| | The flags passed to this method are invalid |
|----------------------|--|
| DVERK_INVALIDELAGS | The hays passed to this method are invalid. |
| DVERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
| DVERR_INVALIDPOINTER | The pointer specified is invalid. |
| DVERR_NOTALLOWED | The object does not have the permission to perform this operation. |
| DVERR_NOTCONNECTED | The Microsoft® DirectPlay® Voice object is not connected. |
| DVERR_NOTINITIALIZED | The <u>IDirectPlayVoiceClient::Initialize</u> or <u>IDirectPlayVoiceServer::Initialize</u> method must be called before calling this method. |

Remarks

The value returned in the *pdvIDTargets* parameter can be player or group DVIDs or the DVID_ALLPLAYERS constant.

If the buffer specified in *pdvIDTargets* is not large enough to store the list of targets, this method returns DVERR_BUFFERTOOSMALL and *pdwNumTargets* is set to the required number of elements.

If there is no target specified, *pdwNumTargets* is set to 0 and the return value is DV_OK.

IDirectPlayVoiceClient::Initialize Method

Initializes the DirectPlayVoiceClient object by associating it with a Microsoft® DirectPlay® object. Additionally, this method registers a message handler with the DirectPlayVoiceClient object.

Syntax

```
HRESULT Initialize(

LPUNKNOWN pVoid,

PDVMESSAGEHANDLER pMessageHandler,

PVOID pUserContext,

PDWORD pdwMessageMask,

DWORD dwMessageMaskElements

);
```

Parameters

pVoid

[in] Pointer to the **IUnknown** interface for the DirectPlay object that this DirectPlayVoiceClient object should use.

pMessageHandler

[in] User-defined callback function that is called when there is a DirectPlayVoiceClient message to be processed. Threads within the DirectPlayVoiceClient object call the callback function, so it will not be called in the context of your process's main thread.

pUserContext

[in] Pointer to an application-defined structure that is passed to the callback function each time the function is called.

pdwMessageMask

[in] Array of **DWORD** values that contains the message identifiers that you want DirectPlay Voice to send to your callback function. If a message identifier is not specified in this array, it is not sent. Each message identifier should appear only once in the array and only valid message identifiers are allowed. For example, <u>DVMSGID_CONNECTRESULT</u> is not valid for the server interface, but is for the client interface. To enable all messages, specify NULL for this value.

dwMessageMaskElements

[in] **DWORD** value specifying the number of elements in the *pdwMessageMask* parameter. If *pdwMessageMask* is NULL, this must be 0.

Return Value

Returns DV_OK if successful, or one of the following error values.

| DVERR_ALREADYINITIALIZED | The object has already been initialized. |
|--------------------------|--|
| DVERR_GENERIC | An undefined error condition occurred. |
| DVERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
| DVERR_INVALIDPOINTER | The pointer specified is invalid. |
| DVERR_NOCALLBACK | This operation cannot be performed because no callback function was specified. |
| DVERR_TRANSPORTNOTINIT | The specified transport is not yet initialized. |

Remarks

This method must be called successfully before <u>IDirectPlayVoiceClient::Connect</u> method is called.

You can call <u>IDirectPlayVoiceClient::SetNotifyMask</u> to change the notify mask during the course of the voice session.

IDirectPlayVoiceClient::SetClientConfig Method

Sets the client configuration.

Syntax

```
HRESULT SetClientConfig(
    PDVCLIENTCONFIG pClientConfig
);
```

Parameters

```
pClientConfig
```

[in] Pointer to a <u>DVCLIENTCONFIG</u> structure that contains the configuration description to set.

Return Value

Returns DV_OK if successful, or one of the following error values.

| DVERR_INVALIDFLAGS | The flags passed to this method are invalid. |
|----------------------|--|
| DVERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
| DVERR_INVALIDPOINTER | The pointer specified is invalid. |
| DVERR_NOTCONNECTED | The Microsoft® DirectPlay® Voice object is not connected. |
| DVERR_NOTINITIALIZED | The <u>IDirectPlayVoiceClient::Initialize</u> or <u>IDirectPlayVoiceServer::Initialize</u> method must be called before calling this method. |
| DVERR_SESSIONLOST | The transport has lost the connection to the session. |

Remarks

You can call this method only after a connection is successfully established with a DirectPlay Voice session.

Calling this method sets all the parameters in the **DVCLIENTCONFIG** structure. Therefore, to leave a setting

unmodified, you must retrieve the current configuration with <u>IDirectPlayVoiceClient::GetClientConfig</u>. Then modify the parameters to change and call **IDirectPlayVoiceClient::SetClientConfig**.

If the session is running in half duplex mode, the members of the **DVCLIENTCONFIG** structure related to recording are ignored.

IDirectPlayVoiceClient::SetNotifyMask Method

Specifies which messages are sent to the message handler.

Syntax

HRESULT SetNotifyMask(PDWORD pdwMessageMask, DWORD dwMessageMaskElements);

Parameters

pdwMessageMask

[in] Pointer to an array of **DWORD** values containing the message identifiers that you want Microsoft® DirectPlay® Voice to send to your callback function. If a message identifier is not specified in this array, it is not sent. Each message identifier should appear only once in the array, and only valid message identifiers are allowed. For example,

DVMSGID_CONNECTRESULT is not valid for the server interface, but is for the client interface. To enable all messages, set this value to NULL.

dwMessageMaskElements

[in] **DWORD** value specifying the number of elements in the *pdwMessageMask* parameter. If *pdwMessageMask* is NULL, this parameter must be 0.

Return Value

Returns DV_OK if successful, or one of the following error values.

| DVERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
|----------------------|--|
| DVERR_INVALIDPOINTER | The pointer specified is invalid. |
| DVERR_NOCALLBACK | This operation cannot be performed because no callback function was specified. |
| DVERR_NOTINITIALIZED | The IDirectPlayVoiceClient::Initialize or IDirectPlayVoiceServer::Initialize method must be called before calling this method. |

IDirectPlayVoiceClient::SetTransmitTargets Method

Specifies which players and/or groups receive audio transmissions from the local client.

Syntax

```
HRESULT SetTransmitTargets(

PDVID pdvIDTargets,

DWORD dwNumTargets,

DWORD dwFlags

);
```

Parameters

pdvIDTargets

[in] Pointer to an array of DVIDs that specify your targets. To target all players in the session, use a single element array, with the value of that element set to DVID_ALLPLAYERS. To target no players, set this parameter to NULL.

dwNumTargets

[in] Number of DVIDs in the array. This value cannot exceed 64. If *pdvIDTargets* is NULL, this parameter must be set to 0.

dwFlags

[in] Reserved. Must be 0.

Return Value

Returns DV_OK if successful, or one of the following error values.

| DVERR_INVALIDFLAGS | The flags passed to this method are invalid. |
|----------------------|--|
| DVERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
| DVERR_INVALIDPOINTER | The pointer specified is invalid. |
| DVERR_INVALIDTARGET | The specified target is not a valid player ID or group ID for this voice session. |
| DVERR_NOTINITIALIZED | The <u>IDirectPlayVoiceClient::Initialize</u> or <u>IDirectPlayVoiceServer::Initialize</u> method must be called before calling this method. |

Remarks

The number of individual targets that you can transmit to is limited to 64. If you exceed this value, the method will fail, and return DVERR_NOTALLOWED. However, you can transmit to more than 64 players. To do this, form the players into groups, and then use the groups as your target.

The *pdvIDTargets* parameter specifies an array of player and/or group DVIDs. There must be no duplicate targets in this parameter, and all entries must be valid DVIDs. If a target contains a player as its individual DVID and through a group that the target belongs to, Microsoft® DirectPlay® Voice ensures duplicate speech packets are not sent to the player.

If the session was created with the DVSESSION_SERVERCONTROLTARGET flag, only the server can set the targets for this local client. A call to this method returns DVERR_NOTALLOWED.

IDirectPlayVoiceServer Interface

Applications use the methods of the **IDirectPlayVoiceServer** interface to manage the host of the voice session.

IDirectPlayVoiceServer Members

| <u>GetCaps</u> | Retrieves the capabilities of the Microsoft® DirectPlay® Voice server for this system. |
|---------------------------|--|
| GetCompressionTypes | Retrieves available compression types for the system. |
| GetSessionDesc | Retrieves the DirectPlay Voice session settings. |
| <u>GetTransmitTargets</u> | Retrieves the transmit targets, if any, of the voice stream for a player in a session. |
| <u>Initialize</u> | Initializes the DirectPlayVoiceServer object by associating it with a DirectPlay object. Additionally, this method registers a message handler with this interface. |
| <u>SetNotifyMask</u> | Specifies which messages are sent to the message handler. |
| SetSessionDesc | Sets the session settings. |
| <u>SetTransmitTargets</u> | Controls the transmission of audio from the client to the specified members of the session. |
| <u>StartSession</u> | Starts an initialized DirectPlay Voice session within a running DirectPlay transport session. This method must be successfully called before the clients can complete a connection to the voice session. |
| StopSession | Stops the DirectPlay Voice session. |

Interface Information

| Inherits from | <u>IUnknown</u> |
|---------------------------|-----------------|
| Header | dvoice.h |
| Minimum operating systems | Windows 98 |

IDirectPlayVoiceServer::GetCaps Method

Retrieves the capabilities of the Microsoft® DirectPlay® Voice server for this system.

Syntax

```
HRESULT GetCaps(
PDVCAPS pDVCaps
);
```

Parameters

pDVCaps

[out] Pointer to the <u>DVCAPS</u> structure that contains the capabilities of the DirectPlayVoiceServer object.

Return Value

Returns DV_OK if successful, or one of the following error values.

DVERR_INVALIDOBJECTThe DirectPlay object pointer is invalid.DVERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DVERR_INVALIDPOINTERThe pointer specified is invalid.

IDirectPlayVoiceServer::GetCompressionTypes Method

Retrieves available compression types for the system.

Syntax

```
HRESULT GetCompressionTypes(

PVOID pData,

PDWORD pdwDataSize,

PDWORD pdwNumElements,

DWORD dwFlags

);
```

Parameters

pData

[out] Pointer to the buffer that receives an array of <u>DVCOMPRESSIONINFO</u> structures that describe the compression types supported by this object.

pdwDataSize

[in] Pointer to a **DWORD** value that contains the size of the buffer, in bytes, passed in the *pData* parameter.

pdwNumElements

[out] Pointer to a **DWORD** value where the method writes the number of elements returned in the array of

DVCOMPRESSIONINFO structures.

```
dwFlags
```

[in] Reserved. Must be 0.

Return Value

Returns DV_OK if successful, or one of the following error values.

| DVERR_BUFFERTOOSMALL | The supplied buffer is not large enough to contain the requested data. |
|----------------------|--|
| DVERR_INVALIDFLAGS | The flags passed to this method are invalid. |
| DVERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
| DVERR_INVALIDPOINTER | The pointer specified is invalid. |

Remarks

If the buffer is not large enough to store the list of compression types, the method returns DVERR_BUFFERTOOSMALL and the *pdwDataSize* parameter is set to the minimum required size.

IDirectPlayVoiceServer::GetSessionDesc Method

Retrieves the Microsoft® DirectPlay® Voice session settings.

Syntax

```
HRESULT GetSessionDesc(
    PDVSESSIONDESC pvSessionDesc
);
```

Parameters

```
pvSessionDesc
[out] Pointer to a <u>DVSESSIONDESC</u> structure to receive the session description.
```

Return Value

Returns DV_OK if successful, or one of the following error values.

| DVERR_INVALIDOBJECT | The DirectPlay object pointer is invalid. |
|----------------------|--|
| DVERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
| DVERR_INVALIDPOINTER | The pointer specified is invalid. |
| DVERR_NOTHOSTING | The object is not the host of the session. |
| DVERR_NOTINITIALIZED | The <u>IDirectPlayVoiceClient::Initialize</u> or <u>IDirectPlayVoiceServer::Initialize</u> method must be called before calling this method. |
| DVERR_SESSIONLOST | The transport has lost the connection to the session. |

Remarks

Before calling this method, make sure to set the **DVSESSIONDESC** structure's **dwSize** member.

A successful call to <u>IDirectPlayVoiceServer::StartSession</u> must be made before this method can be called.

 $\ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.

IDirectPlayVoiceServer::GetTransmitTargets Method

Retrieves the transmit targets, if any, of the voice stream for a player in a session.

Syntax

```
HRESULT GetTransmitTargets(
DVID dvSource,
PDVID pdvIDTargets,
PDWORD pdwNumTargets,
DWORD dwFlags
);
```

Parameters

dvSource

[in] DVID of the user or group whose target is returned.

pdvIDTargets

[out] Array of DVIDs that will contain the current targets of the player or group that were set by the

IDirectPlayVoiceServer::SetTransmitTargets method. You can retrieve the number of targets by specifying NULL for this parameter.

pdwNumTargets

[in, out] Number of DVIDs in the *pdvIDTargets* array. If the call is successful, when the method return, this parameter will be set to the number of elements in the *pdvIDTargets* array. If the array is too small, the method returns DVERR_BUFFERTOOSMALL, and *pdwNumTargets* will be set to the required number of elements. If *pdvIDTargets* is NULL, this must be 0.

dwFlags

[in] Reserved. Must be 0.

Return Value
Returns DV_OK if successful, or one of the following error values.

| DVERR_BUFFERTOOSMALL | The supplied buffer is not large enough to contain the requested data. |
|----------------------|--|
| DVERR_INVALIDFLAGS | The flags passed to this method are invalid. |
| DVERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
| DVERR_INVALIDPOINTER | The pointer specified is invalid. |
| DVERR_NOTALLOWED | The object does not have the permission to perform this operation. |
| DVERR_NOTCONNECTED | The Microsoft® DirectPlay® Voice object is not connected. |
| DVERR_NOTINITIALIZED | The <u>IDirectPlayVoiceClient::Initialize</u> or <u>IDirectPlayVoiceServer::Initialize</u> method must be called before calling this method. |

Remarks

This method can be used only if the DVSESSION_SERVERCONTROLTARGET flag is specified on creation of the DirectPlay Voice session. If the flag is not specified, this method returns DVERR_NOTALLOWED.

IDirectPlayVoiceServer::Initialize Method

Initializes the DirectPlayVoiceServer object by associating it with a Microsoft® DirectPlay® object. Additionally, this method registers a message handler with this interface.

Syntax

```
HRESULT Initialize(

LPUNKNOWN lpVoid,

PDVMESSAGEHANDLER pMessageHandler,

PVOID pUserContext,

LPDWORD lpdwMessageMask,

DWORD dwMessageMaskElements

);
```

Parameters

lpVoid

[in] Pointer to the **IUnknown** interface for the DirectPlay object that this DirectPlayVoiceServer object should use.

pMessageHandler

[in] User-defined callback function that is called when there is a DirectPlayVoiceClient message to process. A thread within the DirectPlayVoiceClient object calls the callback function, so it is not called in the context of your process's main thread.

pUserContext

[in] Pointer to an application-defined structure that is passed to the callback function each time the method is called.

lpdwMessageMask

[in] Array of **DWORD** values that contain the message identifiers that you want DirectPlay Voice to send to your callback function. If a message identifier is not specified in this array, it is not sent. Each message identifier should appear only once in the array, and only valid message identifiers are allowed. For example, <u>DVMSGID_CONNECTRESULT</u> is not valid for the server

interface but is for the client interface. To enable all messages, specify NULL for this value.

dwMessageMaskElements

[in] **DWORD** value that specifies the number of elements in the *lpdwMessageMask* parameter. If *lpdwMessageMask* is NULL, this must be 0.

Return Value

Returns DV_OK if successful, or one of the following error values.

| DVERR_ALREADYINITIALIZED | The object has already been initialized. |
|--------------------------|--|
| DVERR_GENERIC | An undefined error condition occurred. |
| DVERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
| DVERR_INVALIDPOINTER | The pointer specified is invalid. |
| DVERR_NOCALLBACK | This operation cannot be performed because no callback function was specified. |
| DVERR_TRANSPORTNOTINIT | The specified transport is not yet initialized. |

Remarks

You can call <u>IDirectPlayVoiceServer::SetNotifyMask</u> to change the notify mask during the course of the voice session.

IDirectPlayVoiceServer::SetNotifyMask Method

Specifies which messages are sent to the message handler.

Syntax

HRESULT SetNotifyMask(PDWORD pdwMessageMask, DWORD dwMessageMaskElements);

Parameters

pdwMessageMask

[in] Pointer to an array of **DWORD** values that contain the message identifiers that you want Microsoft® DirectPlay® Voice to send to your callback function. If a message identifier is not specified in this array, it is not sent. Each message identifier should appear only once in the array, and only valid message identifiers are allowed. For example,

DVMSGID_CONNECTRESULT is not valid for the server interface but is for the client interface. To enable all messages, specify NULL for this value.

dwMessageMaskElements

[in] **DWORD** value that specifies the number of elements in the *pdwMessageMask* parameter. If *pdwMessageMask* is NULL, this must be 0.

Return Value

Returns DV_OK if successful, or one of the following error values.

| DVERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
|----------------------|--|
| DVERR_INVALIDPOINTER | The pointer specified is invalid. |
| DVERR_NOCALLBACK | This operation cannot be performed because no callback function was specified. |
| DVERR_NOTINITIALIZED | The IDirectPlayVoiceClient::Initialize or IDirectPlayVoiceServer::Initialize method must be called before calling this method. |

IDirectPlayVoiceServer::SetSessionDesc Method

Sets the session settings.

Syntax

```
HRESULT SetSessionDesc(
    PDVSESSIONDESC pSessionDesc
);
```

Parameters

pSessionDesc

[in] Pointer to a <u>DVSESSIONDESC</u> structure that contains the session description.

Return Value

Returns DV_OK if successful, or one of the following error values.

| DVERR_INVALIDOBJECT | The Microsoft® DirectPlay® object pointer is invalid. |
|----------------------|--|
| DVERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
| DVERR_INVALIDPOINTER | The pointer specified is invalid. |
| DVERR_NOTHOSTING | The object is not the host of the session. |
| DVERR_NOTINITIALIZED | The <u>IDirectPlayVoiceClient::Initialize</u> or <u>IDirectPlayVoiceServer::Initialize</u> method must be called before calling this method. |
| DVERR_SESSIONLOST | The transport has lost the connection to the session. |

Remarks

After the DirectPlay voice session has started, only some of the session properties of the **DVSESSIONDESC** structure can be changed. For more information, see **DVSESSIONDESC**.

IDirectPlayVoiceServer::SetTransmitTargets Method

Controls the transmission of audio from the client to the specified members of the session.

Syntax

```
HRESULT SetTransmitTargets(
DVID dvSource,
PDVID pdvIDTargets,
DWORD dwNumTargets,
DWORD dwFlags
);
```

Parameters

dvSource

[in] DVID of the user whose targets are set.

pdvIDTargets

[in] Pointer to an array of DVIDs that specify your targets. To target all players in the session, use a single element array with the value of that element set to DVID_ALLPLAYERS. To target no players, set this parameter to NULL.

dwNumTargets

[in] Number of DVIDs in the array. This value cannot exceed 64. If *pdvIDTargets* is NULL, this must be 0.

```
dwFlags
```

[in] Reserved. Must be 0.

Return Value

Returns DV_OK if successful, or one of the following error values.

DVERR_INVALIDFLAGSThe flags passed to this method are invalid.DVERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DVERR_INVALIDPOINTERThe pointer specified is invalid.

Remarks

The number of individual targets that you can transmit to is limited to 64. If you exceed this value, the method will fail, and return DVERR_NOTALLOWED. However, you can transmit to more than 64 players. To do so, form the players into groups, and then use the group as your target.

There must be no duplicate targets in this parameter, and all entries must be valid DVIDs. If a target contains a player as its individual DVID and through a group that the target belongs to, Microsoft® DirectPlay® Voice ensures duplicate speech packets are not sent to the player.

This method can be used only if the DVSESSION_SERVERCONTROLTARGET flag is specified on creation of the DirectPlay Voice session. If the flag is not specified, this method returns DVERR_NOTALLOWED.

 $\ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.

IDirectPlayVoiceServer::StartSession Method

Starts an initialized Microsoft® DirectPlay® Voice session within a running DirectPlay transport session. This method must be successfully called before the clients can complete a connection to the voice session.

Syntax

```
HRESULT StartSession(
PDVSESSIONDESC pSessionDesc,
DWORD dwFlags
);
```

Parameters

pSessionDesc
[in] Pointer to a DVSESSIONDESC structure that contains the
 session description.
dwFlags

[in] Reserved. Must be 0.

Return Value

Returns DV_OK if successful, or one of the following error values.

| DVERR_ALREADYPENDING | An asynchronous call of this type is already pending. |
|-------------------------------|--|
| DVERR_HOSTING | The object is the host of the session. |
| DVERR_INVALIDFLAGS | The flags passed to this method are invalid. |
| DVERR_INVALIDOBJECT | The DirectPlay object pointer is invalid. |
| DVERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
| DVERR_INVALIDPOINTER | The pointer specified is invalid. |
| DVERR_NOTINITIALIZED | The <u>IDirectPlayVoiceClient::Initialize</u> or <u>IDirectPlayVoiceServer::Initialize</u> method must be called before calling this method. |
| DVERR_COMPRESSIONNOTSUPPORTED | The specified compression type is not supported on the local computer. |
| | |

Remarks

The **IDirectPlayVoiceServer::Initialize** method must be called before this method is called. The voice session can be hosted on any client in the session if the voice session is peer-to-peer. If the voice session is not peer-to-peer, it must be hosted on the transport client, which is the host of a active transport session.

The **DVSESSIONDESC** structure contains the type of voice session to start. The type of voice session can have a dramatic effect on the CPU and bandwidth usage for both the client and the server. To use the default compression type, set the **DVSESSIONDESC** structure's **guidCT** member to DPVCTGUID_DEFAULT.

IDirectPlayVoiceServer::StopSession Method

Stops the Microsoft® DirectPlay® Voice session.

Syntax

```
HRESULT StopSession(
    DWORD dwFlags
);
```

Parameters

dwFlags

[in] Flag. The following flag can be set.
 DVFLAGS_NOHOSTMIGRATE
 The host will not migrate regardless of session and transport settings. Use this flag when you want to shut down the voice session completely.

Return Value

Returns DV_OK if successful, or one of the following error values.

| DVERR_ALREADYPENDING | An asynchronous call of this type is already pending. |
|----------------------|--|
| DVERR_INVALIDFLAGS | The flags passed to this method are invalid. |
| DVERR_INVALIDOBJECT | The DirectPlay object pointer is invalid. |
| DVERR_INVALIDPARAM | One or more of the parameters passed to the method are invalid. |
| DVERR_NOTHOSTING | The object is not the host of the session. |
| DVERR_NOTINITIALIZED | The <u>IDirectPlayVoiceClient::Initialize</u> or <u>IDirectPlayVoiceServer::Initialize</u> method must be called before calling this method. |
| DVERR_SESSIONLOST | The transport has lost the connection to the session. |

Remarks

This method returns DVERR_ALREADYPENDING if it is called while another thread is processing a

IDirectPlayVoiceServer::StopSession request.

IDirectPlayVoiceTest Interface

Applications use the <u>IDirectPlayVoiceTest::CheckAudioSetup</u> method of the **IDirectPlayVoiceTest** interface to test the Microsoft® DirectPlay® Voice audio configuration.

IDirectPlayVoiceTest Members

<u>CheckAudioSetup</u> Runs the Audio Setup Wizard on the specified devices. This wizard runs a series of tests on the devices to determine if they are capable of full duplex audio and to ensure that the microphone is plugged in and working correctly on the capture device.

Interface Information

| Inherits from | <u>IUnknown</u> |
|---------------------------|-----------------|
| Header | dvoice.h |
| Minimum operating systems | Windows 98 |

IDirectPlayVoiceTest::CheckAudioSetup Method

Runs the Audio Setup Wizard on the specified devices. This wizard runs a series of tests on the devices to determine if they are capable of full duplex audio and to ensure that the microphone is plugged in and working correctly on the capture device.

Syntax

```
HRESULT CheckAudioSetup(
const GUID *pguidPlaybackDevice,
const GUID *pguidCaptureDevice,
HWND hwndParent,
DWORD dwFlags
```

);

Parameters

pguidPlaybackDevice

[in] Pointer to the globally unique identifier (GUID) that identifies the playback device to test. If NULL is passed for this parameter, Microsoft® DirectPlay® Voice tests the system default playback device defined by Microsoft DirectSound®. You can also pass one of the DirectSound default GUIDs: DSDEVID DefaultPlayback

The system default playback device.

DSDEVID_DefaultVoicePlayback

The default voice playback device.

pguidCaptureDevice

[in] Pointer to the GUID that identifies the capture device to test. If NULL is passed for this parameter, DirectPlay Voice tests the system default capture device defined by DirectSound. You can also pass one of the DirectSound default GUIDs:

DSDEVID_DefaultCapture

The default system capture device. You can also specify

this device by passing a NULL pointer in the device GUID parameter.

DSDEVID_DefaultVoiceCapture

The default voice communications capture device. Typically, this is a secondary device such as a USB headset with microphone.

hwndParent

[in] The test wizard invoked by this method is modal. If the calling application has a window that should be the parent window of the wizard, it should pass a handle to that window in this parameter. If the calling application does not have a window, it can pass NULL. If the DVFLAGS_QUERYONLY flag is specified, this parameter is not used and the application can pass NULL.

dwFlags

[in] Flags. The following flags can be set.

DVFLAGS_QUERYONLY

Audio setup is not run. Instead, the method checks the registry to see if the devices have been tested. If the devices have not been tested, the method returns DVERR_RUNSETUP. If the devices have been tested, the method returns DV_FULLDUPLEX if the devices support full duplex audio, or DV_HALFDUPLEX if the devices do not support full duplex audio.

DVFLAGS_ALLOWBACK

Passing this flag enables the **Back** button on the wizard's Welcome page. If the user clicks the **Back** button on the Welcome page, the wizard exits, and

IDirectPlayVoiceTest::CheckAudioSetup returns DVERR_USERBACK.

Return Value

Returns DV_OK, DV_FULLDUPLEX, DV_HALFDUPLEX if successful, or one of the following error values.

DVERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DVERR_RUNSETUPThe specified audio configuration has not been tested. Call the
IDirectPlayVoiceTest::CheckAudioSetup method.

Remarks

This method contains user interface (UI) elements and displays dialog boxes. If the DVFLAGS_QUERYONLY flag is specified, the tests are not actually run and no UI is raised. Instead, the registry is checked to determine the results of a previous test of these devices.

If the user cancels the wizard, the

IDirectPlayVoiceTest::CheckAudioSetup call returns DVERR_USERCANCEL. The calling application can then handle the situation appropriately. For example, in DirectPlay Voice part of the gaming options control panel application, if the user clicks **Cancel**, the dialog box displays a message indicating that voice cannot be used because the wizard has been canceled.

This method might return DVERR_INVALIDDEVICE if the device specified does not exist. Also, if you specify the default device and this method still returns this error, then there are no sound devices on the system.

IDP8SimControl Interface

Applications use methods of the **IDP8SimControl** interface to create a DP8Sim session and test applications under a variety of network conditions.

IDP8SimControl Members

| <u>ClearAllStatistics</u> | Clears all of the current DP8Sim statistics. |
|---------------------------|---|
| <u>Close</u> | Closes the open connection to a session. This method must be called on any object successfully initialized with <u>IDP8SimControl::Initialize</u> . |
| <u>GetAllParameters</u> | Retrieves all of the current DP8Sim settings. |
| <u>GetAllStatistics</u> | Retrieves all of the current DP8Sim statistics. |
| <u>Initialize</u> | Initializes the IDP8SimControl interface. This method must be called before calling any other methods of this interface. |
| SetAllParameters | Modifies all of the current DP8Sim settings. |

Interface Information

| Inherits from | <u>IUnknown</u> |
|---------------------------|----------------------------|
| Header | dp8sim.h |
| Minimum operating systems | Windows 98, Pocket PC 2002 |

IDP8SimControl::ClearAllStatistics Method

Clears all of the current DP8Sim statistics.

Syntax

```
HRESULT ClearAllStatistics(
    const DWORD dwFlags
);
```

Parameters

dwFlags [in] Reserved. Must be 0.

Return Value

Returns DP8SIM_OK if successful, or one of the following error values.

DP8SIMERR_INVALIDFLAGSThe flags passed to this method are invalid.DP8SIMERR_INVALIDOBJECTThe DP8Sim control object specified is invalid.DP8SIMERR_NOTINITIALIZEDThe DP8Sim control object has not been initialized.

IDP8SimControl::Close Method

Closes the open connection to a session. This method must be called on any object successfully initialized with <u>IDP8SimControl::Initialize</u>.

Syntax

HRESULT Close(const DWORD *dwFlags*);

Parameters

dwFlags [in] Reserved. Must be 0.

Return Value

Returns DP8SIM_OK if successful, or one of the following error values.

DP8SIMERR_INVALIDFLAGSThe flags passed to this method are invalid.DP8SIMERR_INVALIDOBJECTThe DP8Sim control object specified is invalid.DP8SIMERR_NOTINITIALIZEDThe DP8Sim control object has not been initialized.

Remarks

This method will cancel any operations still outstanding.

IDP8SimControl::GetAllParameters Method

Retrieves all of the current DP8Sim settings.

Syntax

```
HRESULT GetAllParameters(
DP8SIM_PARAMETERS *const pdp8spSend,
DP8SIM_PARAMETERS *const pdp8spReceive,
const DWORD dwFlags
```

);

Parameters

pdp8spSend

[out] Pointer to a <u>DP8SIM_PARAMETERS</u> structure to be filled with the current send parameters.

pdp8spReceive

[out] Pointer to a **DP8SIM_PARAMETERS** structure to be filled with the current receive parameters.

dwFlags

[in] Reserved. Must be 0.

Return Value

Returns DP8SIM_OK if successful, or one of the following error values.

DP8SIMERR_INVALIDFLAGSThe flags passed to this method are invalid.DP8SIMERR_INVALIDOBJECTThe DP8Sim control object specified is invalid.DP8SIMERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DP8SIMERR_INVALIDPOINTERA pointer specified as a parameter is invalid.DP8SIMERR_NOTINITIALIZEDThe DP8Sim control object has not been initialized.

IDP8SimControl::GetAllStatistics Method

Retrieves all of the current DP8Sim statistics.

Syntax

```
HRESULT GetAllStatistics(
DP8SIM_STATISTICS *const pdp8ssSend,
DP8SIM_STATISTICS *const pdp8ssReceive,
const DWORD dwFlags
```

);

Parameters

pdp8ssSend

[out] Pointer to a <u>DP8SIM_STATISTICS</u> structure to be filled with the current send statistics.

pdp8ssReceive

[out] Pointer to a **DP8SIM_STATISTICS** structure to be filled with the current receive statistics.

dwFlags

[in] Reserved. Must be 0.

Return Value

Returns DP8SIM_OK if successful, or one of the following error values.

DP8SIMERR_INVALIDFLAGSThe flags passed to this method are invalid.DP8SIMERR_INVALIDOBJECTThe DP8Sim control object specified is invalid.DP8SIMERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DP8SIMERR_INVALIDPOINTERA pointer specified as a parameter is invalid.DP8SIMERR_NOTINITIALIZEDThe DP8Sim control object has not been initialized.
IDP8SimControl::Initialize Method

Initializes the <u>IDP8SimControl</u> interface. This method must be called before calling any other methods of this interface.

Syntax

HRESULT Initialize(const DWORD *dwFlags*);

Parameters

dwFlags [in] Reserved. Must be 0.

Return Value

Returns DP8SIM_OK if successful, or one of the following error values.

| DP8SIMERR_ALREADYINITIALIZED | The DP8Sim control object has already been initialized. |
|------------------------------|---|
| DP8SIMERR_INVALIDFLAGS | The flags passed to this method are invalid. |
| DP8SIMERR_INVALIDOBJECT | The DP8Sim control object specified is invalid. |
| DP8SIMERR_MISMATCHEDVERSION | A different version of DP8Sim is already in use on this system. |

Remarks

Call this method first after using **CoCreateInstance** to obtain the **IDP8SimControl** interface.

IDP8SimControl::SetAllParameters Method

Modifies all of the current DP8Sim settings.

Syntax

```
HRESULT SetAllParameters(
const DP8SIM_PARAMETERS *const pdp8spSend,
const DP8SIM_PARAMETERS *const pdp8spReceive,
const DWORD dwFlags
```

);

Parameters

pdp8spSend

[out] Pointer to a <u>DP8SIM_PARAMETERS</u> structure containing the send parameters to set.

pdp8spReceive

[out] Pointer to a **DP8SIM_PARAMETERS** structure containing the receive parameters to set.

dwFlags

[in] Reserved. Must be 0.

Return Value

Returns DP8SIM_OK if successful, or one of the following error values.

DP8SIMERR_INVALIDFLAGSThe flags passed to this method are invalid.DP8SIMERR_INVALIDOBJECTThe DP8Sim control object specified is invalid.DP8SIMERR_INVALIDPARAMOne or more of the parameters passed to the method are invalid.DP8SIMERR_INVALIDPOINTERA pointer specified as a parameter is invalid.DP8SIMERR_NOTINITIALIZEDThe DP8Sim control object has not been initialized.

Functions

This section contains functions that are used to create Microsoft® DirectPlay® interface objects.

Functions

- <u>DirectPlay8AddressCreate</u>
- DirectPlay8Create
- DirectPlay8LobbyCreate
- <u>DirectPlayVoiceCreate</u>

DirectPlay8AddressCreate Function

Not currently supported.

Users should instead use **CoCreateInstance** to create the object.

DirectPlay8Create Function

Not currently supported.

Users should instead use **CoCreateInstance** to create the object.

DirectPlay8LobbyCreate Function

Not currently supported.

Users should instead use **CoCreateInstance** to create the object.

DirectPlayVoiceCreate Function

Not currently supported.

Users should instead use **CoCreateInstance** to create the object.

Callback Functions

This section contains Microsoft® DirectPlay® prototype callback functions that are used in Peer-to-Peer, Client/Server, Lobbied, and Voice sessions.

Functions

- PDVMESSAGEHANDLER
- <u>PFNDPNMESSAGEHANDLER</u>

PDVMESSAGEHANDLER Prototype

PDVMESSAGEHANDLER is an application-defined callback function used by the <u>IDirectPlayVoiceClient</u> and <u>IDirectPlayVoiceServer</u> interfaces to send messages to the user.

Syntax

```
typedef HRESULT (CALLBACK *PDVMESSAGEHANDLER)(
  LPVOID pvUserContext,
  DWORD dwMessageType,
  LPVOID lpMessage
);
```

Parameters

pvUserContext

Pointer to the application-defined structure that will be passed to this callback function. This is defined in the *IpUserContext* parameter of the <u>IDirectPlayVoiceServer::Initialize</u> and <u>IDirectPlayVoiceClient::Initialize</u> methods.

dwMessageType

One of the following message types. DVMSGID_CONNECTRESULT DVMSGID_CREATEVOICEPLAYER DVMSGID_DELETEVOICEPLAYER DVMSGID_DISCONNECTRESULT DVMSGID_GAINFOCUS DVMSGID_HOSTMIGRATED DVMSGID_HOSTMIGRATED DVMSGID_LOCALHOSTSETUP DVMSGID_LOSTFOCUS DVMSGID_OUTPUTLEVEL DVMSGID_PLAYEROUTPUTLEVEL DVMSGID_PLAYERVOICESTART DVMSGID_PLAYERVOICESTOP

```
DVMSGID_RECORDSTART
DVMSGID_RECORDSTOP
DVMSGID_SESSIONLOST
DVMSGID_SETTARGETS
IpMessage
```

Structure containing message information.

Return Value

See the documentation for the individual messages for appropriate return values. Unless otherwise noted, this function should return DV_OK.

Remarks

When implementing this callback function, you must first view the message type returned in the *dwMessageType* parameter and then cast the message structure (*lpMessage*) to that type to obtain message information. Some messages don't have a defined structure because they have no parameters. For these messages, the *lpMessage* parameter is NULL.

Note This function may be called on multiple different threads at the same time. It must thus be threadsafe and reentrant.

All message structures have the same name as the corresponding message types except the prefix is DVMSG_ instead of DVMSGID_. For example, the structure for DVMSGID_RECORDSTART is DVMSG_RECORDSTART.

The structure sent to the message handler is valid only for the duration of the call. Therefore, if you want to use any of the

information passed into the function after the handler function has returned you must make a copy of the data.

Callback messages from the same player are serialized. When you receive a message from a player, you will not receive another until you have handled the first message, and the callback function has returned.

Only messages that are specified in the message mask through a call to the IDirectPlayVoiceClient::Initialize, IDirectPlayVoiceServer::Initialize,

IDirectPlayVoiceClient::SetNotifyMask and IDirectPlayVoiceServer::SetNotifyMask methods are sent to this callback function.

The <u>DVMSGID_GAINFOCUS</u> and <u>DVMSGID_LOSTFOCUS</u> message structures have not been implemented in this release of Microsoft® DirectPlay®.

Prototype Information

Headerdvoice.hImport libraryNoneMinimum operating systemsWindows 98, Pocket PC 2002

PFNDPNMESSAGEHANDLER Prototype

PFNDPNMESSAGEHANDLER is an application-defined callback

function used by the <u>IDirectPlay8Peer</u>, <u>IDirectPlay8Client</u>, <u>IDirectPlay8Server</u>, <u>IDirectPlay8LobbyClient</u>, and <u>IDirectPlay8LobbiedApplication</u> interfaces to process messages.

Syntax

typedef HRESULT (CALLBACK *PFNDPNMESSAGEHANDLER)(
 PVOID pvUserContext,
 DWORD dwMessageType,
 PVOID pMessage
);

Parameters

pvUserContext

Pointer to the application-defined structure that will be passed to this callback function. This is defined in the *pvUserContext* parameter of the **Initialize** method.

dwMessageType

One of the following DPN_ message types that are generated by the **IDirectPlay8Peer**, **IDirectPlay8Client**, and **IDirectPlay8Server** interfaces. Each interface uses a different subset of the available DPN_ messages. Refer to the interface documentation for details. Additionally, if the application supports Microsoft® DirectPlay® lobby functionality, this parameter can specify one of the following DPL_ message types that are generated by the **IDirectPlay8LobbyClient** and **IDirectPlay8LobbiedApplication** interfaces. Each lobby interface uses a different subset of the available DPL_ messages. Refer to the interface documentation for details. DPN_MSGID_ADD_PLAYER_TO_GROUP DPN_MSGID_ASYNC_OP_COMPLETE

DPN MSGID CLIENT INFO DPN MSGID CONNECT COMPLETE DPN MSGID CREATE GROUP DPN MSGID CREATE PLAYER DPN MSGID DESTROY GROUP DPN MSGID DESTROY PLAYER DPN MSGID ENUM HOSTS QUERY DPN MSGID ENUM HOSTS RESPONSE DPN MSGID GROUP INFO DPN MSGID HOST MIGRATE DPN MSGID INDICATE CONNECT DPN MSGID INDICATED CONNECT ABORTED DPN_MSGID_PEER_INFO DPN MSGID RECEIVE DPN MSGID REMOVE PLAYER FROM GROUP DPN MSGID RETURN BUFFER DPN MSGID SEND COMPLETE DPN MSGID SERVER INFO DPN MSGID TERMINATE SESSION DPL_MSGID_CONNECT DPL_MSGID_CONNECTION_SETTINGS DPL MSGID DISCONNECT DPL MSGID RECEIVE DPL_MSGID_SESSION_STATUS pMessage Structure containing message information.

Return Value

See the documentation for the individual messages for appropriate return values. Unless otherwise noted, this function should return S_OK.

Remarks

This function must be threadsafe because it might be called reentrantly through multiple threads.

Callback messages from the same player are serialized. When you receive a message from a player, you will not receive another until you have handled the first message, and the callback function has returned.

The message structures have the same name as the message type except the "DPN_MSGID" is replaces with "DPNMSG". For example, the <u>DPN_MSGID_TERMINATE_SESSION</u> message type uses the **DPNMSG_TERMINATE_SESSION** message structure to convey the actual message information.

When implementing this callback function, first view the message type returned in the *dwMessageType* parameter and then cast the message structure (*pMessage*) to that type to obtain message information. Some messages don't have a defined structure because they have no parameters. For these messages, the *pMessage* parameter is NULL.

Prototype Information

| Header | dplay8.h |
|---------------------------|----------------------------|
| Import library | None |
| Minimum operating systems | Windows 98, Pocket PC 2002 |

System Messages

This section contains Microsoft® DirectPlay® messages that are received by a DirectPlay callback message handler. DirectPlay uses these messages to convey information from the system to a DirectPlay application.

- <u>DirectPlay Core Messages</u>
- <u>DirectPlay Lobby Messages</u>
- DirectPlay Voice Messages

DirectPlay Core Messages

The following messages can be processed by the DirectPlay core callback message handlers.

Messages

- <u>DPN_MSGID_ADD_PLAYER_TO_GROUP</u>
- <u>DPN_MSGID_APPLICATION_DESC</u>
- <u>DPN_MSGID_ASYNC_OP_COMPLETE</u>
- <u>DPN_MSGID_CLIENT_INFO</u>
- <u>DPN_MSGID_CONNECT_COMPLETE</u>
- <u>DPN_MSGID_CREATE_GROUP</u>
- <u>DPN_MSGID_CREATE_PLAYER</u>
- <u>DPN_MSGID_CREATE_SENDER_CONTEXT</u>
- <u>DPN_MSGID_CREATE_THREAD</u>
- <u>DPN_MSGID_DESTROY_GROUP</u>
- <u>DPN_MSGID_DESTROY_PLAYER</u>
- <u>DPN_MSGID_DESTROY_THREAD</u>
- <u>DPN_MSGID_ENUM_HOSTS_QUERY</u>
- <u>DPN_MSGID_ENUM_HOSTS_RESPONSE</u>
- <u>DPN_MSGID_GROUP_INFO</u>
- <u>DPN_MSGID_HOST_MIGRATE</u>
- <u>DPN_MSGID_INDICATE_CONNECT</u>
- <u>DPN_MSGID_INDICATED_CONNECT_ABORTED</u>
- <u>DPN_MSGID_NAT_RESOLVER_QUERY</u>
- DPN_MSGID_PEER_INFO
- DPN_MSGID_RECEIVE
- <u>DPN_MSGID_REMOVE_PLAYER_FROM_GROUP</u>
- <u>DPN_MSGID_RETURN_BUFFER</u>
- <u>DPN_MSGID_SEND_COMPLETE</u>

- DPN_MSGID_SERVER_INFO
- <u>DPN_MSGID_TERMINATE_SESSION</u>

DirectPlay Lobby Messages

The following messages are handled by lobby client and lobbied application callback message handlers.

Lobby Messages

- DPL_MSGID_CONNECT
- <u>DPL_MSGID_CONNECTION_SETTINGS</u>
- DPL_MSGID_DISCONNECT
- DPL_MSGID_RECEIVE
- DPL_MSGID_SESSION_STATUS

DirectPlay Voice Messages

The following messages are handled by DirectPlay voice callback message handlers.
Voice Messages

- <u>DVMSGID_CONNECTRESULT</u>
- <u>DVMSGID_CREATEVOICEPLAYER</u>
- <u>DVMSGID_DELETEVOICEPLAYER</u>
- <u>DVMSGID_DISCONNECTRESULT</u>
- <u>DVMSGID_GAINFOCUS</u>
- <u>DVMSGID_HOSTMIGRATED</u>
- DVMSGID_INPUTLEVEL
- <u>DVMSGID_LOCALHOSTSETUP</u>
- DVMSGID_LOSTFOCUS
- <u>DVMSGID_OUTPUTLEVEL</u>
- DVMSGID_PLAYEROUTPUTLEVEL
- <u>DVMSGID_PLAYERVOICESTART</u>
- <u>DVMSGID_PLAYERVOICESTOP</u>
- DVMSGID_RECORDSTART
- DVMSGID_RECORDSTOP
- <u>DVMSGID_SESSIONLOST</u>
- **DVMSGID_SETTARGETS**

DPL_MSGID_CONNECT Message

Microsoft® DirectPlay® generates a **DPL_MSGID_CONNECT** message when a lobby client connects to the lobbied application through the <u>IDirectPlay8LobbyClient::ConnectApplication</u> method.

The **DPL_MESSAGE_CONNECT** structure is passed with the **DPL_MSGID_CONNECT** message.

Syntax

typedef struct _DPL_MESSAGE_CONNECT {
 DWORD dwSize;
 DPNHANDLE hConnectId;
 PDPL_CONNECTION_SETTINGS pdplConnectionSettings;
 PVOID pvLobbyConnectData;
 DWORD dwLobbyConnectDataSize;
 PVOID pvConnectionContext;
}

} DPL_MESSAGE_CONNECT, *PDPL_MESSAGE_CONNECT;

Members

dwSize

Size of the **DPL_MESSAGE_CONNECT** message structure. The application must set this member before it uses the structure.

hConnectId

Handle used to identify the connection. This handle is used in subsequent calls to <u>IDirectPlay8LobbyClient::Send</u> and <u>IDirectPlay8LobbyClient::ReleaseApplication</u>.

pdplConnectionSettings

Pointer to a <u>DPL_CONNECTION_SETTINGS</u> structure with connection information.

pvLobbyConnectData

Pointer to lobby connection data.

dwLobbyConnectDataSize

Variable of type **DWORD** specifying the size of the data contained in the **pvLobbyConnectData** member.

pvConnectionContext

Context value associated with this connection. For lobbied applications, set this parameter when this message is received in your message handler to associate the context value with the connection. This may be set to NULL to disable context values.

Remarks

Return from the message callback function with DPN_OK.

Message Information

Headerdplobby8.hMinimum operating systemsWindows 98, Pocket PC 2002

DPL_MSGID_CONNECTION_SETTINGS Message

The DPL_MSGID_CONNECTION_SETTINGS message is sent from the lobby client to the lobby application when IDirectPlay8LobbyClient::SetConnectionSettings is called. It is also sent from the lobby application to the lobby client when IDirectPlay8LobbiedApplication::SetConnectionSettings is called.

The **DPL_MESSAGE_CONNECTION_SETTINGS** structure is passed with the **DPL_MSGID_CONNECTION_SETTINGS** message.

Syntax

typedef struct _DPL_MESSAGE_CONNECTION_SETTINGS { DWORD dwSize; DPNHANDLE hSender; PDPL_CONNECTION_SETTINGS pdplConnectionSettings; PVOID pvConnectionContext;

} DPL_MESSAGE_CONNECTION_SETTINGS, *PDPL_MESSAGE_C

Members

dwSize

Contains the size of the

DPL_MESSAGE_CONNECTION_SETTINGS structure. It should be set to sizeof(

DPL_MESSAGE_CONNECTION_SETTINGS).

hSender

Contains the handle to the connection that sent this message. **pdplConnectionSettings**

Contains a pointer to a <u>DPL_CONNECTION_SETTINGS</u> structure describing the connection settings for the specified connection.

pvConnectionContext

Pointer to a context value that has been set for the connection.

Remarks

Return from the message callback function with DPN_OK.

The contents of the message are valid only for the duration of the message callback. Therefore, if you want to use the data contained in the message, you must make a copy before returning. In addition, if you want to use the addressing objects you must call **AddRef** on each address to ensure you retain a reference.

For lobbied applications, the context value is set through the **pvConnectionContext** member of the <u>DPL_MESSAGE_CONNECT</u> structure. When your message handler receives this message, whatever you set this member to before returning will be the context value for that connection.

For lobby clients, the *pvConnectionContext* parameter in the <u>IDirectPlay8LobbyClient::ConnectApplication</u> method will be used as the connection's context value if the connection is successful.

Context values are not shared between the lobby client and lobbied application. For example, if you set your context value for a lobby connection in your <u>IDirectPlay8LobbyClient</u> interface to pointer A and in your <u>IDirectPlay8LobbiedApplication</u> interface you set it to pointer B, indications in your **IDirectPlay8LobbyClient** interface will have pointer A as their context value and, in your

IDirectPlay8LobbiedApplication interface, pointer B will be the context value.

You can also set your context values to NULL if you do not want to use this feature.

Message Information

Headerdplobby8.hMinimum operating systemsWindows 98, Pocket PC 2002

 $\ensuremath{\textcircled{}^{\circ}}$ 2003 Microsoft Corporation. All rights reserved.

DPL_MSGID_DISCONNECT Message

Microsoft® DirectPlay® generates a **DPL_MSGID_DISCONNECT** message when a lobby client disconnects from a lobbied application through the <u>IDirectPlay8LobbyClient::ReleaseApplication</u> method.

The **DPL_MESSAGE_DISCONNECT** structure contains information for the **DPL_MSGID_DISCONNECT** system message.

Syntax

typedef struct _DPL_MESSAGE_DISCONNECT {
 DWORD dwSize;
 DPNHANDLE hDisconnectId;
 HRESULT hrReason;
 PVOID pvConnectionContext;
} DPL MESSAGE DISCONNECT, *PDPL MESSAGE DISCONNEC^{*}

Members

dwSize

Size of the **DPL_MESSAGE_DISCONNECT** message structure. The application must set this member before it uses the structure.

hDisconnectId

Handle specifying the disconnection identifier (ID).

hrReason

Reason for the disconnection.

DPN_OK

It was a standard disconnection.

DPNERR_CONNECTIONLOST

This will be set if the process running the client or application exited abnormally.

pvConnectionContext

Context value that has been set for the connection.

Remarks

Return from the message callback function with DPN_OK.

For lobbied applications, the context value is set through the **pvConnectionContext** member of the <u>DPL_MESSAGE_CONNECT</u> structure. When your message handler receives this message, whatever you set this member to before returning will be the context value for that connection.

For lobby clients, the *pvConnectionContext* parameter in the <u>IDirectPlay8LobbyClient::ConnectApplication</u> method will be used as the connection's context value if the connection is successful.

Context values are not shared between lobby client and lobbied application. For example, if you set your context value for a lobby connection in your IDirectPlay8LobbyClient interface to pointer A and in your IDirectPlay8LobbiedApplication interface you set it to pointer B, indications in your IDirectPlay8LobbyClient interface will have pointer A as their context value and, in your IDirectPlay8LobbiedApplication interface, pointer B will be the context value.

You can also set your context values to NULL if you do not want to use this feature.

Message Information

Headerdplobby8.hMinimum operating systemsWindows 98, Pocket PC 2002

DPL_MSGID_RECEIVE Message

Microsoft® DirectPlay® generates the **DPL_MSGID_RECEIVE** message when the target receives a message sent by the <u>IDirectPlay8LobbyClient::Send</u> or <u>IDirectPlay8LobbiedApplication::Send</u> method.

The **DPL_MESSAGE_RECEIVE** structure contains information for the **DPL_MSGID_RECEIVE** system message.

Syntax

typedef struct _DPL_MESSAGE_RECEIVE {
 DWORD dwSize;
 DPNHANDLE hSender;
 BYTE *pBuffer;
 DWORD dwBufferSize;
 PVOID pvConnectionContext;
} DPL MESSAGE RECEIVE, *PDPL MESSAGE RECEIVE;

Members

dwSize

Size of the **DPL_MESSAGE_RECEIVE** message structure. The application must set this member before it uses the structure.

hSender

Handle of the client that sent the message.

pBuffer

Pointer to message data.

dwBufferSize

Size of the message data contained in the **pBuffer**.

pvConnectionContext

Context value that has been set for the connection.

Remarks

Return from the message callback function with DPN_OK.

For lobbied applications, the context value is set through the **pvConnectionContext** member of the **DPL_MESSAGE_CONNECT** structure. When your message handler receives this message, whatever you set this member to before returning will be the context value for that connection.

For lobby clients, the *pvConnectionContext* parameter in the <u>IDirectPlay8LobbyClient::ConnectApplication</u> method will be used as the connection's context value if the connection is successful.

Context values are not shared between the lobby client and lobbied application. For example, if you set your context value for a lobby connection in your <u>IDirectPlay8LobbyClient</u> interface to pointer A, and in your <u>IDirectPlay8LobbiedApplication</u> interface you set it to pointer B, indications in your **IDirectPlay8LobbyClient** interface will have pointer A as their context value, and in your **IDirectPlay8LobbiedApplication** interface, pointer B will be the context value.

You can also set your context values to NULL if you do not want to use this feature.

Message Information

Headerdplobby8.hMinimum operating systemsWindows 98, Pocket PC 2002

DPL_MSGID_SESSION_STATUS Message

Microsoft® DirectPlay® generates the DPL_MSGID_SESSION_STATUS message when the session has been updated with a call to the IDirectPlay8LobbiedApplication::UpdateStatus method.

The DPL MESSAGE SESSION STATUS structure contains the information for the **DPL_MSGID_SESSION_STATUS** system message.

Syntax

typedef struct DPL MESSAGE SESSION STATUS { DWORD dwSize; **DPNHANDLE** hSender; DWORD dwStatus; PVOID pvConnectionContext;

} DPL MESSAGE SESSION STATUS, *PDPL MESSAGE SESSIOI

Members

dwSize

Size of the DPL MESSAGE SESSION STATUS message structure. The application must set this member before it uses the structure.

hSender

The handle of the application that sent the status update message.

dwStatus

Updated status of the session. This member can be set to one of the following values.

DPLSESSION CONNECTED

The lobbied application is currently connected to a session. DPLSESSION COULDNOTCONNECT

The lobbied application could not connect to the session. DPLSESSION DISCONNECTED

The lobbied application is currently disconnected from the session.

DPLSESSION_TERMINATED

The connection between session host and the lobbied application has been terminated.

DPLSESSION_HOSTMIGRATED

The host of a peer-to-peer session has migrated. The local client is not the new host.

DPLSESSION_HOSTMIGRATEDHERE

The host of a peer-to-peer session has migrated. The local client is the new host.

pvConnectionContext

Context value that has been set for the connection.

Remarks

Return from the message callback function with DPN_OK.

For lobbied applications, the context value is set through the **pvConnectionContext** member of the **DPL_MESSAGE_CONNECT** structure. When your message handler receives this message, whatever you set this member to before returning will be the context value for that connection.

For lobby clients, the *pvConnectionContext* parameter in the <u>IDirectPlay8LobbyClient::ConnectApplication</u> method will be used as the connection's context value if the connection is successful.

Context values are not shared between lobby client and lobbied application. For example, if you set your context value for a lobby connection in your <u>IDirectPlay8LobbyClient</u> interface to pointer A and in your <u>IDirectPlay8LobbiedApplication</u> interface you set it to pointer B, indications in your **IDirectPlay8LobbyClient** interface will have pointer A as their context value and in your **IDirectPlay8LobbiedApplication** interface pointer B will be the context value.

You can also set your context values to NULL if you do not want to use this feature.

Message Information

Headerdplobby8.hMinimum operating systemsWindows 98, Pocket PC 2002

DPN_MSGID_ADD_PLAYER_TO_GROUP Message

Microsoft® DirectPlay® generates the

DPN_MSGID_ADD_PLAYER_TO_GROUP message when a player has been added to a group in a peer-to-peer or client/server session.

The **DPNMSG_ADD_PLAYER_TO_GROUP** structure contains information for the **DPN_MSGID_ADD_PLAYER_TO_GROUP** system message.

Syntax

typedef struct _DPNMSG_ADD_PLAYER_TO_GROUP {
 DWORD dwSize;
 DPNID dpnidGroup;
 PVOID pvGroupContext;
 DPNID dpnidPlayer;
 PVOID pvPlayerContext;
} DPNMSG ADD PLAYER TO GROUP, *PDPNMSG ADD PLAYEF

Members

dwSize

Size of this structure.

dpnidGroup

DPNID of the group to add the player.

pvGroupContext

Group context value.

dpnidPlayer

DPNID of the player added to the group.

pvPlayerContext

Player context value.

Remarks

Return from the message callback function with DPN_OK.

Message Information

Headerdplay8.hMinimum operating systemsWindows 98, Pocket PC 2002

DPN_MSGID_APPLICATION_DESC Message

This message indicates that the application description has been changed. There is no accompanying structure.

Remarks

Return from the message callback function with DPN_OK.

To determine the new application description, call the **GetApplicationDesc** method exposed by <u>IDirectPlay8Peer</u>, <u>IDirectPlay8Client</u>, or <u>IDirectPlay8Server</u> interfaces.

Message Information

Headerdplobby8.hMinimum operating systemsWindows 98, Pocket PC 2002

DPN_MSGID_ASYNC_OP_COMPLETE Message

Microsoft® DirectPlay® generates the

DPN_MSGID_ASYNC_OP_COMPLETE message when an asynchronous request has completed.

The **DPNMSG_ASYNC_OP_COMPLETE** structure contains information for the **DPN_MSGID_ASYNC_OP_COMPLETE** system message.

Syntax

typedef struct _DPNMSG_ASYNC_OP_COMPLETE {
 DWORD dwSize;
 DPNHANDLE hAsyncOp;
 PVOID pvUserContext;
 HRESULT hResultCode;
} DPNMSG ASYNC OP COMPLETE, *PDPNMSG ASYNC OP CC

Members

dwSize Size of this structure. hAsyncOp Asynchronous operation handle.

pvUserContext

Supplied user context.

hResultCode

HRESULT indicating the result of the asynchronous operation.

Remarks

Return from the message callback function with DPN_OK.

Message Information

Headerdplay8.hMinimum operating systemsWindows 98, Pocket PC 2002

 $\ensuremath{\textcircled{}^{\circ}}$ 2003 Microsoft Corporation. All rights reserved.

DPN_MSGID_CLIENT_INFO Message

Microsoft® DirectPlay® generates the **DPN_MSGID_CLIENT_INFO** message when client data is modified during a client/server session.

The **DPNMSG_CLIENT_INFO** structure contains information for the **DPN_MSGID_CLIENT_INFO** system message.

Syntax

typedef struct _DPNMSG_CLIENT_INFO {
 DWORD dwSize;
 DPNID dpnidClient;
 PVOID pvPlayerContext;
} DPNMSG_CLIENT_INFO, *PDPNMSG_CLIENT_INFO;

Members

dwSize Size of this structure. dpnidClient DPNID of the client for client information. pvPlayerContext Player context value.

Remarks

Return from the message callback function with DPN_OK.

Message Information

Headerdplay8.hMinimum operating systemsWindows 98, Pocket PC 2002

DPN_MSGID_CONNECT_COMPLETE Message

Microsoft® DirectPlay® generates the

DPN_MSGID_CONNECT_COMPLETE message when the connection attempt has been completed in a peer-to-peer or client/server session.

The **DPNMSG_CONNECT_COMPLETE** structure contains information for the **DPN_MSGID_CONNECT_COMPLETE** system message.

Syntax

typedef struct _DPNMSG_CONNECT_COMPLETE { DWORD dwSize; DPNHANDLE hAsyncOp; PVOID pvUserContext; HRESULT hResultCode; PVOID pvApplicationReplyData; DWORD dwApplicationReplyDataSize; DPNID dpnidLocal; DPNMSG_CONNECT_COMPLETE_*PDPNMSG_CONNECT_COM

} DPNMSG_CONNECT_COMPLETE, *PDPNMSG_CONNECT_COM

Members

dwSize

Size of this structure.

hAsyncOp

Asynchronous operation handle.

pvUserContext

User context supplied when the <u>IDirectPlay8Peer::Connect</u> or <u>IDirectPlay8Client::Connect</u> methods are called.

hResultCode

HRESULT describing the result of the connection attempt. See the Return Values section in the **IDirectPlay8Peer::Connect** or **IDirectPlay8Client::Connect** method for more information. Additionally, <u>DPNERR_PLAYERNOTREACHABLE</u> will be returned if a player has tried to join a peer-to-peer session where at least one other existing player in the session cannot connect to the joining player.

pvApplicationReplyData

Connection reply data returned from the host or server.

dwApplicationReplyDataSize

Size of the data, in bytes, of the **pvApplicationReplyData** member.

dpnidLocal

Specifies the DPNID of the local player.

Remarks

Return from the message callback function with DPN_OK.

This message is generated whether or not the connection was successful.

Message Information

Headerdplay8.hMinimum availabilityDirectX 9.0Minimum operating systemsWindows 2000, Windows 98

DPN_MSGID_CREATE_GROUP Message

Microsoft® DirectPlay® generates the **DPN_MSGID_CREATE_GROUP** message when a group is created.

The **DPNMSG_CREATE_GROUP** structure contains information for the **DPN_MSGID_CREATE_GROUP** system message.

Syntax

typedef struct _DPNMSG_CREATE_GROUP { DWORD dwSize; DPNID dpnidGroup; DPNID dpnidOwner; PVOID pvGroupContext; PVOID pvOwnerContext;

} DPNMSG_CREATE_GROUP, *PDPNMSG_CREATE_GROUP;

Members

dwSize

Size of this structure.

dpnidGroup

DPNID of the of the created group.

dpnidOwner

DPNID of the of the group's owner. This value is only set for groups that have the DPNGROUP_AUTODESTRUCT flag set in the **dwGroupFlags** member of the <u>DPN_GROUP_INFO</u> structure.

pvGroupContext

Group context value.

pvOwnerContext

Owner context value.

Remarks

Return from the message callback function with DPN_OK.

The only method of setting the group context value is through this system message. Once set, group context values cannot be changed.

Message Information

Headerdplay8.hMinimum availabilityDirectX 9.0Minimum operating systemsWindows 2000, Windows 98

DPN_MSGID_CREATE_PLAYER Message

Microsoft® DirectPlay® generates the **DPN_MSGID_CREATE_PLAYER** message when a player is added to a peer-to-peer or client/server session.

The **DPNMSG_CREATE_PLAYER** structure contains information for the **DPN_MSGID_CREATE_PLAYER** system message.

Syntax

typedef struct _DPNMSG_CREATE_PLAYER {
 DWORD dwSize;
 DPNID dpnidPlayer;
 PVOID pvPlayerContext;
} DPNMSG_CREATE_PLAYER, *PDPNMSG_CREATE_PLAYER;

Members

dwSize Size of this structure. dpnidPlayer DPNID of the player that was added to the session. pvPlayerContext Player context value.

Remarks

Return from the message callback function with DPN_OK.

The only method of setting the player context value is through this message. You can either set the player context value directly, through this message, or indirectly through

DPN_MSGID_INDICATE_CONNECT. Once a player context value
has been set, it cannot be changed.

Message Information

Headerdplay8.hMinimum operating systemsWindows 98, Pocket PC 2002

© 2003 Microsoft Corporation. All rights reserved.

DPN_MSGID_CREATE_SENDER_CONTEXT Message

Microsoft® DirectPlay® generates a DPN_MSGID_CREATE_SENDER_CONTEXT message when another player is recognized by the local player.

Syntax

typedef struct _DPNMSG_CREATE_SENDER_CONTEXT {
 DWORD dwSize;
 PVOID pvSenderContext;
} DPNMSG_CREATE_SENDER_CONTEXT, *PDPNMSG_CREATE_\$

Members

dwSize Size of this structure. pvSenderContext Sender context value.

Message Information

Headerdplay8.hMinimum operating systemsWindows 98, Pocket PC 2002

© 2003 Microsoft Corporation. All rights reserved.

DPN_MSGID_CREATE_THREAD Message

Microsoft® DirectPlay® generates the **DPN_MSGID_CREATE_THREAD** message when a new thread is created.

The **DPNMSG_CREATE_THREAD** structure contains information for the **DPN_MSGID_CREATE_THREAD** system message.

Syntax

typedef struct _DPNMSG_CREATE_THREAD { DWORD dwSize; DWORD dwFlags; DWORD dwProcessorNum; PVOID pvUserContext;

} DPNMSG_CREATE_THREAD, *PDPNMSG_CREATE_THREAD;

Members

dwSize
 Size of this structure.

 dwFlags
 Reserved. Must be 0.

 dwProcessorNum
 Specifies the processor index to which this thread is bound.

pvUserContext

Thread context value.

Remarks

Return from the message callback function with DPN_OK.

Message Information

Headerdplay8.hMinimum availabilityDirectX 9.0

Minimum operating systems Windows 2000, Windows 98

 $\ensuremath{\textcircled{\sc 0}}$ 2003 Microsoft Corporation. All rights reserved.

DPN_MSGID_DESTROY_GROUP Message

Microsoft® DirectPlay® generates the

DPN_MSGID_DESTROY_GROUP message when a group is destroyed from a peer-to-peer or client/server session.

The **DPNMSG_DESTROY_GROUP** structure contains information for the **DPN_MSGID_DESTROY_GROUP** system message.

Syntax

typedef struct _DPNMSG_DESTROY_GROUP {
 DWORD dwSize;
 DPNID dpnidGroup;
 PVOID pvGroupContext;
 DWORD dwReason;
} DPNMSG DESTROY GROUP, *PDPNMSG DESTROY GROUP;

Members

dwSize

Size of this structure.

dpnidGroup

DPNID of the group deleted from the session.

pvGroupContext

Group context value.

dwReason

The following flag can be set to indicate why the player was destroyed.

DPNDESTROYGROUPREASON_SESSIONTERMINATED The group is being destroyed because the session was terminated.

Remarks

Return from the message callback function with DPN_OK.

Message Information

Headerdplay8.hMinimum operating systemsWindows 98, Pocket PC 2002

© 2003 Microsoft Corporation. All rights reserved.

DPN_MSGID_DESTROY_PLAYER Message

Microsoft® DirectPlay® generates the

DPN_MSGID_DESTROY_PLAYER message when a player leaves a peer-to-peer or client/server session.

The **DPNMSG_DESTROY_PLAYER** structure contains information for the **DPN_MSGID_DESTROY_PLAYER** system message.

Syntax

typedef struct _DPNMSG_DESTROY_PLAYER {
 DWORD dwSize;
 DPNID dpnidPlayer;
 PVOID pvPlayerContext;
 DWORD dwReason;
} DPNMSG DESTROY PLAYER, *PDPNMSG DESTROY PLAYER;

Members

dwSize Size of this structure.

dpnidPlayer

DPNID of the player deleted from the session.

pvPlayerContext

Player context value.

dwReason

One of the following flags indicating why the player was destroyed.

DPNDESTROYPLAYERREASON_NORMAL

The player is being deleted for normal reasons.

DPNDESTROYPLAYERREASON_CONNECTIONLOST

The player is being deleted because the connection was lost.

DPNDESTROYPLAYERREASON_SESSIONTERMINATED

The player is being deleted because the session was terminated. DPNDESTROYPLAYERREASON_HOSTDESTROYEDPLAYER The player is being deleted because the host called <u>IDirectPlay8Peer::DestroyPeer</u>.

Remarks

Return from the message callback function with DPN_OK.

In client/server mode, this message is received only by the server. In peer-to-peer mode, all players receive this message.

When the server closes a session, it receives a **DPN_MSGID_DESTROY_PLAYER** message for all connected players. Because the server knows that it is disconnecting, this is normal behavior, and the **dwReason** member of the associated structure is set to DPNDESTROYPLAYERREASON_NORMAL. The DPNDESTROYPLAYERREASON_SESSIONTERMINATED value is only set for unexpected disconnections.

You might receive <u>DPN_MSGID_CREATE_PLAYER</u> and DPN_MSGID_DESTROY_PLAYER messages on different threads. However, you will not receive a DPN_MSGID_DESTROY_PLAYER message before your callback function has returned from receiving a DPN_MSGID_CREATE_PLAYER message.

Message Information

Headerdplay8.hMinimum operating systemsWindows 98, Pocket PC 2002

© 2003 Microsoft Corporation. All rights reserved.

DPN_MSGID_DESTROY_THREAD Message

Microsoft® DirectPlay® generates the

DPN_MSGID_DESTROY_THREAD message when a thread is shut down.

The **DPNMSG_DESTROY_THREAD** structure contains information for the **DPN_MSGID_DESTROY_THREAD** system message.

Syntax

typedef struct _DPNMSG_DESTROY_THREAD {
 DWORD dwSize;
 DWORD dwProcessorNum;
 PVOID pvUserContext;
} DPNMSG DESTROY THREAD, *PDPNMSG DESTROY THREAD

Members

dwSize
 Size of this structure.
dwProcessorNum
 Specifies the processor index to which this thread is bound.
pvUserContext
 Thread context value.

Remarks

Return from the message callback function with DPN_OK.

Message Information

Headerdplay8.hMinimum availabilityDirectX 9.0Minimum operating systemsWindows 2000, Windows 98

 $\ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.

DPN_MSGID_ENUM_HOSTS_QUERY Message

Microsoft® DirectPlay® sends the

DPN_MSGID_ENUM_HOSTS_QUERY message to the host's message handler when a peer or client is enumerating the available hosts.

The **DPNMSG_ENUM_HOSTS_QUERY** structure contains information for the **DPN_MSGID_ENUM_HOSTS_QUERY** system message.

Syntax

typedef struct _DPNMSG_ENUM_HOSTS_QUERY {
 DWORD dwSize;
 IDirectPlay8Address *pAddressSender;
 IDirectPlay8Address *pAddressDevice;
 PVOID pvReceivedData;
 DWORD dwReceivedDataSize;
 DWORD dwMaxResponseDataSize;
 PVOID pvResponseData;
 DWORD dwResponseDataSize;
 PVOID pvResponseContext;
} DPNMSG ENUM HOSTS QUERY, *PDPNMSG ENUM HOSTS (

Members

dwSize

Size of this structure.

pAddressSender

Pointer an <u>IDirectPlay8Address</u> interface specifying the address of the sender. You must call <u>IDirectPlay8Address::AddRef</u> to increment the interface's reference count. Call <u>IDirectPlay8Address::Release</u> when you no longer need the interface.

pAddressDevice

Pointer an IDirectPlay8Address interface specifying the

address of the device. You must call

IDirectPlay8Address::AddRef to increment the interface's reference count. Call **IDirectPlay8Address::Release** when you no longer need the interface.

pvReceivedData

Pointer to the data received from the enumeration.

dwReceivedDataSize

Size of the data pointed to in the **pvReceivedData** member. **dwMaxResponseDataSize**

Maximum allowed size for the enumeration response.

pvResponseData

Pointer to the response data from the enumeration. This data must be valid beyond the scope of the callback message handler. It cannot be stack-based. You will receive a <u>DPN_MSGID_RETURN_BUFFER</u> message when DirectPlay is finished with this buffer.

dwResponseDataSize

Size of the data pointed to in the **pvResponseData** member.

pvResponseContext

Pointer to a response context value. This value will be passed to the host's message handler with the

DPN_MSGID_RETURN_BUFFER message as the **pvUserContext** member of the associated structure.

Remarks

Return from the message callback function with DPN_OK.

When you respond normally to this query, DirectPlay will send you a **DPN_MSGID_RETURN_BUFFER** message once the buffer is no longer needed. You can then safely free the buffer.

You can reject the query by returning a value that is not equal to DPN_OK. However, when you reject a query, DirectPlay does not send a reply, does not need a reply buffer, and does not generate a **DPN_MSGID_RETURN_BUFFER** message.

Message Information

Headerdplay8.hMinimum operating systemsWindows 98, Pocket PC 2002

 $\ensuremath{\textcircled{}^{\circ}}$ 2003 Microsoft Corporation. All rights reserved.

DPN_MSGID_ENUM_HOSTS_RESPONSE Message

Microsoft® DirectPlay® sends the

DPN_MSGID_ENUM_HOSTS_RESPONSE message to a peer or client's message handler to convey the host's response to an enumeration request.

The **DPNMSG_ENUM_HOSTS_RESPONSE** structure contains information for the **DPN_MSGID_ENUM_HOSTS_RESPONSE** system message.

Syntax

typedef struct _DPNMSG_ENUM_HOSTS_RESPONSE {
 DWORD dwSize;
 IDirectPlay8Address *pAddressSender;
 IDirectPlay8Address *pAddressDevice;
 const DPN_APPLICATION_DESC *pApplicationDescription;
 PVOID pvResponseData;
 DWORD dwResponseDataSize;
 PVOID pvUserContext;
 DWORD dwRoundTripLatencyMS;
} DPNMSG ENUM HOSTS RESPONSE, *PDPNMSG ENUM HOST

Members

dwSize

Size of this structure.

pAddressSender

Pointer to an <u>IDirectPlay8Address</u> interface specifying the address of the host responding to the enumeration. You must call <u>IDirectPlay8Address::AddRef</u> to increment the interface's reference count. Call <u>IDirectPlay8Address::Release</u> when you no longer need the interface.

pAddressDevice

Pointer an **IDirectPlay8Address** interface specifying the address of the device. You must call

IDirectPlay8Address::AddRef to increment the interface's reference count. Call **IDirectPlay8Address::Release** when you no longer need the interface.

pApplicationDescription

Pointer to a <u>DPN_APPLICATION_DESC</u> structure containing the application description.

pvResponseData

Pointer to the response data from the enumeration.

dwResponseDataSize

Size of the data pointed to in the **pvResponseData** member. **pvUserContext**

Pointer to the user context value. This value is the same as the user context value passed to <u>IDirectPlay8Peer::EnumHosts</u> or <u>IDirectPlay8Client::EnumHosts</u>.

dwRoundTripLatencyMS

Latency measured in milliseconds.

Remarks

Return from the message callback function with DPN_OK.

Because there is no buffer to fill, this message does not generate a <u>DPN_MSGID_RETURN_BUFFER</u> message.

Message Information

Headerdplay8.hMinimum operating systemsWindows 98, Pocket PC 2002

© 2003 Microsoft Corporation. All rights reserved.

DPN_MSGID_GROUP_INFO Message

Microsoft® DirectPlay® generates the **DPN_MSGID_GROUP_INFO** message when group data is modified during a peer-to-peer or client/server session.

The **DPNMSG_GROUP_INFO** structure contains information for the **DPN_MSGID_GROUP_INFO** system message.

Syntax

typedef struct _DPNMSG_GROUP_INFO {
 DWORD dwSize;
 DPNID dpnidGroup;
 PVOID pvGroupContext;
} DPNMSG GROUP INFO, *PDPNMSG GROUP INFO;

Members

dwSize Size of this structure. dpnidGroup DPNID of the group for group information. pvGroupContext Group context value.

Remarks

Return from the message callback function with DPN_OK.

Message Information

Headerdplay8.hMinimum operating systemsWindows 98, Pocket PC 2002

 $\ensuremath{\textcircled{\sc 0}}$ 2003 Microsoft Corporation. All rights reserved.

DPN_MSGID_HOST_MIGRATE Message

Microsoft® DirectPlay® generates the **DPN_MSGID_HOST_MIGRATE** message if the DPNSESSION_MIGRATE_HOST flag is set in the <u>DPN_APPLICATION_DESC</u> structure and the host has migrated.

The **DPNMSG_HOST_MIGRATE** structure contains information for the **DPN_MSGID_HOST_MIGRATE** system message.

Syntax

typedef struct _DPNMSG_HOST_MIGRATE {
 DWORD dwSize;
 DPNID dpnidNewHost;
 PVOID pvPlayerContext;
} DPNMSG HOST MIGRATE, *PDPNMSG HOST MIGRATE;

Members

dwSize Size of this structure. dpnidNewHost DPNID of the player that is now hosting the session. pvPlayerContext Player context value.

Remarks

Return from the message callback function with DPN_OK.

Message Information

Headerdplay8.hMinimum operating systemsWindows 98, Pocket PC 2002

 $\ensuremath{\textcircled{\sc 0}}$ 2003 Microsoft Corporation. All rights reserved.

DPN_MSGID_INDICATED_CONNECT_ABORTED Message

Microsoft® DirectPlay® generates the

DPN_MSGID_INDICATED_CONNECT_ABORTED message if a player's connection drops after it was indicated on the host, but prior to being added to the session though <u>DPN_MSGID_CREATE_PLAYER</u>.

The **DPNMSG_INDICATED_CONNECT_ABORTED** structure contains information for the **DPN_MSGID_INDICATED_CONNECT_ABORTED** system message.

Syntax

typedef struct _DPNMSG_INDICATED_CONNECT_ABORTED {
 DWORD dwSize;
 PVOID pvPlayerContext;

} DPNMSG_INDICATED_CONNECT_ABORTED, *PDPNMSG_INDIC

Members

dwSize Size of this structure. pvPlayerContext Player context value.

Remarks

Return from the message callback function with DPN_OK.

Message Information

Headerdplay8.hMinimum operating systemsWindows 98, Pocket PC 2002

 $\ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.

DPN_MSGID_INDICATE_CONNECT Message

Microsoft® DirectPlay® generates the

DPN_MSGID_INDICATE_CONNECT message when a player attempts to connect to a peer-to-peer or client/server session.

The **DPNMSG_INDICATE_CONNECT** structure contains information for the **DPN_MSGID_INDICATE_CONNECT** system message.

Syntax

typedef struct _DPNMSG_INDICATE_CONNECT {
 DWORD dwSize;
 PVOID pvUserConnectData;
 DWORD dwUserConnectDataSize;
 PVOID pvReplyData;
 DWORD dwReplyDataSize;
 PVOID pvReplyContext;
 PVOID pvPlayerContext;
 IDirectPlay8Address *pAddressPlayer;
 IDirectPlay8Address *pAddressDevice;
} DPNMSG INDICATE CONNECT, *PDPNMSG_INDICATE_CONNE

Members

dwSize

Size of this structure.

pvUserConnectData

Data of the connecting player.

dwUserConnectDataSize

Size of the data, in bytes, contained in the **pvUserConnectData** member.

pvReplyData

Connection reply data. This data must be valid beyond the scope of the callback message handler. You will receive a

DPN_MSGID_RETURN_BUFFER message when DirectPlay is

finished with this buffer.

dwReplyDataSize

Size of the data, in bytes, contained in the **pvReplyData** member.

pvReplyContext

Buffer context for **pvReplyData**. This value will be passed to the host's message handler with the

DPN_MSGID_RETURN_BUFFER message as the

pvUserContext member of the associated structure.

pvPlayerContext

Player context preset.

pAddressPlayer

Pointer to an <u>IDirectPlay8Address</u> interface for the connecting player. You must call <u>IDirectPlay8Address::AddRef</u> to increment the interface's reference count. Call

<u>IDirectPlay8Address::Release</u> when you no longer need the interface.

pAddressDevice

Pointer to an **IDirectPlay8Address** interface for the device receiving the connect attempt. You must call

IDirectPlay8Address::AddRef to increment the interface's reference count. Call **IDirectPlay8Address::Release** when you no longer need the interface.

Remarks

Return DPN_OK to allow the player to join the session. Any other return value will reject the requested connection.

The **hResultCode** member of the structure associated with the <u>DPN_MSGID_CONNECT_COMPLETE</u> message that is sent to the player requesting a connection will be set to S_OK if the connection was successful. If the connection is rejected, **hResultCode** will be set to <u>DPNERR_HOSTREJECTEDCONNECTION</u>, not the value you return from this message. When an **DPN_MSGID_INDICATE_CONNECT** notification arrives on the host player's message handler, setting **pvPlayerContext** before returning the thread will preset the player context value on the respective <u>DPN_MSGID_CREATE_PLAYER</u> notification. This feature allows you to pass a player context value to **DPN_MSGID_CREATE_PLAYER**.

If you set a player context value, that value is not frozen until the subsequent **DPN_MSGID_CREATE_PLAYER** message has been processed. You thus have the option of modifying this player context value when you process **DPN_MSGID_CREATE_PLAYER**.

If a client drops the connection after the server has processed the DPN_MSGID_INDICATE_CONNECT message but before it has processed DPN_MSGID_CREATE_PLAYER, the server will receive a DPN_MSGID_INDICATED_CONNECT_ABORTED message. If you receive this message, free any memory that you allocated while processing DPN_MSGID_INDICATE_CONNECT. When DPN_MSGID_CREATE_PLAYER has been processed, this memory should be freed when you process DPN_MSGID_DESTROY_PLAYER.

If you specify a value for **pvUserConnectData**, you will subsequently be sent a **DPN_MSGID_RETURN_BUFFER** message to notify you that you can safely free the buffer.

Message Information

Headerdplay8.hMinimum operating systemsWindows 98, Pocket PC 2002

© 2003 Microsoft Corporation. All rights reserved.
DPN_MSGID_NAT_RESOLVER_QUERY Message

Microsoft® DirectPlay® generates the

DPN_MSGID_NAT_RESOLVER_QUERY message when a player using the Network Address Translation (NAT) Resolver calls **Host**,

EnumHosts, or Connect.

The **DPNMSG_NAT_RESOLVER_QUERY** structure contains information for the **DPN_MSGID_NAT_RESOLVER_QUERY** system message.

Syntax

typedef struct _DPNMSG_NAT_RESOLVER_QUERY { DWORD dwSize; IDirectPlay8Address pAddressSender; IDirectPlay8Address pAddressDevice; WCHAR pwszUserString;

} DPNMSG_NAT_RESOLVER_QUERY, *PDPNMSG_NAT_RESOLVE

Members

dwSize Size of this structure.

pAddressSender

Address of the client that sent the query.

pAddressDevice

Address of the device that received the query.

pwszUserString

User specified string or NULL if no string was specified.

Remarks

Return from the message callback function with DPN_OK

A player will be using the NAT Resolver if they have called <u>IDirectPlay8Address::AddComponent</u> with the *pwszName* parameter set to DPNA_KEY_NAT_RESOLVER on the device address.

Message Information

Headerdplay8.hMinimum operating systemsWindows 98, Pocket PC 2002

DPN_MSGID_PEER_INFO Message

Microsoft® DirectPlay® generates the **DPN_MSGID_PEER_INFO** message when peer data is modified during a peer-to-peer session.

The **DPNMSG_PEER_INFO** structure contains information for the **DPN_MSGID_PEER_INFO** system message.

Syntax

typedef struct _DPNMSG_PEER_INFO {
 DWORD dwSize;
 DPNID dpnidPeer;
 PVOID pvPlayerContext;
} DPNMSG_PEER_INFO, *PDPNMSG_PEER_INFO;

Members

dwSize Size of this structure. dpnidPeer DPNID of the peer for peer information. pvPlayerContext Player context value.

Remarks

Return from the message callback function with DPN_OK.

Message Information

Headerdplay8.hMinimum operating systemsWindows 98, Pocket PC 2002

DPN_MSGID_RECEIVE Message

Microsoft® DirectPlay® generates the **DPN_MSGID_RECEIVE** message when a message has been processed by the receiver.

The **DPNMSG_RECEIVE** structure contains information for the **DPN_MSGID_RECEIVE** system message.

Syntax

typedef struct _DPNMSG_RECEIVE {
 DWORD dwSize;
 DPNID dpnidSender;
 PVOID pvPlayerContext;
 PBYTE pReceiveData;
 DWORD dwReceiveDataSize;
 DPNHANDLE hBufferHandle;
 DWORD dwReceiveFlags;
} DPNMSG RECEIVE, *PDPNMSG RECEIVE;

Members

dwSize

Size of this structure.

dpnidSender

DPNID of the player that sent the message.

pvPlayerContext

Player context value of the player that sent the message.

pReceiveData

Pointer to the message data buffer. This buffer is normally only valid while the **DPN_MSGID_RECEIVE** message is being processed by the callback message handler.

dwReceiveDataSize

Size of the data, in bytes, of the **pReceiveData** member.

hBufferHandle

Buffer handle to the **pReceiveData** member. If you have

returned DPNSUCCESS_PENDING , pass this value to the appropriate **ReturnBuffer** method to notify DirectPlay to free the buffer.

dwReceiveFlags

The following flags can be specified to describe how messages are received.

DPNRECEIVE_GUARANTEED

The message received was sent guaranteed.

DPNRECEIVE_COALESCED

The message received was coalesced for sending.

Remarks

Return from the message callback function with DPN_OK.

Because you should not spend large amounts of time processing messages, you should copy this data, and process the message. Alternatively, you can return DPNSUCCESS_PENDING from the callback message handler. Doing so transfers ownership of the buffer to the application. If you return DPNSUCCESS_PENDING, you must call <u>IDirectPlay8Peer::ReturnBuffer</u>,

IDirectPlay8Client::ReturnBuffer, or IDirectPlay8Server::ReturnBuffer when you are finished with the buffer. Pass the method the value you receive in the **hBufferHandle** member to identify the buffer. If you fail to call **ReturnBuffer**, you will create a memory leak.

Message Information

Headerdplay8.hMinimum operating systemsWindows 98, Pocket PC 2002

DPN_MSGID_REMOVE_PLAYER_FROM_GROUP Message

Microsoft® DirectPlay® generates the

DPN_MSGID_REMOVE_PLAYER_FROM_GROUP message when a player has been deleted from a group in a peer-to-peer or client/server session.

The **DPNMSG_REMOVE_PLAYER_FROM_GROUP** structure contains information for the **DPN_MSGID_REMOVE_PLAYER_FROM_GROUP** system message.

Syntax

typedef struct _DPNMSG_REMOVE_PLAYER_FROM_GROUP {
 DWORD dwSize;
 DPNID dpnidGroup;
 PVOID pvGroupContext;
 DPNID dpnidPlayer;
 PVOID pvPlayerContext;
} DPNMSG_REMOVE_PLAYER_FROM_GROUP, *PDPNMSG_REM(

Members

dwSize

Size of this structure.

dpnidGroup

DPNID of the group that the player was deleted from.

pvGroupContext

Group context value.

dpnidPlayer

DPNID of the player deleted from the group.

pvPlayerContext

Player context value.

Remarks

Return from the message callback function with DPN_OK.

Message Information

Headerdplay8.hMinimum operating systemsWindows 98, Pocket PC 2002

DPN_MSGID_RETURN_BUFFER Message

Microsoft® DirectPlay® generates the **DPN_MSGID_RETURN_BUFFER** message when DirectPlay is done with a user buffer.

The **DPNMSG_RETURN_BUFFER** structure contains information for the **DPN_MSGID_RETURN_BUFFER** message.

Syntax

typedef struct _DPNMSG_RETURN_BUFFER { DWORD dwSize; HRESULT hResultCode; PVOID pvBuffer; PVOID pvUserContext;

} DPNMSG_RETURN_BUFFER, *PDPNMSG_RETURN_BUFFER;

Members

dwSize

Size of this structure.

hResultCode

Return value of the operation. This will be set to DPNERR_ENUMRESPONSETOOLARGE if the response to a <u>DPN_MSGID_ENUM_HOSTS_QUERY</u> message is too large.

pvBuffer

Pointer to the buffer being returned.

pvUserContext

Context value associated with the buffer.

Remarks

Return from the message callback function with DPN_OK.

Message Information

Headerdplay8.hMinimum operating systemsWindows 98, Pocket PC 2002

DPN_MSGID_SEND_COMPLETE Message

Microsoft® DirectPlay® generates the **DPN_MSGID_SEND_COMPLETE** message when an asynchronous send message request has completed.

The **DPNMSG_SEND_COMPLETE** structure contains information for the **DPN_MSGID_SEND_COMPLETE** system message.

Syntax

typedef struct _DPNMSG_SEND_COMPLETE {
 DWORD dwSize;
 DPNHANDLE hAsyncOp;
 PVOID pvUserContext;
 HRESULT hResultCode;
 DWORD dwSendTime;
 DWORD dwFirstFrameRTT;
 DWORD dwFirstRetryCount;
 DWORD dwSendCompleteFlags;
 const DPN_BUFFER_DESC pBuffers;
 DWORD dwNumBuffers;
} DPNMSG SEND COMPLETE, *PDPNMSG SEND COMPLETE;

Members

dwSize

Size of this structure.

hAsyncOp

Asynchronous operation handle.

pvUserContext

User context supplied in the <u>IDirectPlay8Client::Send</u>, <u>IDirectPlay8Peer::SendTo</u> and <u>IDirectPlay8Server::SendTo</u> methods.

hResultCode

HRESULT indicating the result of the send message request. **dwSendTime**

Total time, in milliseconds, between send call and completion. dwFirstFrameRTT

The measured round-trip time for this message, if it is available. It will only be available for reliable messages that arrive on the first try and therefore do not need to be retransmitted. For all other messages this field is set to -1. For large messages that span multiple frames, this value will reflect the measured roundtrip time for the first frame sent.

dwFirstRetryCount

For reliable messages, this gives the number of times DirectPlay had to retransmit the message before it was successfully delivered. For unreliable messages, this field is always set to -1.

dwSendCompleteFlags

Specify the following flags that describe how the message was sent.

DPNSENDCOMPLETE_GUARANTEED

The message was sent guaranteed.

DPNSENDCOMPLETE_COALESCED

The message was coalesced for sending.

pBuffers

Pointer to array of <u>DPN_BUFFER_DESC</u> structures sent.

dwNumBuffers

Number of **DPN_BUFFER_DESC** structures in the **pBuffers** array.

Remarks

Return from the message callback function with DPN_OK.

If the DPNSEND_NOCOPY flag was specified when IDirectPlay8Peer::SendTo, IDirectPlay8Client::Send, or IDirectPlay8Server::SendTo was called, the pBuffers and dwNumBuffers members refer to an array that contains the same buffer pointers and sizes that were passed when the send call was made. If the DPNSEND_NOCOPY flag was not specified, pBuffers is set to NULL and dwNumBuffers is set to 0.

Message Information

Headerdplay8.hMinimum operating systemsWindows 98, Pocket PC 2002

 $\ensuremath{\textcircled{}^{\circ}}$ 2003 Microsoft Corporation. All rights reserved.

DPN_MSGID_SERVER_INFO Message

Microsoft® DirectPlay® generates the **DPN_MSGID_SERVER_INFO** message when server data is modified during a client/server session.

The **DPNMSG_SERVER_INFO** structure contains information for the **DPN_MSGID_SERVER_INFO** system message.

Syntax

typedef struct _DPNMSG_SERVER_INFO {
 DWORD dwSize;
 DPNID dpnidServer;
 PVOID pvPlayerContext;
} DPNMSG_SERVER_INFO, *PDPNMSG_SERVER_INFO;

Members

dwSize Size of this structure. dpnidServer DPNID of the server for server information. pvPlayerContext Player context value.

Remarks

Return from the message callback function with DPN_OK.

Message Information

Headerdplay8.hMinimum operating systemsWindows 98, Pocket PC 2002

DPN_MSGID_TERMINATE_SESSION Message

Microsoft® DirectPlay® generates the

DPN_MSGID_TERMINATE_SESSION message when a session is terminated by the host.

The **DPNMSG_TERMINATE_SESSION** structure contains information for the **DPN_MSGID_TERMINATE_SESSION** system message.

Syntax

typedef struct _DPNMSG_TERMINATE_SESSION { DWORD dwSize; HRESULT hResultCode; PVOID pvTerminateData; DWORD dwTerminateDataSize;

} DPNMSG_TERMINATE_SESSION, *PDPNMSG_TERMINATE_SES

Members

dwSize

Size of this structure.

hResultCode

Specifies how the session was terminated. This member is set to <u>DPNERR_HOSTTERMINATEDSESSION</u> if the session was peer-to-peer, and the host called

IDirectPlay8Peer::TerminateSession. If the session was ended by the host calling IDirectPlay8Peer::Close, or if the host stops responding, hResultCode is set to DPNERR_CONNECTIONLOST.

pvTerminateData

Termination data. If **hResultCode** is set to DPNERR_HOSTTERMINATEDSESSION, **pvTerminateData** points to the data block that the host passed through the *pvTerminateData* parameter of

IDirectPlay8Peer::TerminateSession. dwTerminateDataSize

Size of the data block pointed to by **pvTerminateData**. This member will be zero if **pvTerminateData** is set to NULL.

Remarks

Return from the message callback function with DPN_OK.

In a peer-peer game that permits host-migration, if the current host calls **IDirectPlay8Peer::Close** or stops responding, the session does not terminate. Instead, the host migrates and all nonhost players receive a <u>DPN_MSGID_DESTROY_PLAYER</u> message for the host's players, and a <u>DPN_MSGID_HOST_MIGRATE</u> message for the new host. To prevent host migration, the host must shut down the session by calling **IDirectPlay8Peer::TerminateSession**. When the host terminates a session this way, all players receive a **DPN_MSGID_TERMINATE_SESSION** message with **hResultCode** set to DPNERR_HOSTTERMINATEDSESSION. The session will terminate, generating **DPN_MSGID_DESTROY_PLAYER** messages for every player.

In a peer-peer game that does not permit host-migration, the session is terminated if the host calls **IDirectPlay8Peer::Close**, or stops responding. In that case, **DPN_MSGID_TERMINATE_SESSION** is sent to all players with **hResultCode** set to DPNERR_CONNECTIONLOST. The session will terminate, generating **DPN_MSGID_DESTROY_PLAYER** messages for every player.

In a client/server game, the session is also terminated if the host calls <u>IDirectPlay8Server::Close</u> or stops responding. In that case,

DPN_MSGID_TERMINATE_SESSION is sent to all connected clients with hResultCode set to DPNERR_CONNECTIONLOST. The DPN_MSGID_DESTROY_PLAYER message not sent to clients. If the server disconnected by calling IDirectPlay8Server::Close, it will receive DPN_MSGID_DESTROY_PLAYER messages for all players, including its own. Otherwise, the server will only receive DPN_MSGID_DESTROY_PLAYER for the clients' players.

Note The **DPN_MSGID_TERMINATE_SESSION** message typically arrives before any **DPN_MSGID_DESTROY_PLAYER** messages. However, the order of arrival is not guaranteed.

Message Information

Headerdplay8.hMinimum operating systemsWindows 98

DVMSGID_CONNECTRESULT Message

Microsoft® DirectPlay® Voice generates the

DVMSGID_CONNECTRESULT message when the connect request generated through a call to the <u>IDirectPlayVoiceClient::Connect</u> method has completed.

The **DVMSG_CONNECTRESULT** structure contains information for the **DVMSGID_CONNECTRESULT** message.

Syntax

typedef struct {
 DWORD dwSize;
 HRESULT hrResult;
} DVMSG CONNECTRESULT *LPDVMSG CONNECTRESULT, *PD'

Members

dwSize

Size of the **DVMSG_CONNECTRESULT** message structure. **hrResult**

HRESULT that specifies the outcome of the connection attempt.

Remarks

Return from the message callback function with DV_OK.

This message is sent only if the **IDirectPlayVoiceClient::Connect** method is called asynchronously.

Message Information

```
Header dvoice.h
```

 $\ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.

DVMSGID_CREATEVOICEPLAYER Message

Microsoft® DirectPlay® Voice generates the

DVMSGID_CREATEVOICEPLAYER message when a new player joins the voice session.

The **DVMSG_CREATEVOICEPLAYER** structure contains information for the **DVMSGID_CREATEVOICEPLAYER** message.

Syntax

typedef struct {
 DWORD dwSize;
 DVID dvidPlayer;
 DWORD dwFlags;
 PVOID pvPlayerContext;
} DVMSG CREATEVOICEPLAYER *LPDVMSG CREATEVOICEPLAY

Members

dwSize

Size of the this message structure.

dvidPlayer

DVID of the player who connected.

dwFlags

Flag specifying information about the player:

DVPLAYERCAPS_HALFDUPLEX

The specified player is running in half duplex mode. The player will only be able to receive voice, not transmit it. DVPLAYERCAPS_LOCAL

The player is the local player.

pvPlayerContext

Player context value for the player in the voice session. This value is set through this parameter when this message is received.

Remarks

Return from the message callback function with DV_OK.

Upon connecting to a voice session, clients receive one of these messages for each player in the voice session. These messages are sent only to clients in peer-to-peer voice sessions.

The host receives these messages when players join the voice session.

Players do not join the voice session until they have called <u>IDirectPlayVoiceClient::Connect</u>. Therefore, it is possible for a player to be in the transport session but not part of the voice session.

DirectPlay and DirectPlay Voice sessions are separate entities. While there are guarantees about message order for each interface, there are no guarantees about message order between interfaces.

Message Information

Headerdvoice.hMinimum operating systemsWindows 98

 $\ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.

DVMSGID_DELETEVOICEPLAYER Message

For clients, Microsoft® DirectPlay® Voice generates the **DVMSGID_DELETEVOICEPLAYER** message when a player quits the voice session. This message is available only to clients in peer-to-peer voice sessions.

For the host, DirectPlay Voice generates the

DVMSGID_DELETEVOICEPLAYER message when a player quits the voice session.

The **DVMSG_DELETEVOICEPLAYER** structure contains information for the **DVMSGID_DELETEVOICEPLAYER** system message.

Syntax

typedef struct {
 DWORD dwSize;
 DVID dvidPlayer;
 PVOID pvPlayerContext;
} DVMSG_DELETEVOICEPLAYER *LPDVMSG_DELETEVOICEPLAY

Members

dwSize

Size of the **DVMSG_DELETEVOICEPLAYER** message structure.

dvidPlayer

DVID of player who disconnected.

pvPlayerContext

Pointer to the context value set for the player. This value is set through the **pvPlayerContext** member of the <u>DVMSGID_CREATEVOICEPLAYER</u> structure.

Remarks

Return from the message callback function with DV_OK.

Players do not leave the voice session until they have called <u>IDirectPlayVoiceClient::Disconnect</u> or they have disconnected from the transport session. Therefore, a client might be part of the transport session but not part of the voice session.

DirectPlay and DirectPlay Voice sessions are separate entities. While there are guarantees about message order for each interface, there are no guarantees about message order between interfaces.

Message Information

Headerdvoice.hMinimum operating systemsWindows 98

DVMSGID_DISCONNECTRESULT Message

Microsoft® DirectPlay® Voice generates the **DVMSGID_DISCONNECTRESULT** message when the disconnect request generated through a call to the <u>IDirectPlayVoiceClient::Disconnect</u> method has completed.

The **DVMSG_DISCONNECTRESULT** structure contains information for the **DVMSGID_DISCONNECTRESULT** system message.

Syntax

typedef struct { DWORD dwSize; HRESULT hrResult;

} DVMSG_DISCONNECTRESULT *LPDVMSG_DISCONNECTRESU

Members

dwSize

Size of the **DVMSG_DISCONNECTRESULT** message structure.

hrResult

Result of the disconnect request.

Remarks

Return from the message callback function with DV_OK.

This message is sent only if the

IDirectPlayVoiceClient::Disconnect method is called

asynchronously

Message Information

Headerdvoice.hMinimum operating systemsWindows 98
DVMSGID_GAINFOCUS Message

This message is sent to notify you that you have begun capturing audio. There is no data associated with this message.

Remarks

Return from the message callback function with DV_OK.

This message is sent when an application that has lost capture focus recovers it. Refer to the Microsoft® DirectSound® documentation for more information about capturing audio.

Message Information

Minimum operating systems Windows 98

DVMSGID_HOSTMIGRATED Message

Microsoft® DirectPlay® Voice generates the

DVMSGID_HOSTMIGRATED message when the voice host has migrated.

The **DVMSG_HOSTMIGRATED** structure contains information for the **DVMSGID_HOSTMIGRATED** system message.

Syntax

typedef struct { DWORD dwSize; DVID dvidNewHostID; LPDIRECTPLAYVOICESERVER pdvServerInterface; } DVMSG_HOSTMIGRATED *LPDVMSG_HOSTMIGRATED, *PDVM

Members

dwSize

Size of the DVMSG_HOSTMIGRATED message structure. dvidNewHostID

DVID of the new host.

pdvServerInterface

If the local client has become the new voice session host, this member will point to a newly created <u>IDirectPlayVoiceServer</u> object that can be used by the local client for providing host services. If the local client is not the new host, this member will be NULL. If this parameter points to an **IDirectPlayVoiceServer** interface, you must call <u>IDirectPlayVoiceServer::AddRef</u> to increment the interface's reference count. Call <u>IDirectPlayVoiceServer::Release</u> when you no longer need the interface.

Remarks

Return from the message callback function with DV_OK.

DirectPlay and DirectPlay Voice sessions are separate entities. While there are guarantees about message order for each interface, there are no guarantees about message order between interfaces.

Message Information

Headerdvoice.hMinimum operating systemsWindows 98

DVMSGID_INPUTLEVEL Message

Microsoft® DirectPlay® Voice generates the **DVMSGID_INPUTLEVEL** message periodically to notify the user of the input level from the microphone.

The **DVMSG_INPUTLEVEL** structure contains information for the **DVMSGID_INPUTLEVEL** system message.

Syntax

typedef struct { DWORD dwSize; DWORD dwPeakLevel;

LONG IRecordVolume;

PVOID pvLocalPlayerContext;

} DVMSG_INPUTLEVEL *LPDVMSG_INPUTLEVEL, *PDVMSG_INPU

Members

dwSize

Size of the **DVMSG_INPUTLEVEL** message structure.

dwPeakLevel

Integer value representing peak level across the current frame, which corresponds to approximately 1/10 second of audio stream. The current frame typically lags 50-200 ms behind real-time. This value can range from 0 through 99, with 0 being completely silent and 99 being the highest possible input level.

IRecordVolume

Current recording volume for the client. The value can range from -10,000 to 0. This member is available even when automatic gain control is active.

pvLocalPlayerContext

Pointer to the context value set for the local player. This value is set through the **pvPlayerContext** member of the

DVMSGID_CREATEVOICEPLAYER structure.

Remarks

Return from the message callback function with DV_OK.

The period of notification is set by the **dwNotifyPeriod** member of the <u>DVCLIENTCONFIG</u> structure. If the notification period is set to 0, this message will not be sent. In addition, if the client is running in half duplex mode, this message is not available.

Message Information

Headerdvoice.hMinimum operating systemsWindows 98

DVMSGID_LOCALHOSTSETUP Message

The **DVMSGID_LOCALHOSTSETUP** message is sent when the local client is selected as the new voice host during host migration.

The **DVMSG_LOCALHOSTSETUP** structure contains information for the **DVMSGID_LOCALHOSTSETUP** system message.

Syntax

typedef struct {
 DWORD dwSize;
 PVOID pvContext;
 PDVMESSAGEHANDLER pMessageHandler;
} DVMSG_LOCALHOSTSETUP *LPDVMSG_LOCALHOSTSETUP, *F

Members

dwSize

Size of the **DVMSG_LOCALHOSTSETUP** message structure. **pvContext**

Pointer to the context value you want to set for the new server. **pMessageHandler**

Pointer to the callback function to be used for the new server.

Remarks

Return from the message callback function with DV_OK.

The message is sent before the <u>DVMSGID_HOSTMIGRATED</u> message and gives you the chance to set the callback function and context value that will be used when creating the new host object. If you do not set either of the values, then the new server interface will have no callback function. When the application returns from handling this message, it will receive the **DVMSGID_HOSTMIGRATED** message.

Message Information

Headerdvoice.hMinimum operating systemsWindows 98

DVMSGID_LOSTFOCUS Message

The **DVMSGID_LOSTFOCUS** message is sent to notify you that you have stopped capturing audio. There is no data associated with this message.

Remarks

Return from the message callback function with DV_OK.

This message is sent when an application that has capture focus loses it to another application. Refer to the Microsoft® DirectSound® documentation for more information about capturing audio.

Message Information

Minimum operating systems Windows 98

DVMSGID_OUTPUTLEVEL Message

Microsoft® DirectPlay® Voice generates the **DVMSGID OUTPUTLEVEL** message periodically to notify the user of the output level of playback.

The DVMSG_OUTPUTLEVEL structure contains information for the **DVMSGID OUTPUTLEVEL** system message.

Syntax

typedef struct { DWORD dwSize: DWORD dwPeakLevel; LONG lOutputVolume; PVOID pvLocalPlayerContext;

} DVMSG OUTPUTLEVEL *LPDVMSG OUTPUTLEVEL, *PDVMSG

Members

dwSize

Size of the **DVMSG OUTPUTLEVEL** message structure. dwPeakLevel

Integer representing the current output level of playback. This value must be in the range from 0 through 99. 0 indicates complete silence and 99 indicates the highest possible output level.

IOutputVolume

Current playback volume for the client.

pvLocalPlayerContext

Pointer to the context value set for the local player. This value is set through the **pvPlayerContext** member of the **DVMSG CREATEVOICEPLAYER** message structure.

Remarks

Return from the message callback function with DV_OK.

The period of notification is set by the **dwNotifyPeriod** member of the <u>DVCLIENTCONFIG</u> structure. If the notification period is set to 0, this message will not be sent.

Message Information

Headerdvoice.hMinimum operating systemsWindows 98

DVMSGID_PLAYEROUTPUTLEVEL Message

Microsoft® DirectPlay® Voice generates the

DVMSGID_PLAYEROUTPUTLEVEL message periodically to notify the user of the output level of an individual player's voice stream.

The **DVMSG_PLAYEROUTPUTLEVEL** structure contains information for the **DVMSGID_PLAYEROUTPUTLEVEL** system message.

Syntax

typedef struct { DWORD dwSize; DVID dvidSourcePlayerID; DWORD dwPeakLevel; PVOID pvPlayerContext;

} DVMSG PLAYEROUTPUTLEVEL *LPDVMSG PLAYEROUTPUTLE

Members

dwSize

Size of the **DVMSG_PLAYEROUTPUTLEVEL** message structure.

dvidSourcePlayerID

DVID of the player whose voice is being played back.

dwPeakLevel

Integer representing the current output level of the player's voice stream. This value must be in the range from 0 through 99, with 0 being completely silent and 99 being the highest possible output level.

pvPlayerContext

Pointer to the context value set for the player. This value is set through the **pvPlayerContext** member of the **DVMSGID_CREATEVOICEPLAYER** message structure.

Remarks

Return from the message callback function with DV_OK.

This message is generated while voice is being played back for an individual player. If multiple player voices are being played, one message for each player speaking will be sent each notification period.

The period of notification is set by the **dwNotifyPeriod** member of the <u>DVCLIENTCONFIG</u> structure. If the notification period is set to 0, this message will not be sent.

Message Information

Headerdvoice.hMinimum operating systemsWindows 98

DVMSGID_PLAYERVOICESTART Message

Microsoft® DirectPlay® Voice generates the

DVMSGID_PLAYERVOICESTART message when an incoming audio stream begins playing back.

The **DVMSG_PLAYERVOICESTART** structure contains information for the **DVMSGID_PLAYERVOICESTART** system message.

Syntax

typedef struct {
 DWORD dwSize;
 DVID dvidSourcePlayerID;
 PVOID pvPlayerContext;
} DVMSG_PLAYERVOICESTART *LPDVMSG_PLAYERVOICESTAR1

Members

dwSize

Size of the DVMSG_PLAYERVOICESTART message structure. dvidSourcePlayerID

DVID of the player where the voice transmission originated.

pvPlayerContext

Pointer to the context value set for the player. This value is set through the **pvPlayerContext** member of the **DVMSGID_CREATEVOICEPLAYER** message structure.

Remarks

Return from the message callback function with DV_OK.

Message Information

Header dvoice.h

 $\ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.

DVMSGID_PLAYERVOICESTOP Message

Microsoft® DirectPlay® Voice generates the

DVMSGID_PLAYERVOICESTOP message when an incoming audio stream stops.

The **DVMSG_PLAYERVOICESTOP** structure contains information for the **DVMSGID_PLAYERVOICESTOP** system message.

Syntax

typedef struct {
 DWORD dwSize;
 DVID dvidSourcePlayerID;
 PVOID pvPlayerContext;
} DVMSG_PLAYERVOICESTOP *LPDVMSG_PLAYERVOICESTOP, *

Members

dwSize

Size of the **DVMSG_PLAYERVOICESTOP** message structure. **dvidSourcePlayerID**

DVID of the player where the voice transmission originated.

pvPlayerContext

Pointer to the context value set for the player. This value is set through the **pvPlayerContext** member of the **DVMSGID_CREATEVOICEPLAYER** message structure.

Remarks

Return from the message callback function with DV_OK.

Message Information

Header dvoice.h

 $\ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.

DVMSGID_RECORDSTART Message

Microsoft® DirectPlay® Voice generates the

DVMSGID_RECORDSTART message when audio input on the local client begins.

The **DVMSG_RECORDSTART** structure contains information for the **DVMSGID_RECORDSTART** system message.

Syntax

typedef struct { DWORD dwSize; DWORD dwPeakLevel; PVOID pvLocalPlayerContext;

} DVMSG_RECORDSTART *LPDVMSG_RECORDSTART, *PDVMSG

Members

dwSize

Size of the **DVMSG_RECORDSTART** message structure. **dwPeakLevel**

Voice activation level that caused the transmission to begin. In push-to-talk mode, this value is 0.

pvLocalPlayerContext

Pointer to the context value set for the player. This value is set through the **pvPlayerContext** member of the **DVMSGID_CREATEVOICEPLAYER** message structure.

Remarks

Return from the message callback function with DV_OK.

This message can be sent, for instance, when the voice activation

sensitivity level is exceeded or when a valid target is specified in push-to-talk mode.

Message Information

Headerdvoice.hMinimum operating systemsWindows 98

DVMSGID_RECORDSTOP Message

Microsoft® DirectPlay® Voice generates the **DVMSGID_RECORDSTOP** message when audio input on the local client stops.

The **DVMSG_RECORDSTOP** structure contains information for the **DVMSGID_RECORDSTOP** system message.

Syntax

typedef struct { DWORD dwSize; DWORD dwPeakLevel; PVOID pvLocalPlayerContext;

} DVMSG_RECORDSTOP *LPDVMSG_RECORDSTOP, *PDVMSG_I

Members

dwSize

Size of the **DVMSG_RECORDSTOP** message structure.

dwPeakLevel

Voice activation level that caused the transmission to stop. In push-to-talk mode, this value is 0.

pvLocalPlayerContext

Pointer to the context value set for the player. This value is set through the **pvPlayerContext** member of the **DVMSGID_CREATEVOICEPLAYER** message structure.

Remarks

Return from the message callback function with DV_OK.

This message can be generated when the voice activation sensitivity level is not being reached or when a target is deselected in push-totalk mode.

Message Information

Headerdvoice.hMinimum operating systemsWindows 98

DVMSGID_SESSIONLOST Message

Microsoft® DirectPlay® Voice generates the **DVMSGID_SESSIONLOST** message when the voice session terminates.

The **DVMSG_SESSIONLOST** structure contains information for the **DVMSGID_SESSIONLOST** system message.

Syntax

typedef struct {
 DWORD dwSize;
 HRESULT hrResult;
} DVMSG_SESSIONLOST *LPDVMSG_SESSIONLOST, *PDVMSG_:

Members

dwSize Size of the DVMSG_SESSIONLOST message structure. hrResult HRESULT value indicating why the session was terminated.

Remarks

Return from the message callback function with DV_OK.

DirectPlay and DirectPlay Voice sessions are separate entities. While there are guarantees about message order for each interface, there are no guarantees about message order between interfaces.

Message Information

Headerdvoice.hMinimum operating systemsWindows 98

 $\ensuremath{\textcircled{\sc 0}}$ 2003 Microsoft Corporation. All rights reserved.

DVMSGID_SETTARGETS Message

Microsoft® DirectPlay® Voice generates the **DVMSGID_SETTARGETS** message when the <u>IDirectPlayVoiceClient::SetTransmitTargets</u> or <u>IDirectPlayVoiceServer::SetTransmitTargets</u> methods are called.

The **DVMSG_SETTARGETS** structure contains information for the **DVMSGID_SETTARGETS** system message.

Syntax

typedef struct {
 DWORD dwSize;
 DWORD dwNumTargets;
 PDVID pdvidTargets;
} DVMSG_SETTARGETS *LPDVMSG_SETTARGETS, *PDVMSG_SE

Members

dwSize

Size of the **DVMSG_SETTARGETS** message structure.

dwNumTargets

Number of DVIDs contained in the **pdvidTargets** member.

pdvidTargets

Array of DVIDs specifying the set targets. This can also be set to NULL if there are no targets.

Remarks

Return from the message callback function with DV_OK.

Message Information

Headerdvoice.hMinimum operating systemsWindows 98

 $\ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.
Structures

This section contains ${\tt Microsoft} \ensuremath{\mathbb{R}}$ ${\tt DirectPlay} \ensuremath{\mathbb{R}}$ structures.

Structures

- DP8SIM_PARAMETERS
- DP8SIM_STATISTICS
- DPL_APPLICATION_INFO
- DPL_CONNECT_INFO
- <u>DPL_CONNECTION_SETTINGS</u>
- DPL_PROGRAM_DESC
- DPN_APPLICATION_DESC
- DPN_BUFFER_DESC
- <u>DPN_CAPS</u>
- <u>DPN_CAPS_EX</u>
- <u>DPN_CONNECTION_INFO</u>
- DPN_GROUP_INFO
- DPN_PLAYER_INFO
- <u>DPN_SECURITY_CREDENTIALS</u>
- DPN_SECURITY_DESC
- <u>DPN_SERVICE_PROVIDER_INFO</u>
- <u>DPN_SP_CAPS</u>
- <u>DPNHCAPS</u>
- <u>DVCAPS</u>
- DVCLIENTCONFIG
- DVCOMPRESSIONINFO
- DVSESSIONDESC
- DVSOUNDDEVICECONFIG

^{© 2003} Microsoft Corporation. All rights reserved.

DP8SIM_PARAMETERS Structure

Used to set and retrieve DP8Sim settings.

Syntax

typedef struct _DP8SIM_PARAMTERS {
 DWORD dwSize;
 DWORD dwFlags;
 DWORD dwPacketHeaderSize;
 DWORD dwBandwidthBPS;
 FLOAT fPcketLossPercent;
 DWORD dwMinLatencyMS;
 DWORD dwMaxLatencyMS;
} DP8SIM_PARAMETERS, *PDP8SIM_PARAMETERS;

Members

dwSize

Must be set the to size of this structure, in bytes, before using this structure.

dwFlags

Unused. Must be zero.

dwPacketHeaderSize

Fixed transport packet header size for packet size calculations, or 0 to calculate based on Microsoft® DirectPlay® headers and payload only. The following values are defined for convenience. DP8SIMPACKETHEADERSIZE_IP

Represents the standard Internet Protocol (IP) header size, 20 bytes.

DP8SIMPACKETHEADERSIZE_UDP

Represents the standard User Datagram Protocol (UDP) header size, 8 bytes.

DP8SIMPACKETHEADERSIZE_IP_UDP

Represents the standard IP and UDP header size, 28 bytes. This is the recommended value.

dwBandwidthBPS

Bandwidth limit in bytes per second. Set to 0 for no limit.

fPcketLossPercent

Percentage of packets to drop. Can be a value between 0.0 and 100.0.

dwMinLatencyMS

Minimum artificial latency, in milliseconds. This value is in addition to any latency cause by bandwidth settings.

dwMaxLatencyMS

Maximum artificial latency, in milliseconds. This value is in addition to any latency cause by bandwidth settings.

Structure Information

Headerdp8sim.hMinimum operating systemsWindows 98, Pocket PC 2002

© 2003 Microsoft Corporation. All rights reserved.

Used to set and retrieve DP8Sim statistics.

Syntax

typedef struct _DP8SIM_STATISTICS {
 DWORD dwSize;
 DWORD dwFlags;
 DWORD dwTransmittedPackets;
 DWORD dwTransmittedBytes;
 DWORD dwDroppedPackets;
 DWORD dwDroppedBytes;
 DWORD dwTotalDelayMS;
} DP8SIM_STATISTICS, *PDP8SIM_STATISTICS;

Members

dwSize

Must be set the to size of this structure, in bytes, before using this structure.

dwFlags

Reserved. Must be zero.

dwTransmittedPackets

Number of packets sent or received.

dwTransmittedBytes

Number of bytes sent or received.

dwDroppedPackets

Number of packets intentionally dropped.

dwDroppedBytes

Number of bytes intentionally dropped.

dwTotalDelayMS

Total number of milliseconds delay added due to bandwidth limitations or random latency settings.

Structure Information

Headerdp8sim.hMinimum operating systemsWindows 98, Pocket PC 2002

© 2003 Microsoft Corporation. All rights reserved.

DPL_APPLICATION_INFO Structure

Used in the *pEnumData* parameter of the

IDirectPlay8LobbyClient::EnumLocalPrograms method to describe the lobbied application.

Syntax

typedef struct _DPL_APPLICATION_INFO {
 GUID guidApplication;
 PWSTR pwszApplicationName;
 DWORD dwNumRunning;
 DWORD dwNumWaiting;
 DWORD dwFlags;
} DPL_APPLICATION_INFO, *PDPL_APPLICATION_INFO;

Members

guidApplication

Variable of type **GUID** specifying the lobbied application.

pwszApplicationName

Pointer to a variable of type **WSTR** containing the name of the lobbied application.

dwNumRunning

Number of instances of the application.

dwNumWaiting

Number of clients waiting to connect to the lobbied application.

dwFlags

Reserved. Must be 0.

Structure Information

Headerdplobby8.hMinimum operating systemsWindows 98, Pocket PC 2002

 $\ensuremath{\textcircled{\sc 0}}$ 2003 Microsoft Corporation. All rights reserved.

DPL_CONNECT_INFO Structure

Used to specify connection information for a lobby client when connecting to the lobby application in the

IDirectPlay8LobbyClient::ConnectApplication method.

Syntax

typedef struct _DPL_CONNECT_INFO {
 DWORD dwSize;
 DWORD dwFlags;
 GUID guidApplication;
 PDPL_CONNECTION_SETTINGS pdplConnectionSettings;
 PVOID pvLobbyConnectData;
 DWORD dwLobbyConnectDataSize;
} DPL_CONNECT_INFO, *PDPL_CONNECT_INFO;

Members

dwSize

Size of the **DPL_CONNECT_INFO** structure. The application must set this member before it uses the structure.

dwFlags

One of the following flags, which determine connection behavior. DPLCONNECT_LAUNCHNEW

Launches a new instance of the application.

DPLCONNECT_LAUNCHNOTFOUND

Launches a new instance of the application only if there is currently no application running that can supply launch settings.

guidApplication

Variable of type globally unique identifier (GUID) specifying the application.

pdplConnectionSettings

Contains the connection settings you want to associate with the connection when it is established.

pvLobbyConnectData

Pointer to connection data passed to the lobbied application.

dwLobbyConnectDataSize

Variable of type **DWORD** specifying the size of the data buffer in the **pvLobbyConnectData** member.

Structure Information

Headerdplobby8.hMinimum operating systemsWindows 98, Pocket PC 2002

© 2003 Microsoft Corporation. All rights reserved.

DPL_CONNECTION_SETTINGS Structure

Used to specify the settings you want to associate with a connection. These settings contain all the information required to create, initialize and connect/host a Microsoft® DirectPlay® object.

Syntax

typedef struct _DPL_CONNECTION_SETTINGS {
 DWORD dwSize;
 DWORD dwFlags;
 DPN_APPLICATION_DESC dpnAppDesc;
 IDirectPlay8Address *pdp8HostAddress;
 IDirectPlay8Address *pdp8DeviceAddresses;
 DWORD cNumDeviceAddresses;
 PWSTR pwszPlayerName;
} DPL CONNECTION SETTINGS, *PDPL CONNECTION SETTING

Members

dwSize

Size of the **DPL_CONNECTION_SETTINGS** structure. The application must set this to sizeof(

DPL_CONNECTION_SETTINGS) before using this structure.

dwFlags

The following flag can be specified.

DPLCONNECTSETTINGS_HOST

The application should host the session.

dpnAppDesc

Pointer to the application description that should be passed to the **Connect** or **Host** call when DirectPlay initialized.

pdp8HostAddress

If DPLCONNECTSETTINGS_HOST is not specified, this is the address of the session the client should connect to. If DPLCONNECTSETTINGS_HOST is specified, this member must be NULL.

ppdp8DeviceAddresses

This structure contains an array of pointers to device addresses. If DPLCONNECTSETTINGS_HOST is specified, this member will contain the addresses the host should listen on. If DPLCONNECTSETTINGS_HOST is not specified, this member will contain the address of the devices the client should use when connecting.

cNumDeviceAddresses

Number of addresses specified in the **ppdp8DeviceAddresses** member.

pwszPlayerName

Can be used to pass the player name you want the DirectPlay object to use when launching. This member can be NULL.

Structure Information

Headerdplobby8.hMinimum operating systemsWindows 98, Pocket PC 2002

© 2003 Microsoft Corporation. All rights reserved.

DPL_PROGRAM_DESC Structure

Describes a Microsoft® DirectPlay® lobby-aware application.

Syntax

typedef struct _DPL_PROGRAM_DESC {
 DWORD dwSize;
 DWORD dwFlags;
 GUID guidApplication;
 PWSTR pwszApplicationName;
 PWSTR pwszCommandLine;
 PWSTR pwszCommandLine;
 PWSTR pwszDescription;
 PWSTR pwszDescription;
 PWSTR pwszExecutableFilename;
 PWSTR pwszExecutablePath;
 PWSTR pwszLauncherFilename;
 PWSTR pwszLauncherPath;
} DPL_PROGRAM_DESC, *PDPL_PROGRAM_DESC;

Members

dwSize

Size of the **DPL_PROGRAM_DESC** structure. The application must set this member before it uses the structure.

dwFlags

Reserved. Must be 0.

guidApplication

Variable of type **GUID** specifying the application.

pwszApplicationName

Pointer to the application name.

pwszCommandLine

Pointer to the command-line arguments.

pwszCurrentDirectory

Pointer to the directory that should be set as the application's working directory..

pwszDescription

Pointer to the application description.

pwszExecutableFilename

Pointer to the file name of the application executable.

pwszExecutablePath

Pointer to the path of the application executable.

pwszLauncherFilename

Pointer to the file name of the launcher executable.

pwszLauncherPath

Pointer to the path of the launcher executable.

Structure Information

Headerdplobby8.hMinimum operating systemsWindows 98, Pocket PC 2002

© 2003 Microsoft Corporation. All rights reserved.

DPN_APPLICATION_DESC Structure

Describes the settings for a Microsoft® DirectPlay® application.

Syntax

typedef struct _DPN_APPLICATION_DESC {
 DWORD dwSize;
 DWORD dwFlags;
 GUID guidInstance;
 GUID guidApplication;
 DWORD dwMaxPlayers;
 DWORD dwCurrentPlayers;
 WCHAR *pwszSessionName;
 WCHAR *pwszPassword;
 PVOID pvReservedData;
 DWORD dwReservedDataSize;
 PVOID pvApplicationReservedDataSize;
} DPN_APPLICATION_DESC, *PDPN_APPLICATION_DESC;

Members

dwSize

Size of the **DPN_APPLICATION_DESC** structure. The application must set this member before it uses the structure.

dwFlags

One of the following flags describing application behavior. DPNSESSION_CLIENT_SERVER

This type of session is client/server. This flag cannot be combined with DPNSESSION_MIGRATE_HOST.

DPNSESSION_MIGRATE_HOST

Used in peer-to-peer sessions, enables host migration. This flag cannot be combined with

DPNSESSION_CLIENT_SERVER.

DPNSESSION_NODPNSVR

Do not forward enumerations to your host from DPNSVR. See <u>Using the DirectPlay DPNSVR Application</u> for details. DPNSESSION REQUIREPASSWORD

The session is password protected. If this flag is set, **pwszPassword** must be set to a valid string.

DPNSESSION_NOENUMS

Do not allow DirectPlay to start enumeration queries. DPNSESSION_FAST_SIGNED

Add an 8 byte value to each packet. The packet must contain this value before a receiver accepts it.

DPNSESSION_FULL_SIGNED

Add a rolling 8 byte SHA1 hash value based on the contents of the packet. The packet must have the correct SHA1 signature before it is accepted.

guidInstance

Globally unique identifier (GUID) that is generated by DirectPlay at startup . This member is an [out] parameter when calling <u>IDirectPlay8Peer::GetApplicationDesc</u>,

IDirectPlay8Client::GetApplicationDesc, or

IDirectPlay8Server::GetApplicationDesc. It is an optional [in] parameter when calling the IDirectPlay8Peer::Connect and IDirectPlay8Client::Connect methods. It must be set to GUID NULL when you call the

IDirectPlay8Peer::SetApplicationDesc or

IDirectPlay8Server::SetApplicationDesc methods. You cannot obtain this GUID by calling the IDirectPlay8Server::Host or IDirectPlay8Peer::Host methods. You must obtain the GUID by calling a GetApplicationDesc method.

guidApplication

Application **GUID**.

dwMaxPlayers

Variable of type **DWORD**, specifying the maximum number of players allowed in the session. Set this member to 0 to specify an unlimited number of players.

dwCurrentPlayers

Variable of type **DWORD** specifying the number of players currently connected to the session. This member is is set only by the **IDirectPlay8Peer::GetApplicationDesc**, **IDirectPlay8Client::GetApplicationDesc**, and IDirectPlay8Server::GetApplicationDesc methods.

pwszSessionName

Pointer to a variable of type **WCHAR** specifying the name of the session. This member is set by the host or server only for informational purposes. A client cannot use this name to connect to a host or server.

pwszPassword

Pointer to a variable of type **WCHAR** specifying the Unicode password that is required to connect to the session. This must be NULL if the DPNSESSION_REQUIREPASSWORD is not set in the **dwFlags** member.

pvReservedData

Pointer to DirectPlay reserved data. An application should never modify this value.

dwReservedDataSize

Variable of type **DWORD** specifying the size of data contained in the **pvReservedData** member. An application should never modify this value.

pvApplicationReservedData

Pointer to application-specific reserved data. This value is optional and may be set to NULL.

dwApplicationReservedDataSize

Variable of type **DWORD** specifying the size of the data in the **pvApplicationReservedData** member. This value is optional and may be set to 0.

Remarks

Multiple instances of the application can run simultaneously in a session. If multiple instances are running, each will have a unique **DPN_APPLICATION_DESC** structure associated with it. "Application" refers to a specific instance of an application.

The dwMaxPlayers, pvApplicationReservedData, dwApplicationReservedDataSize, pwszPassword, and pwszSessionName members can be set when calling the

IDirectPlay8Peer::Host, IDirectPlay8Server::Host, IDirectPlay8Peer::SetApplicationDesc, or IDirectPlay8Server::SetApplicationDesc methods.

Setting either the DPNSESSION_FAST_SIGNED or the DPNSESSION_FULL_SIGNED flag prevents players using Microsoft DirectX® 8.1 or earlier versions from being able to connect to the session. These flags cannot be used together.

Security Alert When connecting to a passwordprotected session, the data in the **pwszPassword** member is transmitted in clear text to the host.

Structure Information

Headerdplay8.hMinimum operating systemsWindows 98, Pocket PC 2002

© 2003 Microsoft Corporation. All rights reserved.

DPN_BUFFER_DESC Structure

Used by Microsoft® DirectPlay® for generic buffer information.

Syntax

typedef struct _BUFFERDESC {
 DWORD dwBufferSize;
 BYTE *pBufferData;
} DPN_BUFFER_DESC, BUFFERDESC;

Members

dwBufferSize

Variable of type **DWORD** that specifies the size of the data buffer in the **pBufferData** member.

pBufferData

Pointer to a variable of type **BYTE** that contains the buffer data.

Structure Information

Headerdplay8.hMinimum operating systemsWindows 98, Pocket PC 2002

© 2003 Microsoft Corporation. All rights reserved.

DPN_CAPS Structure

Used when setting and retrieving general parameters for Microsoft® DirectPlay®.

Syntax

```
typedef struct _DPN_CAPS {
   DWORD dwSize;
   DWORD dwFlags;
   DWORD dwConnectTimeout;
   DWORD dwConnectRetries;
   DWORD dwTimeoutUntilKeepAlive;
} DPN_CAPS, *PDPN_CAPS;
```

Members

dwSize

This value must be set to the size of the structure.

dwFlags

Reserved, this must be 0.

dwConnectTimeout

Number of milliseconds DirectPlay should wait before it retries a connection request.

dwConnectRetries

Number of connection retries DirectPlay should make during the connection process.

dwTimeoutUntilKeepAlive

Number of milliseconds DirectPlay waits since the last time it received a packet from an endpoint, before it sends a keep alive message. The actual delay can vary from the specified value to as much as twice the specified value.

Structure Information

Header dplay8.h

 $\ensuremath{\textcircled{}^{\circ}}$ 2003 Microsoft Corporation. All rights reserved.

Used when setting and retrieving parameters to tune the Microsoft® DirectPlay® protocol.

Syntax

typedef struct _DPN_CAPS_EX { DWORD dwSize; DWORD dwFlags; DWORD dwConnectTimeout; DWORD dwConnectRetries; DWORD dwConnectRetries; DWORD dwTimeoutUntilKeepAlive; DWORD dwMaxRecvMsgSize; DWORD dwMaxRecvMsgSize; DWORD dwNumSendRetries; DWORD dwNumSendRetries; DWORD dwNumSendRetryInterval; DWORD dwDropThresholdRate; DWORD dwDropThresholdRate; DWORD dwThrottleRate; DWORD dwNumHardDisconnectSends; DWORD dwMaxHardDisconnectPeriod; } DPN CAPS EX, *PDPN CAPS EX;

Members

dwSize

Value, which must be set to the size of the structure.

dwFlags

Reserved; this must be 0.

dwConnectTimeout

Number of milliseconds DirectPlay should wait before it retries a connection request.

dwConnectRetries

Number of connection retries DirectPlay should make during the connection process.

dwTimeoutUntilKeepAlive

Number of milliseconds DirectPlay waits since the last time it

received a packet from an endpoint, before it sends a keep-alive message. The actual delay can vary from the specified value to as much as twice the specified value.

dwMaxRecvMsgSize

Maximum size of received messages, in bytes.

dwNumSendRetries

Maximum number of times applications can try to send a message before the connection is considered dead.

dwMaxSendRetryInterval

Maximum number of milliseconds between attempts to resend messages.

dwDropThresholdRate

Percentage of dropped packets allowed before <u>Message</u> <u>Throttling</u> is applied.

dwThrottleRate

Percentage to reduce the send queue when applying message throttling.

dwNumHardDisconnectSends

Number of hard disconnect frames to send when the DPNCLOSE_IMMEDIATE flag is specified.

dwMaxHardDisconnectPeriod

Maximum time, in milliseconds, between each hard disconnect frame sent.

Remarks

When using the **DPN_CAPS_EX** structure with <u>IDirectPlay8Peer</u> or

IDirectPlay8Client objects, the dwMaxRecvMsgSize member is

ignored. Callback functions for IDirectPlay8Peer and

IDirectPlay8Client will receive messages of any size, even those larger than **dwMaxRecvMsgSize**.

Structure Information

Headerdplay8.hMinimum operating systemsWindows 98, Pocket PC 2002

 $\ensuremath{\textcircled{\sc 0}}$ 2003 Microsoft Corporation. All rights reserved.
DPN_CONNECTION_INFO Structure

Used to retrieve statistics for the connection between you and a remote computer that you are connected to.

Syntax

typedef struct DPN CONNECTION INFO { DWORD dwSize: DWORD dwRoundTripLatencyMS; DWORD dwThroughputBPS; DWORD dwPeakThroughputBPS; DWORD dwBytesSentGuaranteed; DWORD dwPacketsSentGuaranteed; DWORD dwBytesSentNonGuaranteed; DWORD dwPacketsSentNonGuaranteed; DWORD dwBytesRetried; DWORD dwPacketsRetried: DWORD dwBytesDropped; DWORD dwPacketsDropped; DWORD dwMessagesTransmittedHighPriority; DWORD dwMessagesTimedOutHighPriority; DWORD dwMessagesTransmittedNormalPriority; DWORD dwMessagesTimedOutNormalPriority; DWORD dwMessagesTransmittedLowPriority; DWORD dwMessagesTimedOutLowPriority; DWORD dwBytesReceivedGuaranteed; DWORD dwPacketsReceivedGuaranteed; DWORD dwBytesReceivedNonGuaranteed; DWORD dwPacketsReceivedNonGuaranteed; DWORD dwMessagesReceived; } DPN CONNECTION INFO, *PDPN CONNECTION INFO;

Members

dwSize

Size of the structure.

dwRoundTripLatencyMS

Approximate time, in milliseconds (ms), it takes a packet to reach the remote computer and be returned to the local computer. This number will change throughout the session as link conditions change.

dwThroughputBPS

Approximate throughput, in bytes per second (Bps), for the link. This number will change throughout the session as link conditions change. This value is approximate, and you may want to calculate your own value for greater accuracy.

dwPeakThroughputBPS

Peak throughput, in bytes per second (Bps) for the link. This number will change throughout the session as link conditions change. This value is approximate, and you may want to calculate your own value for greater accuracy.

dwBytesSentGuaranteed

Amount, in bytes, of guaranteed messages that have been sent.

dwPacketsSentGuaranteed

Number of packets of guaranteed messages that have been sent.

dwBytesSentNonGuaranteed

Amount, in bytes, of nonguaranteed messages that have been sent.

dwPacketsSentNonGuaranteed

Number of packets of nonguaranteed messages that have been sent.

dwBytesRetried

Amount, in bytes, of messages that have been retried.

dwPacketsRetried

Amount of packets that have been retried.

dwBytesDropped

Amount, in bytes, of messages that have been dropped.

dwPacketsDropped

Number of packets that have been dropped.

dwMessagesTransmittedHighPriority

Number of high-priority messages that have been transmitted.

dwMessagesTimedOutHighPriority

Number of high-priority messages that have timed out.

dwMessagesTransmittedNormalPriority

Number of normal-priority messages that have been transmitted.

dwMessagesTimedOutNormalPriority

Number of normal-priority messages that have timed out.

dwMessagesTransmittedLowPriority

Number of low-priority messages that have been transmitted.

dwMessagesTimedOutLowPriority

Number of low priority messages that have timed out.

dwBytesReceivedGuaranteed

Amount, in bytes, of guaranteed messages that have been received.

dwPacketsReceivedGuaranteed

Number of packets of guaranteed messages that have been received.

dwBytesReceivedNonGuaranteed

Amount, in bytes, of nonguaranteed messages that have been received.

dwPacketsReceivedNonGuaranteed

Number of packets of nonguaranteed messages that have been received.

dwMessagesReceived

Number of messages that have been received.

Structure Information

Headerdplay8.hMinimum operating systemsWindows 98, Pocket PC 2002

 $\ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.

Describes static group information.

Syntax

typedef struct _DPN_GROUP_INFO {
 DWORD dwSize;
 DWORD dwInfoFlags;
 PWSTR pwszName;
 PVOID pvData;
 DWORD dwDataSize;
 DWORD dwGroupFlags;
} DPN_GROUP_INFO, *PDPN_GROUP_INFO;

Members

dwSize

Variable of type **DWORD** describing the size of this structure. **dwInfoFlags**

Variable of type **DWORD** containing flags that specify the type of information contained in this structure. When the IDirectPlay8Peer::GetGroupInfo or

IDirectPlay8Server::GetGroupInfo method returns, the dwInfoFlags member of the DPN_GROUP_INFO will always have both flags set, even if the corresponding pointers are set to NULL. These flags are used when calling IDirectPlay8Peer::SetGroupInfo, to notify Microsoft® DirectPlay® of which values have changed.

DPNINFO_NAME

The **pwszName** member contains valid data. DPNINFO_DATA

The **pvData** member contains valid data.

pwszName

Pointer to a variable of type **PWSTR** specifying the Unicode name of the group.

pvData

Pointer to the data describing the group.

dwDataSize

Variable of type **DWORD** that specifies the size of the data contained in the **pvData** member.

dwGroupFlags

Variable of type DWORD that can be set to the following description flag.

DPNGROUP_AUTODESTRUCT

Causes the group to be automatically destroyed when the group creator leaves the group.

Remarks

When using this structure in the IDirectPlay8Peer::GetGroupInfo and IDirectPlay8Server::GetGroupInfo methods, dwInfoFlags must be set to 0.

Structure Information

Headerdplay8.hMinimum operating systemsWindows 98, Pocket PC 2002

© 2003 Microsoft Corporation. All rights reserved.

Describes static player information.

Syntax

typedef struct _DPN_PLAYER_INFO {
 DWORD dwSize;
 DWORD dwInfoFlags;
 PWSTR pwszName;
 PVOID pvData;
 DWORD dwDataSize;
 DWORD dwPlayerFlags;
} DPN_PLAYER_INFO, *PDPN_PLAYER_INFO;

Members

dwSize

Variable of type **DWORD** describing the size of this structure. **dwInfoFlags**

Variable of type **DWORD** containing flags that specify the type of information contained in this structure. When the IDirectPlay8Peer::GetPeerInfo or

IDirectPlay8Server::GetClientInfo method returns, the

dwInfoFlags member of the **DPN_PLAYER_INFO** will always have both flags set, even if the corresponding pointers are set to NULL. These flags are used when calling

<u>IDirectPlay8Peer::SetPeerInfo</u>, to notify Microsoft® DirectPlay® which values have changed.

DPNINFO_NAME

The **pwszName** member contains valid data.

DPNINFO_DATA

The **pvData** member contains valid data.

pwszName

Pointer to a variable of type **PWSTR** specifying the Unicode name of the player.

pvData

Pointer to the data describing the player.

dwDataSize

Variable of type **DWORD** that specifies the size of the data contained in the **pvData** member.

dwPlayerFlags

Variable of type **DWORD** that may contain one of the following flags.

DPNPLAYER_LOCAL

This information is for the local player.

DPNPLAYER_HOST

This player is the host for the application.

Remarks

When using this structure in the IDirectPlay8Peer::GetPeerInfo and IDirectPlay8Server::GetClientInfo methods, dwInfoFlags must be set to 0.

When using this structure in the <a>IDirectPlay8Client::SetClientInfo,

IDirectPlay8Peer::SetPeerInfo, or

IDirectPlay8Server::SetServerInfo methods, dwPlayerFlags should be set to zero.

Structure Information

Headerdplay8.hMinimum operating systemsWindows 98, Pocket PC 2002

$\ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.

DPN_SECURITY_CREDENTIALS Structure

Not currently implemented.

 $\ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.

DPN_SECURITY_DESC Structure

Not currently implemented.

 $\ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.

DPN_SERVICE_PROVIDER_INFO Structure

Used when enumerating information for a specific service provider.

Syntax

typedef struct _DPN_SERVICE_PROVIDER_INFO {
 DWORD dwFlags;
 GUID guid;
 WCHAR *pwszName;
 PVOID pvReserved;
 DWORD dwReserved;
} DPN SERVICE PROVIDER INFO, *PDPN SERVICE PROVIDER

Members

dwFlags

Describes the service provider. DPNSPINFO_NETWORKSIMULATORDEVICE Device is available to the DP8Sim service provider.

guid

GUID for the service provider.

pwszName

Name of the service provider.

pvReserved

Reserved. Must be 0.

dwReserved

Reserved. Must be 0.

Structure Information

Headerdplay8.hMinimum operating systemsWindows 98, Pocket PC 2002

© 2003 Microsoft Corporation. All rights reserved.

DPN_SP_CAPS Structure

Used to set and retrieve parameters for service providers.

Syntax

typedef struct _DPN_SP_CAPS {
 DWORD dwSize;
 DWORD dwFlags;
 DWORD dwNumThreads;
 DWORD dwDefaultEnumCount;
 DWORD dwDefaultEnumRetryInterval;
 DWORD dwDefaultEnumTimeout;
 DWORD dwMaxEnumPayloadSize;
 DWORD dwBuffersPerThread;
 DWORD dwSystemBufferSize;
} DPN_SP_CAPS, *PDPN_SP_CAPS;

Members

dwSize

Value must be set to the size of the structure.

dwFlags

Can be a combination of the following flags.

DPNSPCAPS_SUPPORTSDPNSRV

DPNSVR.EXE will provide port sharing for the given SP. Currently this flag is available on Internet Protocol (IP) and Internetwork Packet Exchange (IPX) only. See <u>Using the</u> <u>DirectPlay DPNSVR Application</u> for a further discussion of DPNSVR.

DPNSPCAPS_SUPPORTSBROADCAST

On IP and IPX applications, the service provider has the ability to broadcast to find games if not enough addressing information is passed.

DPNSPCAPS_SUPPORTSALLADAPTERS

The service provider will use all devices on the system.

There is no need to specify a device element. DPNSPCAPS_SUPPORTSTHREADPOOL

The service provider will support the thread pool. DPNSPCAPS_NETWORKSIMULATOR

Specifies the DP8Sim service provider.

dwNumThreads

Number of threads the service provider will use for servicing network requests. The default value for this is based on an algorithm that takes into account the number of processors on the system. Most applications will not need to modify this value.

After a service provider is active in your process, you can only increase this value. Decreasing the value will have no effect. The setting is process wide, which means it will affect your current Microsoft® DirectPlay® object and any other DirectPlay objects in your process.

You can specify a lower value than the default if you call the **SetSPCaps** method before you call an **EnumHosts, Connect**, or **Host** method.

dwDefaultEnumCount

Default enumeration count.

dwDefaultEnumRetryInterval

Default retry interval, in milliseconds.

dwDefaultEnumTimeout

Default enumeration timeout value, in milliseconds.

dwMaxEnumPayloadSize

Maximum size of the payload information that can be sent in the **pvResponseData** member of the structures that accompany the <u>DPN_MSGID_ENUM_HOSTS_QUERY</u> and <u>DPN_MSGID_ENUM_HOSTS_RESPONSE</u> messages.

dwBuffersPerThread

The number of outstanding receive buffers allocated for each DirectPlay thread. If you increase the number of receive buffers, DirectPlay can pull more data out of the operating system buffers. However, you can also increase latency if data is arriving faster than your application can process it.

dwSystemBufferSize

The size of the operating system buffer. This buffer holds data from the communications device when your application cannot process data as fast as it arrives. The purpose of this buffer is to prevent data loss if you receive a sudden burst of data, or if the receive threads are momentarily stalled. Increasing **dwSystemBufferSize** can increase latency if your application cannot process the received data fast enough. You can eliminate the operating system buffer by setting **dwSystemBufferSize** to 0. However, if you do so, you run the risk of losing data if you cannot process the received data as fast as it arrives.

Remarks

The **dwBuffersPerThread** and **dwSystemBufferSize** members are used only by IP and IPX service providers. The default values for these members are set by the service provider. To determine the default value, call the appropriate **GetSPCaps** method. Most applications should use the default values for these two members. They are intended primarily for use by developers writing server applications for massively-multiplayer games.

Structure Information

Headerdplay8.hMinimum operating systemsWindows 98, Pocket PC 2002

© 2003 Microsoft Corporation. All rights reserved.

DPNHCAPS Structure

Used to set and retrieve parameters for service providers.

Syntax

typedef struct _DPNHCAPS { DWORD dwSize; DWORD dwFlags; DWORD dwNumRegisteredPorts; DWORD dwMinLeaseTimeRemaining; DWORD dwRecommendedGetCapsInterval; } DPNHCAPS, *PDPNHCAPS;

Members

dwSize

Size of this structure. Must be filled prior to calling <u>IDirectPlayNATHelp::GetCaps</u>.

dwFlags

Flags indicating capabilities of the Internet gateway server. Possible values are:

DPNHCAPSFLAGS_LOCALFIREWALLPRESENT

At least one network connection has a local firewall present. DPNHCAPSFLAGS_GATEWAYPRESENT

At least one network connection has an Internet gateway present.

DPNHCAPSFLAGS_GATEWAYISLOCAL

A detected Internet gateway is local.

DPNHCAPSFLAGS_PUBLICADDRESSAVAILABLE

At least one server has a valid public address for registered mappings.

DPNHCAPSFLAGS_NOTALLSUPPORTACTIVENOTIFY

At least one available server does not support an active notification mechanism and must be polled.

dwNumRegisteredPorts

Number of ports currently registered, including multiple ports registered at the same time.

dwMinLeaseTimeRemaining

Time remaining on the lease that will expire first, in milliseconds. dwRecommendedGetCapsInterval

Recommended time between calls to

IDirectPlayNATHelp::GetCaps, in milliseconds.

Structure Information

Headerdpnathlp.hMinimum operating systemsWindows 98, Pocket PC 2002

© 2003 Microsoft Corporation. All rights reserved.

DVCAPS Structure

Describes the capabilities of the Microsoft® DirectPlay® VoiceClient object.

Syntax

typedef struct {
 DWORD dwSize;
 DWORD dwFlags;
} DVCAPS, *LPDVCAPS, *PDVCAPS;

Members

dwSize

Must be set the to size of this structure, in bytes, before using this structure.

dwFlags

Reserved. Must be 0.

Structure Information

Headerdvoice8.hMinimum operating systemsWindows 98

© 2003 Microsoft Corporation. All rights reserved.

DVCLIENTCONFIG Structure

Controls the run-time parameters for the client.

Syntax

typedef struct { DWORD dwSize; DWORD dwFlags; LONG IRecordVolume; LONG IPlaybackVolume; DWORD dwThreshold; DWORD dwBufferQuality; DWORD dwBufferAggressiveness; DWORD dwBufferAggressiveness; DWORD dwNotifyPeriod; } DVCLIENTCONFIG, *LPDVCLIENTCONFIG, *PDVCLIENTCONFIG;

Members

dwSize

Must be set to the size of this structure, in bytes, before using this structure.

dwFlags

Combination of the following flags. Possible values include the following:

| | Activates automatic gain control. With automatic |
|---------------------------------|---|
| | gain control, Microsoft® DirectPlay® Voice adjusts |
| | the hardware input volume on your sound card |
| | automatically to get the best input level possible. |
| | You can determine the current input volume by |
| DVCLIENTCONFIG AUTORECORDVOLUME | looking at the IRecordVolume member of this |
| - | structure after calling |
| | IDirectPlayVoiceClient::GetClientConfig, or by |
| | looking at the IRecordVolume member of the |
| | structure that accompanies a |
| | DVMSGID_INPUTLEVEL message. |

| DVCLIENTCONFIG_ECHOSUPPRESSION | Activates the echo suppression mode. This mode reduces echo introduced by configurations with external speakers and extremely sensitive microphones. While remote players' voices are being played back on the local speaker, the microphone is automatically muted. If the local player is transmitting, the playback of remote player voices is buffered until local input stops. After local input stops, playback resumes. |
|--------------------------------|--|
| DVCLIENTCONFIG_MUTEGLOBAL | Mutes playback of the main sound buffer. Only sound buffers created through calls to <u>IDirectPlayVoiceClient::Create3DSoundBuffer</u> will be heard. |
| DVCLIENTCONFIG_PLAYBACKMUTE | Mutes playback of all DirectPlay Voice output and stops playback. This also stops decompression of incoming packets so CPU usage is reduced. Packets are effectively discarded while this flag is specified. |
| DVCLIENTCONFIG_RECORDMUTE | Mutes input from the microphone and stops recording. This also stops compression so CPU usage is reduced. |

In addition to the preceding flags, the method of transmission is controlled by setting only one of the following flags or by not specifying either flag. Possible values include the following:

| DVCLIENTCONFIG_AUTOVOICEACTIVATED | Places the transmission control system into automatic voice activation mode. In this mode, the sensitivity of voice activation is determined automatically by the system. The input level is adaptive, adjusting itself automatically to the input signal. For most applications this should be the setting used. This flag is mutually exclusive with the DVCLIENTCONFIG_MANUALVOICEACTIVATED flag. |
|-------------------------------------|---|
| DVCLIENTCONFIG_MANUALVOICEACTIVATED | Places the transmission control system into manual voice activation mode. In this mode, transmission of voice begins when the input level passes the level specified by the dwThreshold member. When input levels drop below the specified level, transmission stops. This flag is mutually exclusive with the DVCLIENTCONFIG_AUTOVOICEACTIVATED flag. |

If you do not specify either DVCLIENTCONFIG_MANUALVOICEACTIVATED or DVCLIENTCONFIG_AUTOVOICEACTIVATED, the system will operate in push-to-talk mode. In push-to-talk mode, as long as there is a valid target specified the input from the microphone will be transmitted. Voice transmission stops when a NULL target is set or the current target leaves the session or is destroyed.

IRecordVolume

LONG value that specifies to what level the volume of the recording should be set. See the

IDirectSoundBuffer8::SetVolume method for valid values.

If automatic gain control is enabled, this value can be set to DVRECORDVOLUME_LAST, which tells the system to use the current volume as determined by the automatic gain control algorithm. If a value other than DVRECORDVOLUME_LAST is specified in combination with automatic gain control, this value will be used to restart the algorithm at the specified value.

On return from a call to

IDirectPlayVoiceClient::GetClientConfig, this value will contain the current recording volume. When adjusting the recording volume, DirectPlay Voice will adjust the volume for the microphone (if a microphone volume is present for the card) and the master recording volume (if one is present on the card). If neither a microphone volume nor a master record volume is present, DirectPlay Voice will be unable to adjust the recording volume.

IPlaybackVolume

Value indicating to what level the playback volume should be set. Adjusting this volume adjusts both the main buffer and all 3-D sound buffers. See the **IDirectSoundBuffer8::SetVolume** method for valid values. You can specify DVPLAYBACKVOLUME_DEFAULT to use a default value that is appropriate for most situations (full volume).

dwThreshold

Input level used to trigger voice transmission if the DVCLIENTCONFIG_MANUALVOICEACTIVATED flag is specified in the **dwFlags** member. When the flag is specified, this can be set to any value within the range of DVTHRESHOLD_MIN to DVTHRESHOLD_MAX. Additionally, DVTHRESHOLD_DEFAULT can be set to use a default value. If DVCLIENTCONFIG_MANUALVOICEACTIVATED or DVCLIENTCONFIG_AUTOVOICEACTIVATED is not specified in the **dwFlags** member. When the flag is specified, this can be set to any value within the range member of this structure (indicating push-to-talk mode) this value must be set to DVTHRESHOLD_UNUSED.

dwBufferQuality

Not implemented.

dwBufferAggressiveness

Not implemented.

dwNotifyPeriod

Value indicating how often you want to receive

DVMSGID_OUTPUTLEVEL and DVMSGID_INPUTLEVEL (if session is full duplex) messages. If this value is set to 0, these messages are disabled. The value specifies the number of milliseconds between these messages. DVNOTIFYPERIOD_MINPERIOD specifies the minimum allowable period between messages.

Remarks

This structure is first used in the call to

IDirectPlayVoiceClient::Connect, where it sets the initial state of these parameters. The structure can be retrieved after a connection has been made by calling

IDirectPlayVoiceClient::GetClientConfig, and set using IDirectPlayVoiceClient::SetClientConfig.

Structure Information

Headerdvoice8.hMinimum operating systemsWindows 98

© 2003 Microsoft Corporation. All rights reserved.

DVCOMPRESSIONINFO Structure

Describes the attributes of a specific Microsoft® DirectPlay® Voice compression type.

Syntax

typedef struct { DWORD dwSize; GUID guidType; LPWSTR lpszName; LPWSTR lpszDescription; DWORD dwFlags; DWORD dwMaxBitsPerSecond; } DVCOMPRESSIONINFO, *LPDVCOMPRESSIONINFO, *PDVCOMF

Members

dwSize

Must be set the to size of this structure, in bytes, before using this structure.

guidType

GUID used to identify this compression type by DirectPlay Voice.

IpszName

Pointer to a name describing the codec.

IpszDescription

Pointer to a longer name of the codec.

dwFlags

Reserved; must be 0.

dwMaxBitsPerSecond

Maximum number of bits per second claimed by the codec.

Structure Information

| Header | dvoice8.h |
|--------|-----------|
| | |

 $\ensuremath{\mathbb{C}}$ 2003 Microsoft Corporation. All rights reserved.

DVSESSIONDESC Structure

Describes the desired or current session settings for the Microsoft® DirectPlay® Voice server. This structure is used by the voice session host to configure the session, and by the session host and clients to retrieve information about the current session.

Syntax

typedef struct {
 DWORD dwSize;
 DWORD dwFlags;
 DWORD dwSessionType;
 GUID guidCT;
 DWORD dwBufferQuality;
 DWORD dwBufferAggressiveness;
} DVSESSIONDESC, *LPDVSESSIONDESC, *PDVSESSIONDESC;

Members

dwSize

Must be set the to size of this structure, in bytes, before using this structure.

dwFlags

Combination of the following flags.

DVSESSION_NOHOSTMIGRATION

The voice host will not migrate regardless of the transport settings. If this flag is not specified, the voice host will migrate if the transport supports it.

DVSESSION_SERVERCONTROLTARGET

The clients are unable to control the target of their speech. Only the server player can control the target of their speech. If the server does not specify this flag, only the clients can control the target of their speech. This flag can be specified only in multicast and mixing sessions.

dwSessionType

The type of DirectPlay Voice session to run. The DVSESSIONTYPE_PEER flag is not available in client/server sessions; all other flags are valid for all session types. This member can be one of the following values.

DVSESSIONTYPE_PEER

Voice messages will be sent directly between players. DVSESSIONTYPE_MIXING

Voice session will use a mixing server. In this mode of operation, all voice messages are sent to the server, which mixes them and then forwards a single, premixed stream to each client. This reduces the bandwidth and CPU usage on clients significantly at the cost of increased bandwidth and CPU usage on the server.

DVSESSIONTYPE_FOWARDING

Voice messages will be routed through the session host. This will save bandwidth on the clients at the expense of bandwidth usage on the server. This option is only useful if the session host has a high-speed connection.

guidCT

GUID specifying the compression type of the session. To select the default compression codec, set this member to DPVCTGUID DEFAULT.

dwBufferQuality

The buffer quality setting. This member is unused for all session types except mixing sessions. For all sessions except mixing sessions, set this member to DVBUFFERQUALITY_DEFAULT.

Allowable values are between DVBUFFERQUALITY_MIN and DVBUFFERQUALITY_MAX. Additionally, this member can be set to the following value.

DVBUFFERQUALITY_DEFAULT

Specifying this value tells DirectPlay Voice to use the system default for this value, which is adjustable through a registry entry that can also be set through Sounds and Multimedia in Control Panel.

dwBufferAggressiveness
Buffer aggressiveness setting. This member is unused for all session types except mixing sessions. For all sessions except mixing sessions, set this member to DVBUFFERAGGRESSIVENESS_DEFAULT.

Allowable values are between DVBUFFERAGGRESSIVENESS _MIN and DVBUFFERAGGRESSIVENESS _MAX. Additionally, this member can be set to the following value.

DVBUFFERAGGRESSIVENESS_DEFAULT Specifying this value tells DirectPlay Voice to use the system default for this value, which is adjustable through a registry entry that can also be set through Control Panel.

Remarks

The **dwFlags**, **dwSessionType**, and **guidCT** members can only be set when the host starts the voice session. The host can change the buffer settings at any time.

Structure Information

Headerdvoice8.hMinimum operating systemsWindows 98

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

DVSOUNDDEVICECONFIG Structure

Used to set and retrieve information about the sound device configuration and cannot be changed once a connection has been made. After a connection is made, you can retrieve the current sound device configuration by calling <u>IDirectPlayVoiceClient::GetSoundDeviceConfig</u>.

Syntax

typedef struct { DWORD dwSize; DWORD dwFlags; GUID guidPlaybackDevice; LPDIRECTSOUND lpdsPlaybackDevice; GUID guidCaptureDevice; LPDIRECTSOUNDCAPTURE lpdsCaptureDevice; HWND hwndAppWindow; LPDIRECTSOUNDBUFFER lpdsMainBuffer; DWORD dwMainBufferFlags; DWORD dwMainBufferPriority; } DVSOUNDDEVICECONFIG, *LPDVSOUNDDEVICECONFIG, *PDV:

Members

dwSize

Must be set the to size of this structure, in bytes, before using this structure.

dwFlags

A combination of the following flags.

DVSOUNDCONFIG_AUTOSELECT

Tells Microsoft® DirectPlay® Voice to attempt to automatically select (or un-mute) the microphone line in the mixer for the specified recording device.

DVSOUNDCONFIG_HALFDUPLEX

Tells DirectPlay Voice to initialize itself in half-duplex mode.

In half-duplex mode no recording takes place. If the initialization of the sound system fails in full-duplex mode, this flag will be set by the system.

DVSOUNDCONFIG_NORMALMODE

Tells DirectPlay Voice to use Microsoft DirectSound® Normal Mode when initializing the DirectSound object. If this flag is not specified, the DirectSound object is initialized with DirectSound Priority Mode. See documentation for **IDirectSound8::SetCooperativeLevel** for more information. If a valid DirectSound object is specified in the **IpdsPlaybackDevice** member, this flag is ignored.

DVSOUNDCONFIG_SETCONVERSIONQUALITY

Enables better quality audio at the expense of higher CPU usage.

DVSOUNDCONFIG_NORECVOLAVAILABLE

Set by DirectPlay Voice if there are no volume controls available on the recording device you specified. You cannot set this flag.

DVSOUNDCONFIG_NOFOCUS

The voice application will never go out of focus. In other words, the application will never release the sound capture device. Use of this flag is not recommended.

DVSOUNDCONFIG_STRICTFOCUS

The voice application will lose focus whenever its window is not the foreground window.

guidPlaybackDevice

When this structure is used in the

IDirectPlayVoiceClient::GetSoundDeviceConfig method, this

member contains the actual device globally unique identifier

(GUID) used for playback.

When this structure is used in the

IDirectPlayVoiceClient::Connect method, this member specifies the **GUID** of the device used for playback. This must be specified even if the **IpdsPlaybackDevice** member is used. You can also specify the following default GUIDs provided by DirectSound. DSDEVID_DefaultPlayback

The system default playback device.

DSDEVID_DefaultVoicePlayback

The default voice playback device.

IpdsPlaybackDevice

When this structure is used in the

IDirectPlayVoiceClient::Connect method, this member specifies the DirectSound object you want DirectPlay Voice to use for playback. The **GUID** specified in **guidPlaybackDevice** must match the one used to create the device specified by this parameter. If you used NULL when specifying the device when you created your DirectSound object, pass DSDEVID DefaultPlayback for this member.

When this structure is used in the

IDirectPlayVoiceClient::GetSoundDeviceConfig method, this member contains a pointer to the DirectSound object being used by DirectPlay Voice. This will either be a pointer to the object specified when **Connect** was called or a pointer to a newly created and initialized DirectSound object. If you want to use this DirectSound object, you must store the pointer and increment the reference count by calling **AddRef** on the DirectSound interface.

guidCaptureDevice

When this structure is used in

IDirectPlayVoiceClient::Connect method, this member specifies the **GUID** of the device used for capture. This must be specified even if the **IpdsCaptureDevice** member is used. If you used NULL when specifying the device when you created your DirectSoundCapture object, pass DSDEVID DefaultCapture for this member.

When this structure is used in the

IDirectPlayVoiceClient::GetSoundDeviceConfig method, this member will contain the actual device **GUID** used for capture.

IpdsCaptureDevice

When this structure is used in the IDirectPlayVoiceClient::Connect method, this member specifies the DirectSound object you want DirectPlay Voice to use for capture. The GUID specified in guidCaptureDevice must match the one used to create the device specified by this parameter. If you want to have DirectPlay Voice create the DirectSoundCapture object for you, specify NULL for this member.

When this structure is used in the

IDirectPlayVoiceClient::GetSoundDeviceConfig method, this member contains a pointer to the DirectSoundCapture object being used by DirectPlay Voice. This will either be a pointer to the object specified when **Connect** was called or a pointer to a newly created and initialized DirectSoundCapture object. If you want to use this DirectSoundCapture object, you must store the pointer and increment the reference count by calling **AddRef** on the **IDirectSoundCapture8** interface. If the DirectPlay Voice object is operating in half duplex mode, this member will be NULL.

hwndAppWindow

Must be set to the handle of the window that will be used to determine focus for sound playback. See

IDirectSound8::SetCooperativeLevel for information about DirectSound focus. If you do not have a window to use for focus, use GetDesktopWindow to use the desktop window.

IpdsMainBuffer

Pointer to an **IDirectSoundBuffer8** interface, which is used to create the DirectPlay Voice main buffer. This can be either NULL or a user-created DirectSound buffer. If this member is set to NULL, DirectPlay Voice will create a buffer for the main voice buffer. If users specify a buffer here, DirectPlay Voice will use their buffer for the main voice buffer. User-created buffers have the following restrictions.

- The buffer must be 22 kilohertz, 16-bit, Mono format.
- The buffer must be at least 1 second in length.
- The buffer must have been created with the DSBCAPS_GETCURRENTPOSITION2 and DSBCAPS_CTRL3D flags.
- The buffer must not be a primary buffer.
- The buffer must not be playing when it is passed to the DirectPlay Voice software.
- The buffer must not be locked when it is passed to the DirectPlay Voice software.

dwMainBufferFlags

Passed directly to the *dwFlags* parameter of the IDirectSoundBuffer8::Play method when Play is called for the main buffer. The DSBPLAY_LOOPING flag is automatically added to this field. See the documentation on IDirectSoundBuffer8::Play for details. This parameter must be

0 if the **lpdsMainBuffer** member of this structure is NULL.

dwMainBufferPriority

Passed directly to the *dwPriority* parameter of the **IDirectSoundBuffer8::Play** method when **Play** is called on the main buffer. See documentation for **IDirectSoundBuffer8::Play** for more information. This member must be set to 0 if **IpdsMainBufferDesc** is NULL.

Remarks

Note Applications should set the DVSOUNDCONFIG_NOFOCUS or DVSOUNDCONFIG_STRICTFOCUS flags only when strictly necessary. Instead, you should normally use the default behavior that results when neither flag is set.

Structure Information

Headerdvoice8.hMinimum operating systemsWindows 98

 $\ensuremath{\textcircled{\sc 0}}$ 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

Errors are represented by negative values and cannot be combined.

Many of the Microsoft® DirectPlay® samples include a **GetDirectPlayErrStr** function that converts HRESULT values to string names for the DirectPlay errors. You can copy this code into your own applications for diagnostic traces or error reports.

Syntax

typedef enum { DPNSUCCESS PENDING, DPN OK, DPNERR ABORTED, DPNERR ADDRESSING, DPNERR ALREADYCONNECTED, DPNERR ALREADYCLOSING, DPNERR ALREADYDISCONNECTING, DPNERR ALREADYINITIALIZED, DPNERR BUFFERTOOSMALL, DPNERR CANNOTCANCEL, DPNERR CANTCREATEGROUP, DPNERR CANTCREATEPLAYER, DPNERR CANTLAUNCHAPPLICATION, DPNERR CONNECTING, DPNERR CONNECTIONLOST, DPNERR DATATOOLARGE, DPNERR DOESNOTEXIST, DPNERR DPNSVRNOTAVAILABLE, DPNERR ENUMQUERYTOOLARGE, DPNERR ENUMRESPONSETOOLARGE, DPNERR _EXCEPTION, DPNERR GENERIC, DPNERR GROUPNOTEMPTY, DPNERR HOSTREJECTEDCONNECTION, DPNERR HOSTTERMINATEDSESSION, DPNERR INCOMPLETEADDRESS, DPNERR INVALIDADDRESSFORMAT, DPNERR INVALIDAPPLICATION, DPNERR INVALIDCOMMAND, DPNERR INVALIDDEVICEADDRESS, DPNERR INVALIDFLAGS, DPNERR INVALIDGROUP, DPNERR INVALIDHANDLE, DPNERR INVALIDHOSTADDRESS, DPNERR INVALIDINSTANCE, DPNERR INVALIDINTERFACE, DPNERR INVALIDOBJECT, DPNERR INVALIDPARAM, DPNERR INVALIDPASSWORD, DPNERR INVALIDPLAYER, DPNERR INVALIDPOINTER, DPNERR INVALIDPRIORITY, DPNERR INVALIDSTRING, DPNERR INVALIDURL, DPNERR INVALIDVERSION, DPNERR NOCAPS, DPNERR NOCONNECTION, DPNERR NOHOSTPLAYER, DPNERR NOINTERFACE, DPNERR NORESPONSE, DPNERR NOTALLOWED, DPNERR NOTHOST, DPNERR NOTREADY, DPNERR OUTOFMEMORY, DPNERR PENDING. DPNERR PLAYERALREADYINGROUP, DPNERR PLAYERNOTINGROUP, DPNERR PLAYERLOST, DPNERR PLAYERNOTREACHABLE, DPNERR SESSIONFULL, DPNERR TIMEDOUT, DPNERR UNINITIALIZED, DPNERR UNSUPPORTED,

DPNERR USERCANCEL, DV OK, DV FULLDUPLEX, DV HALFDUPLEX, DV PENDING, DVERR BUFFERTOOSMALL, DVERR EXCEPTION, DVERR GENERIC, DVERR INVALIDFLAGS, DVERR INVALIDOBJECT, DVERR INVALIDPARAM, DVERR INVALIDPLAYER, DVERR INVALIDGROUP, DVERR INVALIDHANDLE, DVERR OUTOFMEMORY, DVERR PENDING, DVERR NOTSUPPORTED, DVERR NOINTERFACE, DVERR SESSIONLOST, DVERR NOVOICESESSION, DVERR CONNECTIONLOST, DVERR NOTINITIALIZED, DVERR CONNECTED, DVERR NOTCONNECTED, DVERR CONNECTABORTING, DVERR NOTALLOWED, DVERR INVALIDTARGET, DVERR TRANSPORTNOTHOST, DVERR COMPRESSIONNOTSUPPORTED, DVERR ALREADYPENDING, DVERR ALREADYINITIALIZED, DVERR SOUNDINITFAILURE, DVERR TIMEOUT, DVERR CONNECTABORTED, DVERR NO3DSOUND, DVERR ALREADYBUFFERED, DVERR NOTBUFFERED, DVERR HOSTING, DVERR NOTHOSTING.

DVERR INVALIDDEVICE, DVERR RECORDSYSTEMERROR, DVERR PLAYBACKSYSTEMERROR, DVERR SENDERROR, DVERR USERCANCEL, DVERR UNKNOWN, DVERR RUNSETUP, DVERR INCOMPATIBLEVERSION, DVERR INITIALIZED, DVERR INVALIDPOINTER, DVERR NOTRANSPORT, DVERR NOCALLBACK, DVERR TRANSPORTNOTINIT, DVERR TRANSPORTNOSESSION, DVERR TRANSPORTNOPLAYER, DP8SIM OK, DP8SIMERR ALREADYINITIALIZED, DP8SIMERR INVALIDFLAGS, DP8SIMERR INVALIDOBJECT, DP8SIMERR MISMATCHEDVERSION, DP8SIMERR NOTINITIALIZED, DP8SIMERR INVALIDPARAM, DP8SIMERR INVALIDPOINTER } Return Values;

Constants

DPNSUCCESS_PENDING

An asynchronous operation has reached the point where it is successfully queued.

DPN_OK

The operation completed successfully. This value is equal to the S_OK standard Component Object Model (COM) return value.

DPNERR_ABORTED

The operation was canceled before it could be completed.

DPNERR_ADDRESSING

The address specified is invalid.

DPNERR_ALREADYCONNECTED

The object is already connected to the session.

DPNERR_ALREADYCLOSING

An attempt to call the **Close** method on a session has been made more than once.

DPNERR_ALREADYDISCONNECTING

The client is already disconnecting from the session.

DPNERR_ALREADYINITIALIZED

The object has already been initialized.

DPNERR_BUFFERTOOSMALL

The supplied buffer is not large enough to contain the requested data.

DPNERR_CANNOTCANCEL

The operation could not be canceled.

DPNERR_CANTCREATEGROUP

A new group cannot be created.

DPNERR_CANTCREATEPLAYER

A new player cannot be created.

DPNERR_CANTLAUNCHAPPLICATION

The lobby cannot launch the specified application.

DPNERR_CONNECTING

The method is in the process of connecting to the network.

DPNERR_CONNECTIONLOST

The service provider connection was reset while data was being sent.

DPNERR_DATATOOLARGE

The application data is too large for the service provider's Maximum Transmission Unit.

DPNERR_DOESNOTEXIST

Requested element is not part of the address or the requested application globally unique identifier (GUID) is not registered.

DPNERR_DPNSVRNOTAVAILABLE

Port 6073 is already in use.

DPNERR_ENUMQUERYTOOLARGE

The query data specified is too large.

DPNERR_ENUMRESPONSETOOLARGE

The response to an enumeration query is too large.

DPNERR_EXCEPTION

An exception occurred when processing the request.

DPNERR_GENERIC

An undefined error condition occurred.

DPNERR_GROUPNOTEMPTY

The specified group is not empty.

DPNERR_HOSTREJECTEDCONNECTION

The <u>DPN_MSGID_INDICATE_CONNECT</u> system message returned something other than S_OK in response to a connect request.

DPNERR_HOSTTERMINATEDSESSION

The host in a peer session (with host migration enabled) terminated the session.

DPNERR_INCOMPLETEADDRESS

The address specified is not complete.

DPNERR_INVALIDADDRESSFORMAT

The address format is invalid.

DPNERR_INVALIDAPPLICATION

The GUID supplied for the application is invalid.

DPNERR_INVALIDCOMMAND

The command specified is invalid.

DPNERR_INVALIDDEVICEADDRESS

The address for the local computer or adapter is invalid.

DPNERR_INVALIDFLAGS

The flags passed to this method are invalid.

DPNERR_INVALIDGROUP

The group identifier (ID) is not recognized as a valid group ID for

this game session.

DPNERR_INVALIDHANDLE

The handle specified is invalid.

DPNERR_INVALIDHOSTADDRESS

The specified remote address is invalid.

DPNERR_INVALIDINSTANCE

The GUID for the application instance is invalid.

DPNERR_INVALIDINTERFACE

The interface parameter is invalid. This value will be returned in a connect request if the connecting player was not a client in a client/server game or a peer in a peer-to-peer game.

DPNERR_INVALIDOBJECT

The DirectPlay object pointer is invalid.

DPNERR_INVALIDPARAM

One or more of the parameters passed to the method are invalid.

DPNERR_INVALIDPASSWORD

An invalid password was supplied when attempting to join a session that requires a password.

DPNERR_INVALIDPLAYER

The player ID is not recognized as a valid player ID for this game session.

DPNERR_INVALIDPOINTER

The pointer specified as a parameter is invalid.

DPNERR_INVALIDPRIORITY

The specified priority is not within the range of allowed priorities, which is inclusively from 0 through 65535.

DPNERR_INVALIDSTRING

String specified as a parameter is invalid.

DPNERR_INVALIDURL

Specified string is not a valid DirectPlay URL.

DPNERR_INVALIDVERSION

There was an attempt to connect to an invalid version of DirectPlay.

DPNERR_NOCAPS

The communication link that DirectPlay is attempting to use is not capable of this function.

DPNERR_NOCONNECTION

No communication link was established.

DPNERR_NOHOSTPLAYER

There is currently no player acting as the host of the session.

DPNERR_NOINTERFACE

The interface is not supported.

DPNERR_NORESPONSE

There was no response from the specified target.

DPNERR_NOTALLOWED

This function is not allowed on this object.

DPNERR_NOTHOST

An attempt by the client to connect to a nonhost computer. Additionally, this error value can be returned by a nonhost that tries to set the application description.

DPNERR_NOTREADY

The object is not ready for use.

DPNERR_OUTOFMEMORY

There is insufficient memory to perform the requested operation.

DPNERR_PENDING

Not an error, this return indicates that an asynchronous operation has reached the point where it is successfully queued. SUCCEEDED(DPNERR_PENDING) will return TRUE. This error value has been superseded by DPNERR_SUCCESS, which should be used by all new applications. DPNERR_PENDING is only included for backward compatibility.

DPNERR_PLAYERALREADYINGROUP

The player ID is already included in the group.

DPNERR_PLAYERNOTINGROUP

The player ID is not included in the group.

DPNERR_PLAYERLOST

A player has lost the connection to the session.

DPNERR_PLAYERNOTREACHABLE

A player has tried to join a peer-peer session where at least one other existing player in the session cannot connect to the joining player.

DPNERR_SESSIONFULL

The request to connect to the host or server failed because the maximum number of players allotted for the session has been reached.

DPNERR_TIMEDOUT

The operation could not complete because it has timed out.

DPNERR_UNINITIALIZED

The requested object has not been initialized.

DPNERR_UNSUPPORTED

The function or feature is not available in this implementation or on this service provider.

DPNERR_USERCANCEL

The user canceled the operation.

DV_OK

The request completed successfully.

DV_FULLDUPLEX

The sound card is capable of full-duplex operation.

DV_HALFDUPLEX

The sound card can only be run in half-duplex mode.

DV_PENDING

An asynchronous operation has reached the point where it is successfully queued.

DVERR_BUFFERTOOSMALL

The supplied buffer is not large enough to contain the requested data.

DVERR_EXCEPTION

An exception occurred when processing the request.

DVERR_GENERIC

An undefined error condition occurred.

DVERR_INVALIDFLAGS

The flags passed to this method are invalid.

DVERR_INVALIDOBJECT

The DirectPlay object pointer is invalid.

DVERR_INVALIDPARAM

One or more of the parameters passed to the method are invalid.

DVERR_INVALIDPLAYER

The player ID is not recognized as a valid player ID for this game session.

DVERR_INVALIDGROUP

The group ID is not recognized as a valid group ID for this game session.

DVERR_INVALIDHANDLE

The handle specified is invalid.

DVERR_OUTOFMEMORY

There is insufficient memory to perform the requested operation.

DVERR_PENDING

Not an error, this return indicates that an asynchronous operation has reached the point where it is successfully queued. This error value has been deprecated. It has been replaced by DV_PENDING.

DVERR_NOTSUPPORTED

The operation is not supported.

DVERR_NOINTERFACE

The specified interface is not supported. Could indicate using the wrong version of DirectPlay.

DVERR_SESSIONLOST

The transport has lost the connection to the session.

DVERR_NOVOICESESSION

The session specified is not a voice session.

DVERR_CONNECTIONLOST

The connection to the voice session has been lost.

DVERR_NOTINITIALIZED

The <u>IDirectPlayVoiceClient::Initialize</u> IDirectPlayVoiceClient8::Initialize or <u>IDirectPlayVoiceServer::Initialize</u> IDirectPlayVoiceServer8::Initialize method must be called before calling this method.

DVERR_CONNECTED

The DirectPlay Voice object is connected.

DVERR_NOTCONNECTED

The DirectPlay Voice object is not connected.

DVERR_CONNECTABORTING

The connection is being disconnected.

DVERR_NOTALLOWED

The object does not have the permission to perform this operation.

DVERR_INVALIDTARGET

The specified target is not a valid player ID or group ID for this voice session.

DVERR_TRANSPORTNOTHOST

The object is not the host of the voice session.

DVERR_COMPRESSIONNOTSUPPORTED

The specified compression type is not supported on the local computer.

DVERR_ALREADYPENDING

An asynchronous call of this type is already pending.

DVERR_ALREADYINITIALIZED

The object has already been initialized.

DVERR_SOUNDINITFAILURE

A failure was encountered initializing the sound card.

DVERR_TIMEOUT

The operation could not be performed in the specified time.

DVERR_CONNECTABORTED

The connect operation was canceled before it could be completed.

DVERR_NO3DSOUND

The local computer does not support 3-D sound.

DVERR_ALREADYBUFFERED

There is already a user buffer for the specified ID.

DVERR_NOTBUFFERED

There is no user buffer for the specified ID.

DVERR_HOSTING

The object is the host of the session.

DVERR_NOTHOSTING

The object is not the host of the session.

DVERR_INVALIDDEVICE

The specified device is invalid.

DVERR_RECORDSYSTEMERROR

An error in the recording system occurred.

DVERR_PLAYBACKSYSTEMERROR

An error in the playback system occurred.

DVERR_SENDERROR

An error occurred while sending data.

DVERR_USERCANCEL

The user canceled the operation.

DVERR_UNKNOWN

An unknown error occurred.

DVERR_RUNSETUP

The specified audio configuration has not been tested. Call the IDirectPlayVoiceTest::CheckAudioSetup method.

DVERR_INCOMPATIBLEVERSION

The client connected to a voice session that is incompatible with the host.

DVERR_INITIALIZED

The Initialize method failed because the object has already been initialized.

DVERR_INVALIDPOINTER

The pointer specified is invalid.

DVERR_NOTRANSPORT

The specified object is not a valid transport.

DVERR_NOCALLBACK

This operation cannot be performed because no callback function was specified.

DVERR_TRANSPORTNOTINIT

The specified transport is not yet initialized.

DVERR_TRANSPORTNOSESSION

The specified transport is valid but is not connected/hosting.

DVERR_TRANSPORTNOPLAYER

The specified transport is connected/hosting but no local player exists.

DP8SIM_OK

The DP8Sim control object was successfully initialized.

DP8SIMERR_ALREADYINITIALIZED

The DP8Sim control object has already been initialized.

DP8SIMERR_INVALIDFLAGS

The flags passed to this method are invalid.

DP8SIMERR_INVALIDOBJECT

The DP8Sim control object specified is invalid.

DP8SIMERR_MISMATCHEDVERSION

A different version of DP8Sim is already in use on this system.

DP8SIMERR_NOTINITIALIZED

The DP8Sim control object has not been initialized.

DP8SIMERR_INVALIDPARAM

One or more of the parameters passed to the method are invalid.

DP8SIMERR_INVALIDPOINTER

A pointer specified as a parameter is invalid.

Remarks

The following table lists the interfaces to which the error codes listed apply.

Interface IDirectPlay8Address IDirectPlay8AddressIP IDirectPlay8Client IDirectPlay8LobbiedApplication IDirectPlay8LobbyClient IDirectPlay8Peer IDirectPlay8Server IDirectPlayVoiceClient IDirectPlayVoiceServer IDirectPlayVoiceTest IDP8SimControl

For a list of the error codes each method can return, see the individual method descriptions.

Enumerated Type Information

Minimum operating systems Windows 98, Windows NT 2000, Windows XP

© 2003 Microsoft Corporation. All rights reserved.

Microsoft DirectX 9.0 SDK Update (Summer 2003)

Basic Networking

This section covers some basic networking technology topics that you need to understand to write Microsoft® DirectPlay® applications. For a general discussion of networking technology, see one of the standard texts on the subject, such as *Computer Networks* by Andrew Tannenbaum.

- DirectPlay Addressing
- DirectPlay Protocol
- Optimizing Network Usage
- Network Address Translation, Firewalls, and Proxies
- <u>Using the DirectX Protocol in an Application</u>

© 2003 Microsoft Corporation. All rights reserved.