[MC-DPL8R]: DirectPlay 8 Protocol: Reliable Specification

Revision Summary

Date	Revision History	Revision Class	Comments
08/10/2007	0.1	Major	Initial Availability
09/28/2007	0.2	Minor	Updated the technical content.
10/23/2007	0.3	Minor	Updated the technical content.
11/30/2007	1.0	Major	Updated and revised the technical content.

Date	Revision History	Revision Class	Comments
01/25/2008	2.0	Major	Updated and revised the technical content.

Table of Contents

1 Introduction

The DirectPlay 8 Protocol is intended for use in multiplayer game communication. It provides mixed unreliable and reliable messages over existing datagram protocols, such as the **User Datagram Protocol (UDP)**.

1.1 Glossary

The following terms are defined in [MS-GLOS]:

Little-Endian
Local Area Network (LAN)
Maximum Transmission Unit (MTU)
Partner

The following terms are specific to this document:

Acknowledgement (ACK): A signal passed between communicating processes or computers to signify successful receipt of a transmission. The DirectPlay 8 Protocol acknowledges packet **sequence IDs**.

CFRAME: A DirectPlay 8 Protocol command frame.

Coalesced Payload: A special form of payload that consists of multiple traditional payloads combined into a single packet.

DFRAME: A DirectPlay 8 Protocol data frame.

DirectX: Microsoft **DirectX** is a collection of application programming interfaces that handle tasks related to multimedia, especially game programming and video, on Microsoft platforms.

Next Receive: The next 8-bit packet **sequence ID** expected to be received, indicating **acknowledgement** of all packets up to this ID. This is typically represented as a field named **bNRcv** in packet structures.

Next Send: The next 8-bit packet **sequence ID** that will be sent. This is represented as **bNSeq** in the **Selective Acknowledgement (SACK)** packet structure, which does not have a **sequence ID** of its own. DirectPlay 8 Protocol implementations also keep an internal counter so that IDs can be assigned in order.

Round-Trip Time (RTT): The time that it takes a packet to be sent to a remote **partner** and for that **partner's** **acknowledgement** to arrive back at the original sender. This is a measurement of latency between **partners**.

Selective Acknowledgement (SACK): A mechanism that indicates successful receipt of packets beyond the **Next Receive** indicator. **Next Receive** reports all packets prior to when its **sequence ID** has been received, but subsequent packets may have arrived out of order or with gaps in the sequence. **SACK** masks enable the receiver to acknowledge these packets so that they do not need to be retried, in addition to the packets that were truly lost.

Send Mask: A bit-mask mechanism that indicates that previously sent packets may have been dropped, were marked as unreliable, and will never be retried.

Sequence ID: A monotonically increasing 8-bit identifier for packets. This is typically represented as a field named **bSeq** in packet structures.

Transmission Control Protocol (TCP): A connection-oriented protocol used with the Internet Protocol (IP) to send data in the form of message units between computers over the Internet. **TCP** handles keeping track of the individual units of data (called packets) into which a message is divided for efficient routing through the Internet.

User Datagram Protocol (UDP): A common connectionless, datagram-oriented protocol that is used with the Internet Protocol (IP).

MAY, SHOULD, MUST, SHOULD NOT, MUST NOT: These terms (in all caps) are used as specified in [RFC2119]. All statements of optional behavior use either MAY, SHOULD, or SHOULD NOT.

1.2 References

1.2.1 Normative References

We conduct frequent surveys of the normative references to assure their continued availability. If you have any issue with finding a normative reference, please contact dochelp@microsoft.com. We will assist you in finding the relevant information. Please check the archive site, http://msdn2.microsoft.com/en-us/library/E4BD6494-06AD-4aed-9823-445E921C9624, as an additional source.

[FIPS180] Federal Information Processing Standards Publication, "Secure Hash Standard", FIPS PUB 180-1, April 1995, http://www.itl.nist.gov/fipspubs/fip180-1.htm

[MS-GLOS] Microsoft Corporation, "Windows Protocols Master Glossary", March 2007.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997, http://www.ietf.org/rfc/rfc2119.txt

1.2.2 Informative References

[MC-DPL8CS] Microsoft Corporation, "DirectPlay 8 Protocol: Core and Service Providers Specification" September 2007.

[MC-DPLHP] Microsoft Corporation, "DirectPlay 8 Protocol: Host and Port Enumeration Specification" September 2007.

[RFC768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, August 1980, http://www.ietf.org/rfc/rfc768.txt

[RFC793] Postel, J., "Transmission Control Protocol: DARPA Internet Program Protocol Specification", RFC 793, September 1981, http://www.ietf.org/rfc/rfc0793.txt

[RFC2581] Allman, M., Paxson, V., and Stevens, W., Network Working Group, "TCP Congestion Control," RFC 2581, April 1999, http://www.ietf.org/rfc/rfc2581.txt

1.3 Protocol Overview (Synopsis)

The DirectPlay 8 Protocol is designed to perform low latency, multiplayer game communication between two **partners**. Its messages are nominally transported over UDP using application-specific port numbers. They are processed by receivers prepared to handle DirectPlay 8 Protocol: Host and Port Enumeration Protocol messages as well, and are distinguished from such messages by their first UDP payload byte being nonzero.

The DirectPlay 8 Protocol assigns a sequence number to each packet that it sends, and the sequence numbers received are acknowledged by the receiver. A sliding window is used to determine how many packets may be outstanding at a time, while waiting for **acknowledgements**.

An acknowledgement can be conveyed through two methods. One way is to bundle it within back traffic from the receiver. If no back traffic is flowing, a (**Selective Acknowledgement (SACK)** command frame packet without upper layer payloads can be sent. If the original sender specifies the PACKET_COMMAND_POLL (acknowledge now) flag in the packet header, the receiver must immediately acknowledge the packet when it arrives, which usually means a SACK packet is required. Whether using a data frame or a SACK packet, the headers indicate the sequence number of the next packet that is expected to be received, which acknowledges that all packets with sequence numbers less than the specified number have been received correctly. Implementations may also include SACK masks in order to acknowledge subsequent packets beyond the specified ID were received out of order.

The DirectPlay 8 Protocol uses combinations of two sets of characteristics for each payload that it sends: reliable/unreliable and sequential/non-sequential. Reliable packets are those that the upper layer deems important to retry if they are lost on the network, whereas unreliable packets are for ephemeral messages that are not critical to operation and do not need to be retried, perhaps because they will be superseded by subsequent messages. Sequential packets are those that must be delivered in order to the upper layer and must wait until any gaps in the sequence due to packet loss are resolved, whereas non-sequential packets may be delivered to the upper layer as soon as they arrive.

A packet is deemed to have been lost if an acknowledgement is not received within a specified time-out typically derived from the current **round-trip time**, or if the receiver explicitly indicates that it encountered a gap in the sequence where the packet would have been using a SACK mask. When the loss is recognized, the implementation either re-sends the original packet with the same sequence number as was previously assigned if it had been marked as reliable, or updates future packets to include a **send mask** that indicates that the data will never be resent if the dropped packet had been marked as unreliable.

The protocol also supports multiple payloads of mixed reliability and sequencing coalesced within a single message to reduce packet overhead. The packet takes on the most restrictive properties of the payloads that it contains (reliable and/or sequential), although individual payloads retain their unique properties. Only the reliable sub-payloads in a **coalesced payload** packet are retried.

1.4 Relationship to Other Protocols

The DirectPlay 8 Protocol requires UDP or a similar datagram-oriented, connectionless protocol. The DirectPlay 8 Protocol is always implemented in conjunction with DirectPlay 8 Host and Port Enumeration Protocol and the DirectPlay 8 Core and Service Providers Protocol. The DirectPlay 8 Protocol is required for use by the DirectPlay 8 Core and Service Providers Protocol.

1.5 Prerequisites/Preconditions

To establish a DirectPlay 8 Protocol connection, the consuming application on one system must have decided to listen for a new incoming connection or connections; that is, to become a server in traditional networking parlance. The application on another system must have discovered this server via external means, such as a game server list or **LAN** broadcast discovery that uses the DirectPlay 8 Host and Port Enumeration Protocol.

1.6 Applicability Statement

The DirectPlay 8 Protocol is intended for use in multiplayer game communication where mixed reliable and unreliable, bidirectional, peer-to-peer traffic is desired. It is not recommended for file transfer, or for applications with robust security needs that cannot provide them at other layers such as IPSec. It is also not intended as a generic replacement for the **Transmission Control Protocol (TCP)**.

1.7 Versioning and Capability Negotiation

This document covers versioning issues in the following areas:

- **Supported Transports**: This protocol should be implemented on top of UDP, but may be implemented on top of any connectionless, datagram-oriented protocol, as discussed in section 2.1.
- **Protocol Versions**: The DirectPlay 8 Protocol has the following three feature version levels.
  - The base implementation may report any version value between 0x00010000 through 0x00010004 in the **dwCurrentProtocolVersion** field in the CONNECT, or CONNECTED messages.
  - Version 0x00010005, which supports all the base features, plus signing.
  - Version 0x00010006, which supports all features, including coalescence.

  These features are defined in section 2.2.
- **Security and Authentication Methods**: The DirectPlay 8 Protocol does not natively provide robust authentication or encryption. It provides optional signing mechanisms that can be used to mitigate Denial of Service (DoS) attacks, as discussed in sections 3.1.4.4 and 3.1.5.2.7.
- **Capability Negotiation**: This protocol detects some features by inspecting the **dwCurrentProtocolVersion** field in the CONNECT, CONNECTED, or CONNECTED_SIGNED messages, as described below in this section.

The fast, or full, signing packet authentication mechanisms are used to mitigate Denial of Service attacks. Either both partners must use the functionality when sending and require it when receiving, or neither partner may use it. Therefore, if the version number in a received CONNECT packet is not at least the 0x00010005 value that indicates signing support, and the listener is using a signing mode, it must ignore the packet. Similarly, if the connector receives a CONNECTED packet instead of a CONNECTED_SIGNED packet, and is expecting to use a signing mode, it must ignore the response.

In order to use the coalescence mechanism to combine multiple, small messages into a single packet, the sender must first make sure that the receiver will understand the packet format. This is signaled by the receiver that indicates a version value of 0x00010006 in its CONNECT, CONNECTED, or CONNECTED_SIGNED message.

1.8 Vendor-Extensible Fields

The DirectPlay 8 Protocol does not have any vendor-extensible fields.

1.9 Standards Assignments

There are no standard assignments for this protocol.

2 Messages

The following sections specify how DirectPlay 8 Protocol messages are transported and DirectPlay 8 Protocol message syntax.

2.1 Transport

DirectPlay 8 Protocol messages are nominally transported over UDP by using application-specific port numbers. They are processed by receivers prepared to handle DirectPlay 8 Protocol: Host and Port Enumeration Protocol messages as well, and are distinguished from such enumeration messages by their first UDP payload byte being nonzero.

The DirectPlay 8 Protocol does not require usage of UDP. It MAY be implemented on any connectionless, datagram-oriented protocol. It is not required, and it is in fact not recommended that the underlying transport provider natively provide reliable messaging.

The DirectPlay 8 Protocol does not negotiate transport providers. If the transport provider to be used is ambiguous, the implementation MUST provide its own mechanism for distinguishing among providers. The DirectPlay 8 Protocol assumes that all remote partners are using the same or binary compatible transport providers.

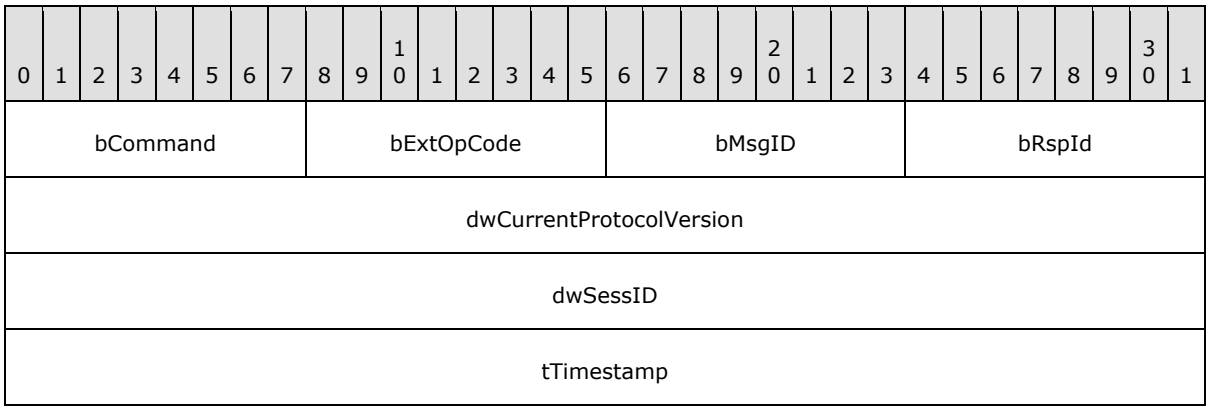
2.2 Message Syntax

2.2.1 Command Frames (CFRAMEs)

Command frames are special control frames that to do not carry application payload data. They are identified by not having the PACKET_COMMAND_DATA flag (0x01) set in their **bCommand** fields.

2.2.1.1 CONNECT

The CONNECT packet is used to request a connection. If accepted, the response is a CONNECTED (section 2.2.1.2) packet or a CONNECTED_SIGNED (section 2.2.1.3) packet, depending on whether packet signing is enabled.



bCommand (1 byte): A command-code bitmask that contains values that are combined by using the bitwise OR operation from the following table. The PACKET_COMMAND_CFRAME flag MUST be set, and the PACKET_COMMAND_POLL flag SHOULD NOT be set, although the recipient SHOULD still respond immediately. All other bits MUST be 0 and the packet MUST be rejected if they are not.

Value	Meaning
0x80	PACKET_COMMAND_CFRAME (command frame)
0x08	PACKET_COMMAND_POLL (acknowledge immediately)

bExtOpCode (1 byte): Extended operation code.

Value	Meaning
0x01	FRAME_EXOPCODE_CONNECT

bMsgID (1 byte): A message identifier used to correlate responses. The initial value SHOULD be 0 and SHOULD be incremented each time the connect packet is retried. The recipient MUST echo the value in **bRspId** when responding.

bRspId (1 byte): Not used in connect packets. This MUST be set to 0 and ignored on receipt.

dwCurrentProtocolVersion (4 bytes): The version number of the requestor's DirectPlay 8 Protocol, in **little-endian** byte order, where the upper 16 bits are considered a major version number and the lower 16 bits are considered a minor version number. The major version number MUST be set to 0x0001; otherwise, the packet MUST be ignored. The minor version number SHOULD <1> be set to 0x0006 to indicate support for all features. The recipient SHOULD be prepared to support older message formats used by earlier minor versions, but MUST ignore this packet if it does not. To ensure security, the packet MUST be ignored if the recipient is using signing but the minor version number is less than 0x0005.

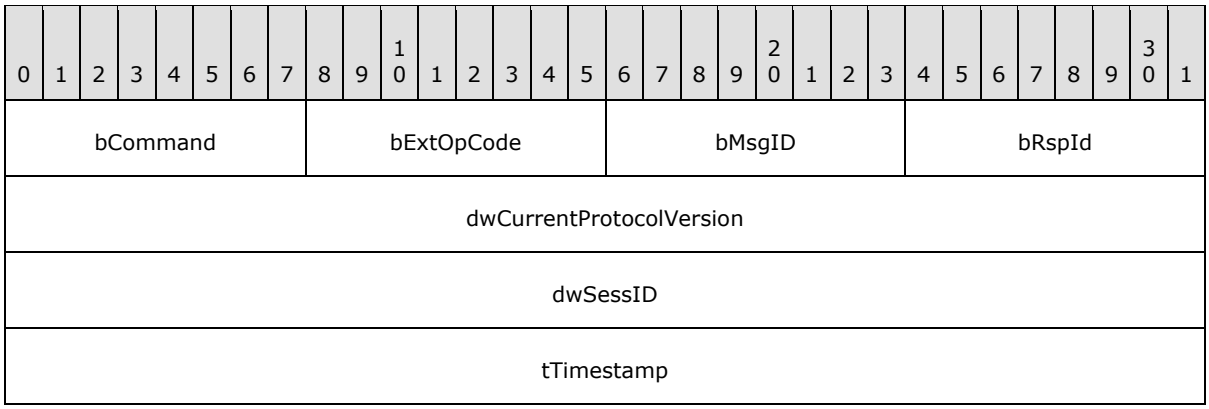
Value	Meaning
0x00010000	Protocol version number 1.0, indicating base functionality.
0x00010005	Protocol version number 1.5, indicating signing support.
0x00010006	Protocol version number 1.6, indicating coalescence support.

dwSessID (4 bytes): The session identifier used to correlate responses. The value is dependent upon the implementation and SHOULD be a random, non-predictable number. This MUST NOT be 0, unless **dwCurrentProtocolVersion** indicates a minor version less than 0x0005. This MUST remain the same value when retrying the CONNECT packet. The recipient MUST echo the value in **dwSessID** when responding.

tTimestamp (4 bytes): Requestor's system tick count, in millisecond units, specified in little-endian byte order.

2.2.1.2 CONNECTED

The CONNECTED packet is used to accept a connection request or complete a connection handshake when signing is not enabled.



bCommand (1 byte): A command-code bitmask that contains values that are combined by using the bitwise OR operation from the following table. The PACKET_COMMAND_CFRAME flag MUST be set. The PACKET_COMMAND_POLL flag MUST be set by a listener accepting a connection request, and MUST NOT be set by a connector completing the connection handshake, although the listener SHOULD still respond immediately in the latter case. All other bits MUST be 0 and the packet MUST be rejected if they are not.

Value	Meaning
0x80	PACKET_COMMAND_CFRAME (command frame)
0x08	PACKET_COMMAND_POLL (acknowledge immediately)

bExtOpCode (1 byte): An extended operation code.

Value	Meaning
0x02	FRAME_EXOPCODE_CONNECTED

bMsgID (1 byte): A message identifier. The initial value SHOULD be 0, and SHOULD be incremented if the packet is retried.

bRspId (1 byte): A response identifier. This value MUST be set to the value of the **bMsgID** field in the CONNECT or CONNECTED message to which this is a response.

dwCurrentProtocolVersion (4 bytes): The version number of the requestor's DirectPlay 8 Protocol, in little-endian byte order, where the upper 16 bits are considered a major version number and the lower 16 bits are considered a minor version number. The major version number MUST be set to 0x0001, or the packet MUST be ignored. The minor version number SHOULD<2> be set to 0x0006 to indicate support for all features. The recipient SHOULD be prepared to support older message formats used by earlier minor versions, but MUST ignore this packet if it does not.

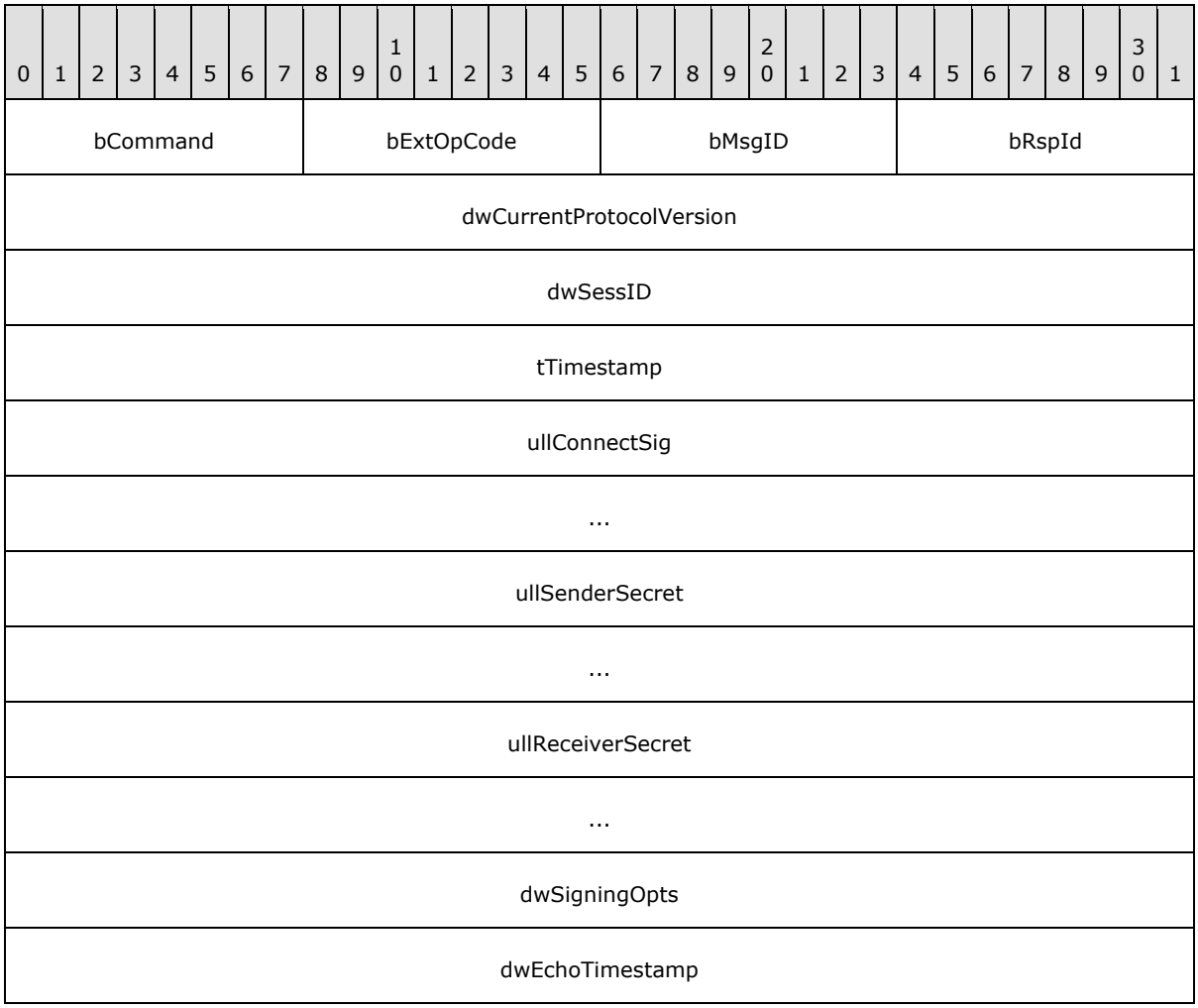
Value	Meaning
0x00010000	Protocol version number 1.0, indicating base functionality.
0x00010005	Protocol version number 1.5, indicating signing support.
0x00010006	Protocol version number 1.6, indicating coalescence support.

dwSessID (4 bytes): The session identifier. This value MUST be set to the value of **dwSessID** specified in the CONNECT or CONNECTED message to which this is a response.

tTimestamp (4 bytes): The sender's system tick count, in millisecond units, specified in little-endian byte order.

2.2.1.3 CONNECTED_SIGNED

The CONNECTED_SIGNED packet is used to accept a connection request or complete a connection handshake when signing is enabled.



bCommand (1 byte): The commandcode bitmask that contains values that are combined by using the bitwise OR operation from the following table. The PACKET_COMMAND_CFRAME flag MUST be set. The PACKET_COMMAND_POLL flag MUST be set by a listener accepting a connection request, and MUST NOT be set by a connector completing the connection handshake, although the listener SHOULD still respond immediately in the latter case. All other bits MUST be 0 and the packet MUST be rejected if they are not.

Value	Meaning
0x80	PACKET_COMMAND_CFRAME (command frame)
0x08	PACKET_COMMAND_POLL (acknowledge immediately)

bExtOpCode (1 byte): An extended operation code.

Value	Meaning
0x03	FRAME_EXOPCODE_CONNECTED_SIGNED

bMsgID (1 byte): The message identifier. The initial value SHOULD be 0, and SHOULD be incremented by connectors if the packet is retried. Listeners SHOULD choose to avoid keeping any state by using the **ullConnectSig** cookie, and MAY always set this to 0.

bRspId (1 byte): The response identifier. This value MUST be set to the value of **bMsgID** field in the CONNECT or CONNECTED_SIGNED message to which this is a response.

dwCurrentProtocolVersion (4 bytes):

The version number of the requestor's DirectPlay 8 Protocol, in little-endian byte order, where the upper 16 bits are considered a major version number and the lower 16 bits are considered a minor version number. The major version number MUST be set to 0x0001 or the packet MUST be ignored. The minor version number SHOULD<3> be set to 0x0006 to indicate support for all features and MUST be set to 0x0005 or higher. The recipient SHOULD be prepared to support older message formats used by earlier minor versions, but MUST ignore this packet if it does not.

Value	Meaning
0x00010000	Protocol version number 1.0, indicating base functionality.
0x00010005	Protocol version number 1.5, indicating signing support.
0x00010006	Protocol version number 1.6, indicating coalescence support.

dwSessID (4 bytes): The session identifier. This value MUST be set to the value of **dwSessID**, as specified in the CONNECT or CONNECTED_SIGNED message to which this is a response.

tTimestamp (4 bytes): The sender's system tick count, in millisecond units, specified in little-endian byte order.

ullConnectSig (8 bytes): The listener cookie used to validate the connect handshake without keeping state. Connectors MUST echo the value specified in the CONNECTED_SIGNED message to which this is a response. Listeners MAY specify any value, and SHOULD generate one that can be used to verify that the connector saw the listener's CONNECTED_SIGNED message. For more information, see section 3.1.5.1.3.

ullSenderSecret (8 bytes): The initial value for generating signatures on packets sent by the connector to the listener, in little-endian byte order. This MUST be 0 when sent by the listener, and MUST be nonzero when sent by the connector. Connectors SHOULD generate a cryptographically-secure random number.

ullReceiverSecret (8 bytes): The initial value for generating signatures on packets sent by the listener to the connector, in little-endian byte order. This MUST be 0 when sent by the listener,

and MUST be nonzero when sent by the connector. Connectors SHOULD generate a cryptographically-secure random number.

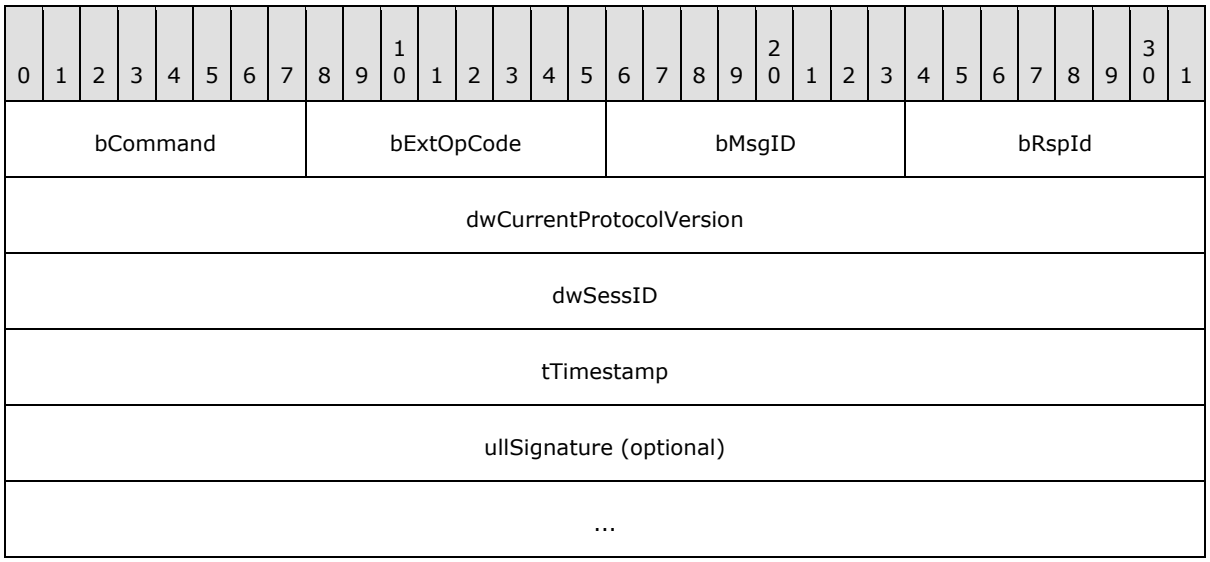
dwSigningOpts (4 bytes): Option flag values, in little-endian byte order. One or the other described flag MUST be set, but not both. All other bits SHOULD be 0 and MUST be ignored upon receipt.

Value	Meaning
0x00000001	PACKET_SIGNING_FAST (use signing cookie only)
0x00000002	PACKET_SIGNING_FULL (sign a digest of packet contents)

dwEchoTimestamp (4 bytes): Echo of the **tTimestamp** field specified in the CONNECT or CONNECTED_SIGNED message to which this is a response.

2.2.1.4 HARD_DISCONNECT

The HARD_DISCONNECT packet is used to quickly disconnect or acknowledge quick disconnection without waiting for remaining packets to be delivered.



bCommand (1 byte): The command-code bitmask that contains values that are combined by using the bitwise OR operation from the following table. The PACKET_COMMAND_CFRAME flag MUST be set. The PACKET_COMMAND_POLL flag SHOULD NOT be set but the recipient SHOULD still respond immediately. All other bits MUST be 0 and the packet MUST be rejected if they are not.

Value	Meaning
0x80	PACKET_COMMAND_CFRAME (command frame)
0x08	PACKET_COMMAND_POLL (acknowledge immediately)

bExtOpCode (1 byte): An extended operation code.

Value	Meaning
0x04	FRAME_EXOPCODE_HARD_DISCONNECT

bMsgID (1 byte): The message identifier. The value SHOULD be the next **CFRAME sequence ID** value for the sender, and MUST be ignored by the recipient.

bRspId (1 byte): The response identifier. This value SHOULD be 0, unless the connection is using PACKET_SIGNING_FULL, in which case it MUST be set to the sequence ID of the next **DFRAME** that would have been sent.

dwCurrentProtocolVersion (4 bytes): The version number of the requestor's DirectPlay 8 Protocol, in little-endian byte order. The value SHOULD match the value previously sent in a CONNECT, CONNECTED, or CONNECTED_SIGNED packet, and MUST be ignored upon receipt.

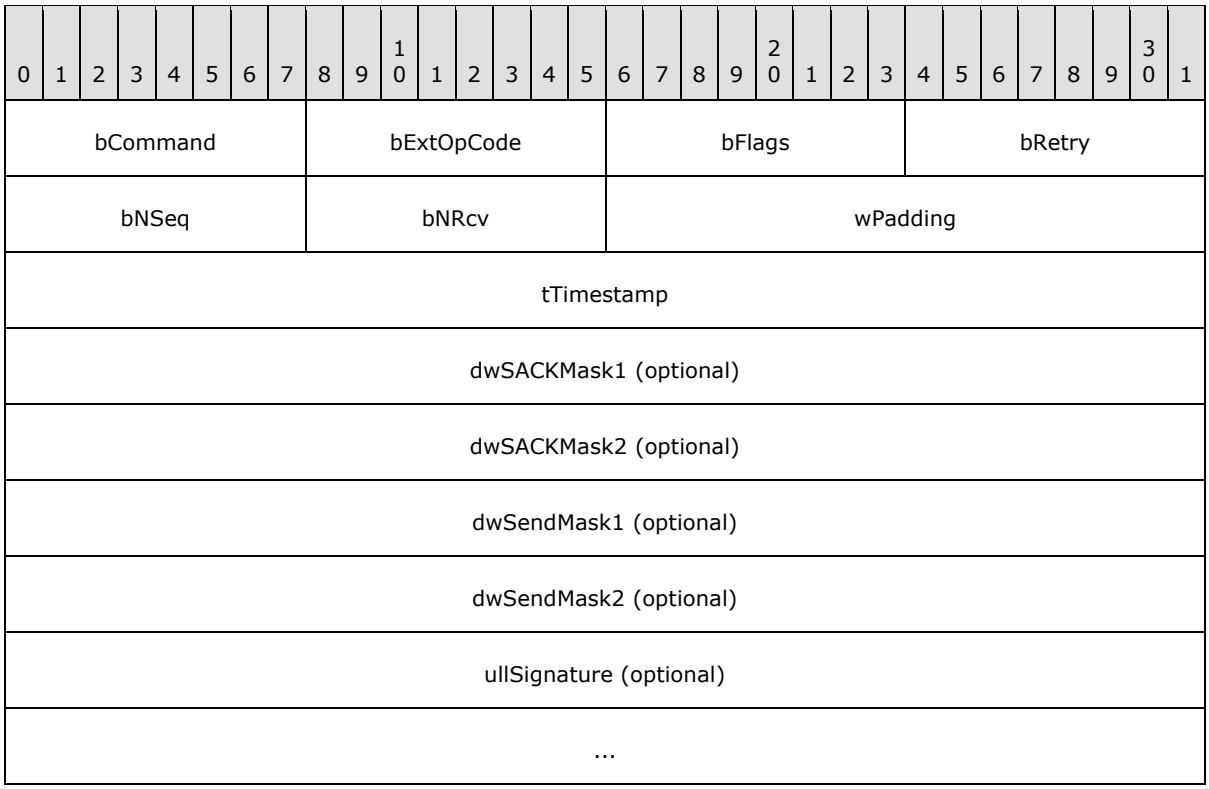
dwSessID (4 bytes): The session identifier. This value MUST be set to the same **dwSessID** value that is specified in the CONNECT message originally associated with the connection.

tTimestamp (4 bytes): The sender's system tick count, in millisecond units, specified in little-endian byte order.

ullSignature (8 bytes): If the connection was established using signing, this MUST be the signature of the packet using the agreed-upon signing algorithm. The packet sequence ID to be used in the calculation is the value in **bRspId**. This field MUST NOT be present if signing is not enabled for the connection.

2.2.1.5 SACK

The SACK packet is used to selectively acknowledge outstanding packets. Packet acknowledgment is typically bundled in all user data packets using the **bSeq** and **bNRec** fields found in the DFRAME header; however, the SACK packet is used when a dedicated acknowledgment is requested (that is, when the PACKET_COMMAND_POLL bit in the **bCommand** header field is set) or when no user data remains for further bundled acknowledgments.



bCommand (1 byte): The command-code bitmask that contains bitwise OR values from the following table. The PACKET_COMMAND_CFRAME flag MUST be set. The PACKET_COMMAND_POLL flag SHOULD NOT be set, and SHOULD be ignored by recipients. All other bits MUST be 0 and the packet MUST be rejected if they are not.

Value	Meaning
0x80	PACKET_COMMAND_CFRAME (command frame)
0x08	PACKET_COMMAND_POLL (acknowledge immediately)

bExtOpCode (1 byte): An extended operation code.

Value	Meaning
0x06	FRAME_EXOPCODE_SACK

bFlags (1 byte): Status flags. The value can be one or more of the following. All other bits MUST be 0. The SACK_FLAGS_RESPONSE flag SHOULD be set and **bRetry** SHOULD be filled in properly.

Value	Meaning
0x01	SACK_FLAGS_RESPONSE (**bRetry** field is valid)
0x02	SACK_FLAGS_SACK_MASK1 (low 32 bits of the SACK mask are present)

Value	Meaning
0x04	SACK_FLAGS_SACK_MASK2 (high 32 bits of the SACK mask are present)
0x08	SACK_FLAGS_SEND_MASK1 (low 32 bits of the send mask are present)
0x10	SACK_FLAGS_SEND_MASK2 (high 32 bits of the send mask are present)

bRetry (1 byte): Boolean that indicates whether the last received packet was a retry. This value MUST be ignored if SACK_FLAGS_RESPONSE is not set. Otherwise, the value SHOULD be 0 if the last received DFRAME for the connection was not marked as a retry, otherwise the value SHOULD be nonzero. Recipients MUST NOT require any particular bit or bits to be set in the nonzero case, only that at least one is set.

bNSeq (1 byte): SACK packets do not have sequence numbers of their own. This field represents the sequence number of the next data frame to send.

bNRcv (1 byte): The expected sequence number of the next packet received. If the SACK_FLAGS_SACK_MASK1 flag is set, the **bNRcv** field is supplemented with an additional DWORD bitmask field that selectively acknowledges frames with sequence numbers higher than **bNRcv**.

wPadding (2 bytes): Unused. This SHOULD be set to 0 and MUST be ignored by the recipient.

tTimestamp (4 bytes): The sender's system tick count, in millisecond units, specified in little-endian byte order.

dwSACKMask1 (4 bytes): The optional low 32 bits of the selective acknowledgement mask, in little-endian byte order. The existence of this field in the packet is dependent upon the **bFlags** field having SACK_FLAGS_SACK_MASK1 set.

dwSACKMask2 (4 bytes): Optional high 32 bits of the selective acknowledgement mask, in little-endian byte order. The existence of this field in the packet is dependent upon the **bFlags** field having SACK_FLAGS_SACK_MASK2 set.

dwSendMask1 (4 bytes): Optional low 32 bits of the send mask, in little-endian byte order. The existence of this field in the packet is dependent upon the **bFlags** field having SACK_FLAGS_SEND_MASK1 set.

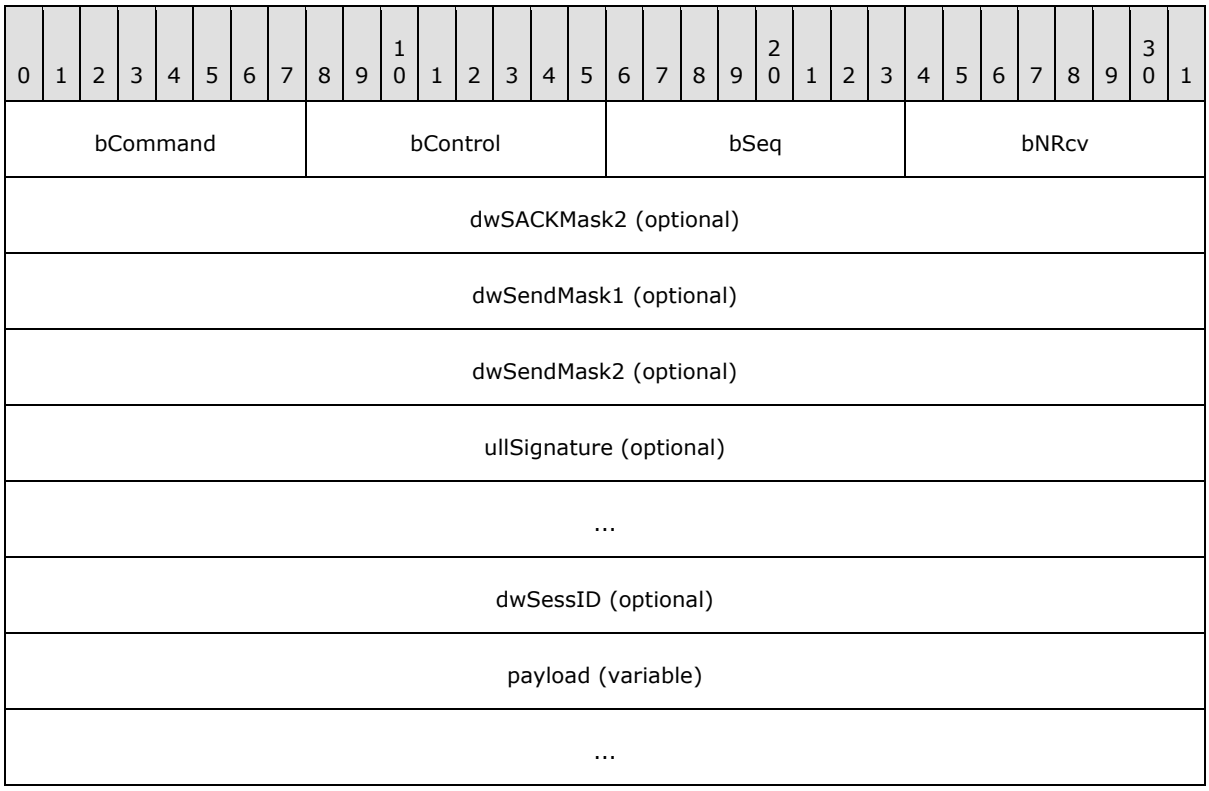
dwSendMask2 (4 bytes): Optional high 32 bits of the send mask, in little-endian byte order. The existence of this field in the packet is dependent upon the **bFlags** field having SACK_FLAGS_SEND_MASK2 set.

ullSignature (8 bytes): If the connection was established using signing, this MUST be the signature of the packet using the agreed-upon signing algorithm. The packet sequence ID to be used in the calculation is the value in **bNSeq**. This field MUST NOT be present if signing is not enabled for the connection.

2.2.2 Data Frames (DFRAMEs)

Data frames exist in the standard connection sequence space and typically carry application payload data. They all are identified by having the PACKET_COMMAND_DATA flag (0x01) set in their **bCommand** field. The total size of the DFRAME header and payload SHOULD be less than the **Maximum Transmission Unit (MTU)** of the underlying protocols and network. If larger messages are to be transmitted, the implementation SHOULD break the payload into multiple DFRAME packets, send the portions sequentially, and set the PACKET_COMMAND_NEW_MSG and

PACKET_COMMAND_END_MSG flags on the first and final DFRAMEs. Message payloads split this way MUST NOT be coalesced with other payloads.



bCommand (1 byte): Command field. The PACKET_COMMAND_DATA flag MUST be set. If the packet is a KeepAlive, then the PACKET_COMMAND_RELIABLE, PACKET_COMMAND_SEQUENTIAL, and PACKET_COMMAND_END_MSG flags MUST be set. If the packet contains coalesced payloads, then the PACKET_COMMAND_NEW_MSG and PACKET_COMMAND_END_MSG flags MUST be set. All other flags are optional.

Value	Meaning
0x01	PACKET_COMMAND_DATA (frame contains user data)
0x02	PACKET_COMMAND_RELIABLE (frame SHOULD be delivered reliably)
0x04	PACKET_COMMAND_SEQUENTIAL (frame SHOULD be indicated sequentially)
0x08	PACKET_COMMAND_POLL (partner SHOULD acknowledge immediately)
0x10	PACKET_COMMAND_NEW_MSG (data frame is first in message)
0x20	PACKET_COMMAND_END_MSG (data frame is last in message)
0x40	PACKET_COMMAND_USER_1 (first consumer-controlled flag)
0x80	PACKET_COMMAND_USER_2 (second consumer-controlled flag)

bControl (1 byte): Control field. The following flags MAY be specified.

Value	Meaning
0x01	PACKET_CONTROL_RETRY (indicates whether the frame is a retry for this sequence number)
0x02	PACKET_CONTROL_KEEPALIVE_OR_CORRELATE (0x00010005 versions and up, indicates that the frame is a keep-alive frame) (versions less than 0x00010005 requests a dedicated acknowledgement from the receiver)
0x04	PACKET_CONTROL_COALESCE (0x00010006 versions and up, packet contains multiple payloads as described in section 2.2.3)
0x08	PACKET_CONTROL_END_STREAM (last packet in the stream; indicates disconnect)
0x10	PACKET_CONTROL_SACK1 (low 32 bits of the Selective Acknowledgement mask are present)
0x20	PACKET_CONTROL_SACK2 (high 32 bits of the Selective Acknowledgement mask are present)
0x40	PACKET_CONTROL_SEND1 (low 32 bits of the cancel-send mask are present)
0x80	PACKET_CONTROL_SEND2 (high 32 bits of the cancel-send mask are present)

bSeq (1 byte): The sequence number of the packet.

bNRcv (1 byte): The expected sequence number of the next packet received.

dwSACKMask2 (4 bytes): Optional high 32 bits of the Selective Acknowledgement mask, in little-endian byte order. The existence of this field in the packet is dependent upon the **bFlags** field having SACK_FLAGS_SACK_MASK2 set.

dwSendMask1 (4 bytes): Optional low 32 bits of the send mask, in little-endian byte order. The existence of this field in the packet is dependent upon the **bFlags** field having SACK_FLAGS_SEND_MASK1 set.

dwSendMask2 (4 bytes): Optional high 32 bits of the send mask, in little-endian byte order. The existence of this field in the packet is dependent upon the **bFlags** field having SACK_FLAGS_SEND_MASK2 set.

ullSignature (8 bytes): If the connection was established using signing, this MUST be the signature of the packet using the agreed-upon signing algorithm. The packet sequence ID to be used in the calculation is the value in **bSeq**. This field MUST NOT be present if signing is not enabled for the connection.

dwSessID (4 bytes): The session identifier. If the packet is marked as PACKET_CONTROL_KEEPALIVE on connections reported as version 0x00010005 or higher, this MUST be present and MUST be the only payload data for the packet. It MUST NOT be present otherwise. The value MUST be set to the same **dwSessID** value specified in the CONNECT message originally associated with the connection.

payload (variable): Consumer payload data. The payload size is the total UDP frame size minus the amount of data consumed by data frame headers up to this point. If the PACKET_CONTROL_COALESCE flag is set, the payload is not a single message or portion of a message, but is instead organized according to the coalesced payload format, as specified in section 2.2.3.

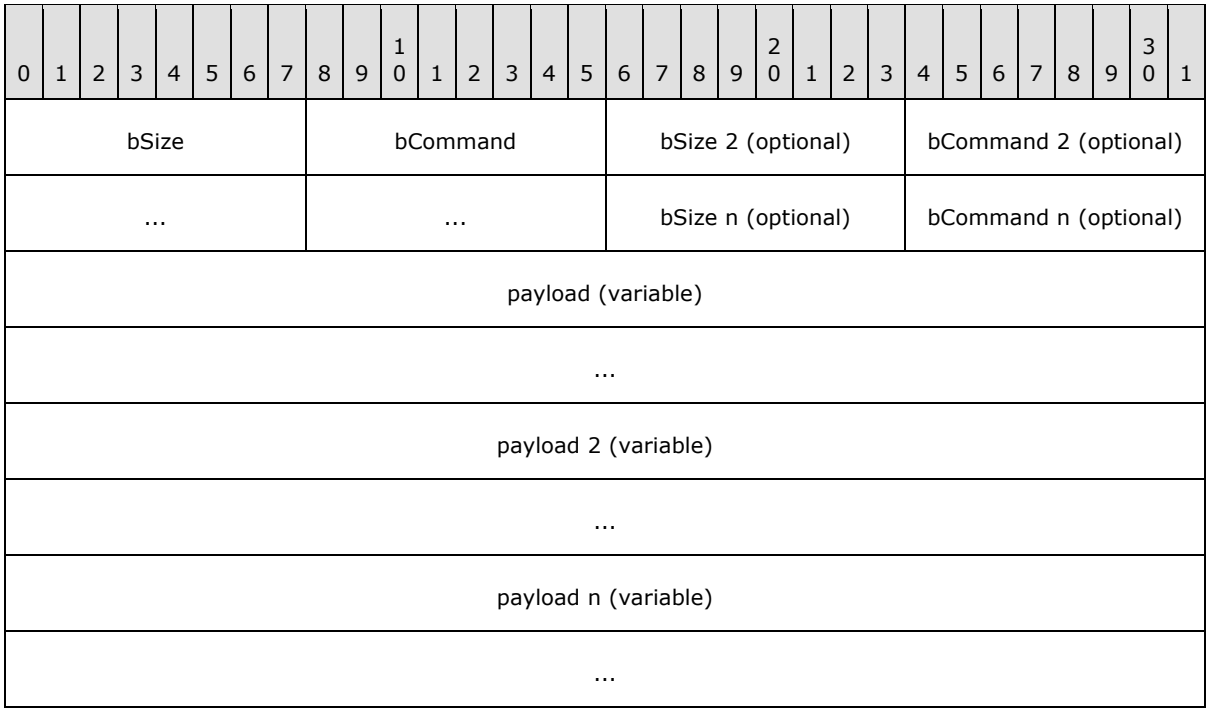
2.2.3 Coalesced Payloads

Coalesced payloads are a special form of payload within standard data frames. When the PACKET_CONTROL_COALESCE flag is set on the outer data frame header **bControl** field, the payload is interpreted using this format. Frames with coalesced payloads MUST have the PACKET_COMMAND_NEW_MSG and PACKET_COMMAND_END_MSG flags set on the outer data frame header **bCommand** field.

Between one and 32 two-byte headers are placed at the beginning of the buffer. The buffer MUST NOT contain more than 32 coalesce headers. If there is an odd number of coalesce headers, two extra bytes of zero padding MUST be added at the end to align the subsequent data on a 32 bit boundary. The last non-padded coalesce header MUST have the PACKET_COMMAND_END_COALESCE flag set in its **bCommand** field.

Following the headers are one to 32 payloads, where the sizes of each are indicated in the corresponding headers that were added in the same order. If the payload size is not a multiple of 32 bits, and it is not the last payload in the message, then 1 to 3 bytes of zero padding MUST be added to align the beginning of the next payload on a 32-bit boundary. The sizes indicated in the coalesce headers MUST NOT include any padding so as to preserve the message size as originally sent. The receiver MUST infer alignment padding when processing the payloads and SHOULD indicate the messages to the consumer using the unpadded size.

The total size of the DFRAME with coalesced payloads SHOULD NOT be larger than the MTU of the underlying protocol and network. Each individual payload MUST NOT be larger than what is fit in the coalesced DFRAME.



bSize (1 byte): The least significant 8 bits of the size of the coalesced payload. The value is combined with the optional PACKET_COMMAND_COALESCE_BIG_1, PACKET_COMMAND_COALESCE_BIG_2, and PACKET_COMMAND_COALESCE_BIG_3 flags to determine the actual size of the payload. This MUST NOT be larger than what can fit in a

standard data frame, including any size already used to store previous coalesce headers and payloads.

bCommand (1 byte): Command field for the coalesced message. The PACKET_COMMAND_DATA flag MUST be set. If the packet is a KeepAlive, the PACKET_COMMAND_RELIABLE, PACKET_COMMAND_SEQUENTIAL, and PACKET_COMMAND_END_MSG flags MUST be set. All other flags are optional.

Value	Meaning
0x01	PACKET_COMMAND_END_COALESCE (this is the final coalesced payload in the frame)
0x02	PACKET_COMMAND_RELIABLE (payload SHOULD be delivered reliably)
0x04	PACKET_COMMAND_SEQUENTIAL (payload SHOULD be indicated sequentially)
0x08	PACKET_COMMAND_COALESCE_BIG_1 (bit 9 of the coalesced payload size)
0x10	PACKET_COMMAND_COALESCE_BIG_2 (bit 10 of the coalesced payload size)
0x20	PACKET_COMMAND_COALESCE_BIG_3 (bit 11 of the coalesced payload size, the most significant bit)
0x40	PACKET_COMMAND_USER_1 (first consumer-controlled flag)
0x80	PACKET_COMMAND_USER_2 (second consumer-controlled flag)

bSize 2 (1 byte): See **bSize** above.

bCommand 2 (1 byte): See **bCommand** above.

bSize n (1 byte): See **bSize** above.

bCommand n (1 byte): See **bCommand** above.

payload (variable): Consumer payload data.

payload 2 (variable): See **payload** above.

payload n (variable): See **payload** above.

3 Protocol Details

3.1 Common Details

Although there are no traditional client or server roles after a connection is established, a system MUST initially determine whether to establish an outbound connection or listen for an inbound one. After both sides have seen a CONNECTED message from the other, they can mark the connection as established. The following diagram shows the states for a system performing the DirectPlay 8 Protocol unsigned connect sequence.

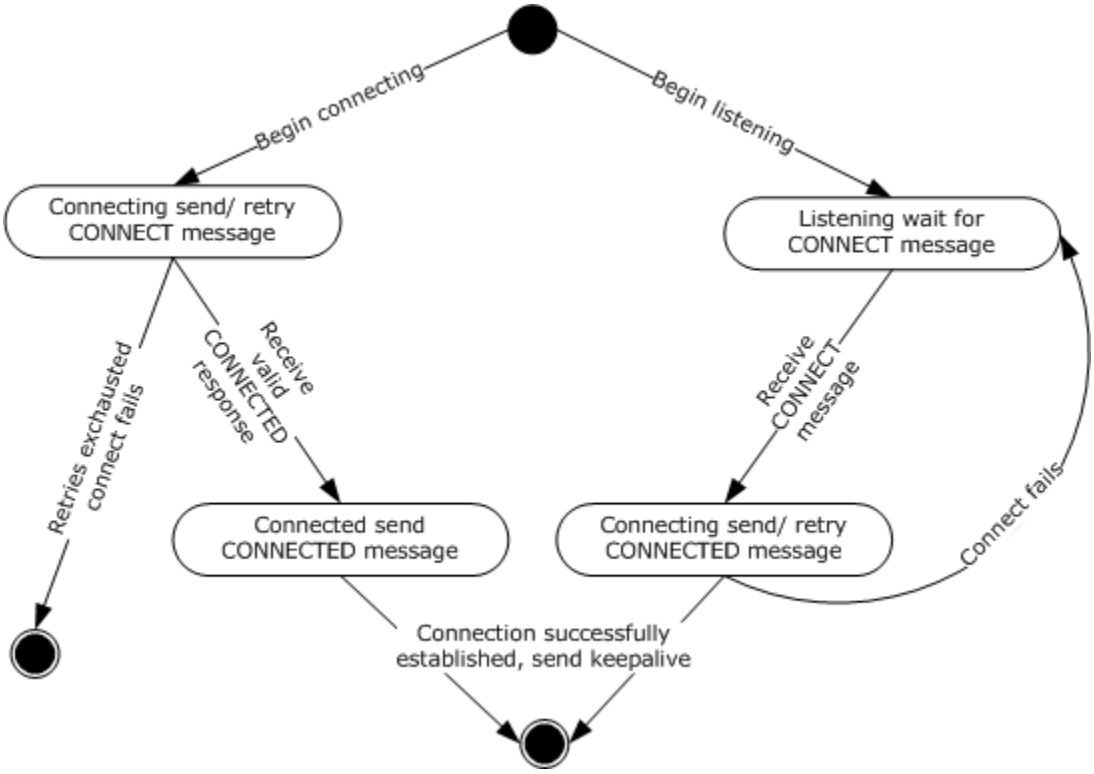


Figure 1: DirectPlay8 Reliable Protocol system states

When connected, the protocol behaves identically for both the connecting and listening systems.

3.1.1 Abstract Data Model

This section describes a conceptual model of possible data organization that an implementation maintains to participate in this protocol. The described organization is provided to facilitate the explanation of how the protocol behaves. This document does not mandate that implementations adhere to this model as long as their external behavior is consistent with that described in this document.

dwSessID: The session ID used to establish the connection. This value is referenced in HARD_DISCONNECTCFRAMEs and KeepAlive DFRAMEs.

Local Secrets: The 64-bit current local secret and previous local secret values for use when sending over fast and full signed connections.

Local Secret Modifier: The 64-bit current local secret modifier value for use when sending for full signed connections. This MUST be derived from the most recently sent reliable message with the lowest sequence ID.

Next Receive: The next 8-bit packet sequence ID expected to be received, indicating acknowledgement of all packets up to this ID.

Next Send: The next 8-bit packet sequence ID that will be sent.

Remote Secrets: The 64-bit current remote secret and previous remote secret values for use when receiving over fast and full signed connections.

Remote Secret Modifier: The 64-bit current remote secret modifier value for use when receiving for full signed connections. This MUST be derived from the most recently received reliable message with the lowest sequence ID.

Retry Counter: A per-packet counter of how many times the individual CONNECT, CONNECTED, or reliable DFRAME has been retried.

RTT**:** A recent sample or running average of the round-trip time for the connection. This SHOULD be used to schedule retries.

Send Mask**:** A sliding window bitmask that indicates unreliable DFRAMEs that will not be retried. The window base reference SHOULD be the current Next Send, and work backward for up to 64 bits (messages).

3.1.2 Timers

3.1.2.1 Connect Retry Timer

The timer used to retry CONNECT and CONNECTED messages if no response is received. Implementations MAY retry as many times as desired, at any frequency desired. RECOMMENDED values are for the first retry to be 200 msec, doubling every subsequent retry, with a cap at five seconds and 14 retries.

3.1.2.2 Delayed Acknowledgement Timer

The timer used to reduce frequency of dedicated acknowledgements (ACK) so that they can be piggybacked onto return traffic or multiple receives can be covered by a single reply. The RECOMMENDED value is 100 msec for normal ACKs and 20 msec when acknowledging out-of-order or duplicate packets, but it MAY be any value that maximizes ACK coalescence opportunity without introducing an undesirable latency under the particular application circumstances.

3.1.2.3 Delayed Send Mask Timer

The timer used to reduce frequency of dedicated packets that contain a send mask so that the mask can be piggybacked onto additional traffic. The RECOMMENDED value is 40 msec but it MAY be any value that maximizes a send mask coalescence opportunity under the particular application circumstances.

3.1.2.4 Hard Disconnect Timer

The timer used to space multiple hard disconnect packets over time to increase the likelihood that one or more arrive. The RECOMMENDED value is one half of the current round-trip time (RTT), with a minimum value of 10 msec. The maximum interval SHOULD be capped at 500 msec, but MAY be any value that is appropriate for particular application requirements or network circumstances.

3.1.2.5 Retry Timer

The timer used to track when a message SHOULD be considered to have been dropped and either needs to be retried or causes a send mask to be sent. The RECOMMENDED values are for the first retry to be 2.5 round-trip time (RTT) plus the delayed acknowledgement (ACK) time out (nominally 100 msec). It is also RECOMMENDED that there be linear backoff for the second and third retries, exponential backoff for subsequent retries 4-8, and an overall cap at five seconds and 10 retries.

3.1.2.6 KeepAlive Timer

The timer used to send a minimal reliable packet to keep the connection alive when no traffic has been received. The RECOMMENDED value is 25 seconds of inactivity.

3.1.3 Initialization

There are no special initialization procedures for the DirectPlay 8 Protocol.

3.1.4 Higher-Layer Triggered Events

3.1.4.1 Listening

Higher layers that want to accept new connections SHOULD place the DirectPlay 8 Protocol in listening mode. The protocol SHOULD begin treating CONNECT messages from previously unknown sources as attempts to establish a connection, and respond with CONNECTED or CONNECTED_SIGNED messages as described in section 3.1.5.1.1.

3.1.4.2 Connecting

Higher layers that want to establish a new connection SHOULD cause the DirectPlay 8 Protocol to send a CONNECT message to the desired destination. The connect timer SHOULD also be scheduled to trigger resending a CONNECT message if no response is received.

3.1.4.3 Disconnecting Gracefully

Higher layers that no longer want to communicate over an established connection SHOULD cause the DirectPlay 8 Protocol to send a reliable DFRAME message with the PACKET_CONTROL_END_STREAM flag set. This SHOULD occur after all messages previously queued by the higher layer have been sent to avoid data loss for the upper layer. This PACKET_CONTROL_END_STREAM packet SHOULD be a separate packet with no payload, but MAY be the final queued data packet.

If the connection was established with signing, the DFRAME MUST be signed appropriately as described in section 3.1.4.4.

Implementations MUST NOT send any additional DFRAMEs after sending a packet with the PACKET_CONTROL_END_STREAM flag set to other than retries. It MUST be prepared to continue receiving and acknowledging packets until the remote partner sends its own PACKET_CONTROL_END_STREAM packet.

When the DFRAME that contains a sender's PACKET_CONTROL_END_STREAM indicator has been acknowledged, and the remote partner's own PACKET_CONTROL_END_STREAM has been received and the acknowledgement (ACK) sent, the connection SHOULD be considered terminated.

Higher layers MAY also choose to hard disconnect, as described in section 3.1.4.5.

3.1.4.4 Sending Application Data

Higher layers pass messages to the DirectPlay 8 Protocol for transmission over an established connection. The protocol SHOULD send the packets by using the reliable or unreliable behavior requested.

The higher layer MAY request that PACKET_COMMAND_USER_1, PACKET_COMMAND_USER_2, both flags, or neither flag be set in the **bCommand** field. On reception, the DirectPlay 8 Protocol implementation MUST pass the presence or absence of these bits unchanged to the upper layer and MUST NOT interpret their meaning.

If the message is smaller than the maximum transmission unit (MTU) size supported, the packet MUST have the PACKET_COMMAND_NEW_MSG and PACKET_COMMAND_END_MSG flags set in **bCommand**, and the payload is eligible for coalescence with other payloads if the receiver's version number and the payload sizes allow.

If the message is larger than the MTU size supported, the protocol SHOULD split the message into multiple packets. The implementation SHOULD fill each packet to the maximum size allowed with any remainder in the final packet, but MAY divide the payload portions in any manner it chooses, such as equal portions in all packets. The packets MUST be transmitted in order, and other messages MUST NOT be transmitted until the last packet of the large message is sent. The first packet in the series MUST have the PACKET_COMMAND_NEW_MSG flag set and the last MUST have the PACKET_COMMAND_END_MSG flag set in **bCommand**.

If the connection was established with fast signing, the DFRAME MUST contain a **ullSignature** field set to the 64-bit local secret associated with the local sender; that is, the same value as the CONNECTED_SIGNED frame's **ullSenderSecret** field if the local system performed an outbound connection, or the same value as the CONNECTED_SIGNED frame's **ullReceiverSecret** field if the local system received an inbound connection.

If the connection was established with full signing, the DFRAME MUST contain a **ullSignature** field set to the first 64 bits of the SHA-1 signature digest, as specified in [FIPS180]. The digest MUST be calculated from the following data, in sequence.

1. The entire packet to be sent, extending from the beginning of the DFRAME header and concluding with the final byte of the final mask, payload, or coalesced payload as appropriate, except with the DFRAME **ullSignature** bytes zeroed.

2. The 64-bit current or previous local secret, in little-endian byte order. The local secret to use when validating MUST be selected according to the following logic.

    1. If the packet is not a retry, use the current local secret.

    2. For retried packets, if the next new sequence ID that will be sent is less than 64, and the packet being retried has a sequence ID that is greater than or equal to 192, use the previous local secret.

    3. For all other retried packets, use the current local secret.

For full-signed connections, local secrets are also modified once each time that the 8-bit sequence space wraps to avoid signing all data with the same value. The modification is performed using a modifier value derived from the lowest sequenced reliable payload sent with a sequence ID of less than 192 that is not a KeepAlive. If the lowest sequenced packet that uses the PACKET_COMMAND_RELIABLE flag contains coalesced payloads, the first sub-payload that is marked PACKET_COMMAND_RELIABLE is used to generate the modifier. If no non-KeepAlive reliable payload is sent with a sequence ID between 0 and 191 inclusive, the previous local secret modifier value is reused. The local secret modifier value is initialized to the secret associated with the sender

when the connection was established; that is, it begins with the same value as the CONNECTED_SIGNED frame's **ullSenderSecret** field if the local system performed an outbound connection, and it begins with the same value as the CONNECTED_SIGNED frame's **ullReceiverSecret** field if the local system received an inbound connection.

Upon sending the packet with sequence ID 255 on a full-signed connection, the sender MUST advance the secret by making the current local secret become the previous local secret, and then setting the new current local secret to the first 64 bits of a SHA-1 digest (as specified in [FIPS180]) of the following data, in sequence.

1. The previous 64-bit local secret, in little-endian byte order.

2. The 64-bit local secret modifier value, in little-endian byte order.

DirectPlay 8 Protocol implementations MUST NOT allow more than 64 packets on the network simultaneously. Additional packets SHOULD be queued on the sender until an acknowledgement (ACK) for an earlier sequence ID is received.

Implementations SHOULD also implement TCP-friendly congestion control mechanisms, such as initially allowing only two packets on the network, and gradually increasing the window by one as ACKs arrive without packet loss.

3.1.4.5 Hard Disconnects

Higher layers that want to terminate the connection as quickly as possible SHOULD NOT initiate the graceful disconnect described in section 3.1.4.3. Instead the implementation SHOULD cancel all previous queued messages and terminate any packets that are still waiting for acknowledgements (ACK). It MUST then send one or more HARD_DISCONNECT messages separated by brief intervals to make a best-effort attempt at informing the remote partner of this abrupt termination. It is RECOMMENDED that three HARD_DISCONNECT messages be sent, using the hard disconnect timer to schedule the subsequent attempts as specified in section 3.1.2.4.

If the connection was established with signing, the HARD_DISCONNECT CFRAME MUST be signed appropriately BY using the mechanism described in section 3.1.4.4.

Implementations MUST NOT send any additional data after initiating a hard disconnect. They MAY continue receiving packets until the remote partner acknowledges the termination request by sending its own hard disconnect packets.

When either a HARD_DISCONNECT packet from the remote partner is received or the maximum number of local HARD_DISCONNECT packets have been sent and the final time out has elapsed, the implementation SHOULD consider the connection terminated. See sections 3.1.5.1.4 and 3.1.6.4.

3.1.5 Message Processing Events and Sequencing Rules

When a packet arrives, the recipient SHOULD first check whether it is large enough to be a minimal DFRAME (4 bytes), and that the first byte has the low bit (PACKET_COMMAND_DATA) set. If so, it MUST process the message as a DFRAME (section 3.1.5.2) data frame. Otherwise, if the data is at least 12 bytes and the first byte is either 0x80 or 0x88 (PACKET_COMMAND_CFRAME or PACKET_COMMAND_CFRAME | PACKET_COMMAND_POLL), it MUST process the message as a CFRAME (section 3.1.5.1) command frame. Otherwise the message is not a valid DirectPlay 8 Protocol message and MUST be ignored or passed to other protocols.

3.1.5.1 CFRAMEs

Command frames MUST be handled according to the type of command, that is, the second byte of the packet listed as **bExtOpcode**. If it is not one of the known values in this section, the packet MUST be discarded.

3.1.5.1.1 CONNECT

If the **bExtOpcode** field indicates FRAME_EXOPCODE_CONNECT (0x01), the source address (for example, IPv4 address and port type when running on UDP) for the message SHOULD be checked. If the address corresponds to one with an existing fully-established connection, it SHOULD be ignored. If the address is for a previously received inbound connection that has not completed the handshake process, and if the **dwSessID** field matches the previously received CONNECT, another CONNECTED message SHOULD be sent immediately; otherwise the packet SHOULD be ignored. If the address is for a previously established outbound connection that has not completed the handshake process, the packet SHOULD be ignored.

If the source address does not correspond to any existing connection, it SHOULD be treated as a new connection attempt. If the recipient is not allowing connections, the packet MUST be ignored. Otherwise, it MUST check the **dwCurrentProtocolVersion** field for compatibility and reject incompatible version numbers, as indicated in section 2.2.1.1. If the recipient will require fast or full signing on the connection, it MUST also validate that **dwSessID** is not 0.

If the recipient is not enforcing signing, it SHOULD allocate resources for the new connection and send a CONNECTED response. This includes setting the connect retry timer to continue retrying the CONNECTED reply until either a valid CONNECTED response arrives from the connector, or the maximum number of retries elapses and the connection is terminated.

If the recipient is enforcing signing, it SHOULD NOT allocate resources, but instead send a CONNECTED_SIGNED response that uses a cookie value in its **ullConnectSig** field that can be used to subsequently verify that the connector saw the CONNECTED_SIGNED reply. This is described in more detail in section 3.1.5.1.3.

3.1.5.1.2 CONNECTED

If the **bExtOpcode** field indicates FRAME_EXOPCODE_CONNECTED (0x02), the source address (for example, IPv4 address and port type when running on UDP) for the message SHOULD be checked. If the address does not correspond to one with an existing partially or fully-established connection, it SHOULD be ignored.

If the source address matches that of a previously initiated outbound connection that has not completed the handshake process, the **dwSessID** field MUST match that of the previously sent CONNECT packet, and the PACKET_COMMAND_POLL flag MUST be set in the **bCommand** field before the packet can be accepted. If the connector is enforcing signing, this unsigned response SHOULD be ignored. Otherwise, the connection SHOULD be considered established and a CONNECTED response sent to confirm this connection.

If the source address matches that of a previously initiated inbound connection that has not completed the handshake process, the **dwSessID** field MUST match that of the previously received CONNECT, and the PACKET_COMMAND_POLL flag MUST NOT be set in the **bCommand** field before the packet can be accepted. If the connector is enforcing signing, this unsigned response SHOULD be ignored. Otherwise, the connection SHOULD be considered established.

If the source address matches that of an established connection, the **dwSessID** field MUST match the one used to establish the connection, and the PACKET_COMMAND_POLL flag MUST be set. If so,

this connector's previous CONNECTED response was apparently lost and the connector SHOULD send a duplicate CONNECTED packet. Otherwise, the packet SHOULD be ignored.

3.1.5.1.3 CONNECTED_SIGNED

If the **bExtOpcode** field indicates FRAME_EXOPCODE_CONNECTED_SIGNED (0x03), the source address (for example, IPv4 address and port type when running on UDP) for the message SHOULD be checked.

If the source address matches that of a previously initiated outbound connection that has not completed the handshake process, the **dwSessID** field MUST match that of the previously sent CONNECT packet, and the PACKET_COMMAND_POLL flag MUST be set in the **bCommand** field before the packet can be accepted. The **dwSigningOpts** field MUST have either the PACKET_SIGNING_FAST or the PACKET_SIGNING_FULL flag set, but not both, and the one set MUST exactly match the connector's desired signing mode. If the connector did not intend to use signing, this signed response SHOULD be ignored. Otherwise, the connection SHOULD be considered established, random sender and receiver signing secrets SHOULD be generated, and a CONNECTED_SIGNED response SHOULD be sent to confirm this connection. This CONNECTED_SIGNED response MUST NOT set the PACKET_COMMAND_POLL flag. A reliable KeepAlive data frame MUST also then be scheduled to ensure that the remote side that did not allocate resources yet is prompted to complete the connection establishment if the CONNECTED_SIGNED response is dropped.

If the source address matches that of a previously initiated inbound connection that has not completed the handshake process, the **dwSessID** field MUST match that of the previously received CONNECT packet, and the PACKET_COMMAND_POLL flag MUST NOT be set in the **bCommand** field before the packet can be accepted. If the connector is not using signing, this confirmation SHOULD be ignored. Lastly, the **ullConnectSig** cookie signature field SHOULD be validated to ensure that the sender saw the previous CONNECTED_SIGNED packet. If the signature is not valid, the packet MUST be dropped. Otherwise, the connection SHOULD be allocated and considered established, and the sender and receiver secrets provided SHOULD be saved. A reliable KeepAlive data frame SHOULD also be scheduled to immediately update round-trip time (RTT) measurements.

If the source address matches that of a previously established connection, and the **dwSessID** field does not match the one used to establish the connection, or the PACKET_COMMAND_POLL flag is not set, or the client is not intending to use signing, then this packet MUST be ignored. Otherwise, a duplicate CONNECTED_SIGNED confirmation SHOULD be sent.

If the source address does not match any existing connection, the packet SHOULD be ignored. Note that if the implementation does not allocate resources when receiving the first CONNECT packet, the CONNECTED_SIGNED packets intended for previously initiated inbound connections that have not completed the handshake process would thus not match any existing connection. The **ullConnectSig** cookie field is used to determine whether this is the case, and if the cookie and thus the source address is validated, it SHOULD be handled as previously described.

This document does not prescribe any particular method for generating or validating **ullConnectSig** cookies. If the implementation chooses to utilize this field, it SHOULD incorporate the sender's source address (for example, IP address and port when using a UDP transport provider), the **dwSessID** value, and a time-dependent secret that only the listener knows that uses a cryptographically secure algorithm that makes it difficult to guess.

3.1.5.1.4 HARD_DISCONNECT

If the **bExtOpcode** field indicates FRAME_EXOPCODE_HARD_DISCONNECT (0x04), the source address (for example, IPv4 address and port type when running on UDP) for the message SHOULD

be checked. If the address does not correspond to one with a fully-established connection, it SHOULD be ignored. If the connection used signing, the signature MUST be valid or else the packet MUST be ignored. If a HARD_DISCONNECT message has already been received for the connection, additional actions SHOULD NOT be taken.

If the local node initiated the hard disconnect sequence, the received HARD_DISCONNECT is treated as an acknowledgement (ACK) of the previously sent HARD_DISCONNECT. The hard disconnect timer SHOULD be cancelled, and connection termination SHOULD be considered complete.

Otherwise, the local node is receiving a new request to hard-terminate the connection, and SHOULD abort all outstanding sends, then send three HARD_DISCONNECT ACK packets immediately. The connection SHOULD then be considered terminated.

3.1.5.1.5 SACK

If the **bExtOpcode** field indicates FRAME_EXOPCODE_SACK (0x06), the source address (for example, IPv4 address and port type when running on UDP) for the message SHOULD be checked. If the address does not correspond to one with a fully-established connection, it MUST be ignored. If the connection used signing, the signature MUST be valid; otherwise, the packet MUST be ignored. The **bNSeq**, **bNRcv**, optional selective acknowledgement (SACK), and optional send mask fields SHOULD then be processed using the standard rules in sections 3.1.5.2.1 through 3.1.5.2.4.

A successfully validated SACK packet SHOULD count as a valid receive and thus restart the KeepAlive timer.

3.1.5.2 DFRAMEs

Data frames SHOULD be checked to see if they were sent from an address (for example, IPv4 address and port tuple when running on UDP) to which there is an established connection. If the address is unknown, the packet MUST be discarded.

If the connection enabled signing, and the data frame is not properly signed, it MUST be discarded.

If the remote system reported a version 0x00010005 or higher, and the PACKET_CONTROL_KEEPALIVE bit was set in **bControl**, the packet is a KeepAlive and MUST contain the 32-bit session identifier as a payload. This identifier value MUST match the **dwSessID** value used to establish the connection. The PACKET_CONTROL_COALESCE flag MUST NOT be set in **bControl** on KeepAlives. This payload MUST NOT be indicated to the upper layer as data.

All data frames, KeepAlive or otherwise, MUST have their **bNRcv**, **bSeq**, optional SACK mask, and optional send mask fields processed as described in sections 3.1.5.2.1 through 3.1.5.2.4. After the sequencing information is validated, and processing indicates that the data is either in sequence or was not marked as PACKET_COMMAND_SEQUENTIAL in **bCommand**, the implementation SHOULD report the data payload or coalesced data payloads, if any, to its consumer. The exception is version 0x00010005 and higher KeepAlives, which SHOULD be treated as if they have a 0-byte payload as previously noted.

All successfully validated DFRAME packets SHOULD count as a valid receive and thus restart the KeepAlive timer (section 3.1.2.6).

3.1.5.2.1 Send Sequence ID (bSeq) Validation and Processing

The **bSeq** field MUST either be the next sequence ID expected by the receiver, or be within 63 packets beyond the expected ID. If the sequence ID is outside this range, a SACK packet SHOULD be sent that indicates the current expected state. If the PACKET_COMMAND_POLL flag is set in **bCommand**, this packet SHOULD be sent immediately. Otherwise, the dedicated ACK timer

SHOULD be set using the short time out (nominally 20 msec) and the SACK packet SHOULD be sent at that time. The payload of the received packet MUST then be ignored.

After all other validation is performed and the packet has not been rejected, this **bSeq** value MUST become the new expected sequence ID. Receiving a retried packet with the same **bSeq** value is treated as a duplicate, using the logic in the previous paragraph.

Sequence IDs start at zero for every connection. Therefore each partner's Next Send value from which **bSeq** is generated MUST start at zero for every connection.

3.1.5.2.2 Acknowledged Sequence ID (bNRcv) Processing

The **bNRcv** field acknowledges reception of previously sent frames that are less than the specified ID. All DFRAME packets that had been sent with **bSeq** values less than **bNRcv**, accounting for 8-bit counter wrapping, no longer need to be remembered, and their retry timers SHOULD be canceled.

Sequence IDs start at zero for every connection. Therefore, each partner's Next Receive value from which **bNRcv** is generated MUST start at zero for every connection.

3.1.5.2.3 SACK Mask Processing

When one or both of the optional SACK mask 32-bit fields is present and one or more bits are set, the sender is indicating that it received a packet or packets with sequence IDs higher than **bNRcv** out of order, presumably due to packet loss. The two 32-bit little-endian fields MUST be considered as one 64-bit field, where **dwSACKMask1** is the low 32 bits, and **dwSACKMask2** is the high 32 bits. If one or the other field is not present, its contents MUST be considered as all zero.

The receiver of a SACK mask SHOULD loop through each bit of the combined 64-bit value, from the least significant to most significant. Each bit corresponds to a sequence ID after **bNRcv**, and if that bit is set, it indicates that the corresponding packet was received out of order. Implementations SHOULD avoid retrying those packets in the future.

Implementations SHOULD also shorten the retry timer for the first frame of the window to 10 msec to speed recovery from the packet loss indicated by a SACK mask.

3.1.5.2.4 Send Mask Processing

When one or both of the optional send mask 32-bit fields is present, and one or more bits are set, the sender is indicating that it sent an unreliable packet or packets for which it did not receive acknowledgements yet. The two 32-bit little-endian fields MUST be considered as one 64-bit field, where **dwSendMask1** is the low 32 bits, and **dwSendMask2** is the high 32 bits. If one or the other field is not present, its contents MUST be considered as all zero.

The receiver of a send mask SHOULD loop through each bit of the combined 64-bit value, from the least significant to most significant. Each bit corresponds to a sequence ID prior to **bSeq**, and if that bit that is set, it indicates that the corresponding packet had been sent unreliably, and will not be retried. If the recipient of the send mask had not received the packet and had not already processed a send mask that identifies the sequence ID, it SHOULD consider the packet as dropped and release its placeholder in the sequence. That is, any sequential messages that could not be indicated because of the gap in the sequence where the unreliable packet had been SHOULD now be reported to the upper layer.

3.1.5.2.5 Coalesced Payload Processing

When a data frame arrives with the PACKET_CONTROL_COALESCE flag set in **bControl**, the data payload is made of one or more coalesced payloads instead of a single payload. These sub-payloads

begin with an array of 16-bit headers described in section 2.2.3, followed by an optional 16 bits of padding to ensure 32-bit alignment. Following this array are each of the actual payloads, all with 32-bit alignment padding except for the final payload. The recipient MUST ensure that the number of sub-payloads indicated in the array is valid and does not report sizes larger than that of the remainder of the packet. It SHOULD then report each individual payload to the upper layer as if it had arrived in its own packet. These MUST be reported in the same order in which they were placed in the packet to preserve sequencing.

3.1.5.2.6 Large (Multi-Packet) Payload Processing

When a data frame arrives with the PACKET_COMMAND_NEW_MSG flag set but not the PACKET_COMMAND_END_MSG flag set in **bCommand**, the complete data payload spans more than one packet. The PACKET_CONTROL_COALESCE flag MUST NOT also be set. The receiver SHOULD save the received payload without indicating it to the upper layer, and prepare to collect future packets.

Future data frames that arrive without the PACKET_COMMAND_END_MSG flag MUST NOT also have PACKET_CONTROL_COALESCE enabled. Their payloads SHOULD be added to the buffer or list.

When the next data frame in sequence that contains the PACKET_COMMAND_END_MSG flag arrives without any gaps in the sequence space, the implementation SHOULD indicate the payloads to the upper layer as a contiguous buffer in the same order in which they arrived.

If a data frame arrives in sequence after one with the PACKET_COMMAND_END_MSG flag set but it does not have the PACKET_COMMAND_NEW_MSG set, the receiver SHOULD behave as if the PACKET_COMMAND_NEW_MSG flag was set, but MAY terminate the connection. Similarly, if a data frame arrives in sequence with the PACKET_COMMAND_NEW_MSG flag set but the previous one did not have the PACKET_COMMAND_END_MSG flag set, the receiver SHOULD behave as if the previous packet had the PACKET_COMMAND_END_MSG flag set, but MAY terminate the connection.

An implementation MAY provide an upper limit of how many packets MAY be used to send a large message. If it receives the specified number of packets without any missing sequence numbers, and it still has not encountered a PACKET_COMMAND_END_MSG flag, it MAY terminate the connection. The limit SHOULD be appropriate for and is specific to the application's desired resource consumption and sending patterns; no particular value is recommended in this document.

3.1.5.2.7 Signature Processing

All frames that are associated with a signed connection MUST have an **ullSignature** field present and it MUST be validated. The method of validation depends on the signing mode that was agreed upon when the connection was established.

For fast-signed connections, the 64-bit signature MUST match exactly the secret that was associated with the remote partner when the connection was set up; otherwise, the packet MUST be discarded.

For full-signed connections, the 64- bit signature MUST match exactly the first 64 bits of the expected SHA-1 signature digest or else the packet MUST be discarded. The digest MUST be calculated from the following data, in sequence.

1. The entire received packet, extending from the beginning of the CFRAME or DFRAME header and concluding with the final byte of the final mask, payload, or coalesced payload as appropriate, except with the CFRAME or DFRAME **ullSignature** bytes zeroed.

2. The 64-bit current or previous remote secret, in little-endian byte order. The remote secret to use when validating MUST be selected according to the following logic.

1. If the next expected receive sequence ID is greater than or equal to 192, use the previous remote secret.

2. If the next expected receive sequence ID is less than 64 and the received **bSeq** value is greater than or equal to 192, use the previous remote secret.

3. Otherwise, use the current remote secret.

For full-signed connections, remote secrets are also modified once each time the 8-bit sequence space wraps to avoid signing all data with the same value. The modification is performed using a modifier value derived from the lowest sequenced reliable payload received with a sequence ID of less than 192 that is not a KeepAlive. If the lowest sequenced packet using the PACKET_COMMAND_RELIABLE flag contains coalesced payloads, the first sub-payload that is marked PACKET_COMMAND_RELIABLE is used to generate the modifier. If no non-KeepAlive reliable payload is received with a sequence ID between 0 and 191 inclusive, the previous remote secret modifier value is reused. The remote secret modifier value is initialized to the secret associated with the sender when the connection was established; that is, it begins with the same value as the CONNECTED_SIGNED frame's **ullSenderSecret** field if the local system received an inbound connection, and it begins with the same value as the CONNECTED_SIGNED frame's **ullReceiverSecret** field if the local system performed an outbound connection.

Upon receiving the packet with sequence ID 192 on a full-signed connection, and there are no missing sequence IDs, the receiver MUST advance the secret by making the current remote secret become the previous remote secret, and then setting the new current remote secret to the first 64 bits of a SHA-1 digest of the following data, in sequence.

1. The previous 64-bit remote secret, in little-endian byte order.

2. The 64-bit remote secret modifier value, in little-endian byte order.

3.1.6 Timer Events

3.1.6.1 Connect Retry Timer

When the connect retry timer expires, a new CONNECT or CONNECTED message SHOULD be sent, depending on the current state. The connect retry timer SHOULD then be rescheduled for the next period. It is RECOMMENDED that the retry period start at 200 msec, double every time, with a maximum of five seconds and 14 retries.

If the maximum number of retries has already been attempted when the timer expires, the connection attempt MUST be considered as failed. If the connection was initiated from an inbound CONNECT packet arriving on a listening system, the listener MAY choose to go back to listening if it did not allow additional connection attempts while the failed attempt was in progress.

3.1.6.2 Delayed Acknowledgement Timer

When the delayed acknowledgement timer expires without having been cancelled, the system SHOULD send a dedicated SACK message that contains the current connection state information. Any active delayed send mask timer SHOULD then be cancelled.

3.1.6.3 Delayed Send Mask Timer

When the delayed send mask timer expires without having been cancelled, the system SHOULD send a dedicated SACK message that contains the current connection state information. Any active delayed acknowledgement timer SHOULD then be cancelled.

3.1.6.4 Hard Disconnect Timer

When the hard disconnect timer expires without having been cancelled, the system SHOULD send another HARD_DISCONNECT packet. If the maximum number of resends has already occurred, the connection SHOULD be considered terminated. Otherwise, the hard disconnect timer SHOULD be rescheduled.

3.1.6.5 Retry Timer

When the retry timer elapses without having been cancelled, and the associated packet was reliable, the DFRAME SHOULD be resent and the retry timer SHOULD then be scheduled for the next period. It is RECOMMENDED that the retry period start at 2.5 times RTT plus the delayed acknowledgement time out (nominally 100 msec), and that there be linear backoff for the second and third retries, exponential backoff for subsequent retries 4-8, and an overall cap at five seconds and 10 retries.

If the maximum number of retries has already been attempted when the timer expires, the connection MUST be considered as lost. All other in-progress sends MUST be discarded, and the upper layer SHOULD be informed of the disconnection.

When the retry timer elapses without having been cancelled, and the associated packet was unreliable, the packet's sequence ID SHOULD be remembered as requiring a send mask, and a delayed send mask timer SHOULD be scheduled to transmit this information.

For reliable packets that contained coalesced reliable and unreliable sub-payloads, only the reliable sub-payloads SHOULD be retried. All unreliable sub-payloads MUST be removed from the packet.

Retried packets MUST always contain the latest DFRAME header information, except that **bSeq** MUST be the sequence ID originally assigned to the packet, and if the send mask is present, it MUST be relative to that **bSeq** value.

For connections that enabled full-signing, retried packets MUST always be properly re-signed whenever any header information is updated, or unreliable coalesced sub-payloads are removed.

3.1.6.6 KeepAlive Timer

When the KeepAlive timer expires without having been cancelled, the system SHOULD send a KeepAlive message. A KeepAlive message is a reliable DFRAME with no application payload. If both partners indicated a **dwCurrentProtocolVersion** value of 0x00010005 or higher, the DFRAME MUST have the PACKET_CONTROL_KEEPALIVE flag set and it MUST have the 32-bit **dwSessID** field present and set to the connection's session ID. Otherwise, there MUST NOT be any payload.

After the reliable KeepAlive message begins transmitting, it MUST behave like all other reliable DFRAMEs with respect to time outs and maximum retries. The implementation SHOULD reschedule the KeepAlive timer to expire again after another period of inactivity.

3.1.7 Other Local Events

No other local events affect the DirectPlay 8 Protocol.

4 Protocol Examples

The following sections describe several operations as used in common scenarios to illustrate the function of the DirectPlay 8 Protocol.

4.1 Sample Connection Sequence

The following five frames show an example of the multiple-step connection sequence and subsequent KeepAlive messages used when "Connector" initiates a connection to "Listener". The bytes are assumed to be a UDP payload (UDP and other headers not shown here).

1. Connector to Listener

CFRAME - CONNECT, ACK now, message ID 0x00, response ID 0x00, version 0x00010006, session ID 0x79C9AEC6, timestamp 0x2367369D

```
88 01 00 00 06 00 01 00 C6 AE C9 79 9D 36 67 23   ...........y.6g#
```

2. Listener to Connector

CFRAME - CONNECTED, ACK now, message ID 0x00, response ID 0x00, version 0x00010006, session ID 0x79C9AEC6, timestamp 0x0004DFE1

```
88 02 00 00 06 00 01 00 C6 AE C9 79 E1 DF 04 00   ...........y....
```

3. Connector to Listener

CFRAME - CONNECTED, message ID 0x01, response ID 0x00, version 0x00010006, session ID 0x79C9AEC6, timestamp 0x2367369D

```
80 02 01 00 06 00 01 00 C6 AE C9 79 9D 36 67 23   ...........y.6g#
```

4. Connector to Listener

DFRAME, reliable, sequential, complete message, not a retry, not coalesced, not final packet, no SACK or send masks, sequence ID 0, Next Receive 0, acknowledgement (ACK) now, KeepAlive for session ID 0x79C9AEC6

```
3F 02 00 00 C6 AE C9 79                           ?......y
```

5. Listener to Connector

DFRAME, reliable, sequential, complete message, not a retry, not coalesced, not final packet, no SACK or send masks, sequence ID 0, Next Receive 0, ACK now, KeepAlive for session ID 0x79C9AEC6

```
3F 02 00 00 C6 AE C9 79                          ?......y
```

4.2 Sample Upper Layer Data Transmission and Acknowledgement

The following two frames show an example of an upper layer sending a payload from "Partner A" to "Partner B", and receiving an acknowledgement in the reverse direction. The bytes are assumed to be a UDP payload (UDP and other headers not shown here).

1. Partner A to Partner B

DFRAME, unreliable, sequential, ACK now, complete message, not a retry, not coalesced, not final packet, no SACK or send masks, sequence ID 5, Next Receive 3, 5 byte payload "ABCDE"

```
3D 00 05 03 01 41 42 43 44 45         =....ABCDE
```

2. Partner B to Partner A

CFRAME - SACK, Next Send 3, Next Receive 5, not a retry, no SACK or send masks, timestamp 0x00115D07

```
80 06 01 00 03 06 00 00 07 5D 11 00   .........]..
```

5 Security

The following sections specify security considerations for implementers of the DirectPlay 8 Protocol.

5.1 Security Considerations for Implementers

The DirectPlay 8 Protocol optionally uses the SHA-1 hashing algorithm (as specified in [FIPS180]), which has been shown to have weaknesses. However, the protocol is not intended for use in applications that demand robust security without IPSec or other lower-level security mechanisms already in place.

5.2 Index of Security Parameters

Security parameter	Section
SHA-1 digest	3.1.4.4 and 3.1.5.2.7

6 Appendix A: Windows Behavior

The information in this specification is applicable to the following versions of Windows:

- Windows XP
- Windows Server 2003
- Windows Vista

Exceptions, if any, are noted below. Unless otherwise specified, any statement of optional behavior in this specification prescribed using the terms SHOULD or SHOULD NOT implies Windows behavior in accordance with the SHOULD or SHOULD NOT prescription. Unless otherwise specified, the term MAY implies that Windows does not follow the prescription.

<1> Section 2.2.1.1: Windows Server 2003 and Windows XP, without the **DirectX** 9 or later runtime installed, report versions less than 0x00010005, and do not support signing or coalescence.

<2> Section 2.2.1.2: Windows Server 2003 and Windows XP , without the DirectX 9 or later runtime installed, report versions less than 0x00010005, and do not support signing or coalescence.

<3> Section 2.2.1.3: Windows Server 2003 and Windows XP, without the DirectX 9 or later runtime installed, report versions less than 0x00010005, and do not support signing or coalescence.

7 Index