[MS-RDPEA]:
Remote Desktop Protocol:
Audio Output Virtual Channel Extension

Revision Summary

Date	Revision History	Revision Class	Comments
07/20/2007	0.1	Major	MCPP Milestone 5 Initial Availability
09/28/2007	0.2	Minor	Made technical and editorial changes based on feedback.
10/23/2007	0.3	Minor	Made technical and editorial changes based on feedback.

Date	Revision History	Revision Class	Comments
11/30/2007	0.4	Minor	Made technical and editorial changes based on feedback.
01/25/2008	0.4.1	Editorial	Revised and edited the technical content.

Table of Contents

1 Introduction

This document specifies the Remote Desktop Protocol: Audio Output Virtual Channel Extension to the Remote Desktop Protocol. This protocol seamlessly transfers audio data from a **server** to a **client**.

1.1 Glossary

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

Client
Protocol Data Unit (PDU)
RC4
Server
SHA-1 Hash

The following terms are specific to this document:

Audio Format: A data structure used to define waveform-audio data. The actual structure of individual formats is opaque to this protocol. For more information, see [MSDN-AUDIOFORMAT].

Virtual Channel: A static transport used for lossless communication between a **client** and a **server** component over a main data connection, in 1600-byte chunks, as specified in Static Virtual Channels in [MS-RDPBCGR].

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.

[FIPS180-2] Federal Information Processing Standards Publication, "Secure Hash Standard", FIPS PUB 180-2, August 2002, http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf

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

[MS-RDPBCGR] Microsoft Corporation, "Remote Desktop Protocol: Basic Connectivity and Graphics Remoting Specification", June 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

[SCHNEIER] Schneier, B., "Applied Cryptography, Second Edition", John Wiley and Sons, 1996, ISBN: 0471117099.

If you have any trouble finding [SCHNEIER], please check here.

1.2.2 Informative References

[MMRK] Microsoft Corporation, "Multimedia Registration Kit Revision 3.0", http://support.microsoft.com/default.aspx?scid=KB;EN-US;Q120253

[MSDN-AUDIOFORMAT] Microsoft Corporation, "WAVEFORMATEX", http://msdn2.microsoft.com/en-us/library/ms713497.aspx

[MSDN-FOURCC] Microsoft Corporation, "Registered FOURCC Codes and WAVE Formats", http://msdn2.microsoft.com/en-us/library/ms867195.aspx

[MSDN-getsockname] Microsoft Corporation, "getsockname Function", http://msdn2.microsoft.com/en-us/library/ms738543.aspx

1.3 Protocol Overview (Synopsis)

This section provides a high-level overview of the operation of Remote Desktop Protocol: Audio Output Virtual Channel Extension. The purpose of the protocol is to transfer audio data from the server to the client. For example, when the server plays an audio file, this protocol is used by the server to transfer the audio data to the client. The client may then play the audio.

1.3.1 Audio Redirection Protocol Transport Options

Remote Desktop Protocol: Audio Output Virtual Channel Extension information may be exchanged between the client and server via two different transport methods:

- Static Virtual Channels, as specified in [MS-RDPBCGR].
- User Datagram Protocol (UDP).

Static Virtual Channels may be used to transmit all information between client and server and must be used for some sequences. For certain sequences, however, UDP may be used as well. Throughout this document, references are made to sending data over static virtual channels and over UDP.

1.3.2 Audio Redirection Protocol

Remote Desktop Protocol: Audio Output Virtual Channel Extension is divided into three distinct sequences:

- Initialization Sequence (section 1.3.2.1)

  The connection is established and capabilities and settings are exchanged.

- Data Transfer Sequences (section 1.3.2.2)

  Audio data is transferred.

- Audio Setting Transfer Sequences (section 1.3.2.3)

  Changes to audio settings are transferred.

1.3.2.1 Initialization Sequence

The initialization sequence has the following goals:

1. Establish the client and server protocol versions and capabilities.

2. Establish a list of **audio formats** common to both the client and the server. All audio data transmits in a format specified in this list.

3. Determine if UDP may be used to transmit audio data.

Initially, the server sends a Server Audio Formats and Version PDU, specifying its protocol version and supported audio formats to the client. In response, the client sends a Client Audio Formats and Version PDU. At this point, the server and client have each other's versions, each other's capabilities, and a synchronized list of supported audio formats.

If the client wants to accept data over UDP, the client advertises a port to be used for UDP traffic. Given the client's port, the server attempts to use UDP to send a Training PDU to the client over the port. The client in turn attempts to reply with its own Training Confirm PDU. The server then attempts to send a private key (using a Crypt Key PDU) to the client, using the audio static virtual channel. This key will be used to encrypt some data sent over UDP. If all of the preceding steps succeed, then data transfer sequences are sent over UDP. If any of the preceding steps fail, then data transfer sequences are sent over static virtual channels.

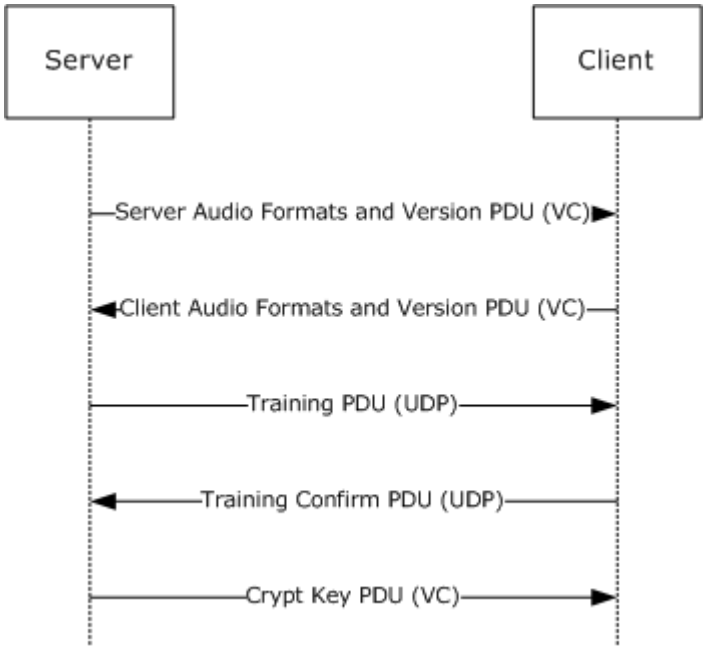


Figure 1: Initialization sequence using UDP for data transfer

If all data transfer sequences are to be sent over static virtual channels, the server and client exchange a Training PDU and a Training Confirm PDU over static virtual channels.

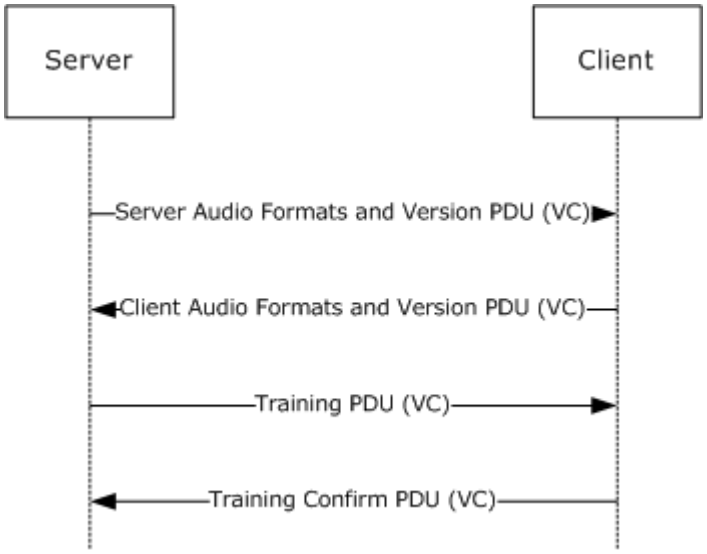


Figure 2: Initialization sequence using static virtual channels for data transfer

1.3.2.2 Data Transfer Sequences

The data transfer sequences have the goal of transferring audio data from the server to the client. Two different protocols exist for the data transfer sequences: one protocol transfers over static virtual channels, and another transfers over UDP.

The data transfer sequence over static virtual channels has a very simple protocol. The server sends two consecutive packets of audio data: a WaveInfo PDU (section 2.2.3.3) and a Wave PDU (section 2.2.3.4). Upon consuming the audio data, the client sends back a Wave Confirm PDU (section 2.2.3.8) to the server to notify the server that it has consumed the audio data.

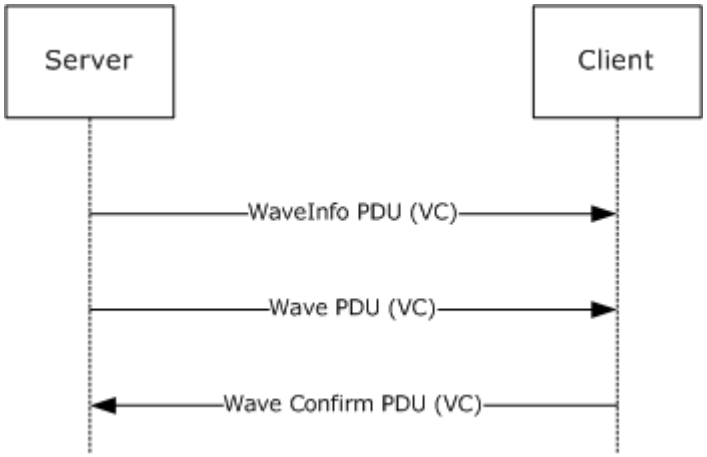


Figure 3: Data transfer sequence over static virtual channels

The protocol for the data transfer sequence over UDP is a little more involved. Similar to the protocol over static virtual channels, the server sends a chunk of audio data to the client. When the client finishes consuming the audio data, the client sends back a Wave Confirm PDU to the server.

The difference with the protocol used over static virtual channels is how the server sends the audio data.

If either the client or server version is less than 5, then the server sends audio data using a Wave Encrypt PDU (section 2.2.3.5). Upon consumption of the audio data, the client sends a Wave Confirm PDU to the server.

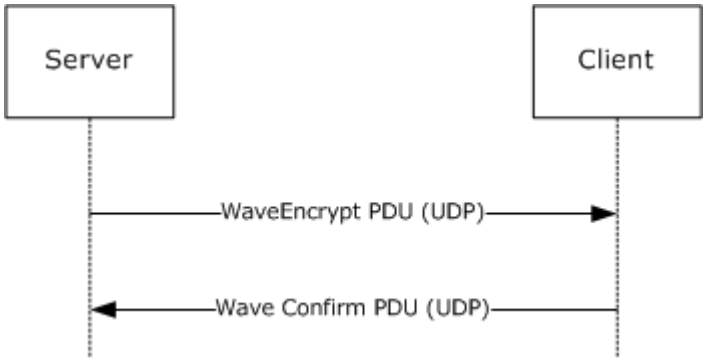


Figure 4: Data transfer sequence over UDP

If the client and server versions are both at least 5, then another method can be used to send audio data over UDP. This method involves the server sending the audio data in successive **PDUs**. All PDUs (except for the final one) are UDP Wave PDUs (section 2.2.3.6). The final PDU is a UDP Wave Last PDU (section 2.2.3.7). Given these PDUs, the client reconstructs the audio data sample. Upon consumption of audio data, the client sends a Wave Confirm PDU to the server.

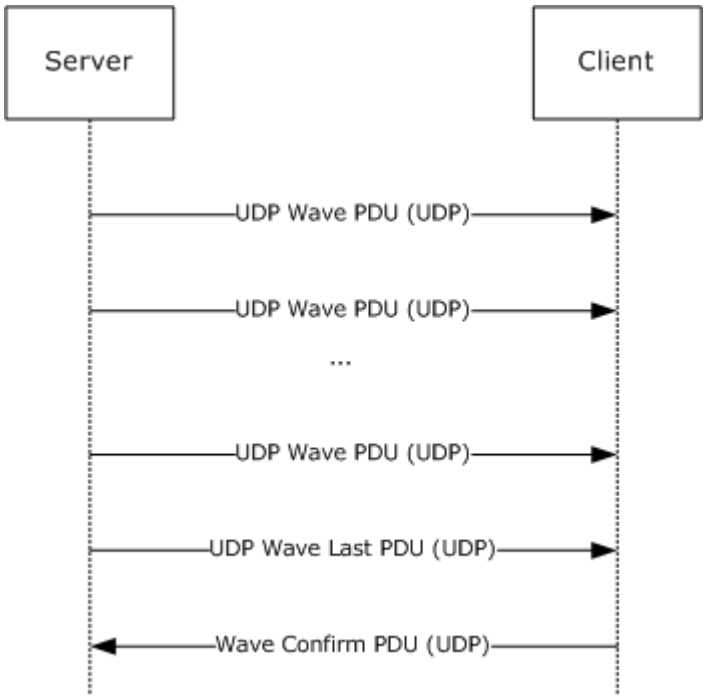


Figure 5: Data transfer sequence over UDP when protocol version is at least 5

During the initialization sequence (section 1.3.2.1), the server uses the Crypt Key PDU (section 2.2.2.3) to send a 32-byte private key over a static virtual channel to the client . Some audio data is encrypted using this key.

At the end of the audio data transfer, the server notifies the client by sending a Close PDU (section 2.2.3.9) over a static virtual channel.

1.3.2.3 Audio Setting Transfer Sequences

The audio setting transfer sequence has the goal of transferring audio setting changes from the server to the client. Two different settings may be redirected: Volume and Pitch. All audio setting transfer sequences are sent over static virtual channels. The settings are redirected using the Volume PDU (section 2.2.4.1) and Pitch PDU (section 2.2.4.2), respectively.

1.4 Relationship to Other Protocols

The Remote Desktop Protocol: Audio Output Virtual Channel Extension is embedded in a static virtual channel transport, as specified in [MS-RDPBCGR].

1.5 Prerequisites/Preconditions

The Remote Desktop Protocol: Audio Output Virtual Channel Extension operates only after the static virtual channel transport (as specified in [MS-RDPBCGR]) is fully established. If the static virtual channel transport is terminated, no other communication occurs over the Remote Desktop Protocol: Audio Output Virtual Channel Extension.

1.6 Applicability Statement

The Remote Desktop Protocol: Audio Output Virtual Channel Extension is designed to be run within the context of a Remote Desktop Protocol **virtual channel** established between a client and server. This protocol is applicable when the client is required to play audio that is playing on the server.

1.7 Versioning and Capability Negotiation

The Remote Desktop Protocol: Audio Output Virtual Channel Extension is capability-based. The client and the server exchange capabilities during the protocol Initialization Sequence (as specified in section 1.3.2.1).

After the capabilities have been received and stored, the client and the server do not send PDUs or data formats that cannot be processed by the other.

1.8 Vendor-Extensible Fields

This protocol contains no vendor-extensible fields.

1.9 Standards Assignments

This protocol contains no standards assignments.

2 Messages

The following sections specify how Remote Desktop Protocol: Audio Output Virtual Channel Extension messages are transported and Remote Desktop Protocol: Audio Output Virtual Channel Extension message syntax.

2.1 Transport

This protocol is designed to operate over two transports:

1. A static virtual channel, as specified in [MS-RDPBCGR]. The virtual channel name is "RDPSND". The Remote Desktop Protocol layer manages the creation, setup, and transmission of data over the virtual channel.

2. UDP, where the port is advertised in the Client Audio Formats and Version PDU (section 2.2.2.2).

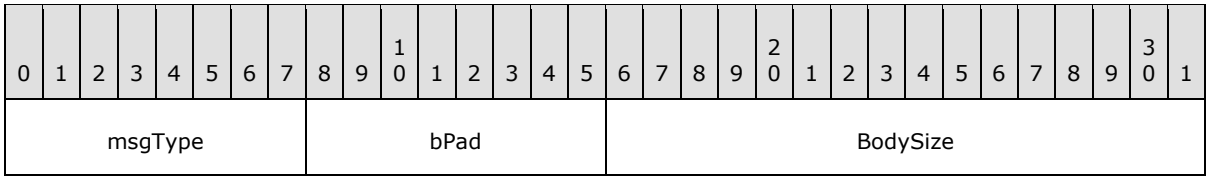
The Initialization Sequence (section 1.3.2.1) and Audio Setting Transfer Sequence (section 1.3.2.3) MUST operate over static virtual channels. The Data Transfer Sequence (section 1.3.2.2) can operate over either UDP or static virtual channels. The following sections of this document specify when to send Data Transfer Sequence messages over UDP and when to send them over static virtual channels.

2.2 Message Syntax

The following sections contain Remote Desktop Protocol: Audio Output Virtual Channel Extension message syntax.

2.2.1 RDPSND PDU Header (SNDPROLOG)

The RDPSND PDU header is present in many audio protocol data units (PDUs). It is used to identify the PDU type, specify the length of the PDU, and convey message flags.



msgType (1 byte): An 8-bit unsigned integer that specifies the type of audio PDU that follows the **BodySize** field.

Value	Meaning
SNDC_CLOSE 0x01	Close PDU
SNDC_WAVE 0x02	WaveInfo PDU
SNDC_SETVOLUME 0x03	Volume PDU
SNDC_SETPITCH 0x04	Pitch PDU

Value	Meaning
SNDC_WAVECONFIRM 0x05	Wave Confirm PDU
SNDC_TRAINING 0x06	Training PDU or Training Confirm PDU
SNDC_FORMATS 0x07	Server Audio Formats and Version PDU or Client Audio Formats and Version PDU
SNDC_CRYPTKEY 0x08	Crypt Key PDU
SNDC_WAVEENCRYPT 0x09	Wave Encrypt PDU

bPad (1 byte): An 8-bit unsigned integer. Unused. The value in this field is arbitrary and MUST be ignored upon receipt.

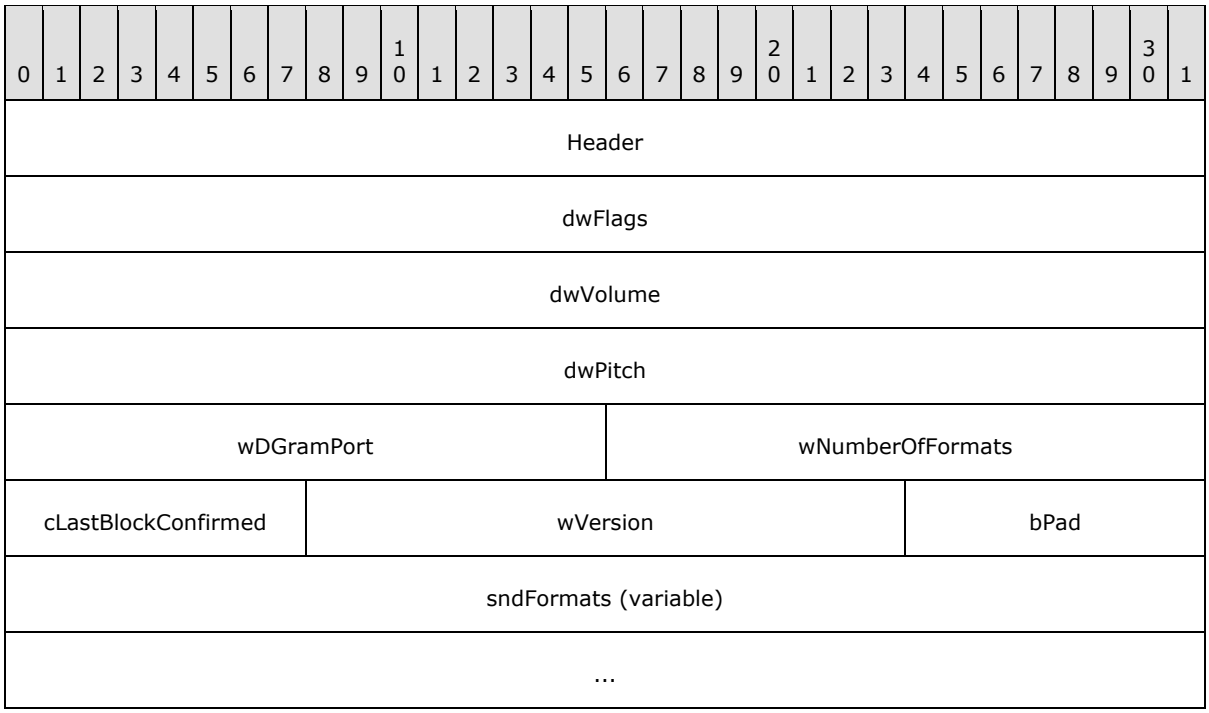
BodySize (2 bytes): A 16-bit unsigned integer. If **msgType** is not set to 0x02 (SNDC_WAVE), then this field specifies the size, in bytes, of the data which follows the RDPSND PDU header. If **msgType** is set to 0x02 (SNDC_WAVE), then the representation of **BodySize** is explained in the **Header** field in section 2.2.3.3.

2.2.2 Initialization Sequence

The following sections contain Remote Desktop Protocol: Audio Output Virtual Channel Extension message syntax for the initialization sequence, as specified in section 1.3.2.1.

2.2.2.1 Server Audio Formats and Version PDU (SERVER_AUDIO_VERSION_AND_FORMATS)

The Server Audio Formats and Version PDU is a protocol data unit (PDU) used by the server to send version information and a list of supported audio formats to the client. This PDU MUST be sent using static virtual channels.



Header (4 bytes): A RDPSND PDU Header (section 2.2.1). The **msgType** field of the RDPSND PDU Header MUST be set to SNDC_FORMATS (0x07).

dwFlags (4 bytes): A 32-bit unsigned integer. Unused. The value in this field is arbitrary and MUST be ignored upon receipt.

dwVolume (4 bytes): A 32-bit unsigned integer. Unused. The value in this field is arbitrary and MUST be ignored upon receipt.

dwPitch (4 bytes): A 32-bit unsigned integer. Unused. The value in this field is arbitrary and MUST be ignored upon receipt.

wDGramPort (2 bytes): A 16-bit unsigned integer. Unused. The value in this field is arbitrary and MUST be ignored upon receipt.

wNumberOfFormats (2 bytes): A 16-bit unsigned integer. Number of **AUDIO_FORMAT** structures contained in the **sndFormats** array.

cLastBlockConfirmed (1 byte): An unsigned 8-bit integer specifying the ID of the last block of audio data that was confirmed by the client. If the client has not previously confirmed any audio data, then the value sent by the server is arbitrary.

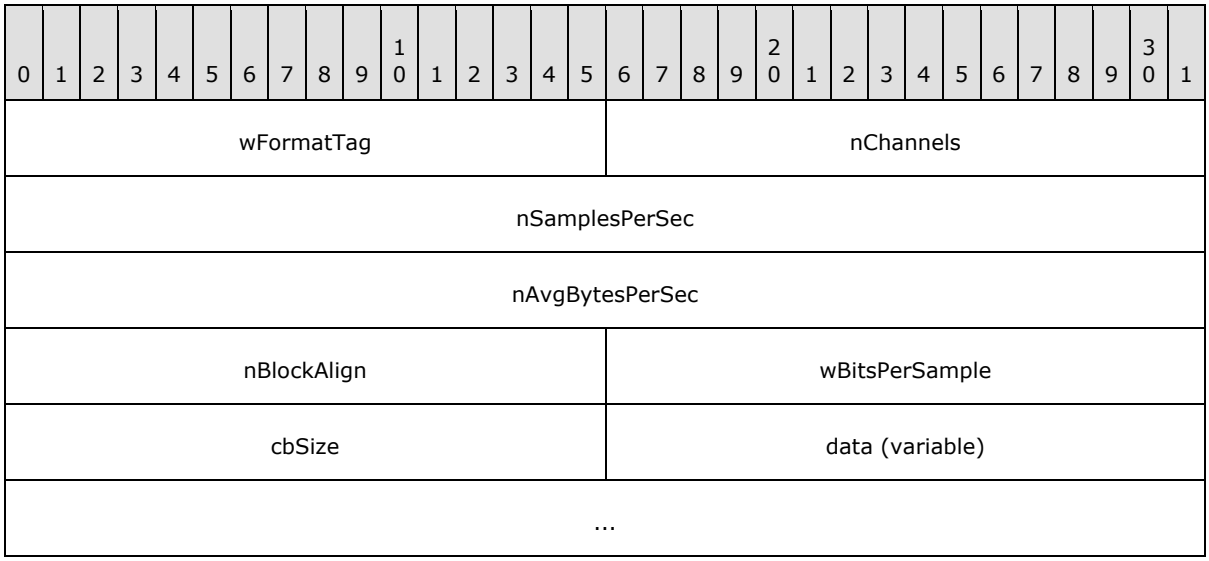
wVersion (2 bytes): A 16-bit unsigned integer that contains the version of the protocol supported by the server.<1>

bPad (1 byte): An 8-bit unsigned integer. Unused. The value in this field is arbitrary and MUST be ignored upon receipt.

sndFormats (variable): A variable-sized array of audio formats supported by the server, each conforming in structure to the **AUDIO_FORMAT** structure. The number of formats in the array is **wNumberOfFormats**.

2.2.2.1.1 Audio Format (AUDIO_FORMAT)

The **AUDIO_FORMAT** structure is used to describe a supported audio format.



wFormatTag (2 bytes): An unsigned 16-bit integer identifying the compression format of the audio format. See [MSDN-FOURCC] for more information on compression formats.

nChannels (2 bytes): An unsigned 16-bit integer that specifies the number of channels of the audio format.

nSamplesPerSec (4 bytes): An unsigned 32-bit integer that specifies the number of audio samples per second in the audio format.

nAvgBytesPerSec (4 bytes): An unsigned 32-bit integer that specifies the average number of bytes the audio format uses to encode one second of audio data.

nBlockAlign (2 bytes): An unsigned 16-bit integer that specifies the minimum atomic unit of data needed to process audio of this format. See [MSDN-AUDIOFORMAT] for more information on block alignment semantics.

wBitsPerSample (2 bytes): An unsigned 16-bit integer that specifies the number of bits needed to represent a sample.

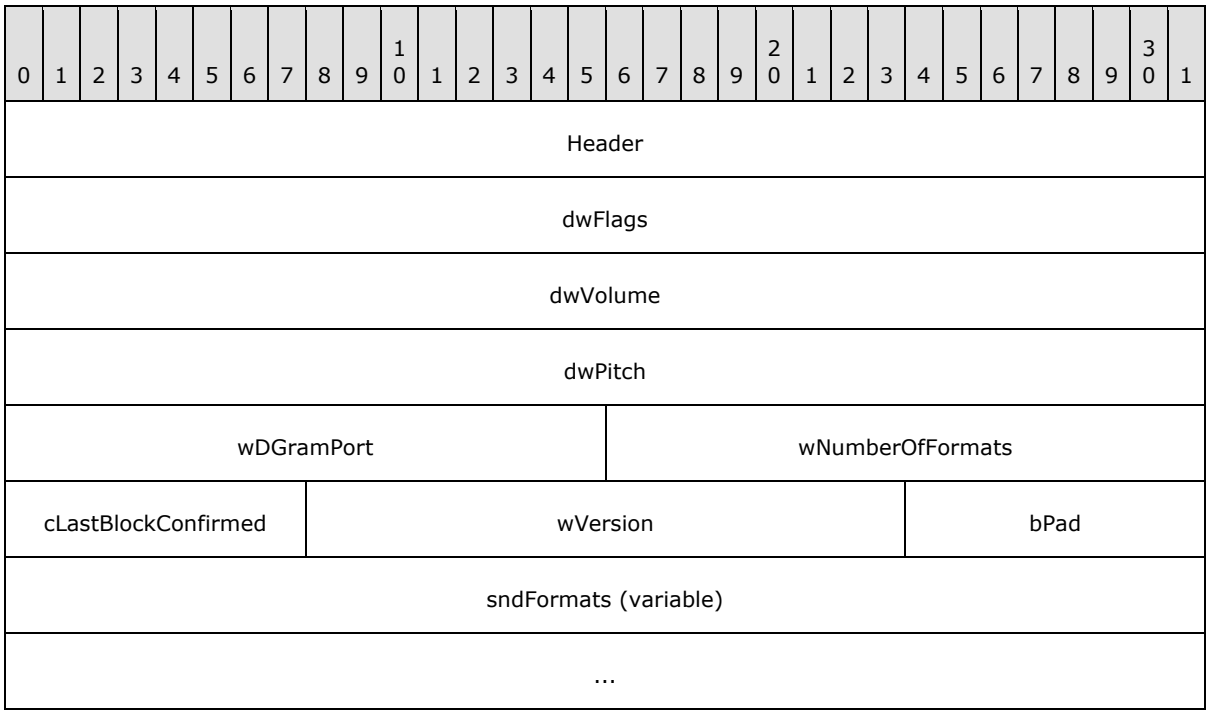
cbSize (2 bytes): An unsigned 16-bit integer specifying the size of the **data** field.

data (variable): Extra data specific to the audio format. See [MMRK] for more information on registering format information. The size of **data**, in bytes, is **cbSize**.

2.2.2.2 Client Audio Formats and Version PDU (CLIENT_AUDIO_VERSION_AND_FORMATS)

The Client Audio Formats and Version PDU is a protocol data unit (PDU) used to send version information, capabilities, and a list of supported audio formats from the client to the server. After the server has sent its version and a list of supported audio formats to the client, the client sends back a Client Audio Formats and Version PDU to the server containing its version and a list of

formats that both the client and server support. This PDU MUST be sent using static virtual channels.



Header (4 bytes): A RDPSND PDU Header (section 2.2.1). The **msgType** field of the RDPSND PDU Header MUST be set to SNDC_FORMATS (0x07).

dwFlags (4 bytes): A 32-bit unsigned integer that specifies the general capability flags. **dwFlags** MUST be a one or more of the following flags combined with a logical OR operator.

Value	Meaning
TSSNDCAPS_ALIVE 0x00000001	The client is capable of consuming audio data. This flag MUST be set for audio data to be transferred.
TSSNDCAPS_VOLUME 0x00000002	The client is capable of applying a volume change to all audio data received.
TSSNDCAPS_PITCH 0x00000004	The client is capable of applying a pitch change to all audio data received.

dwVolume (4 bytes): A 32-bit unsigned integer. If the TSSNDCAPS_VOLUME flag is not set in the **dwFlags** field, then the **dwVolume** field MUST be ignored. If the TSSNDCAPS_VOLUME flag is set in the **dwFlags** field, then the **dwVolume** field specifies the initial volume of the audio stream. The low-order word contains the left-channel volume setting, and the high-order word contains the right-channel setting. A value of 0xFFFF represents full volume, and a value of 0x0000 is silence.

If a device does not support both left and right volume control, the low-order word of **dwVolume** specifies the volume level, and the high-order word is ignored.

This value is to be interpreted logarithmically. This means the perceived increase in volume is the same when increasing the volume level from 0x5000 to 0x6000 as it is from 0x4000 to 0x5000.

dwPitch (4 bytes): A 32-bit unsigned integer. If the TSSNDCAPS_PITCH flag is not set in the **dwFlags** field, then the **dwPitch** field MUST be ignored. If the TSSNDCAPS_PITCH flag is set in the **dwFlags** field, then the **dwPitch** field specifies the initial pitch of the audio stream. The pitch is specified as a fixed-point value. The high-order word contains the signed integer part of the number, and the low-order word contains the fractional part. A value of 0x8000 in the low-order word represents one-half, and 0x4000 represents one-quarter. For example, the value 0x00010000 specifies a multiplier of 1.0 (no pitch change), and a value of 0x000F8000 specifies a multiplier of 15.5.

wDGramPort (2 bytes): A 16-bit unsigned integer that, if set to a non-zero value, specifies the client port that the server MUST use to send data over UDP. A zero value means UDP is not supported.

wNumberOfFormats (2 bytes): A 16-bit unsigned integer specifying the number of AUDIO_FORMAT structures contained in an **sndFormats** array.

cLastBlockConfirmed (1 byte): An 8-bit unsigned integer. Unused. The value in this field is arbitrary and MUST be ignored upon receipt.

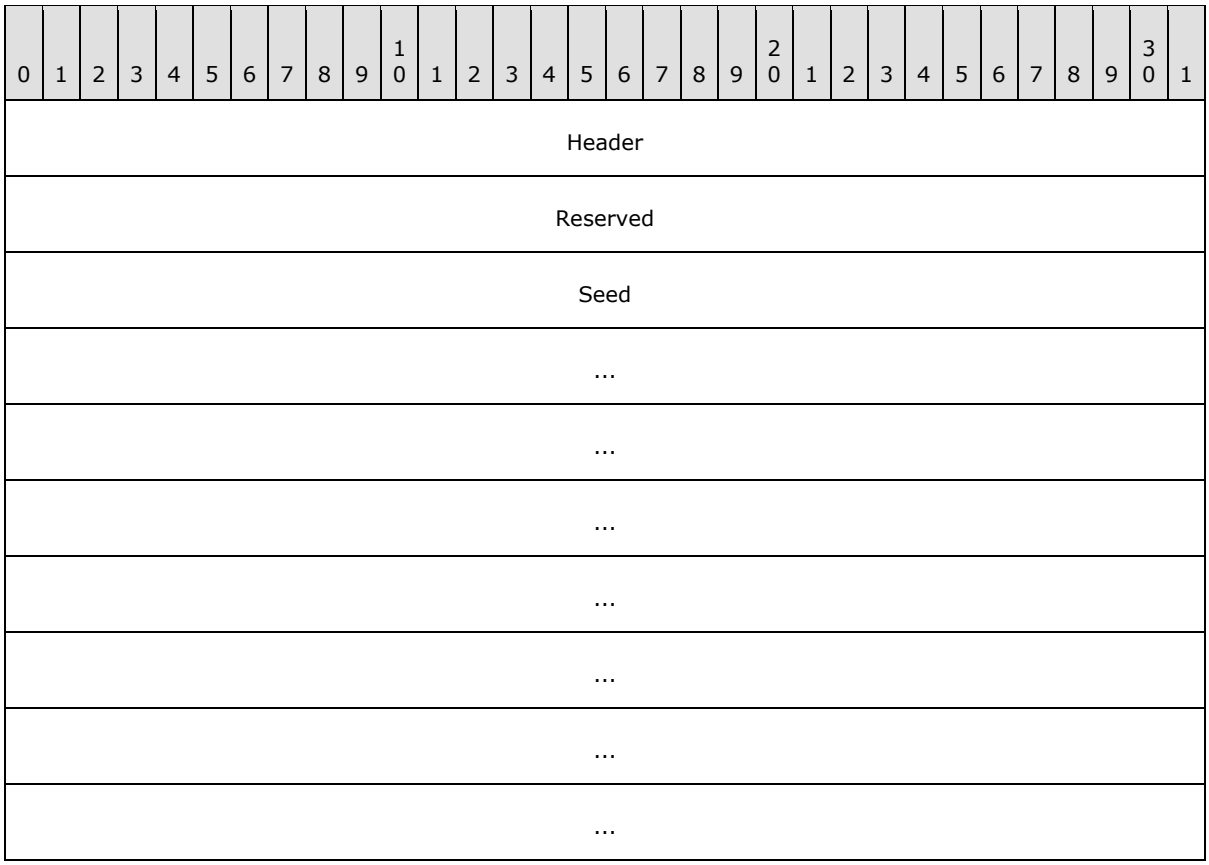
wVersion (2 bytes): A 16-bit unsigned integer specifying the version of the protocol supported by the client.<2>

bPad (1 byte): An 8-bit unsigned integer. Unused. The value in this field is arbitrary and MUST be ignored upon receipt.

sndFormats (variable): A variable sized array of audio formats supported by the client and the server, each conforming in structure to the AUDIO_FORMAT. Each audio format MUST also appear in the Server Audio Formats and Version PDU list of audio formats just sent by the server. The number of formats in the array is **wNumberOfFormats**.

2.2.2.3 Crypt Key PDU (SNDCRYPT)

The Crypt Key PDU is a protocol data unit (PDU) used to send a 32-byte key from the server to the client. The key is used to encrypt some audio data sent over UDP. This PDU MUST be sent using static virtual channels.



Header (4 bytes): A RDPSND PDU Header (section 2.2.1). The **msgType** field of the RDPSND PDU Header MUST be set to SNDC_CRYPTKEY (0x0008).

Reserved (4 bytes): A 32-bit unsigned integer. Unused. The value in this field is arbitrary and MUST be ignored upon receipt.

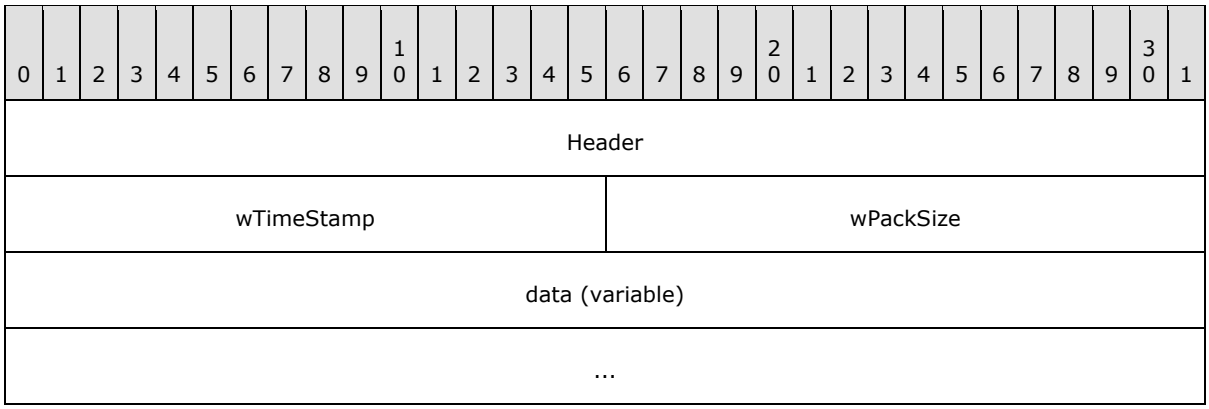
Seed (32 bytes): A 32-byte symmetric key used for encryption and decryption of audio data sent over UDP. When sending a Wave Encrypt PDU, the key will be used to encrypt the audio data. When sending a UDP Wave PDU and a UDP Wave Last PDU there is no encrypted audio data and the key will be used instead to generate a signature.

2.2.3 Data Sequence

The following sections contain Remote Desktop Protocol: Audio Output Virtual Channel Extension message syntax for Data Transfer Sequences, as specified in section 1.3.2.2. To receive audio data from the server, the client MUST have set the flag TSSNDCAPS_ALIVE (0x0000001) in the Client Audio Formats and Version PDU sent during the initialization sequence (section 1.3.2.1)

2.2.3.1 Training PDU (SNDTRAINING)

The Training PDU is a protocol data unit (PDU) used by the server to ping the client. In response, the client MUST immediately send a Training Confirm PDU to the server. The server uses the sending and receiving of these packets for diagnostic purposes. This PDU can be sent using static virtual channels or UDP.



Header (4 bytes): A RDPSND PDU Header (section 2.2.1). The **msgType** field of the RDPSND PDU Header MUST be set to SNDC_TRAINING (0x06).

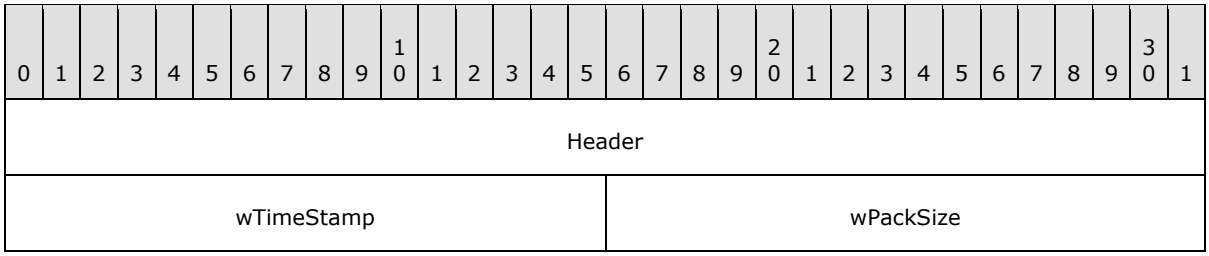
wTimeStamp (2 bytes): A 16-bit unsigned integer. In the Training PDU this value is arbitrary.

wPackSize (2 bytes): A 16-bit unsigned integer. If the size of **data** is non-zero, then this field specifies the size, in bytes, of the entire PDU. If the size of **data** is 0, then **wPackSize** MUST be 0.

data (variable): Unused. The value in this field is arbitrary and MUST be ignored upon receipt.

2.2.3.2 Training Confirm PDU (SNDTRAININGCONFIRM)

The Training Confirm PDU is a protocol data unit (PDU) sent by the client to confirm the reception of a Training PDU. This PDU MAY be sent using static virtual channels or UDP.



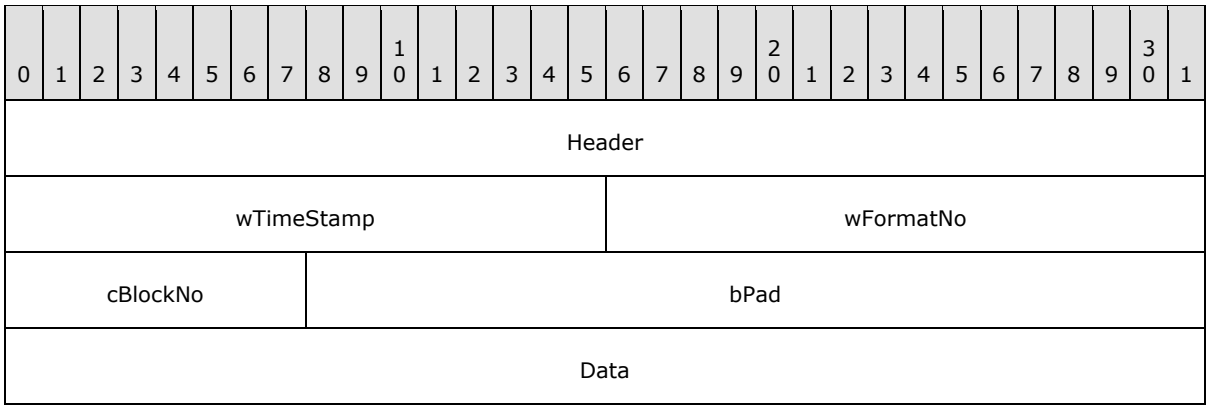
Header (4 bytes): A RDPSND PDU Header (section 2.2.1). The **msgType** field of the RDPSND PDU Header MUST be set to SNDC_TRAINING (0x06).

wTimeStamp (2 bytes): A 16-bit unsigned integer. This value MUST be set to the same value as the **wTimeStamp** field in the Training PDU received from the server.

wPackSize (2 bytes): A 16-bit unsigned integer. This value MUST be set to the same value as the **wPackSize** field in the Training PDU received from the server.

2.2.3.3 WaveInfo PDU (SNDWAVINFO)

The WaveInfo PDU is the first of two consecutive protocol data units (PDUs) used to transmit audio data over static virtual channels. This packet contains information about the audio data and a small portion of the audio data itself. This PDU MUST be sent using static virtual channels.



Header (4 bytes): A RDPSND PDU Header (section 2.2.1). The **msgType** field of the RDPSND PDU Header MUST be set to SNDC_WAVE (0x02). The **BodySize** field of the RDPSND PDU Header is the size of the WaveInfo PDU plus the size of the **data** field of the Wave PDU that immediately follows this packet minus the size of the Header.

wTimeStamp (2 bytes): A 16-bit unsigned integer representing the timestamp of the audio data.

wFormatNo (2 bytes): A 16-bit unsigned integer that represents an index into the list of audio formats exchanged between the client and server during the initialization phase, as described in section 3.1.1.1. The format located at that index is the format of the audio data in this PDU and the Wave PDU that immediately follows this packet.

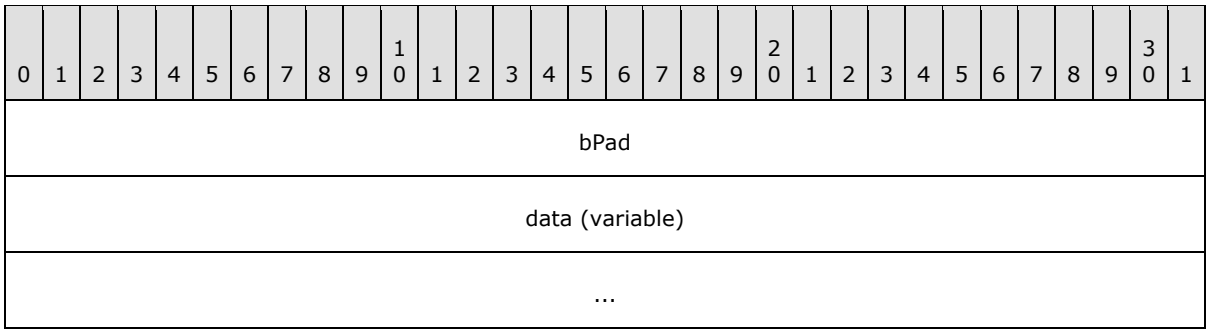
cBlockNo (1 byte): An 8-bit unsigned integer specifying the block ID of the audio data. When the client notifies the server that it has consumed the audio data, it sends a Wave Confirm PDU (section 2.2.3.8) containing this field in its **cConfirmedBlockNo** field.

bPad (3 bytes): A 24-bit unsigned integer. Unused. The value in this field is arbitrary and MUST be ignored upon receipt.

Data (4 bytes): The first four bytes of audio data. The rest of the audio data arrives in the next PDU, which MUST be a Wave PDU. The audio data MUST be in the audio format from the list of formats exchanged during the Initialization Sequence (section 1.3.2.1) found at the index specified in the **wFormatNo** field.

2.2.3.4 Wave PDU (SNDWAV)

The Wave PDU is the second of two consecutive protocol data units (PDUs) used to transmit audio data over static virtual channels. This packet contains the rest of the audio data not sent in the WaveInfo PDU. This PDU MUST be sent using static virtual channels.

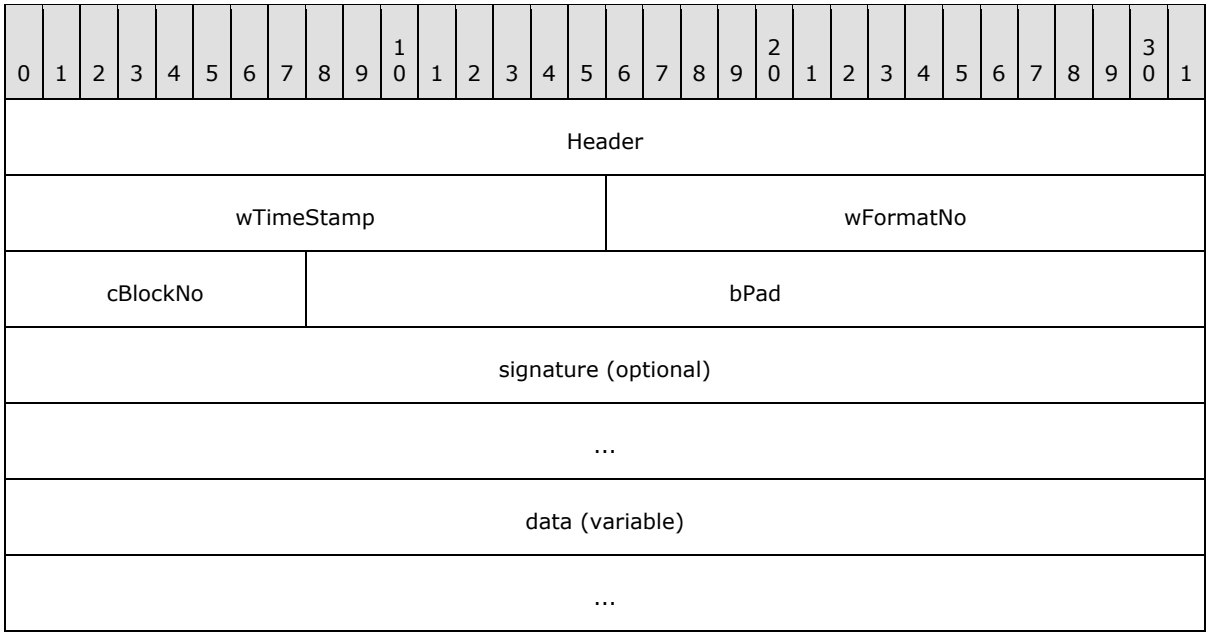


bPad (4 bytes): A 32-bit unsigned integer that MUST be set to 0x00000000.

data (variable): The rest of the audio data. The size of the audio data MUST be equal to the **BodySize** field of the RDPSND PDU header of the WaveInfo PDU that immediately preceded this packet, minus the size of the preceding WaveInfo PDU packet (not including the size of its Header field). The format of the audio data MUST be the format specified in the list of formats exchanged during the Initialization Sequence and found at the index specified in the **wFormatNo** field of the preceding WaveInfo PDU.

2.2.3.5 Wave Encrypt PDU (SNDWAVCRYPT)

The Wave Encrypt PDU is a protocol data unit (PDU) used to send audio data from the server to the client. This PDU MUST be sent over UDP.



Header (4 bytes): A RDPSND PDU Header (section 2.2.1). The **msgType** field of the RDPSND PDU header MUST be set to SNDC_WAVEENCRYPT (0x09).

wTimeStamp (2 bytes): A 16-bit unsigned integer representing the timestamp of the audio data.

wFormatNo (2 bytes): A 16-bit unsigned integer that represents an index into the list of formats exchanged between the client and server during the initialization phase, as described in section 3.1.1.1.

cBlockNo (1 byte): An 8-bit unsigned integer specifying the block ID of the audio data. When the client notifies the server that it has consumed the audio data, it sends a Wave Confirm PDU containing this field in its **cConfirmedBlockNo** field.

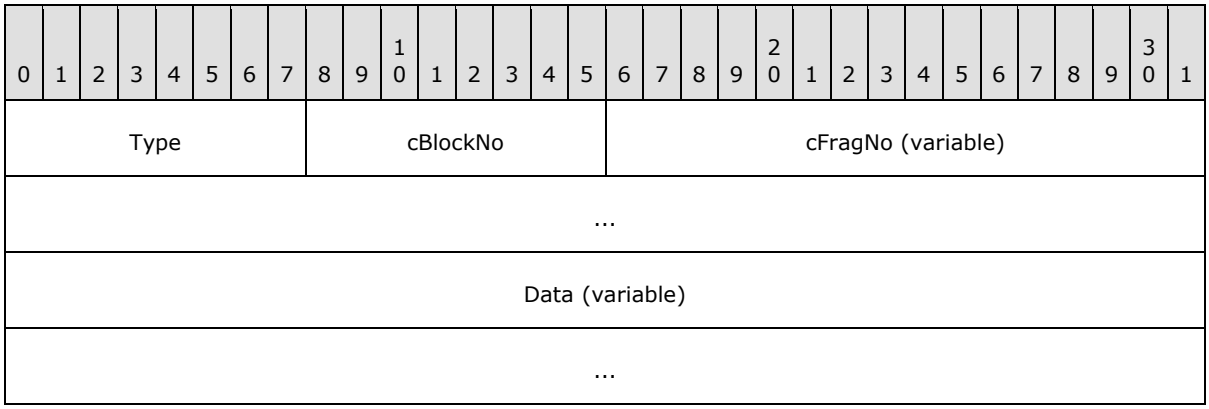
bPad (3 bytes): A 24-bit unsigned integer. Unused. The value in this field is arbitrary and MUST be ignored upon receipt.

signature (8 bytes): An 8-byte digital signature. If the version of either the server or the client is less than 5, then this field MUST NOT exist. If the version of the server and the client are at least 5, then this field MUST exist. An explanation of how this field is created is specified in section 3.3.5.2.1.3.

data (variable): Encrypted audio data. The audio data MUST be in the format specified by the **wFormatNo** and MUST be encrypted. For an explanation of how the data is encrypted, see section 3.3.5.2.1.3.

2.2.3.6 UDP Wave PDU (SNDUDPWAVE)

The UDP Wave PDU is a protocol data unit (PDU) used to send a fragment of audio data from the server to the client. This packet is only used when the client and server versions are both at least 5. This PDU MUST be sent over UDP.



Type (1 byte): An 8-bit unsigned integer. This field MUST be set to SNDC_UDPWAVE (0x0A).

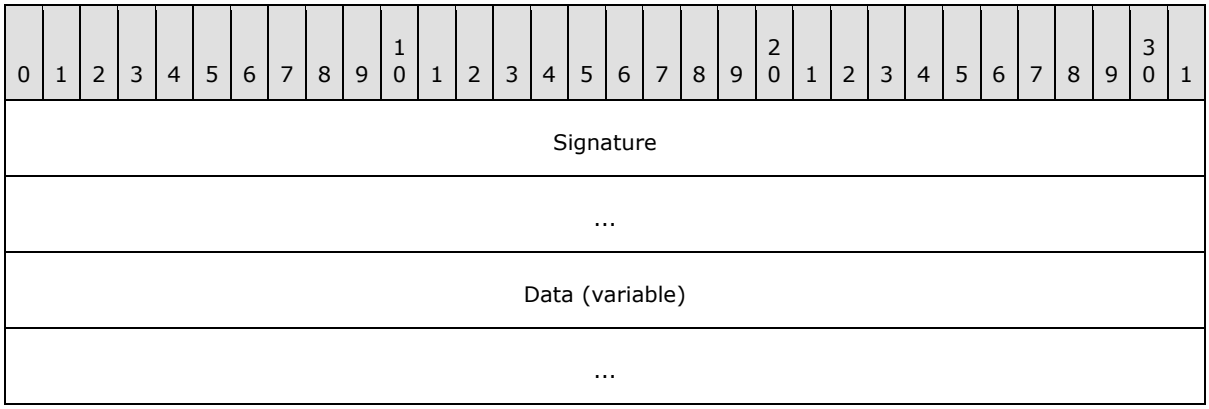
cBlockNo (1 byte): An 8-bit unsigned integer specifying the block ID of the audio data. When the client notifies the server that it has consumed the audio data, it sends a Wave Confirm PDU containing this field in its **cConfirmedBlockNo** field.

cFragNo (variable): An 8-bit or 16-bit unsigned integer specifying the order of the audio data fragment in the overall audio sample. The 0x80 bit of the first byte is used to determine if the field is one or two bytes in length. If the first byte is less than 0x80, then the field is 1 byte. If the first byte is greater than or equal to 0x80, then this field is 2 bytes. To calculate the value of the field, the second byte holds 8 low-order bits, while the first byte holds 7 high-order bits.

Data (variable): A portion of an Audio FragData structure. Several UDP Wave PDUs and a UDP Wave Last PDU contain pieces of a structure conforming to Audio FragData. This algorithm is specified in section 3.2.5.2.1.5.

2.2.3.6.1 Audio FragData (AUDIO_FRAGDATA)

The Audio FragData structure is used to describe the data that is fragmented and sent in several UDP Wave PDUs and a final UDP Wave Last PDU.

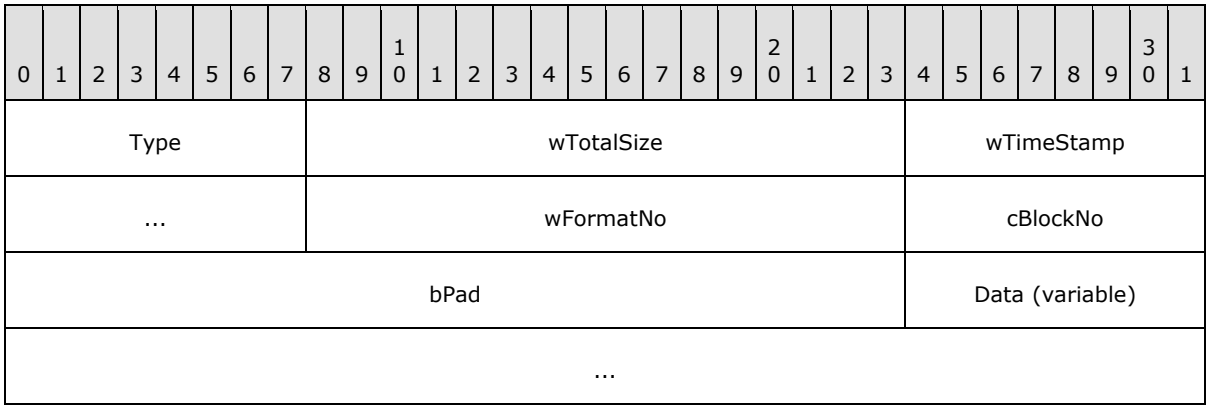


Signature (8 bytes): An 8-byte digital signature. The algorithm for creating this field is the same as creating the **signature** field of a Wave Encrypt PDU as specified in section 3.3.5.2.1.3.

Data (variable): Audio data. The format of the audio data MUST be the format specified in the **wFormatNo** field of the UDP Wave Last PDU that sends the final piece of this structure.

2.2.3.7 UDP Wave Last PDU (SNDUDPWAVELAST)

The UDP Wave Last PDU is a protocol data unit (PDU) used to send the final fragment of audio data from the server to the client . This packet is only used when the client and server versions are both at least 5. This PDU MUST be sent over UDP.



Type (1 byte): An 8-bit unsigned integer. This field MUST be set to SNDC_UDPWAVELAST (0x0B).

wTotalSize (2 bytes): A 16-bit unsigned integer that represents the total size of the audio data sent in successive PDUs. The amount of audio data in previous UDP Wave PDUs plus the amount of audio data in this PDU MUST be equivalent to **wTotalSize**.

wTimeStamp (2 bytes): A 16-bit unsigned integer representing the timestamp of the audio data.

wFormatNo (2 bytes): A 16-bit unsigned integer that represents an index into the list of formats exchanged between the client and server during the initialization phase, as described in section 3.1.1.1.

cBlockNo (1 byte): An 8-bit unsigned integer specifying the block id of the audio data. When the client notifies the server that it has consumed the audio data, it sends a Wave Confirm PDU containing this field in its **cConfirmedBlockNo** field.

bPad (3 bytes): A 24-bit unsigned integer. Unused. The value in this field is arbitrary and MUST be ignored upon receipt.

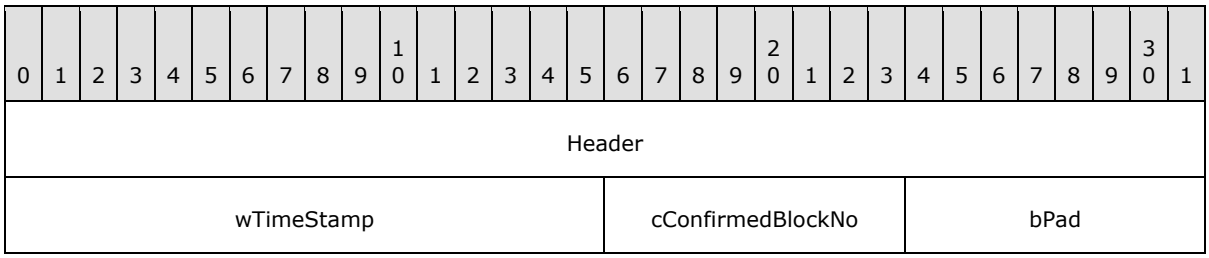
Data (variable): A portion of an Audio FragData. Several UDP Wave PDUs and a UDP Wave Last PDU will contain pieces of a structure conforming to Audio FragData, as specified in section 3.2.5.2.1.5.

2.2.3.8 Wave Confirm PDU (SNDWAV_CONFIRM)

The Wave Confirm PDU is a protocol data unit (PDU) that MUST be sent by the client to the server immediately after the following two events occur:

- An audio data sample is received from the server, whether using a WaveInfo PDU and Wave PDU, a Wave Encrypt PDU, or several UDP Wave PDUs followed by a UDP Wave Last PDU.
- The audio data sample is emitted to completion by the client.

This PDU can be sent using static virtual channels or UDP.



Header (4 bytes): A RDPSND PDU Header (section 2.2.1). The **msgType** field of the RDPSND PDU Header MUST be set to SNDC_WAVECONFIRM (0x05).

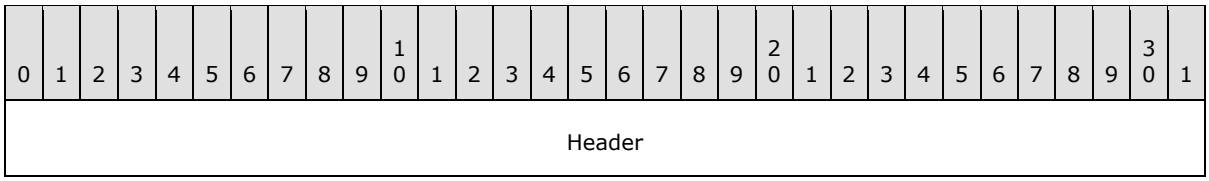
wTimeStamp (2 bytes): A 16-bit unsigned integer. See section 3.2.5.2.1.6 for details of how this field is set.

cConfirmedBlockNo (1 byte): An 8-bit unsigned integer that MUST be the same as the **cBlockNo** field of the UDP Wave Last PDU (section 2.2.3.7), the Wave Encrypt PDU (section 2.2.3.5) or the WaveInfo PDU (section 2.2.3.3) just received from the server.

bPad (1 byte): An unsigned 8-bit integer. Unused. The value in this field is arbitrary and MUST be ignored upon receipt.

2.2.3.9 Close PDU (SNDCLOSE)

The Close PDU is a protocol data unit (PDU) sent by the server to notify the client that audio streaming has stopped. This PDU MUST be sent using static virtual channels.



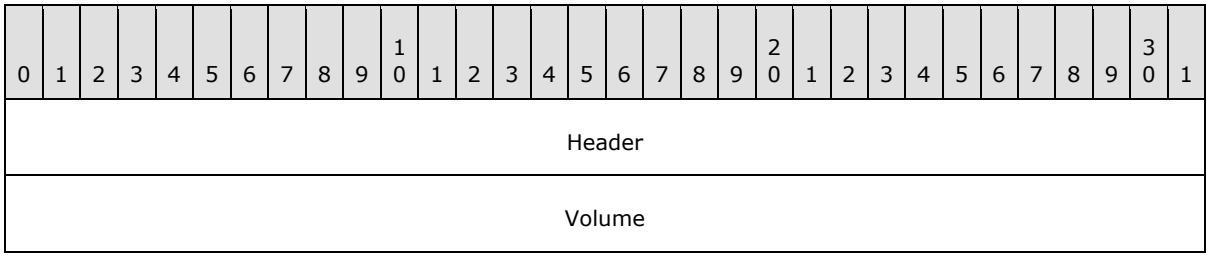
Header (4 bytes): A RDPSND PDU Header (section 2.2.1). The **msgType** field of the RDPSND PDU Header MUST be set to SNDC_CLOSE (0x01).

2.2.4 Audio Setting Transfer Sequences

The following sections contain message syntax for the audio setting transfer sequence, as specified in section 1.3.2.3.

2.2.4.1 Volume PDU (SNDVOL)

The Volume PDU is a protocol data unit (PDU) sent from the server to the client to specify the volume to be set on the audio stream. For this packet to be sent, the client MUST have set the flag TSSNDCAPS_VOLUME (0x0000002) in the Client Audio Formats and Version PDU (section 2.2.2.2) sent during the initialization sequence (section 1.3.2.1).

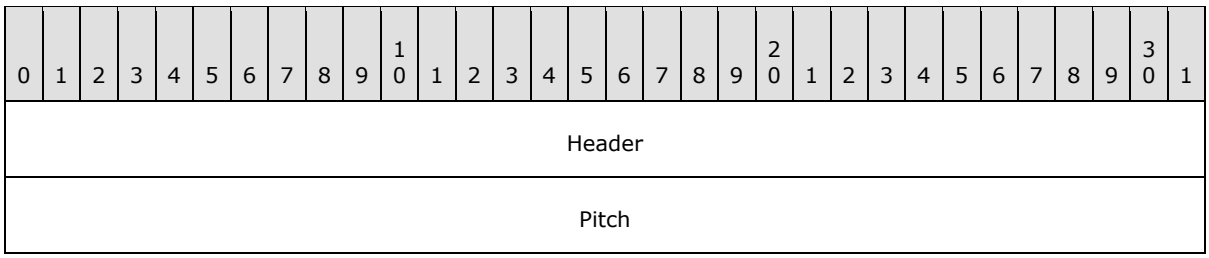


Header (4 bytes): A RDPSND PDU Header (section 2.2.1). The **msgType** field of the RDPSND PDU Header MUST be set to SNDC_VOLUME (0x03).

Volume (4 bytes): A 32-bit unsigned integer specifying the volume to be set on the audio stream. See the **dwVolume** field in section 2.2.2.2 for semantics of the data in this field.

2.2.4.2 Pitch PDU (SNDPITCH)

The Pitch PDU is a protocol data unit (PDU) sent from the server to the client to specify the pitch to be set on the audio stream. For this packet to be sent, the client MUST have set the flag TSSNDCAPS_PITCH (0x0000004) in the Client Audio Formats and Version PDU (section 2.2.2.2) sent during the initialization sequence (section 1.3.2.1).



Header (4 bytes): A RDPSND PDU Header (section 2.2.1). The **msgType** field of the RDPSND PDU Header MUST be set to SNDC_PITCH (0x04).

Pitch (4 bytes): A 32-bit unsigned integer. Although the server may send this PDU, the client MUST ignore it.

3 Protocol Details

The following sections specify protocol details, including abstract data models and message processing rules.

3.1 Common Details

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 an 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.

3.1.1.1 Audio Format List

The **Audio Format List** is the list of formats that the client sends to the server in the Client Audio Formats and Version PDU. This list MUST be maintained throughout the duration of the protocol. The **wFormatNo** field of the Wave Info PDU, the Wave Encrypt PDU, and the UDP Wave Last PDU is an index into this list.

3.1.1.2 Crypt Key

The Crypt Key is a key used by the client and the server for two purposes:

1. To encrypt and decrypt data in a Wave Encrypt PDU.
2. To create the **signature** field for an Audio FragData and Wave Encrypt PDU.

A specification for both purposes is specified in section 3.3.5.2.1.3.

3.1.2 Timers

No timers are used.

3.1.3 Initialization

Before protocol operation can commence, the static virtual channel MUST be established using the parameters specified in section 2.1.

3.1.4 Higher-Layer Triggered Events

3.1.4.1 Playing Audio

When audio is played on the server (for example, the server opens an MP3 file in Windows Media Player), the server MUST start redirecting the audio data. If the initialization sequence (section 1.3.2.1) has not transpired, the server MUST start the initialization sequence and then proceed to start the data transfer sequence (section 1.3.2.2).

3.1.5 Message Processing Events and Sequencing Rules

Malformed, unrecognized, and out-of-sequence packets MUST be ignored by the server and the client.

There are no timeouts for receiving a reply for any request.

3.1.6 Timer Events

No common timer events are used.

3.1.7 Other Local Events

There are no common local events.

3.2 Client Details

3.2.1 Abstract Data Model

The abstract data model is specified in section 3.1.1.

3.2.2 Timers

No timers are used.

3.2.3 Initialization

Initialization is specified in section 3.1.3.

3.2.4 Higher-Layer Triggered Events

No client higher-layer triggered events are used.

3.2.5 Message Processing Events and Sequencing Rules

3.2.5.1 Initialization Sequence

Initialization messages exchange the basic information to establish the connection, and to perform capabilities negotiation. Initialization ensures that the server and client both know which messages are supported. Future versions of the protocol MAY support new messages that current versions do not support. As a result, this negotiation is important to ensure that no messages are sent from one side that the other cannot interpret.

3.2.5.1.1 Messages

3.2.5.1.1.1 Processing a Server Audio Formats and Version PDU

The structure and fields of the Server Audio Formats and Version PDU are specified in section 2.2.2.1.

The Server Audio Formats and Version PDU MUST be the first message received by the client in the protocol sequence. The client uses the version of the server to discover which messages are supported by the protocol.

3.2.5.1.1.2 Sending a Client Audio Formats and Version PDU

The structure and fields of the Client Audio Formats and Version PDU are specified in section 2.2.2.2.

The client MUST acknowledge the Server Audio Formats and Version PDU message by sending its own version and capabilities information, in a Client Audio Formats and Version PDU. The list of formats sent by the client MUST be a subset of the list of formats that was sent by the server in the preceding Server Audio Formats and Version PDU. Formats that do not appear in the server list may not be sent by the client in this message.

The list of formats sent by the client will be referenced in the data transfer sequence. The **wFormatNo** field of the WaveInfo PDU, the UDP Wave Last PDU, and the Wave Encrypt PDU messages all represent an index into this list. A value of **I** refers to the **I**th format of this list and means that the audio data is encoded in the **I**th format of the list.

If the client wants to allow the server to send audio data over UDP (as described in the data transfer sequence) then the client MUST set the **wDGramPort** field to a valid non-zero UDP port on the client machine. However, setting the **wDGramPort** field to a valid non-zero UDP port on the client machine does not guarantee that the server will send audio data over UDP. The server MAY <3> send all audio data over static virtual channels and no data over UDP.

If the client does not want to allow the server to send audio data over UDP, thereby forcing all audio data to be sent over static virtual channels, the client MUST set the **wDGramPort** field to 0.

3.2.5.1.1.3 Processing a Training PDU

The structure and fields of the Training PDU are specified in section 2.2.3.1.

The Training PDU MAY be sent by the server at any time and during any sequence, not just during the initialization sequence. The only prerequisite is that version exchange MUST have occurred.

If the client advertises a UDP port during version exchange, then the Training PDU MAY<4> be sent over UDP or over static virtual channels. Any subsequent audio data SHOULD be sent over the same transport method used to send the Training PDU by the server.

The client MUST respond with a Training Confirm PDU using the same transport on which the Training PDU was received.

3.2.5.1.1.4 Sending a Training Confirm PDU

The structure and fields of the Training Confirm PDU are specified in section 2.2.3.2.

A Training Confirm PDU MUST NOT be sent unless the client has just received a Training PDU from the server. The **wTimeStamp** and **wPackSize** field MUST be set to the same value as the **wTimeStamp** and **wPackSize** field of the Training PDU just received.

3.2.5.1.1.5 Processing a Crypt Key PDU

The structure and fields of the Crypt Key PDU are specified in section 2.2.2.3.

A Crypt Key PDU MUST only be received over static virtual channels.

The following steps MUST have occurred before a Crypt Key PDU can be sent:

1. The client advertised a local UDP port to be used for the transfer of audio data during version exchange.
2. The server successfully sent a Training PDU over UDP to the client.
3. The client successfully replied by sending a Training Confirm PDU over UDP to the server.

This key will be used to help digitally sign pieces of audio data and to help encrypt pieces of audio data.

3.2.5.2 Data Transfer Sequence

The data transfer sequence messages are used to send audio data from the server to the client.

3.2.5.2.1 Messages

3.2.5.2.1.1 Processing a WaveInfo PDU

The structure and fields of the WaveInfo PDU are specified in section 2.2.3.3.

A WaveInfo PDU and a Wave PDU, sent consecutively by the server, combine to form an audio sample. The client reproduces the sample by taking the four bytes of audio data in the **data** field of the WaveInfo PDU, and prepending it to what is in the **data** field of the Wave PDU.

The **wFormatNo** field of the WaveInfo PDU is an index into the list of formats sent by the client in the Client Audio Formats and Version PDU. A value of **i** means the format of the audio data is the **i**th format of that list.

After consuming the data, the client MUST respond by sending a Wave Confirm PDU to the server. The **cConfirmedBlockNo** field of the Wave Confirm PDU MUST be identical to the **cBlockNo** field of the WaveInfo PDU.

If a packet for **cBlockNo n** is lost and an audio sample is constructed for a **cBlockNo** that is greater than **n**, the client abandons all packets associated with **cBlockNo n** and quits processing that sample.

This PDU MUST have been sent by the server over static virtual channels.

3.2.5.2.1.2 Processing a Wave PDU

The structure and fields of the Wave PDU are specified in section 2.2.3.4.

A WaveInfo PDU and a Wave PDU, sent consecutively by the server, combine to form an audio sample. The client reproduces the sample by taking the four bytes of audio data in the **data** field of the WaveInfo PDU, and prepending it to what is in the **data** field of the Wave PDU.

This PDU MUST have been sent by the server over static virtual channels.

3.2.5.2.1.3 Processing a Wave Encrypt PDU

The structure and fields of the Wave Encrypt PDU are specified in section 2.2.3.5.

Unlike a WaveInfo PDU and Wave PDU, the Wave Encrypt PDU contains the entire audio sample in its **data** field.

The **wFormatNo** field of the Wave Encrypt PDU is an index into the list of formats sent by the client in the Client Audio Formats and Version PDU. A value of **i** means the format of the audio data is the **i**th format of that list.

The client MUST decrypt the data before consuming it. How the server encrypts the data is specified in section 3.3.5.2.1.3.

This PDU MUST have been sent by the server over UDP.

3.2.5.2.1.4 Processing a UDP Wave PDU

The structure and fields of the UDP Wave PDU are specified in section 2.2.3.6.

The client will receive several UDP Wave PDUs and one UDP Wave Last PDU, each containing the same value within the **cBlockNo** field. These PDUs contain the fragments of a sample of audio data. Once the UDP Wave Last PDU and all of the associated UDP Wave PDUs are received, the client may reproduce the entire audio data and consume it. The algorithm for reproducing the sample is specified in section 3.2.5.2.1.5.

If an entire sequence of UDP Wave PDUs and the UDP Wave Last PDU get consumed by the client, the client will disregard any pending UDP Wave PDUs from previous blocks.

This PDU MUST have been sent over UDP and only if the client's version and the server's version are both at least 5.

3.2.5.2.1.5 Processing a UDP Wave Last PDU

The structure and fields of the UDP Wave Last PDU are specified in section 2.2.3.7.

The client receives several UDP Wave PDUs and one UDP Wave Last PDU each containing the same value within the **cBlockNo** field. These PDUs contain the fragments of an Audio FragData structure in the **Data** field.

The client MUST consume the original audio data sample. The sample is recreated as follows:

The UDP Wave Last PDU holds the final fragment of audio data. As a result, its **data** field contains data that belongs at the end of the recreated audio sample.

The **cFragNo** field determines the order of the fragments in the UDP Wave PDUs. The contents of the **Data** field in each of the UDP Wave PDUs must be concatenated in the order determined by the **cFragNo** field. The UDP Wave PDU whose **cFragNo** field is 0 represents the start of the audio data, followed by the PDU whose **cFragNo** is 1, and so on. The **Data** field of the UDP Wave Last PDU holds the audio data that is concatenated as the end of the sample. Concatenating all of these **Data** fields yields an AUDIO_FRAGDATA structure that reproduces the original sample.

The **wFormatNo** field is an index into the list of formats sent by the client in the Client Audio Formats and Version PDU. A value of **i** means the format of the audio data is the **i**th format of that list.

This PDU MUST have been sent over UDP and only if the client's version and the server's version are both at least 5.

3.2.5.2.1.6 Sending a Wave Confirm PDU

The structure and fields of the Wave Confirm PDU are specified in section 2.2.3.8.

The client MUST send a Wave Confirm PDU in response to any audio sample sent by the server. The client MUST send the PDU over the same channel used to receive the audio sample. That is, if the client received a WaveInfo PDU and Wave PDU, then the client MUST send the Wave Confirm PDU over static virtual channels. If the client received a Wave Encrypt PDU, or several UDP Wave PDUs and a UDP Wave Last PDU, then the client MUST send the Wave Confirm PDU over UDP.

The client MUST send the Wave Confirm PDU immediately after consuming the audio data. The **cConfirmedBlockNo** field of the Wave Confirm PDU MUST be identical to the **cBlockNo** field of the PDU that sent the audio data, whether it is a WaveInfo PDU, a Wave Encrypt PDU, or a UDP Wave Last PDU. The **wTimeStamp** field MUST be set to the same field of the originating WaveInfo PDU,

Wave Encrypt PDU or UDP Wave Last PDU. The value of that field MUST be adjusted by adding the time, in milliseconds, between receiving the packet from the network and sending this PDU. This enables the server to calculate the amount of time it takes for the client to receive the audio data PDU and send the confirmation.

3.2.5.2.1.7 Processing a Close PDU

The structure and fields of the Close PDU are specified in section 2.2.3.9. The Close PDU is sent when the server intends to stop rendering audio (for example, just before a disconnect).

Upon receiving the Close PDU, the client MUST not render any audio received after the Close PDU. The client finishes any audio that arrived before this PDU and which remains to be rendered. This PDU signals the end of audio transfer. As a result, the only PDU's that may be sent to the server after receiving a Close PDU are a Training PDU, and a Server Audio Formats and Version PDU (which will restart the entire audio output redirection protocol).

This packet MUST be received over static virtual channels.

3.2.5.3 Settings Transfer Sequence

The Settings Transfer Sequence messages are used to send audio settings changes from the server to the client. These packets are sent anytime after the initialization sequence or any time before the server sends a Close PDU.

3.2.5.3.1 Messages

3.2.5.3.1.1 Processing a Volume PDU

The structure and fields of the Volume PDU are specified in section 2.2.4.1.

On receiving a Volume PDU, the client MUST adjust the volume to the value specified in the **Volume** field.

3.2.5.3.1.2 Processing a Pitch PDU

The structure and fields of the Pitch PDU are specified in section 2.2.4.2.

On receiving a Pitch PDU, the client does nothing.

3.2.6 Timer Events

No client timer events are used.

3.2.7 Other Local Events

No additional client events are used.

3.3 Server Details

3.3.1 Abstract Data Model

The abstract data model is specified in section 3.1.1.

3.3.2 Timers

No server timers are used.

3.3.3 Initialization

Initialization is specified in section 3.1.3.

3.3.4 Higher-Layer Triggered Events

The server must play and stream audio. For example, if a user opens an audio file in a media player, the server initiates this protocol and begins streaming the audio.

3.3.5 Message Processing Events and Sequencing Rules.

3.3.5.1 Initialization Sequence

3.3.5.1.1 Messages

3.3.5.1.1.1 Sending a Server Audio Formats and Version PDU

The structure and fields of the Server Audio Formats and Version PDU are specified in section 2.2.2.1.

This MUST be the first message the server sends to the client.

3.3.5.1.1.2 Processing a Client Audio Formats and Version PDU

The structure and fields of the Client Audio Formats and Version PDU are specified in section 2.2.2.2. The server MUST receive this message prior to receiving any other message sent by the client.

The list of formats sent by the client are referenced in the data transfer sequence. The **wFormatNo** field of the WaveInfo PDU, the UDP Wave Last PDU, and the Wave Encrypt PDU all represent an index into this list. A value of **i** refers to the **i**th format of this list which means that the audio data is encoded in the **i**th format of the list.

The **wDGramPort** field holds the value of the port that the server MUST use to send data over UDP. If the value is set to 0, then the server MUST use static virtual channels for the data transfer sequence. If the field is NOT set to 0, then the server MAY <5> use UDP.

Although the **dwPitch** field specifies the initial pitch on the client, the server does nothing with this value.

3.3.5.1.1.3 Sending a Training PDU

The structure and fields of the Training PDU are specified in section 2.2.3.1.

The server MAY send the Training PDU at any time and during any sequence, not just during the initialization sequence. The only prerequisite is that version exchange MUST have occurred.

If the client advertises a UDP port during version exchange, then the server MAY<6> choose to send the Training PDU over UDP, but does not have to.

During the initialization sequence, the server sends a Training PDU and receives a Training Confirm PDU for diagnostic purposes.

3.3.5.1.1.4 Processing a Training Confirm PDU

The structure and fields of the Training Confirm PDU are specified in section 2.2.3.2.

A Training Confirm PDU is received ONLY if the server sends a Training PDU. The **wTimeStamp** and **wPackSize** fields MUST contain the same value as the corresponding fields in the Training PDU sent by the server.

The server uses this PDU for diagnostic purposes.

3.3.5.1.1.5 Sending a Crypt Key PDU

The structure and fields of the Crypt Key PDU are specified in section 2.2.2.3.

A Crypt Key PDU MUST only be sent over static virtual channels. The server sends this PDU only if it intends to use UDP for the data transfer sequence. To use UDP, the client MUST have advertised a valid port during version exchange, and the server MUST have successfully sent a Training PDU and received a Training Confirm PDU from the client over UDP.

3.3.5.2 Data Transfer Sequence

The data transfer sequence messages are used to send audio data from the server to the client.

As specified in section 1.3.2.2, there are three distinct sequences for the exchange of audio data.

1. The first involves sending a WaveInfo PDU and a Wave PDU, and receiving a Wave Confirm PDU over static virtual channels.
2. The second involves sending a Wave Encrypt PDU and receiving a Wave Confirm PDU over UDP.
3. The third involves sending several UDP Wave PDUs and a UDP Wave Last PDU, and receiving a Wave Confirm PDU over UDP.

If the client does not advertise a valid port for UDP during version exchange, then the first sequence MUST be used.

If the client does advertise a valid port for UDP and the version of either the client or server are below 5, then the first or second sequence MAY<7> be used.

If the client does advertise a valid port for UDP and the version of both the client and the server are at least 5, then any of the three sequences MAY<8> be used.

For the data transfer sequence to take place, the client MUST have set the TSSNDCAPS_ALIVE (0x0000001) flag in the Client Audio Formats and Version PDU.

Once a particular sequence is selected for use by the server, that sequence SHOULD be used throughout the protocol. Any malformed packets MUST be ignored.

3.3.5.2.1 Messages

3.3.5.2.1.1 Sending a WaveInfo PDU

The structure and fields of the WaveInfo PDU are specified in section 2.2.3.3.

A WaveInfo PDU and a Wave PDU, sent consecutively by the server, combine to form an audio sample. The audio sample MUST be greater than four bytes. The first four bytes of the audio sample are placed in the **data** field of this PDU. The remaining data is sent in the **data** field of the Wave PDU that immediately follows this PDU.

The **BodySize** field of the RDPSND PDU Header of this PDU MUST be set to 8 bytes more than the size of the entire audio sample.

The **cBlockNo** field MUST be one more than the **cBlockNo** field of the last audio sample sent. If the value of the last **cBlockNo** was 255, then the value of **cBlockNo** for this PDU MUST be 0. If this is the first audio sample sent, then the **cBlockNo** field MUST be one more than the **cLastConfirmedBlockNo** field of the Server Audio Formats and Version PDU sent by the server to the client.

The **wFormatNo** field is an index into the list of formats sent by the client in the Client Audio Formats and Version PDU. A value of **i** means the format of the audio data is the **i**th format of that list.

This PDU MUST be sent over static virtual channels.

3.3.5.2.1.2 Sending a Wave PDU

The structure and fields of the Wave PDU are specified in section 2.2.3.4.

A WaveInfo PDU and a Wave PDU, sent consecutively by the server, combine to form an audio sample.

This PDU MUST be sent over static virtual channels.

3.3.5.2.1.3 Sending a Wave Encrypt PDU

The structure and fields of the Wave Encrypt PDU are specified in section 2.2.3.5.

Unlike a WaveInfo PDU and Wave PDU, the Wave Encrypt PDU contains the entire audio sample in the **data** field.

The **cBlockNo** field MUST be set as specified in section 3.3.5.2.1.1.

The **wFormatNo** field is an index into the list of formats sent by the client in the Client Audio Formats and Version PDU. A value of **i** means the format of the audio data is the **i**th format of that list.

The audio data MUST be encrypted. Given:

- The original audio data of the same size
- And given a 36-byte number, where:
  - the first 32 bytes are the field **Seed**, exchanged in the Crypt Key PDU during the initialization sequence
  - the 33rd byte is **cBlockNo**
  - the final three bytes are 0x000000

An **SHA-1 hash** algorithm (as specified in [FIPS180-2]) is run over this 36-byte number and the field **data** to produce a 20-byte hash. The original audio data is encrypted with **RC4** (as specified in [SCHNEIER]) using this 20 byte hash as a key.

If the client and server versions are both at least 5, then the **signature** field MUST exist. Otherwise, the field MUST NOT exist. This is how the signature is created. Given:

- A 36-byte number, where:
  - the first 32 bytes are the field **Seed**, exchanged in the Crypt Key PDU during the initialization sequence
  - the 33rd byte is **cBlockNo**
  - and the final three bytes are 0x000000

An SHA-1 hash algorithm is run over this 36-byte number and the field **data** to produce a 20-byte hash. The value of this field is set to the first 8 bytes of this hash.

This PDU MUST be sent over UDP.

3.3.5.2.1.4 Sending a UDP Wave PDU

The structure and fields of the UDP Wave PDU are specified in section 2.2.3.6.

If the client and server's versions are both at least 5, the server MAY choose to send an Audio FragData structure, using several PDUs. All PDUs, except for the final one, MUST be UDP Wave PDUs. The final PDU MUST be a UDP Wave Last PDU. The **cFragNo** value of each UDP Wave PDU corresponds to the order of the fragment Audio FragData structure. The very first fragment at the beginning of the audio sample MUST have a **cFragNo** value of 0, and each successive fragment MUST have a **cFragNo** value that is 1 more than the preceding fragment.

The **cBlockNo** field of all UDP Wave PDUs holding fragments of an audio sample MUST be the same. The **cBlockNo** field MUST be set as specified in section 3.3.5.2.1.1.

This PDU is sent over UDP and only if the client and server's versions are both at least 5.

3.3.5.2.1.5 Sending a UDP Wave Last PDU

The structure and fields of the UDP Wave Last PDU are specified in section 2.2.3.7.

The **cBlockNo** field MUST be set as specified in section 3.3.5.2.1.1.

The **wFormatNo** field is an index into the list of formats sent by the client in the Client Audio Formats and Version PDU. A value of **i** means the format of the audio data is the **i**th format of that list.

This PDU is sent over UDP and only if the client and server's versions are both at least 5.

3.3.5.2.1.6 Processing a Wave Confirm PDU

The structure and fields of the Wave Confirm PDU are specified in section 2.2.3.8.

Upon receiving a Wave Confirm PDU, the server knows that the client consumed the audio sample that has a **cBlockNo** value identical to **cConfirmedBlockNo**.

3.3.5.2.1.7 Sending a Close PDU

The structure and fields of the Close PDU are specified in section 2.2.3.9.

To stop sending audio, the server sends this PDU.

This packet MUST be sent over static virtual channels.

3.3.5.3 Audio Settings Transfer Sequence

The audio settings transfer sequence messages are used to send audio setting changes from the server to the client.

3.3.5.3.1 Messages

3.3.5.3.1.1 Sending a Volume PDU

The structure and fields of the Volume PDU are specified in section 2.2.4.1.

For the server to send this packet, the client MUST have had the TSSNDCAPS_VOLUME (0x00000002) flag set in the **dwFlags** field of the Client Audio Formats and Version PDU sent during the initialization sequence.

3.3.5.3.1.2 Sending a Pitch PDU

The structure and fields of the Pitch PDU are specified in section 2.2.4.2.

For the server to send this packet, the client MUST have had the TSSNDCAPS_PITCH (0x00000004) flag set in the **dwFlags** field of the Client Audio Formats and Version PDU sent during the initialization sequence.

3.3.6 Timer Events

No server timer events are used.

3.3.7 Other Local Events

No additional server events are used.

4 Protocol Examples

The following sections describe several operations used in common scenarios to illustrate the function of the Remote Desktop Protocol: Audio Output Virtual Channel Extension.

4.1 Annotated Initialization Sequence

The following is an annotated dump of an initialization sequence using static virtual channels for data transfer, as specified in section 1.3.2.1.

4.1.1 Server Audio Formats and Version PDU

The following is an annotated dump of a Server Audio Formats and Version PDU.

```
00000000 07 2b 90 00 08 fb 8b 00 e0 f1 09 00 70 27 1f 77  .+..........p'.w
00000010 00 00 05 00 ff 05 00 00 01 00 02 00 22 56 00 00  ............"V..
00000020 88 58 01 00 04 00 10 00 00 00 06 00 02 00 22 56  .X............"V
00000030 00 00 44 ac 00 00 02 00 08 00 00 00 07 00 02 00  ..D.............
00000040 22 56 00 00 44 ac 00 00 02 00 08 00 00 00 02 00  "V..D...........
00000050 02 00 22 56 00 00 27 57 00 00 00 04 04 00 20 00  ..."V..'W...... .
00000060 f4 03 07 00 00 01 00 00 00 02 00 ff 00 00 00 00  ................
00000070 c0 00 40 00 f0 00 00 00 cc 01 30 ff 88 01 18 ff  ..@.......0.....
00000080 11 00 02 00 22 56 00 00 b9 56 00 00 00 04 04 00  ....."V...V......
00000090 02 00 f9 03

07 -> SNDPROLOG::Type = SNDC FORMATS (7)
2b -> SNDPROLOG::bPad = 0x2b
90 00 -> SNDPROLOG::BodySize = 0x90 = 144 bytes

08 fb 8b 00 -> SERVER_AUDIO_VERSION_AND_FORMATS::dwFlags = 0x008bfb08
e0 f1 09 00 -> SERVER_AUDIO_VERSION_AND_FORMATS::dwVolume = 0x0009f1e0
70 27 1f 77 -> SERVER_AUDIO_VERSION_AND_FORMATS::dwPitch = 0x771f2770
00 00 -> SERVER AUDIO VERSION AND FORMATS::wDGramPort = 0
05 00 -> SERVER AUDIO VERSION AND FORMATS::wNumberOfFormats = 5
ff -> SERVER AUDIO VERSION AND FORMATS::cLastBlockConfirmed = 0xff = 255
05 00 -> SERVER AUDIO VERSION AND FORMATS::wVersion = 5
00 -> SERVER_AUDIO_VERSION_AND_FORMATS::bPad = 0
01 00 02 00 22 56 00 00 88 58 01 00 04 00 10 00 00 00 -> AUDIO FORMAT
01 00 -> AUDIO FORMAT::wFormatTag = 1
   02 00 -> AUDIO FORMAT::nChannels = 2
   22 56 00 00 -> AUDIO FORMAT::nSamplesPerSec = 0x5622 = 22050
   88 58 01 00 -> AUDIO_FORMAT::nAvgBytesPerSec = 0x15888 = 88200
   04 00 -> AUDIO_FORMAT::nBlockAlign = 0x0004 = 4
   10 00 -> AUDIO FORMAT::wBitsPerSample = 0x10 = 16
   00 00 -> AUDIO FORMAT::cbSize = 0
06 00 02 00 22 56 00 00 44 ac 00 00 02 00 08 00 00 00 -> AUDIO FORMAT
   06 00 -> AUDIO FORMAT::wFormatTag = 6
   02 00 -> AUDIO_FORMAT::nChannels = 2
   22 56 00 00 -> AUDIO_FORMAT::nSamplesPerSec = 0x5622 = 22050
   44 ac 00 00 -> AUDIO FORMAT::nAvgBytesPerSec = 0x44ac = 44100
   02 00 -> AUDIO FORMAT::nBlockAlign = 2
   08 00 -> AUDIO FORMAT::wBitsPerSample = 8
   00 00 -> AUDIO FORMAT::cbSize = 0
07 00 02 00 22 56 00 00 44 ac 00 00 02 00 08 00 00 00 -> AUDIO_FORMAT
   07 00 -> AUDIO_FORMAT::wFormatTag = 7
   02 00 -> AUDIO FORMAT::nChannels = 2
   22 56 00 00 -> AUDIO FORMAT::nSamplesPerSec = 0x5622 = 22050
   44 ac 00 00 -> AUDIO FORMAT::nAvgBytesPerSec = 0x44ac = 44100
   02 00 -> AUDIO FORMAT::nBlockAlign = 2
   08 00 -> AUDIO_FORMAT::wBitsPerSample = 8
   00 00 -> AUDIO_FORMAT::cbSize = 0
02 00 02 00 22 56 00 00 27 57 00 00 00 04 04 00 20 00 f4 03 07 00 00 01
```

```
00 00 00 02 00 ff 00 00 00 00 c0 00 40 00 f0 00 00 00 cc 01 30 ff 88 01
18 ff -> AUDIO_FORMAT
   02 00 -> AUDIO_FORMAT::wFormatTag = 2
   02 00 -> AUDIO FORMAT::nChannels = 2
   22 56 00 00 -> AUDIO FORMAT::nSamplesPerSec = 0x5622 = 22050
   27 57 00 00 -> AUDIO FORMAT::nAvgBytesPerSec = 0x5727 = 22311
   00 04 -> AUDIO FORMAT::nBlockAlign = 0x400 = 1024
   04 00 -> AUDIO_FORMAT::wBitsPerSample = 4
   20 00 -> AUDIO_FORMAT::cbSize = 0x20 = 32
f4 03 07 00 00 01 00 00 00 02 00 ff 00 00 00 00 c0 00 40 00 f0 00 00
00 cc 01 30 ff 88 01 18 ff -> data
11 00 02 00 22 56 00 00 b9 56 00 00 00 04 04 00 02 00 f9 03 -> AUDIO FORMAT
   11 00 -> AUDIO FORMAT::wFormatTag = 0x11 = 17
   02 00 -> AUDIO_FORMAT::nChannels = 2
   22 56 00 00 -> AUDIO_FORMAT::nSamplesPerSec = 0x5622 = 22050
   b9 56 00 00 -> AUDIO FORMAT::nAvgBytesPerSec =0x56b9 = 22201
   00 04 -> AUDIO FORMAT::nBlockAlign = 0x400 = 1024
   04 00 -> AUDIO FORMAT::wBitsPerSample = 4
   02 00 -> AUDIO_FORMAT::cbSize = 2
   f9 03 -> AUDIO_FORMAT::data
```

4.1.2 Client Audio Formats and Version PDU

The following is an annotated dump of a Client Audio Formats and Version PDU.

```
00000000 07 00 90 00 03 00 00 00 ff ff ff ff 00 f7 f9 00  ................
00000010 00 00 05 00 28 05 00 7c 01 00 02 00 22 56 00 00  ....(..|....."V..
00000020 88 58 01 00 04 00 10 00 00 00 06 00 02 00 22 56  .X............."V
00000030 00 00 44 ac 00 00 02 00 08 00 00 00 07 00 02 00  ..D.............
00000040 22 56 00 00 44 ac 00 00 02 00 08 00 00 00 02 00  "V..D...........
00000050 02 00 22 56 00 00 27 57 00 00 00 04 04 00 20 00  ..."V..'W...... .
00000060 f4 03 07 00 00 01 00 00 00 02 00 ff 00 00 00 00  ................
00000070 c0 00 40 00 f0 00 00 00 cc 01 30 ff 88 01 18 ff  ..@.......0.....
00000080 11 00 02 00 22 56 00 00 b9 56 00 00 00 04 04 00  ...."V...V......
00000090 02 00 f9 03

07 -> SNDPROLOG::Type = SNDC FORMATS (7)
00 -> SNDPROLOG::bPad = 0
90 00 -> SNDPROLOG::BodySize = 0x90 = 144 bytes

03 00 00 00 -> CLIENT AUDIO VERSION AND FORMATS::dwFlags = 0x00000003
0x03
= 0x01 |
  0x03
= TSSNDCAPS_ALIVE |
  TSSNDCAPS_VOLUME
ff ff ff ff -> CLIENT AUDIO VERSION AND FORMATS::dwVolume = 0xffffffff
00 f7 f9 00 -> CLIENT AUDIO VERSION AND FORMATS::dwPitch = 0x00f9f700
00 00 -> CLIENT AUDIO VERSION AND FORMATS::wDGramPort = 0
05 00 -> CLIENT AUDIO VERSION AND FORMATS::wNumberOfFormats = 5
28 -> CLIENT_AUDIO_VERSION_AND_FORMATS::cLastBlockConfirmed = 0x28
05 00 -> CLIENT_AUDIO_VERSION_AND_FORMATS::wVersion = 5
7c -> CLIENT AUDIO VERSION AND FORMATS::bPad = 0x7c
01 00 02 00 22 56 00 00 88 58 01 00 04 00 10 00 00 00 -> AUDIO FORMAT
  01 00 -> AUDIO FORMAT::wFormatTag = 1
  02 00 -> AUDIO FORMAT::nChannels = 2
  22 56 00 00 -> AUDIO_FORMAT::nSamplesPerSec = 0x5622 = 22050
  88 58 01 00 -> AUDIO_FORMAT::nAvgBytesPerSec = 0x15888 = 88200
  04 00 -> AUDIO FORMAT::nBlockAlign = 0x0004 = 4
  10 00 -> AUDIO FORMAT::wBitsPerSample = 0x10 = 16
  00 00 -> AUDIO FORMAT::cbSize = 0
06 00 02 00 22 56 00 00 44 ac 00 00 02 00 08 00 00 00 -> AUDIO_FORMAT
  06 00 -> AUDIO_FORMAT::wFormatTag = 6
```

```
  02 00 -> AUDIO FORMAT::nChannels = 2
  22 56 00 00 -> AUDIO_FORMAT::nSamplesPerSec = 0x5622 = 22050
  44 ac 00 00 -> AUDIO_FORMAT::nAvgBytesPerSec = 0x44ac = 44100
  02 00 -> AUDIO FORMAT::nBlockAlign = 2
  08 00 -> AUDIO FORMAT::wBitsPerSample = 8
  00 00 -> AUDIO FORMAT::cbSize = 0
07 00 02 00 22 56 00 00 44 ac 00 00 02 00 08 00 00 00 -> AUDIO FORMAT
  07 00 -> AUDIO_FORMAT::wFormatTag = 7
  02 00 -> AUDIO_FORMAT::nChannels = 2
  22 56 00 00 -> AUDIO FORMAT::nSamplesPerSec = 0x5622 = 22050
  44 ac 00 00 -> AUDIO FORMAT::nAvgBytesPerSec = 0x44ac = 44100
  02 00 -> AUDIO FORMAT::nBlockAlign = 2
  08 00 -> AUDIO FORMAT::wBitsPerSample = 8
  00 00 -> AUDIO_FORMAT::cbSize = 0
02 00 02 00 22 56 00 00 27 57 00 00 00 04 04 00 20 00 f4 03 07 00 00 01 00 00 00 02 00 ff
00 00 00 00 c0 00 40 00 f0 00 00 00 cc 01 30 ff 88 01 18 ff -> AUDIO FORMAT
  02 00 -> AUDIO FORMAT::wFormatTag = 2
  02 00 -> AUDIO FORMAT::nChannels = 2
  22 56 00 00 -> AUDIO FORMAT::nSamplesPerSec = 0x5622 = 22050
  27 57 00 00 -> AUDIO_FORMAT::nAvgBytesPerSec = 0x5727 = 22311
  00 04 -> AUDIO FORMAT::nBlockAlign = 0x400 = 1024
  04 00 -> AUDIO FORMAT::wBitsPerSample = 4
  20 00 -> AUDIO FORMAT::cbSize = 0x20 = 32
  f4 03 07 00 00 01 00 00 00 02 00 ff 00 00 00 00 c0 00 40 00 f0 00 00 00 cc 01 30 ff 88
01 18 ff -> data
11 00 02 00 22 56 00 00 b9 56 00 00 00 04 04 00 02 00 f9 03 -> AUDIO_FORMAT
  11 00 -> AUDIO_FORMAT::wFormatTag = 0x11 = 17
  02 00 -> AUDIO FORMAT::nChannels = 2
  22 56 00 00 -> AUDIO FORMAT::nSamplesPerSec = 0x5622 = 22050
  b9 56 00 00 -> AUDIO FORMAT::nAvgBytesPerSec =0x56b9 = 22201
  00 04 -> AUDIO FORMAT::nBlockAlign = 0x400 = 1024
  04 00 -> AUDIO_FORMAT::wBitsPerSample = 4
  02 00 -> AUDIO_FORMAT::cbSize = 2
  f9 03 -> AUDIO_FORMAT::data
```

4.1.3 Training PDU

The following is an annotated dump of a Training PDU.

```
00000000 06 23 fc 03 da 89 00 04 52 90 49 f1 f1 bb e9 eb  .#......R.I.....
00000010 b3 a6 db 3c 87 0c 3e 99 24 5e 0d 1c 06 b7 47 de  ...<..>.$^....G.
00000020 b3 12 4d c8 43 bb 8b a6 1f 03 5a 7d 09 38 25 1f  ..M.C.....Z}.8%.
00000030 5d d4 cb fc 96 f5 45 3b 13 0d 89 0a 1c db ae 32  ].....E;.......2
…
000003d0 20 9a 50 ee 40 78 36 fd 12 49 32 f6 9e 7d 49 dc   .P.@x6..I2..}I.
000003e0 ad 4f 14 f2 44 40 66 d0 6b c4 30 b7 32 3b a1 22  .O..D@f.k.0.2;."
000003f0 f6 22 91 9d e1 8b 1f da b0 ca 99 02 b9 72 9d 49  ."...........r.I

06 -> SNDPROLOG::Type = SNDC TRAINING (6)
23 -> SNDPROLOG::bPad = 0x23
fc 03 -> SNDPROLOG::BodySize = 0x3fc
da 89 -> SNDTRAINING::wTimeStamp = 0x89da
00 04 -> SNDTRAINING::wPackSize = 0x400 = 1024 bytes
52 90 49 … 72 9d 49 -> SNDTRAINING::data
```

4.1.4 Training Confirm PDU

The following is an annotated dump of a Training Confirm PDU.

```
00000000 06 55 04 00 da 89 00 04                          .#......
```

```
06 -> SNDPROLOG::Type = SNDC_TRAINING (6)
55 -> SNDPROLOG::bPad = 0x55
04 00 -> SNDPROLOG::BodySize = 0x4
da 89 -> SNDTRAINING::wTimeStamp = 0x89da
00 04 -> SNDTRAINING::wPackSize = 0x400 = 1024 bytes
```

4.2 Annotated Static Virtual Channel Data Transfer Sequence

The following is an annotated dump of a data transfer sequence over static virtual channels, as specified in section 1.3.2.2.

4.2.1 WaveInfo PDU

The following is an annotated dump of a WaveInfo PDU.

```
00000000 02 7e 51 02 d7 ad 0f 00 08 00 00 00 20 48 17 d6  .~Q.........H...

02 -> SNDPROLOG::Type = SNDC WAVE (2)
7e -> SNDPROLOG::bPad = 0x7e
51 02 -> SNDPROLOG::BodySize = 0x251 = 593 bytes

d7 ad -> SNDWAVINFO::wTimeStamp = 0xadd7
0f 00 -> SNDWAVINFO::wFormatNo = 0xf = format #15
08 -> SNDWAVINFO::cBlockNo = 8
00 00 00 -> SNDWAVINFO::bPad = 0
20 48 17 d6 -> SNDWAVINFO::data
```

4.2.2 Wave PDU

The following is an annotated dump of a Wave PDU.

```
00000000 00 00 00 00 84 02 80 24 49 92 24 89 02 80 24 49  .......$I.$...$I
00000010 92 24 89 02 80 24 49 92 24 89 02 80 24 49 92 24  .$...$I.$...$I.$
00000020 09 82 74 61 4d 28 00 48 92 24 49 92 28 00 48 92  ..taM(.H.$I.(.H.
...
00000030 0f 7b de 20 b2 2a 72 74 37 d9 bc dd 5f 4d 0b 58  .{. .*rt7... M.X
00000040 a5 05 a9 12 3c 19 40 59 6a 48 aa 4e 11 4c 51 63  ....<.@YjH.N.LQc
00000050 55 cd 57 1f f8 91 ba 48 aa                       U.W....H.

00 00 00 00 -> SNDWAVE::Type = 0
84 02 80… ba 48 aa -> SNDWAVE::data
```

4.2.3 Wave Confirm PDU

The following is an annotated dump of a Wave Confirm PDU.

```
00000000 05 39 04 00 b7 5a 08 77                          .9...Z.w

05 -> SNDPROLOG::Type = SNDC_WAVECONFIRM
39 -> SNDPROLOG::bPad = 0x39
04 00 -> SNDPROLOG::BodySize = 0x4 = 4 bytes

b7 5a -> SNDWAV_CONFIRM::wTimeStamp = 0x5ab7
08 -> SNDWAV_CONFIRM::cConfirmedBlockNo = 8
```

```
77 -> SNDWAV_CONFIRM::bPad = 0x77
```

4.3 Annotated UDP Data Transfer Sequence

The following is an annotated dump of a data transfer sequence over UDP when the client and server versions are both less than 5, as specified in section 1.3.2.2.

4.3.1 Wave Encrypt PDU

The following is an annotated dump of a Wave Encrypt PDU.

```
00cd7dd0  09 e0 b8 03 b4 d0 2d 00 24 00 00 00 fd 19 07 55
...
00cd8180  a6 ba 89 4c 36 f2 56 89 dd c0 42 78

09 -> SNDPROLOG::Type = SNDC_WAVEENCRYPT (9)
e0 -> SNDPROLOG::bPad = 0xe0
b8 03 -> SNDPROLOG::BodySize = 0x3b8 = 952 bytes
b4 d0 -> SNDWAVCRYPT::wTimeStamp = 0xd0b4
2d 00 -> SNDWAVCRYPT::wFormatNo = 0x2d = format #45
24 -> SNDWAVCRYPT::cBlockNo = 0x24 = 36
00 00 00 -> SNDWAVCRYPT::bPad
fd 19 07 55 ... c0 42 78  -> SNDWAVCRYPT::data
```

4.3.2 Wave Confirm PDU

The following is an annotated dump of a Wave Confirm PDU.

```
00000000 05 25 04 00 b7 5a 24 22                         .9...Z.w

05 -> SNDPROLOG::Type = SNDC_WAVECONFIRM
25 -> SNDPROLOG::bPad = 0x39
04 00 -> SNDPROLOG::BodySize = 0x4 = 4 bytes

b7 5a -> SNDWAV_CONFIRM::wTimeStamp = 0x5ab7
24 -> SNDWAV_CONFIRM::cConfirmedBlockNo = 0x24 = 36
22 -> SNDWAV_CONFIRM::bPad = 0x22
```

4.4 Annotated UDP Data Transfer Sequence

The following is an annotated dump of a data transfer sequence over UDP when the client and server versions are both at least 5, as specified in section 1.3.2.2.

4.4.1 UDP Wave PDU

The following is an annotated dump of a UDP Wave PDU.

```
00000000  0a 00 00 87 27 b8 77 78 21 b9 e8 00 00 00 00 00
00000010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

0a -> SNDUDPWAVE::Type = SNDC_UDPWAVE (0xa)
00 -> SNDUDPWAVE::cBlockNo = 0
```

```
00 -> SNDUDPWAVE::cFragNo = 0
87 27 b8 ... 00 00 00 -> SNDUDPWAVE::data
```

4.4.2 UDP Wave Last PDU

The following is an annotated dump of a UDP Wave Last PDU.

```
00000000  0b 08 20 b9 1a 04 00 00 00 00 00 00 00 00 00 00
00000010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

0b -> SNDUDPWAVELAST::Type
08 20 -> SNDUDPWAVELAST::wTotalSize = 0x2008
b9 1a -> SNDUDPWAVELAST::wTimeStamp = 0x1ab9
04 00 -> SNDUDPWAVELAST::wFormatNo = 0x4
00 -> SNDUDPWAVELAST::cBlockNo = 0
00 00 00 -> SNDUDPWAVELAST::bPad
00 00 00 ... 00 00 00 -> SNDUDPWAVELAST::data
```

4.4.3 Wave Confirm PDU

The following is an annotated dump of a Wave Confirm PDU.

```
00000000 05 25 04 00 b7 2a 00 22

05 -> SNDPROLOG::Type = SNDC_WAVECONFIRM
25 -> SNDPROLOG::bPad = 0x39
04 00 -> SNDPROLOG::BodySize = 0x4 = 4 bytes

b7 2a -> SNDWAV_CONFIRM::wTimeStamp = 0x2ab7
00 -> SNDWAV_CONFIRM::cConfirmedBlockNo = 0
22 -> SNDWAV_CONFIRM::bPad = 0x22
```

5 Security

5.1 Security Considerations for Implementers

All static virtual channel traffic is secured by the underlying core Remote Desktop Protocol. An overview of the implemented security-related mechanisms is specified in [MS-RDPBCGR] section 5.

When audio data is sent using UDP Wave PDUs and UDP Wave Last PDUs, the audio is not encrypted during transmission between the client and the server. However, verification that the audio data has been transmitted intact is possible since these PDUs are signed. Sending audio data using this UDP sequence is not recommended since the audio data is not encrypted. Instead, static virtual channels should be used.

When audio data is sent using Wave Encrypt PDUs, the audio data is encrypted using RC4 and SHA-1. Additionally, verification that the audio data has been transmitted intact is not possible since these PDUs are not signed. Sending audio data using this UDP sequence is not recommended since SHA-1 is proven insecure. Instead, static virtual channels should be used.

5.2 Index of Security Parameters

None.

6 Appendix A: Windows Behavior

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

- Windows Server 2008
- Windows Server 2003
- Windows 2000 Server
- Windows Vista
- Windows XP

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 2.2.2.1:

Client and Server Version	Meaning
0x02	Windows XP
0x05	Windows 2000
0x05	Windows Server 2003
0x05	Windows Vista
0x05	Windows Server 2008

<2> Section 2.2.2.2:

Client and Server Version	Meaning
0x02	Windows XP
0x05	Windows 2000
0x05	Windows Server 2003
0x05	Windows Vista
0x05	Windows Server 2008

<3> Section 3.2.5.1.1.2: Windows 2000, Windows XP, and Windows Server 2003 clients try to negotiate UDP using the API function getsockname. If the function is successful, then Windows clients advertise a UDP port. Otherwise, the clients will use static virtual channels. For more information on the getsockname function, see [MSDN-getsockname].

Windows Vista and Windows Server 2008 always use static virtual channels.

<4> Section 3.2.5.1.1.3: In Windows 2000 Server, Windows XP and Windows Server 2003, if a client advertises a UDP port during version exchange, then the Training PDUs are first sent over UDP. Static virtual channels are used only if the communication over UDP fails.

In Windows Vista and Windows Server 2008, the training PDU is always sent over static virtual channels even if the client has advertised a UDP port.

<5> Section 3.3.5.1.1.2: Windows 2000 Server, Windows Server 2003 and Windows XP use UDP. Windows Vista and Windows Server 2008 always use static virtual channels.

<6> Section 3.3.5.1.1.3: Windows 2000, Windows XP, and Windows Server 2003 clients try to negotiate UDP by using the API function getsockname. If the function is successful, then Windows clients advertise a UDP port. Otherwise, the clients will use static virtual channels. For more information on the getsockname function, see [MSDN-getsockname].

Windows Vista and Windows Server 2008 always use static virtual channels.

<7> Section 3.3.5.2: Windows 2000, Windows XP, and Windows Server 2003 clients try to negotiate UDP by using the API function getsockname. If the function is successful, then Windows clients advertise a UDP port. Otherwise, the clients will use static virtual channels. For more information on the getsockname function, see [MSDN-getsockname].

Windows Vista and Windows Server 2008 always use static virtual channels.

<8> Section 3.3.5.2: Windows 2000, Windows XP, and Windows Server 2003 clients try to negotiate UDP by using the API function getsockname. If the function is successful, then Windows clients advertise a UDP port. Otherwise, the clients will use static virtual channels. For more information on the getsockname function, see [MSDN-getsockname].

When sending an audio sample via UDP, Windows servers will use either Wave Encrypt PDU or UDP Wave PDU and UDP Wave Last PDU, depending on the size of the sample, the maximum datagram size for the UDP socket, and system configuration parameters. Typically, samples over 1,460 bytes in size will be sent by using UDP Wave PDU and UDP Wave Last PDU.

Windows Vista and Windows Server 2008 always use static virtual channels.

7 Index