[MS-DRMND]:

Windows Media Digital Rights Management (WMDRM): Network Devices Protocol

---

Intellectual Property Rights Notice for Open Specifications Documentation

- **Technical Documentation.** Microsoft publishes Open Specifications documentation ("this documentation") for protocols, file formats, data portability, computer languages, and standards support. Additionally, overview documents cover inter-protocol relationships and interactions.
- **Copyrights**. This documentation is covered by Microsoft copyrights. Regardless of any other terms that are contained in the terms of use for the Microsoft website that hosts this documentation, you can make copies of it in order to develop implementations of the technologies that are described in this documentation and can distribute portions of it in your implementations that use these technologies or in your documentation as necessary to properly document the implementation. You can also distribute in your implementation, with or without modification, any schemas, IDLs, or code samples that are included in the documentation. This permission also applies to any documents that are referenced in the Open Specifications documentation.
- **No Trade Secrets**. Microsoft does not claim any trade secret rights in this documentation.
- **Patents**. Microsoft has patents that might cover your implementations of the technologies described in the Open Specifications documentation. Neither this notice nor Microsoft's delivery of this documentation grants any licenses under those patents or any other Microsoft patents. However, a given Open Specifications document might be covered by the Microsoft Open Specifications Promise or the Microsoft Community Promise. If you would prefer a written license, or if the technologies described in this documentation are not covered by the Open Specifications Promise or Community Promise, as applicable, patent licenses are available by contacting iplg@microsoft.com.
- **License Programs**. To see all of the protocols in scope under a specific license program and the associated patents, visit the Patent Map.
- **Trademarks**. The names of companies and products contained in this documentation might be covered by trademarks or similar intellectual property rights. This notice does not grant any licenses under those rights. For a list of Microsoft trademarks, visit www.microsoft.com/trademarks.
- **Fictitious Names**. The example companies, organizations, products, domain names, email addresses, logos, people, places, and events that are depicted in this documentation are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

Reservation of Rights. All other rights are reserved, and this notice does not grant any rights other than as specifically described above, whether by implication, estoppel, or otherwise.

Tools. The Open Specifications documentation does not require the use of Microsoft programming tools or programming environments in order for you to develop an implementation. If you have access to Microsoft programming tools and environments, you are free to take advantage of them. Certain Open Specifications documents are intended for use in conjunction with publicly available standards specifications and network programming art and, as such, assume that the reader either is familiar with the aforementioned material or has immediate access to it.

Support. For questions and support, please contact dochelp@microsoft.com.

Revision Summary

Date	Revision History	Revision Class	Comments
4/23/2010	0.1	Major	First Release.
6/4/2010	0.1.1	Editorial	Changed language and formatting in the technical content.
7/16/2010	1.0	Major	Updated and revised the technical content.
8/27/2010	2.0	Major	Updated and revised the technical content.
10/8/2010	3.0	Major	Updated and revised the technical content.
11/19/2010	4.0	Major	Updated and revised the technical content.
1/7/2011	4.0	None	No changes to the meaning, language, or formatting of the technical content.
2/11/2011	4.0	None	No changes to the meaning, language, or formatting of the technical content.
3/25/2011	4.0	None	No changes to the meaning, language, or formatting of the technical content.
5/6/2011	4.0	None	No changes to the meaning, language, or formatting of the technical content.
6/17/2011	4.1	Minor	Clarified the meaning of the technical content.
9/23/2011	4.1	None	No changes to the meaning, language, or formatting of the technical content.
12/16/2011	5.0	Major	Updated and revised the technical content.
3/30/2012	5.0	None	No changes to the meaning, language, or formatting of the technical content.
7/12/2012	5.0	None	No changes to the meaning, language, or formatting of the technical content.
10/25/2012	5.0	None	No changes to the meaning, language, or formatting of the technical content.
1/31/2013	6.0	Major	Updated and revised the technical content.
8/8/2013	7.0	Major	Updated and revised the technical content.
11/14/2013	7.0	None	No changes to the meaning, language, or formatting of the technical content.
2/13/2014	7.0	None	No changes to the meaning, language, or formatting of the technical content.
5/15/2014	7.0	None	No changes to the meaning, language, or formatting of the technical content.
6/30/2015	8.0	Major	Significantly changed the technical content.
10/16/2015	8.0	None	No changes to the meaning, language, or formatting of the technical content.
7/14/2016	9.0	Major	Significantly changed the technical content.

Date	Revision History	Revision Class	Comments
6/1/2017	9.0	None	No changes to the meaning, language, or formatting of the technical content.

Table of Contents

1 Introduction

Windows Media Digital Rights Management (WMDRM): Network Devices Protocol enables consumers to experience audio and/or video on multiple connected devices in the home, while protecting the rights of the content owner.

WMDRM: Network Devices Protocol extends the reach of protected content to consumer electronic devices on home networks. These devices include digital media receivers, referred to in this document as receivers, which are connected to transmitting devices, referred to in this document as transmitters. WMDRM: Network Devices Protocol enables the receivers to render protected content while enforcing the rights specified by the content owner.

WMDRM: Network Devices Protocol enables protected content to be streamed between the transmitter and the receiver. As a result, consumers can share digital media content (audio, video, and photos) on multiple devices that are connected by IP networks in the home.

This document describes how to support WMDRM: Network Devices Protocol on **receivers** and **transmitters**.

WMDRM: Network Devices Protocol includes procedures that are implemented using specific algorithms and data structures. This specification defines these procedures and how they are mapped onto network protocols, such as **UPnP**, HTTP, **RTSP**, and **RTP**.

WMDRM: Network Devices Protocol requires the use of the UPnP **device** architecture. Procedures that map to UPnP have only a single mapping. Other procedures, such as the Data Transfer procedure, have mappings to both HTTP and RTSP/RTP. Section 2.1.1 specifies how WMDRM: Network Devices Protocol is mapped onto UPnP, HTTP, RTSP/RTP, and **PBDA**.

Sections 1.5, 1.8, 1.9, 2, and 3 of this specification are normative. All other sections and examples in this specification are informative.

1.1 Glossary

This document uses the following terms:

Advanced Encryption Standard (AES): A block **cipher** that supersedes the Data Encryption Standard (DES). AES can be used to protect electronic data. The AES algorithm can be used to encrypt (encipher) and decrypt (decipher) information. **Encryption** converts data to an unintelligible form called ciphertext; decrypting the ciphertext converts the data back into its original form, called plaintext. AES is used in symmetric-key cryptography, meaning that the same key is used for the **encryption** and decryption operations. It is also a block cipher, meaning that it operates on fixed-size blocks of plaintext and ciphertext, and requires the size of the plaintext as well as the ciphertext to be an exact multiple of this block size. AES is also known as the Rijndael symmetric encryption algorithm [FIPS197].

Advanced Systems Format (ASF): An extensible file format that is designed to facilitate streaming digital media data over a network. This file format is used by Windows Media.

American National Standards Institute (ANSI) character set: A character set defined by a code page approved by the American National Standards Institute (ANSI). The term "ANSI" as used to signify Windows code pages is a historical reference and a misnomer that persists in the Windows community. The source of this misnomer stems from the fact that the Windows code page 1252 was originally based on an ANSI draft, which became International Organization for Standardization (ISO) Standard 8859-1 [ISO/IEC-8859-1]. In Windows, the ANSI character set can be any of the following code pages: 1252, 1250, 1251, 1253, 1254, 1255, 1256, 1257, 1258, 874, 932, 936, 949, or 950. For example, "ANSI application" is usually a reference to a non-Unicode or code-page-based application. Therefore, "ANSI character set" is often misused to refer to one of the character sets defined by a Windows code page that can be used as an

active system code page; for example, character sets defined by code page 1252 or character sets defined by code page 950. Windows is now based on Unicode, so the use of ANSI character sets is strongly discouraged unless they are used to interoperate with legacy applications or legacy data.

ASCII: The American Standard Code for Information Interchange (ASCII) is an 8-bit character-encoding scheme based on the English alphabet. ASCII codes represent text in computers, communications equipment, and other devices that work with text. ASCII refers to a single 8-bit ASCII character or an array of 8-bit ASCII characters with the high bit of each character set to zero.

Augmented Backus-Naur Form (ABNF): A modified version of Backus-Naur Form (BNF), commonly used by Internet specifications. ABNF notation balances compactness and simplicity with reasonable representational power. ABNF differs from standard BNF in its definitions and uses of naming rules, repetition, alternatives, order-independence, and value ranges. For more information, see [RFC5234].

authenticated commands: A cryptographically signed command flowing from the WMDRM: Network Devices Protocol Receiver to the WMDRM: Network Devices Protocol Transmitter.

base license: A reference Windows Media DRM **policy** from which a Windows Media DRM **license** is derived.

binary large object (BLOB): A collection of binary data stored as a single entity in a database.

certificate chain: A sequence of certificates, where each certificate in the sequence is signed by the subsequent certificate. The last certificate in the chain is normally a self-signed certificate.

certificate revocation list (CRL): A list of certificates that have been revoked by the **certification authority (CA)** that issued them (that have not yet expired of their own accord). The list must be cryptographically signed by the **CA** that issues it. Typically, the certificates are identified by serial number. In addition to the serial number for the revoked certificates, the CRL contains the revocation reason for each certificate and the time the certificate was revoked. As described in [RFC3280], two types of CRLs commonly exist in the industry. Base CRLs keep a complete list of revoked certificates, while delta CRLs maintain only those certificates that have been revoked since the last issuance of a base CRL. For more information, see [X509] section 7.3, [MSFT-CRL], and [RFC3280] section 5.

certification authority (CA): A third party that issues **public key** certificates. Certificates serve to bind public keys to a user identity. Each user and certification authority (CA) can decide whether to trust another user or CA for a specific purpose, and whether this trust should be transitive. For more information, see [RFC3280].

Certified Output Protection Protocol (COPP): A **device driver** system used to secure high band-width digital communications channels between media applications and display devices.

challenge: A piece of data used to authenticate a user. Typically a challenge takes the form of a **nonce**.

cipher: A cryptographic algorithm used to encrypt and decrypt files and messages.

content: Multimedia data. **content** is always in **ASF**, for example, a single **ASF** music file or a single **ASF** video file. Data in general. A file that an application accesses. Examples of content include web pages and documents stored on either web servers or SMB file servers.

content encryption key: A cryptographic key that is used to encrypt data. **Content encryption keys** are used not only to encrypt content, but also to encrypt other secrets, such as the proximity detection nonce.

content integrity key: A cryptographic key that is used to assert message integrity. **Content encryption keys** are used to calculate the OMAC1 [OMAC] of various messages.

counter mode (CTR): A confidentiality mode that features the application of the forward **cipher** to a set of input blocks, called counters, to produce a sequence of output blocks that are exclusive-ORed (XOR) with the plaintext to produce either ciphertext during the **encryption** process or plaintext during the **decryption** process.

datagram: A style of communication offered by a network transport protocol where each message is contained within a single network packet. In this style, there is no requirement for establishing a session prior to communication, as opposed to a connection-oriented style.

decryption: In cryptography, the process of transforming encrypted information to its original clear text form.

device: Any peripheral or part of a computer system that can send or receive data.

device driver: The software that the system uses to communicate with a device such as a display, printer, mouse, or communications adapter. An abstraction layer that restricts access of applications to various hardware devices on a given computer system. It is often referred to simply as a "driver".

digest: The fixed-length output string from a one-way hash function that takes a variable-length input string and is probabilistically unique for every different input string. Also, a cryptographic checksum of a data (octet) stream.

digital signature: A message authenticator that is typically derived from a cryptographic operation by using an asymmetric algorithm and private key. When a symmetric algorithm is used for this purpose, the authenticator is typically referred to as a Message Authentication Code (MAC).

encryption: In cryptography, the process of obscuring information to make it unreadable without special knowledge.

exchange: A pair of messages, consisting of a request and a response.

file: An entity of data in the file system that a user can access and manage. A **file** must have a unique name in its directory. It consists of one or more streams of bytes that hold a set of related data, plus a set of attributes (also called properties) that describe the **file** or the data within the **file**. The creation time of a **file** is an example of a file attribute.

flags: A set of values used to configure or report options or settings.

flow: A TCP session or **User Datagram Protocol (UDP)** pseudo session, identified by a 5-tuple (source and destination IP and ports, and protocol). By extension, a request/response Internet Control Message Protocol (ICMP) exchange (for example, ICMP echo) is also a **flow**.

globally unique identifier (GUID): A term used interchangeably with universally unique identifier (UUID) in Microsoft protocol technical documents (TDs). Interchanging the usage of these terms does not imply or require a specific algorithm or mechanism to generate the value. Specifically, the use of this term does not imply or require that the algorithms described in [RFC4122] or [C706] must be used for generating the **GUID**. See also universally unique identifier (UUID).

hash function: A function that takes an arbitrary amount of data and produces a fixed-length result (a "hash") that depends only on the input data. A **hash function** is sometimes called a message digest or a digital fingerprint.

Hash-based Message Authentication Code (HMAC): A mechanism for message authentication using cryptographic hash functions. HMAC can be used with any iterative cryptographic hash

function (for example, MD5 and SHA-1) in combination with a secret shared key. The cryptographic strength of HMAC depends on the properties of the underlying hash function.

HRESULT: An integer value that indicates the result or status of an operation. A particular HRESULT can have different meanings depending on the protocol using it. See [MS-ERREF] section 2.1 and specific protocol documents for further details.

interface: A group of related function prototypes in a specific order, analogous to a C++ virtual interface. Multiple objects, of different object class, may implement the same interface. A derived interface may be created by adding methods after the end of an existing interface. In the Distributed Component Object Model (DCOM), all interfaces initially derive from IUnknown.

leaf license: A **license** that specifies rules that augment or restrict the rules in a **root license**. A **leaf license** can be more or less restrictive than a **root license**.

license: A data structure that contains, but is not limited to, policies and an encrypted content key. WMDRM: Network Devices Protocol has four types of licenses: **standard licenses**, **root licenses**, **base licenses**, and **leaf licenses**.

license retrieval: Every time a receiver requests content from a transmitter, the transmitter delivers to the receiver a **standard license**, a **root license** when using license chains, or a **base license** when using license derivation data. This procedure is referred to as license retrieval.

license update: Every time a receiver requests content and does not require a new root or **base license**, this procedure is invoked. This is the procedure in which a transmitter delivers a **leaf license** to the receiver to update a license restriction.

little-endian: Multiple-byte values that are byte-ordered with the least significant byte stored in the memory location with the lowest address.

near: An expression of the relationship between transmitters and receivers. A transmitter and receiver are **near** one another if **proximity detection** can be executed between them.

network byte order: The order in which the bytes of a multiple-byte number are transmitted on a network, most significant byte first (in big-endian storage). This may or may not match the order in which numbers are normally stored in memory for a particular processor.

nonce: A number that is used only once. This is typically implemented as a random number large enough that the probability of number reuse is extremely small. A nonce is used in authentication protocols to prevent replay attacks. For more information, see [RFC2617].

OpenCable Unidirectional Receiver (OCUR): A WMDRM: Network Devices Protocol transmitter that is manufactured in accordance with the specification for OpenCable Unidirectional Receiver Host Device issued by CableLabs [OCDRIP] and is paired with a CableCARD.

packet identifier (PID): The encoding of the identity is dependent on the type of packet being identified.

policy: (1) A set of rules that governs all interactions with an object such as a document or item.

(2) The set of rules that govern the interaction between a subject and an object or resource.

private key: One of a pair of keys used in public-key cryptography. The private key is kept secret and is used to decrypt data that has been encrypted with the corresponding public key. For an introduction to this concept, see [CRYPTO] section 1.8 and [IEEE1363] section 3.1.

Protected Broadcast Driver Architecture (PBDA): A worldwide platform for broadcast TV on the PC. This platform enables the PC-TV hardware ecosystem to integrate virtually any free or premium TV service into Windows Media Center, while satisfying the TV industry's requirements for strong content protection for pay TV.

proximity detection: The procedure in which a transmitter determines if a receiver is **near**.

public key: One of a pair of keys used in public-key cryptography. The public key is distributed freely and published as part of a digital certificate. For an introduction to this concept, see [CRYPTO] section 1.8 and [IEEE1363] section 3.1.

Public Key Cryptography Standards (PKCS): A group of Public Key Cryptography Standards published by RSA Laboratories.

Real-Time Streaming Protocol (RTSP): A protocol used for transferring real-time multimedia data (for example, audio and video) between a server and a client, as specified in [RFC2326]. It is a streaming protocol; this means that **RTSP** attempts to facilitate scenarios in which the multimedia data is being simultaneously transferred and rendered (that is, video is displayed and audio is played).

Real-Time Transport Protocol (RTP): A network transport protocol that provides end-to-end transport functions that are suitable for applications that transmit real-time data, such as audio and video, as described in [RFC3550].

receiver: A **device** that acquires **licenses** and content from transmitters (for example, a digital media receiver).

registration: The procedure in which a transmitter is able to uniquely identify a receiver.

resource: Any component that a computer can access that can read, write, and process data. This includes internal components (such as a disk drive), a service, or an application running on and managed by the cluster on a network that is used to access a file.

revalidation: The procedure in which a receiver that has successfully completed the **registration** process with a transmitter repeats the **registration** process with the same Transmitter.

revocation: The process of invalidating a certificate. For more details, see [RFC3280] section 3.3.

revocation information version (RIV): When a client application or **device** requests a license, the license **challenge** includes the client's **revocation** information structure. This structure contains an **RIV** and the revocation lists that the client supports. To determine whether the client revocation information is current, the license server compares its **RIV** to the client **RIV**.

revocation list: The list of identifiers of software or hardware components to which protected content cannot **flow**. Different content protection systems typically have different formats for representing **revocation lists**.

revocation list update: The procedure in which a receiver requests an updated revocation list from a transmitter.

root certificate: A self-signed certificate that identifies the **public key** of a root **certification authority (CA)** and has been trusted to terminate a **certificate chain**.

root license: A license whose rules can be further restricted or augmented by terms in a **leaf license**. A **root license** can be more or less restrictive than a **leaf license**.

round-trip time (RTT): The time that it takes a packet to be sent to a remote partner and for that partner's acknowledgment to arrive at the original sender. This is a measurement of latency between partners.

secret key: A symmetric encryption key shared by two entities, such as between a user and the domain controller (DC), with a long lifetime. A password is a common example of a secret key. When used in a context that implies Kerberos only, a principal's secret key.

service: A process or agent that is available on the network, offering resources or services for clients. Examples of services include file servers, web servers, and so on.

service or UPnP service: A set of rules that is required to be published by the device and advertised when the device is turned on all the available control points.

Session Description Protocol (SDP): A protocol that is used for session announcement, session invitation, and other forms of multimedia session initiation. For more information see [MS-SDP] and [RFC3264].

Simple Mail Transfer Protocol (SMTP): A member of the TCP/IP suite of protocols that is used to transport Internet messages, as described in [RFC5321].

Simple Service Discovery Protocol (SSDP): Used to detect **UPnP** devices on a network. **SSDP** is maintained by the Internet Engineering Task Force (IETF).

standard license: A stand-alone, self-contained license. A **standard license** is not updated by a leaf license.

symmetric key: A **secret key** used with a cryptographic symmetric algorithm. The key needs to be known to all communicating parties. For an introduction to this concept, see [CRYPTO] section 1.5.

Time-To-Live (TTL): The time duration for which a Server Object is available.

topology: The structure of the connections between members.

transmitter: A **device** that issues **policy** and transfers content to a receiver. An example of a transmitter is a digital media server.

Uniform Resource Locator (URL): A string of characters in a standardized format that identifies a document or resource on the World Wide Web. The format is as specified in [RFC1738].

Universal Plug and Play (UPnP): A set of computer network protocols, published by the UPnP Forum [UPnP], that allow devices to connect seamlessly and that simplify the implementation of networks in home (data sharing, communications, and entertainment) and corporate environments. UPnP achieves this by defining and publishing UPnP device control protocols built upon open, Internet-based communication standards.

User Datagram Protocol (UDP): The connectionless protocol within TCP/IP that corresponds to the transport layer in the ISO/OSI reference model.

XML: The Extensible Markup Language, as described in [XML1.0].

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

1.2 References

Links to a document in the Microsoft Open Specifications library point to the correct section in the most recently published version of the referenced document. However, because individual documents in the library are not updated at the same time, the section numbers in the documents may not match. You can confirm the correct section numbering by checking the Errata.

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.

[ASF] Microsoft Corporation, "Advanced Systems Format Specification", December 2004, http://download.microsoft.com/download/7/9/0/790fecaa-f64a-4a5e-a430-0bccdab3f1b4/ASF_Specification.doc

[CR-WMDRM] Microsoft Corporation, "Compliance and Robustness Rules for Windows Media DRM", https://www.microsoft.com/playready/licensing/compliance/

Note PlayReady replaced Windows Media DRM. Zip archive available.

[FIPS180-2] National Institute of Standards and Technology, "Secure Hash Standard", FIPS PUB 180-2, August 2002, http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf

[FIPS197] FIPS PUBS, "Advanced Encryption Standard (AES)", FIPS PUB 197, November 2001, https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf

[ISO/IEC-13818-1] International Organization for Standardization, "Information Technology -- Generic Coding of Moving Pictures and Associated Audio Information: Systems", ISO/IEC 13818-1:2007, http://www.iso.org/iso/catalogue_detail?csnumber=44169%20

Note There is a charge to download the specification.

[MS-DTYP] Microsoft Corporation, "Windows Data Types".

[MTP] Microsoft Corporation, "Media Transfer Protocol Enhanced", September 2006, https://download.microsoft.com/download/f/1/3/f13df1a5-6ce4-4907-86a0-6ce5c3560639/MTP_Enhanced.pdf

[OCDRIP] Cable Television Laboratories, Inc., "OpenCable Digital Receiver Interface Protocol Specification OC-SP-DRI-Pre-D02-051024", September 2010, https://apps.cablelabs.com/specification/opencable-digital-receiver-interface-protocol

[OMAC] Iwata, T., and Kurosawa, K., "OMAC: One-Key CBC MAC - Addendum", March 2003, http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/omac/omac-ad.pdf

[PKCS1] RSA Laboratories, "PKCS #1: RSA Cryptography Standard", PKCS #1, Version 2.1, June 2002, http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/pkcs-rsa-cryptography-standard.htm

[RFC2045] Freed, N., and Borenstein, N., "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, November 1996, http://www.rfc-editor.org/rfc/rfc2045.txt

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

[RFC2234] Crocker, D. and Overell, P., "Augmented BNF for Syntax Specifications: ABNF", RFC 2234, November 1997, http://www.ietf.org/rfc/rfc2234.txt

[RFC2250] Hoffman, D., Fernando, G., Goyal, V., and Civanlar, M., "RTP Payload Format for MPEG1/MPEG2 Video", RFC 2250, January 1998, http://www.ietf.org/rfc/rfc2250.txt

[RFC2326] Schulzrinne, H., Rao, A., and Lanphier, R., "Real Time Streaming Protocol (RTSP)", RFC 2326, April 1998, http://www.ietf.org/rfc/rfc2326.txt

[RFC2327] Handley, M. and Jacobson, V., "SDP: Session Description Protocol", RFC 2327, April 1998, http://www.ietf.org/rfc/rfc2327.txt

[RFC2397] Masinter, L., "The 'data' URL Scheme", RFC 2397, August 1998, http://www.rfc-editor.org/rfc/rfc2397.txt

[RFC2616] Fielding, R., Gettys, J., Mogul, J., et al., "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999, https://www.rfc-editor.org/info/rfc2616

[RFC2732] Hinden, R., Carpenter, B., and Masinter, L., "Format for Literal IPv6 Addresses in URL's", RFC 2732, December 1999, http://www.ietf.org/rfc/rfc2732.txt

[RFC2821] Klensin, J., "Simple Mail Transfer Protocol", RFC 2821, April 2001, http://www.ietf.org/rfc/rfc2821.txt

[RFC3548] Josefsson, S., Ed., "The Base16, Base32, and Base64 Data Encodings", RFC 3548, July 2003, http://www.rfc-editor.org/rfc/rfc3548.txt

[RFC3550] Schulzrinne, H., Casner, S., Frederick, R., and Jacobson, V., "RTP: A Transport Protocol for Real-Time Applications", STD 64, RFC 3550, July 2003, http://www.ietf.org/rfc/rfc3550.txt

[RFC3555] Casner, S., and Hoschka, P., "MIME Type Registration of RTP Payload Formats", RFC 3555, July 2003, http://www.ietf.org/rfc/rfc3555.txt

[RFC4567] Arkko, J., Lindholm, F., Naslund, M., et al., "Key Management Extensions for Session Description Protocol (SDP) and Real Time Streaming Protocol (RTSP)", RFC 4567, July 2006, http://www.rfc-editor.org/rfc/rfc4567.txt

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006, http://www.rfc-editor.org/rfc/rfc4648.txt

[UPNPARCH1.1] UPnP Forum, "UPnP Device Architecture 1.1", October 2008, http://www.upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.1.pdf

[UPNPCDS1] UPnP Forum, "ContentDirectory:1 Service Template Version 1.01", June 2002, http://www.upnp.org/standardizeddcps/documents/ContentDirectory1.0.pdf

[UPNPCNMGR] UPnP Forum, "ConnectionManager:1 Service Template Versions 1.01", June 2002, http://www.upnp.org/standardizeddcps/documents/ConnectionManager1.0.pdf

[WMRTP] Microsoft Corporation, "RTP Payload Format for Windows Media Audio and Video", http://download.microsoft.com/download/5/5/a/55a7b886-b742-4613-8ea8-d8b8b5c27bbc/RTPPayloadFormat_for_WMAandWMV_v1.doc

[XMR] Microsoft Corporation, "License Agreement for Import Specifications for Windows Media Format 11 Software Development Kit", http://download.microsoft.com/download/f/8/e/f8e42daa-1fd2-4be6-869b-875c3f24e277/WMFSDK%20Import%20Specifications%20License%20_(SAMPLE).pdf

Note There is a charge to download the specification.

1.2.2 Informative References

[MS-ERREF] Microsoft Corporation, "Windows Error Codes".

1.3 Protocol Overview (Synopsis)

The WMDRM: Network Devices Protocol defines the following procedures:

Authorization: The procedure for granting a **receiver** access to content from a transmitter. This procedure is required before a receiver can perform **registration** and access protected content.

Registration and Revalidation: Allows the transmitter to uniquely identify a receiver that is located nearby. Revalidation is the procedure for automatically reregistering a receiver with the transmitter after a certain amount of time has elapsed. Registration and **revalidation** are very

similar, so for the purposes of this document, they will be grouped together in one category. Differences will be pointed out as necessary.

Proximity Detection: Measures the latency between the transmitter and a receiver. If the latency is short enough, then the receiver is considered to be **near** the transmitter. The transmitter will not send protected content to a receiver unless it is near.

License Retrieval: The procedure for sending a **license** to a registered receiver when it requests content from a transmitter.

Data Transfer: Performed between a transmitter and receiver after a license has been sent by the transmitter and enforced by the receiver. The content is encrypted and delivered using HTTP or RTP.

License Management: For this procedure, the transmitter sends an updated license, to the receiver to change the receiver's rights to a **resource**.

Revocation List Update: This procedure, which is only permitted once the registration procedure has successfully completed, occurs when the receiver asks the transmitter for an updated **revocation list**. This procedure can occur at any time.

Authorization is performed after a receiver is discovered by a transmitter, and registration/revalidation is performed every 48 hours, but the **license retrieval** and data transfer procedures occur every time a **file** is requested and played.

All of the procedures rely on messages, or sets of parameters, that are passed back and forth between a transmitter and a receiver. Messages are mapped to the HTTP, **Real-Time Streaming Protocol (RTSP)**, and **UPnP** protocols, except during **proximity detection**, which uses **User Datagram Protocol (UDP)**. The following conceptual sections describe these procedures in further detail.

1.4 Relationship to Other Protocols

Relationship mappings to other protocols are called out in section 2.1.1.

1.5 Prerequisites/Preconditions

Common cryptographic requirements are expected to be implemented as specified in section 2.4.

1.6 Applicability Statement

None.

1.7 Versioning and Capability Negotiation

WMDRM: Network Devices Protocol does not perform version or capability negotiation. The client submits requests to a server that understands the maximum protocol version used by the client.

1.8 Vendor-Extensible Fields

This protocol uses **HRESULT** values as defined in [MS-ERREF] section 2.1. Vendors can define their own HRESULT values, provided they set the **C** bit (0x20000000) for each vendor-defined value, indicating the value is a customer code.

This protocol uses Win32 error codes as defined in [MS-ERREF] section 2.2. Vendors SHOULD reuse those values with their indicated meanings. Choosing any other value runs the risk of a collision in the future.

This protocol uses NTSTATUS values as defined in [MS-ERREF] section 2.3. Vendors are free to choose their own values for this field, as long as the **C** bit (0x20000000) is set, indicating it is a customer code.

This protocol MAY be extended using the Media Transfer Protocol (MTP) Vendor Extension **Identification** field as specified in section 2.2.2.

1.9 Standards Assignments

None.

2 Messages

2.1 Transport

This protocol can be implemented on top of the following transport mechanisms:

- Hyptertext Transport Protocol (HTTP)
- Universal Plug-andPlay (UPnP)
- Real-time Streaming Protocol (RTSP) Real-time Transport Protocol (RTP)
- OpenCable Digital Receiver Interface Protocol (Open Cable DRI)
- Media Transport Protocol (MTP)
- Protected Broadcast Driver Architecture (PBDA)

Rules and recommendations for each transport are discussed in section 2.1.1.

2.1.1 Protocol Mappings

The procedures in the WMDRM: Network Devices Protocol can be mapped to various protocols. The following sections describe which control protocols can be mapped to which procedures.

2.1.1.1 UPnP Mapping

The following sections specify how the authorization and **registration** procedures are mapped to the **UPnP** protocol.

2.1.1.1.1 Authorization

When authorization is implemented using **UPnP**, the rules specified in section 3.1.1.2 MUST be followed.

Authorization uses the X_MS_MediaReceiverRegistrar:1 **UPnP service**, which is defined in section 3.4. An example of the UPnP Authorization procedure is detailed in section 4.1

Receivers MUST use **Simple Service Discovery Protocol (SSDP)** to announce their presence on the network. This is done by transmitting **ssdp:alive** messages, as specified in the UPnP Device Architecture specification [UPNPARCH1.1] section 1.2.2. These messages allow the transmitter to discover a new Receiver.

Whenever a **Receiver** is authorized or unauthorized by a user, a state variable in the X_MS_MediaReceiverRegistrar:1 UPnP service is set by the transmitter. A UPnP message is sent when the state variable is changed. The message notifies all receivers that a receiver has either been authorized or unauthorized. Each receiver MUST then check with the transmitter to determine if it is the **devices** that has been authorized or unauthorized. A receiver uses the X_MS_MediaReceiverRegistrar:1 UPnP service to check with the transmitter.

2.1.1.1.1.1 Receivers

Receivers MUST transmit **ssdp:alive** messages, according to the rules specified in the **UPnP** Device Architecture specification [UPNPARCH1.1].

Receivers MUST respond to the change in the **AuthorizationGrantedUpdateID** state variable in the X_MS_MediaReceiverRegistrar:1 UPnP service (see section 3.4) by invoking the **IsAuthorized** action to determine whether it is the receiver that was authorized.

The **IsAuthorized** action has two arguments: *DeviceID* and *Result*.

The *DeviceID* argument is sent to the transmitter for verification and the transmitter sends the *Result* argument back to the receiver.

The *Result* argument is an integer. Its value is set to 1 if the **device** is currently authorized and to 0 if the device is not authorized.

2.1.1.1.1.2 Transmitters

The **transmitter** MUST perform authorization through the following steps.

1. The transmitter MUST use **SSDP** to discover a receiver when it becomes available on the network.
2. When the transmitter discovers a receiver that supports WMDRM: Network Devices Protocol, the transmitter MUST follow the rules defined in section 3.1.1.2, to determine whether it MUST prompt the user to approve the receiver to access **content** and metadata.
3. **UPnP** devices can be removed from and restored to the network frequently. The transmitter SHOULD NOT prompt the user to approve a **device** every time that device connects to the network. The transmitter SHOULD maintain a record of currently connected UPnP devices and check this record before prompting.
4. If the user approves the receiver, the transmitter MUST set the **AuthorizationGrantedUpdateID** state variable. If the user does not approve the device, the transmitter MUST set the **AuthorizationDeniedUpdateID** state variable.
5. For each **IsAuthorized** action invoked by each receiver, the transmitter MUST send back a response notifying the receiver whether it is authorized or unauthorized, depending on which state variable was set.

2.1.1.1.2 Registration

When **registration** and **revalidation** are implemented using **UPnP**, the rules specified in section 3.1.1.3 MUST be followed.

Registration and revalidation MUST use the X_MS_MediaReceiverRegistrar:1 UPnP service, which is defined in section 6.2.

2.1.1.1.2.1 Receivers

The receiver MUST encode the binary **registration** request message to Base64 format [RFC3548]. The Base64-encoded data block MUST then be sent to the transmitter through the **UPnP** **RegisterDevice** action of the X_MS_MediaReceiverRegistrar:1 **service**.

After the receiver retrieves the Base64-encoded data block in the **RegisterDevice** response from the transmitter, it MUST decode the data block from Base64 to binary [RFC3548]. The resulting **Binary large object (BLOB)** is the registration response message.

The registration request and registration response messages are defined in sections 2.2.1.1 and 2.2.1.2, respectively.

2.1.1.1.2.2 Transmitters

When the **RegisterDevice** action of the transmitter's X_MS_MediaReceiverRegistrar:1 service is invoked, it MUST decode the Base64-encoded data block sent by the receiver from Base64 to binary.

The resulting **BLOB** is the **registration** request message, and it MUST be used as specified in section 2.1.1.1.2. For information on Base64 see [RFC3548]

The transmitter MUST encode the binary registration response message to Base64 format [RFC3548]. The Base64-encoded data block MUST then be sent to the receiver as the response to the **RegisterDevice** action.

The registration request and registration response messages are defined in sections 2.2.1.1 and 2.2.1.2, respectively.

2.1.1.1.3 Error Codes

In **UPnP**, the WMDRM: Network Devices Protocol error codes MUST be mapped to the range of 850-899, where WMDRM: Network Devices Protocol error code 100 maps to UPnP error code 850, 101 maps to 851, and so on. For example, WMDRM: Network Devices Protocol error code 107 (Must Register) maps to UPnP error code 857.

2.1.1.2 HTTP Mappings

The following sections show how the **license retrieval**, data transfer, **license update**, and **revocation list update** procedures are mapped to the HTTP protocol [RFC2616].

2.1.1.2.1 Framing Headers

This section defines a mechanism that allows the **license update** procedure and data transfer to use the same HTTP connection. The mechanism is called Framing Headers and it allows the license response messages (used by the license update procedure) to be interleaved with data blocks of protected content.

The HTTP Framing Headers mechanism is used only when delivering content that is not encapsulated in **ASF**.

When the Framing Headers mechanism is used for data transfer, the entity-body of the HTTP response MUST consist of a series of frames. Each frame starts with a framing header and is followed by either control data or content data.

The framing header is four bytes in length. The first byte MUST always be an **ASCII** dollar sign (0x24) followed by a one-byte frame type identifier, and then followed by a 16-bit **length** field. The **length** field specifies the number of bytes of data in the frame, not including the size of the framing header itself. The length field MUST be represented in **network byte order**.

A control frame uses an ASCII "c" character (0x63) as its type identifier. This frame contains a WMDRM: Network Devices Protocol message. Currently, the only message delivered in a control frame is a license response message, which carries a **leaf license**.

A data frame uses an ASCII "d" character (0x64) as its type identifier. This frame contains data belonging to the protected content being transferred. The transmitter can choose the size of data frames arbitrarily, but the size of a data frame MUST NOT exceed 65535 bytes.

The HTTP response MUST include the "Content-Type" header, and the content type specified in that header MUST be "application/vnd.ms-wmdrm-data-transfer". This indicates the use of the HTTP Framing Header mechanism.

The "application/vnd.ms-wmdrm-data-transfer" content type has one mandatory parameter, called *media*. The value of the *media* parameter MUST be set to the original content type for the content being delivered, including all of the parameters of the original content type. The value of the *media* parameter MUST be enclosed in double quotes.

The following is an example of the "Content-Type" header when the original content type is "video/mpeg":

```
Content-Type: application/vnd.ms-wmdrm-data-transfer; media="video/mpeg"
```

2.1.1.2.2 License Retrieval

When **license retrieval** is implemented using HTTP, the rules that are specified in section 2.1.1.3.1 MUST be followed.

When transferring content that is encapsulated in **ASF** using HTTP, a **standard license** is delivered to the receiver; however, for any other content type, a **root license** is sent to the receiver. The **leaf licenses** that chain to the root license are delivered in control frames during the license update procedure, as specified in section 2.1.1.2.1.

2.1.1.2.2.1 Receivers

The receiver MUST include the HTTP header "Supported: com.microsoft.wmdrm-nd" in the HTTP request message to indicate to the transmitter that it supports WMDRM: Network Devices Protocol.

The receiver MUST begin the **license retrieval** procedure by sending a POST request to the **Uniform Resource Locator (URL)** that the receiver intends to play. The entity-body of the POST request MUST contain a license request message.

The POST request MUST include a "Content-Type: application/vnd.ms-wmdrm-license-request" header to indicate that the body of the POST request contains a license request message.

Unlike the **registration** request and registration response messages, the license request and license response messages MUST NOT be Base64 encoded or decoded [RFC3548].

When the receiver receives the response to the POST request, it MUST verify that the response indicates that the request succeeded, and that the response includes the "Content-Type: application/vnd.ms-wmdrm-license-response" header. This header indicates that the entity-body of the response contains a license response message.

If the "WMDRM-ND" HTTP header is present in the response to the POST request, the receiver MUST remember the value of the *SessionId* parameter in that header. The syntax of the "WMDRM-ND" header is specified in section 2.2. The value of the *SessionId* parameter MUST be sent back by the receiver when requesting data during the data transfer procedure (defined in section 2.1.1.2.5).

The value of the *SessionId* parameter used in the "WMDRM-ND" HTTP header is NOT related to the *Session ID* parameter used in the registration or **proximity detection** procedures.

2.1.1.2.2.2 Transmitters

When the transmitter receives a POST request, it MUST verify that the request includes the header "Content-Type: application/vnd.ms-wmdrm-license-request". The presence of this header indicates that the entity-body of the POST request contains a license request message.

The transmitter MUST process the license request, complying with the rules in section 3.1.1.5. Unless there is an error, the transmitter MUST include a license response message in the entity-body of the response to the POST request.

The response to the POST request MUST include a "Content-Type: application/vnd.ms-wmdrm-license-response" header to indicate that the body of the response contains a License Response message.

Unlike the registration request and registration response messages, the license request and license response messages MUST NOT be Base64 encoded or decoded [RFC3548].

The transmitter MUST include the HTTP header "Supported: com.microsoft.wmdrm-nd" in the HTTP response message to indicate to the receiver that it supports WMDRM: Network Devices Protocol.

The transmitter MUST include a "WMDRM-ND" HTTP header in the response to the POST request, unless the request failed. The syntax of the header is specified in **Augmented Backus-Naur Form (ABNF)** syntax [RFC2234] as follows:

```
"WMDRM-ND:" "SessionId=" <"> session-id <">
session-id = 1*( ALPHA | DIGIT | safe )
```

The *session-id* parameter is an arbitrary string that MUST be echoed back by the receiver when requesting data during the data transfer procedure.

2.1.1.2.3 License Update

When the license update procedure is implemented using HTTP, the rules specified in section 3.1.1.5 MUST be followed.

If the license obtained through the **license retrieval** procedure is a **root license**, the receiver MUST also receive a **leaf license** before it can decrypt the content. Transmitters MUST use the license update mechanism to deliver the leaf license. Transmitters MUST send a license response message with a leaf license before they start delivering protected content in response to a **GET** request from the receiver.

The license response message for the license update MUST be delivered using the Framing Header mechanism defined in section 2.1.1.2.1.

2.1.1.2.3.1 Receivers

When the receiver receives a control frame [$c] as defined in section 2.1.1.2.1. The receiver MUST verify that the control frame contains a license response message. Other message types MAY be ignored.

The receiver MUST process the license response, complying with the rules defined in section 3.2.5.5.

The license response message in a control frame MUST NOT be Base64 encoded [RFC3548].

2.1.1.2.3.2 Transmitters

The transmitter MUST begin the license update procedure by sending a control frame [$c], as defined here to the receiver. The control frame MUST be sent in the entity-body of the response to the GET request that the receiver sent to request data transfer. The control frame MUST contain a license response message with a **leaf license**.

The license response message in a control frame MUST NOT be Base64 encoded [RFC3548].

When the Framing Header mechanism is used, the license delivered as part of the **license retrieval** procedure MUST always be a **root license**. The transmitter MUST deliver a leaf license before it starts delivering the content. As a consequence, if a data transfer request succeeded, the **GET** response MUST be composed of the following frames:

1. Control frame [$c] carrying a license response message with a leaf license.
2. One or more data frames [$d].
3. If a key or a **policy (1)** change during the transmission of the content requires an additional license update, then steps 1 and 2 are repeated with new data for each license update.

2.1.1.2.4 Revocation List Update

When **revocation list update** is implemented using HTTP, the rules specified here MUST be followed.

Revocation list updates can occur at any time and do not change the transmitter state.

2.1.1.2.4.1 Receivers

The receiver MUST begin the **revocation list update** procedure by sending a POST request to the **URL** that the receiver intends to play or is currently playing. The entity-body of the POST request MUST contain a **revocation list** request message.

The POST request MUST include a "Content-Type: application/vnd.ms-wmdrm-extended-operation-request" header to indicate that the body of the POST request contains a revocation list request message.

Unlike the **registration** request and registration response messages, the revocation list request and revocation list response messages MUST NOT be encoded or decoded in Base64 [RFC3548].

When the receiver receives the response to the POST request, it MUST verify that the response indicates that the request succeeded. The receiver MUST also verify that the response includes the "Content-Type: application/vnd.ms-wmdrm-extended-operation-response" header. The presence of this header indicates that the entity-body of the response contains a revocation list response message.

2.1.1.2.4.2 Transmitters

When the transmitter receives a POST request, it MUST verify that the request includes the "Content-Type: application/vnd.ms-wmdrm-extended-operation-request" header. The presence of this header indicates that the entity body of the POST request contains a revocation list request message.

The transmitter MUST process the revocation list request, complying with the rules defined here. Unless there is an error, the transmitter MUST include a revocation list response message in the entity-body of the response to the POST request.

The response to the POST request MUST include a "Content-Type: application/vnd.ms-wmdrm-extended-operation-response" header. This indicates that the body of the response contains a revocation list response message.

Unlike the **registration** request and registration response messages, the revocation list request and revocation list response messages MUST NOT be encoded or decoded in Base64 [RFC3548].

2.1.1.2.5 Data Transfer

When data transfer is implemented using HTTP, the rules that are specified in section 3.1.1.6 MUST be followed.

2.1.1.2.5.1 Common Data Transfer Rules

The receiver MUST begin the data transfer procedure by sending a **GET** request to the **URL** that the receiver intends to play.

If the transmitter's response to the **license retrieval** request contained the "WMDRM-ND" HTTP header, the receiver MUST include this header in the **GET** request. The syntax of the "WMDRM-ND" HTTP header is defined here.

The "WMDRM-ND" HTTP header can be used more than once to replay the content. In other words, if a previous data transfer has been concluded and the receiver wants to issue a new **GET** request, it can use the "WMDRM-ND" header that it used in the previous request. However, sessions typically

expire five minutes after the data transfer is concluded or interrupted. Therefore, receivers MUST handle expired sessions by repeating the license retrieval procedure if the server responds with error code 0x006E (Invalid Session).

Unless there is an error, the entity-body in the transmitter's response MUST contain the protected content. If the protected content is encapsulated as Windows Media-based content, the rules in section 2.1.1.2.5.4 MUST be followed.

For MPEG-2 transport stream content, the rules in section 2.1.1.2.5.2 MUST be followed. For other content types (not MPEG-2 and not encapsulated in **ASF**) the rules in section 2.1.1.2.5.6 MUST be followed.

The entity-body of the response to the **GET** request MUST use the frame header mechanism that is defined in section 2.1.1.2.1 (except for content encapsulated in ASF). Also the transmitter MUST deliver a **leaf license** through the license update procedure before the content is transferred.

2.1.1.2.5.2 MPEG-2 Transport Streams

Content that is encapsulated in MPEG-2 TS packets MUST be encrypted as specified here.

MPEG-2 TS packets are sent over HTTP using the Framing Headers mechanism as defined in section 2.1.1.2.1. The MPEG-2 TS packets MAY be split arbitrarily and do not need to be aligned with the Framing Header of a data frame.

2.1.1.2.5.3 MPEG-2 Elementary Streams

MPEG-2 ES content cannot be delivered over HTTP.

2.1.1.2.5.4 Windows Media-based Content

The transmitter MUST encrypt Windows Media-based content in a specific way. The Windows Media **file** MUST be decrypted by a receiver that implements the rules for decrypting **ASF**-encapsulated content specified here. The samples MUST be encrypted with the **content encryption key** that was included in the **standard license** sent to the receiver in the **license retrieval** procedure.

2.1.1.2.5.5 File Contents

The **Content-Type header** field with the value "video/x-ms-asf" indicates that the body of the response contains data that is an **ASF file**. Other values, such as "video/x-ms-wmv" and "audio/x-ms-wma" are also possible.

2.1.1.2.5.6 Other Content Types

Content that is not encapsulated in MPEG-2 TS packets, is not in MPEG-2 elementary streams, and is not Windows Media-based, MUST be encrypted using the link encryption mode that is specified in section 3.1.5.2.11.

The encrypted data segments MUST be sent over HTTP using the Framing Headers mechanism as defined in section 2.1.1.2.1.

The link encryption procedure MUST use **Advanced Encryption Standard (AES)** [FIPS197] in counter mode. The procedure also MUST be conducted as defined in section 3.1.5.2.1. The data segment corresponds to the content data included within a single framing header.

A data segment descriptor MUST be included at the beginning of each frame. Both the key ID and the AES128 initialization vector extensions that are defined in section 3.1.5.2.3 MUST be included in the data segment descriptor. Transmitters MUST use a block ID of 0 for each frame. However, receivers MUST be able to handle different block ID values for future extensibility.

The data segment descriptor MUST be inserted immediately after the framing header, yet the data segment descriptor itself MUST NOT be encrypted. Only the content data that follows it can be encrypted.

Since a frame can only carry a maximum of 65535 bytes, the combined size of the data segment and the data segment descriptor MUST NOT exceed this limit.

The data segment descriptor MUST be added to the beginning of each frame and is present regardless of whether the content in a particular frame has been encrypted. Transmitters MUST encrypt all content; however, receivers MUST be able to handle unencrypted content for future extensibility.

Because the entity-body of the response to the GET request uses the HTTP framing header mechanism (section 2.1.1.2.1), control frames are interleaved with data frames, allowing the transmitter to perform the license update procedure as needed.

2.1.1.2.6 Error Codes

Specific Windows Media DRM-ND error codes are indicated by the "WMDRM-ND-Status" HTTP header. This header takes the form of a three-digit status code that is followed by a space character and an error description string. The error description string MUST be enclosed in double quotes.

When a transmitter indicates an error code using the "WMDRM-ND-Status" HTTP header, the status code in the HTTP response itself MUST also indicate that the HTTP request failed. The transmitter uses the HTTP status code "500 Internal Server Error" in this case.

2.1.1.3 RTSP Mappings

When Windows Media DRM-ND is used with **RTSP**, the feature tag "com.microsoft.wmdrm-nd" is used to identify support for Windows Media DRM-ND. This feature tag can be included on the RTSP "Require" header in any RTSP request sent by the receiver or by the transmitter. It MUST be included in the "Supported" header in RTSP requests and responses.

If the transmitter or Receiver receives an RTSP request that includes the "Require" header, it MUST recognize "com.microsoft.wmdrm-nd" as one of the features that it supports.

The following sections specify how the **license retrieval**, data transfer, license update, and **revocation list update** procedures are mapped to the RTSP protocol [RFC2326].

2.1.1.3.1 License Retrieval

When **license retrieval** is implemented using **RTSP**, the rules that are specified here MUST be followed.

The transmitter MUST send a **root license** to the receiver in response to a valid license request message. The **leaf license** MUST be sent as part of a license update before content is transferred.

2.1.1.3.1.1 Receivers

The receiver MUST include the **RTSP** header Supported: "com.microsoft.wmdrm-nd" in its RTSP request messages to indicate to the transmitter that it supports WMDRM: Network Devices Protocol .

The receiver MUST begin the **license retrieval** procedure by sending a DESCRIBE request to the **URL** that the receiver intends to play. The entity-body of the DESCRIBE request MUST contain a license request message.

The DESCRIBE request MUST include a "Content-Type: application/vnd.ms-wmdrm-license-request" header to indicate that the body of the DESCRIBE request contains a license request message.

The DESCRIBE request SHOULD include a "Require: com.microsoft.wmdrm-nd" header to indicate that the receiver expects the transmitter to be compliant with WMDRM: Network Devices Protocol .

Unlike the **registration** request message, the license request message MUST NOT be Base64 encoded [RFC3548].

When the receiver receives the response to the DESCRIBE request, it MUST verify that the response indicates that the request succeeded and that the **Session Description Protocol (SDP)** [RFC2327] description contained in the response includes the "a=key-mgmt:wmdrm-nd" attribute.

The receiver MUST verify that the "a=key-mgmt:wmdrm-nd" SDP attribute contains a URL using the data URL scheme that is defined in RFC 2397 [RFC2397], and that the MIME type of the "data" URL is "application/vnd.ms-wmdrm-license-response". This MIME type indicates that the data URL contains a license response message.

Typically, the license response is encoded in Base64 format [RFC3548]. The data URL itself will indicate if Base64 encoding has been used. In that case, the receiver MUST apply Base64 decoding when extracting the license response message from the data URL [RFC3548].

2.1.1.3.1.2 Transmitters

When the transmitter receives a DESCRIBE request, it MUST verify that the request includes the "Content-Type: application/vnd.ms-wmdrm-license-request" header. The presence of this header indicates that the entity-body of the DESCRIBE request contains a license request message.

The transmitter MUST process the license request, complying with the rules defined in section 2.2.1.4.1. Unless there is an error, the transmitter MUST reply with a **Session Description Protocol (SDP)** description that includes the license response message.

The SDP description returned by the transmitter MUST include the license response message encoded in a "data" **URL** according to the data URL scheme that is defined in RFC 2397 [RFC2397]. The data that is contained in the data URL MUST be Base64 encoded [RFC3548] and the MIME type of the data URL MUST be set to "application/vnd.ms-wmdrm-license-response".

The data URL MUST be inserted at the SDP session level by using an "a=key-mgmt" attribute. The attribute MUST comply with the following syntax:

```
a=key-mgmt:wmdrm-nd url
```

The *url* parameter is the data URL described in the previous paragraphs.

The "a=key-mgmt" attribute is described in [RFC4567].

2.1.1.3.2 License Updates

When the license update procedure is implemented using **RTSP**, the following rules MUST be followed.

The license response message in the **SDP** description contains only a **root license**. Before the receiver can decrypt the content, it MUST also receive a **leaf license**. As a result, transmitters MUST send an ANNOUNCE request with a leaf license before they start delivering **RTP** packets in response to a PLAY request from the receiver.

2.1.1.3.2.1 Receivers

When the receiver receives an ANNOUNCE request, it MUST verify that the request includes the "Content-Type: application/vnd.ms-wmdrm-license-response" header. The presence of this header indicates that the entity-body of the ANNOUNCE request contains a license response message.

The receiver MUST process the license response, complying with the following rules. Unless there is an error, the receiver MUST reply to the ANNOUNCE request with **RTSP** status code 200.

The license response message in an ANNOUNCE request MUST NOT be Base64 encoded [RFC3548].

Receivers MUST be prepared to handle the case when an ANNOUNCE request is received after the time when the key ID in the **leaf license** is required, because network congestion or Transmitter overload can delay the delivery of the ANNOUNCE request.

2.1.1.3.2.2 Transmitters

The transmitter MUST begin the license update procedure by sending an ANNOUNCE request to the **URL** that the receiver intends to play, or is currently playing. The entity-body of the ANNOUNCE request MUST contain a license response message with a **leaf license**.

The ANNOUNCE request MUST include a "Content-Type: application/vnd.ms-wmdrm-license-response" header to indicate that the body of the ANNOUNCE request contains a license response message.

The license response message in an ANNOUNCE request MUST NOT be Base64 encoded [RFC3548].

2.1.1.3.3 Revocation List Update

When the **revocation list update** procedure is implemented using **RTSP**, the rules that are specified here MUST be followed.

Revocation list updates can occur at any time and do not change the transmitter state.

2.1.1.3.3.1 Receivers

The receiver MUST begin the **revocation list update** procedure by sending a **GET_PARAMETER** request to the **URL** that the receiver intends to play or is currently playing. The entity-body of the **GET_PARAMETER** request MUST contain a revocation list request message.

The **GET_PARAMETER** request MUST include a "Content-Type: application/vnd.ms-wmdrm-extended-operation-request" header to indicate that the body of the **GET_PARAMETER** request contains a revocation list request message.

Unlike the **registration** request and registration response messages, the revocation list request and revocation list response messages MUST NOT be encoded or decoded in Base64 [RFC3548].

When the receiver checks the response to the **GET_PARAMETER** request. The receiver MUST verify that the response indicates that the request succeeded, and that the response includes the "Content-Type: application/vnd.ms-wmdrm-extended-operation-response" header. The presence of this header indicates that the entity-body of the response contains a revocation list response message.

2.1.1.3.3.2 Transmitters

When the transmitter receives a **GET_PARAMETER** request, it MUST verify that the request includes the "Content-Type: application/vnd.ms-wmdrm-extended-operation-request" header. The presence of this header indicates that the entity-body of the **GET_PARAMETER** request contains a revocation list request message.

The only extended operation currently defined is the **revocation list request**.

The transmitter MUST process the revocation list request, complying with the rules here. Unless there is an error, the transmitter MUST include a revocation list response message in the entity-body of the response to the **GET_PARAMETER** request.

The response to the **GET_PARAMETER** request MUST include a "Content-Type: application/vnd.ms-wmdrm-extended-operation-response" header to indicate that the body of the response contains a revocation list response message.

Unlike the **registration** request and registration response messages, the revocation list request and revocation list response messages MUST NOT be encoded or decoded in Base64 [RFC3548].

2.1.1.3.4 Error Codes

Specific WMDRM: Network Devices Protocol error codes are indicated by the **RTSP** header "WMDRM-ND-Status", which takes the form of a 3-digit status code, followed by a space character and an error description string. The error description string MUST be enclosed in double-quotes.

When a transmitter or receiver indicates an error code using the "WMDRM-ND-Status" RTSP header, the status code in the RTSP response itself MUST also indicate that the RTSP request failed. The transmitter or receiver SHOULD use the RTSP status code "500 Internal Server Error" in this case.

2.1.1.3.5 Data Transfer

When data transfer is implemented using **RTSP**, the rules specified in section 3.1.1.6 MUST be followed.

2.1.1.3.5.1 MPEG-2 Transport Streams

Content that is encapsulated in MPEG-2 TS packets MUST be encrypted as specified here.

MPEG-2 TS packets MUST be encapsulated in **RTP** using the MP2T RTP payload format, which is defined in RFC 2250 [RFC2250].

2.1.1.3.5.2 MPEG-2 Elementary Streams

MPEG-2 ES content MUST be encrypted as specified here.

The **RTP** Payload Format for Windows Media Audio and Video [WMRTP] MUST be used to carry MPEG-2 ES content when protected by WMDRM: Network Devices Protocol. The mapping of MPEG-2 ES content to RTP is defined in sections 2.1.1.3.6, 2.1.1.3.7, and 2.1.1.3.8.

2.1.1.3.5.3 Windows Media-based Content

The **ASF** Sample Encryption Mode defined here MUST be used with Windows Media-based content. When operating in this mode, each media sample is encrypted separately prior to the generation of each **RTP** packet.

When RTP is carrying content protected by WMDRM: Network Devices Protocol, the RTP Payload Format for Windows Media Audio and Video [WMRTP] MUST be used, and the following values and fields MUST be set in the RTP packet:

The **Bit Field 2 Present** bit (B2P) in the Packet Info section MUST be set to 1, to indicate that a **Bit Field 2** is present.

The **Encryption** bit (E) in the Bit Field 2 of the "MAU Properties" section MUST be set to 1.

The **Extension Present** bit (X) in the "MAU Timing" section MUST be set to 1, to indicate the presence of Extension fields.

The "Encrypted Payload Boundary" extension MUST NOT be present.

A "WMDRM Initialization Vector" extension MUST be included.

The following values MUST be set:

- The **Extension Type** MUST be set to 2.
- The **Extension Length** MUST be set to 8 (meaning 8 bytes or 64 bits).
- The **Extension Data** MUST be set to the sample ID value obtained from the ASF payload extension system as defined in section 3.1.5.2.9.
- This extension MUST be included for the first payload of every MAU. If the MAU is fragmented into multiple payloads, this extension SHOULD be present only in the first payload.

A "WMDRM Key ID" extension MUST be included. The following values MUST be set:

- The **Extension Type** MUST be set to 3.
- The **Extension Length** MUST be set to 16 (meaning 16 bytes or 128 bits).
- The **Extension Data** MUST be set with the key ID value from the ASF Content Encryption object.
- This extension MUST be included for the first payload in each multiple-payload RTP packet to ensure that the receiver always knows about the current key ID even if some RTP packets are lost.

2.1.1.3.5.4 Other Content Types

Content that is not encapsulated in MPEG-2 TS packets, is not MPEG-2 ES, and is not Windows Media-based, MUST be encrypted using the link-encryption mode specified here.

The link-encryption procedure MUST use **AES** in counter mode and conducted according to the procedure defined here. The data segment corresponds to the **RTP** packet, with the exception of the RTP header.

A data segment descriptor MUST be included at the end of each RTP packet. Both the key ID and the AES128 Initialization Vector extensions defined here MUST be included in the data segment descriptor. Transmitters MUST use a block ID of 0 for each RTP packet; however, receivers MUST be able to handle different block ID values for future extensibility.

The data segment descriptor MUST be added to the end of the RTP packet and is present regardless of whether the content in a particular RTP packet has been encrypted. Transmitters MUST encrypt all content; however, receivers MUST be able to handle unencrypted content for future extensibility.

The following is an example of an RTP packet that has been encrypted.

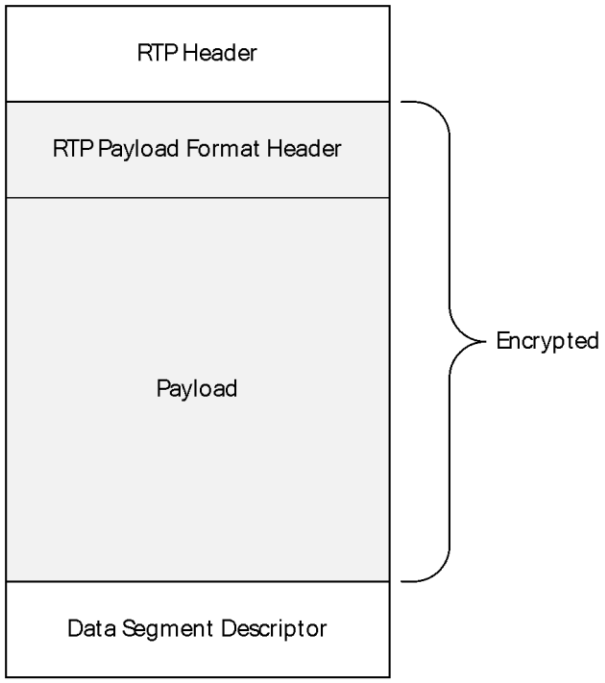


Figure 1: Encrypted RTP Packet

2.1.1.3.6 RTP Encapsulation of MPEG-2 Elementary Streams

This section defines how MPEG-2 ES content is encapsulated in the **RTP** Payload Format for Windows Media Audio and Video [WMRTP] and how the encryption parameters are communicated using **SDP**.

General RTP encapsulation rules for MPEG-2 ES Content

Media Access Unit (MAU): A Media Access Unit (MAU) for encapsulation of MPEG-2 ES content with the payload format for Windows Media Audio and Video [WMRTP] is defined as a single frame of video or an audio frame.

When MPEG video headers are present, they MUST precede the subsequent frame. This requirement is similar to the fragmentation rules outlined in Section 3.1 of [RFC2250]. Specifically:

Unlike [RFC2250], if a MAU containing video is fragmented, there is no requirement to perform the fragmentation at a slice boundary.

MAU Fragmentation: MAUs can be fragmented across multiple Real-Time Transport Protocol (RTP) packets for different reasons. The most common reasons are:

- RTP packet size restrictions.
- Differences in encryption parameters for specific portions of the MAU.

RTP Header Fields: The **RTP Header** fields MUST be interpreted according to section 3.5 of the Payload Format for Windows Media Audio and Video [WMRTP]. The following clarifications apply:

- The **Timestamp** field in the **RTP header** MUST be set to the PTS of the sample with an accuracy of 90 kHz.
- The **Payload Type (PT)** field MUST be set according to out-of-band negotiation mechanisms (for example, using SDP).

Packet Specific Info section: The Packet Specific Info section MUST be interpreted according to section 3.6.1 of the Payload Format for Windows Media Audio and Video [WMRTP].

The following clarifications apply:

- The presence of the **Send Time** field is optional.
- The presence of the **Correspondence** field is optional.
- The **Bit Field 2 Present** bit (B2P) MUST be set in case the payload contains a portion of a MAU which is encrypted, or a fragment of a MAU which is encrypted.

MAU Properties section: The MAU Properties section MUST be interpreted according to section 3.6.2 of [WMRTP]. The following clarifications apply:

- The **Sync Point** bit (S) MUST be set when the MAU contains a video I-frame or an audio frame.
- The **Discontinuity** bit (D1) MUST be set when one or more MAUs are missing. For example, when video frames were dropped by a frame dropping transmitter.
- The use of the **Droppable** bit (D2) is optional. Defining the cases in which it SHOULD be used is outside of the scope of this specification.
- The **Encryption** bit (E) MUST be set in case the payload contains a portion of a MAU that is encrypted or a fragment of a MAU that is encrypted.

MAU Timing section: The MAU Timing section MUST be interpreted according to section 3.6.3 of the RTP Payload Format for Windows Media Audio and Video specification [WMRTP]. The following clarifications apply:

- The **Decode Time** field is optional. If used, it MUST contain the DTS of the MAU.
- The **Presentation Time** field is optional.
- The **NPT** field is optional.
- The **Extension Present** bit (X) MUST be set in case the payload contains a portion of a MAU which is encrypted, or a fragment of a MAU which is encrypted.

2.1.1.3.7 Delivering encrypted MPEG-2 ES payloads

The **RTP** Payload Format for Windows Media Audio and Video [WMRTP] allows for a single MAU to be encrypted according to different encryption parameters. That includes the ability to have fragments of a single MAU that are encrypted while others may be left in the clear.

In such cases, a single RTP packet can carry multiple payloads of the same MAU, each with different encryption parameters. For details, please refer to sections 3.2 and 3.3 of the RTP Payload Format for Windows Media Audio and Video specification [WMRTP].

An MAU or a fragment of an MAU that is encrypted MUST have the following values and fields set according to the specification:

- The **Bit Field 2 Present** bit (B2P) in the Packet Info section MUST be set to 1, to indicate that a **Bit Field 2** is present.
- The **Encryption** bit (E) in the MAU Properties section MUST be set to 1, to indicate that the payload is encrypted.
- The **Extension Present** bit (X) in the "MAU Timing" section MUST be set to 1, to indicate the presence of Extension fields.

- The "Encrypted Payload Boundary" extension MUST NOT be present.
- A "WMDRM Initialization Vector" extension MUST be included.

The following values MUST be set:

- The **Extension Type** MUST be set to 2.
- The **Extension Length** MUST be set to 8 (meaning 64 bits) if the **Extension Data** field contains only a data segment ID, or 16 (meaning 128 bits) if the **Extension Data** field contains both a data segment ID and a block ID.
- The **Extension Data** MUST be set with the data segment ID value as defined in section 3.1.5.2.1 in case the initial block ID is zero. If the initial block ID is different from zero, then the **Extension Data** MUST be set to the data segment ID followed by the initial block ID.
- This extension MUST be included for each encrypted payload of a MAU.

A "WMDRM Key ID" extension MUST be included. The following values MUST be set:

- The **Extension Type** MUST be set to 3.
- The **Extension Length** MUST be set to 16 (meaning 128 bits).
- The **Extension Data** MUST be set with the key ID value from the license that corresponds to this MAU.

The "WMDRM Initialization Vector" and "WMDRM Key ID" extensions MUST be included for the first payload of a new MAU in each multiple-payload RTP packet that contains multiple MAUs. This ensures that the receiver always knows about the current key ID even if some RTP packets are lost.

2.1.1.3.8 SDP Description

The use of the **RTP** Payload Format for Windows Media Audio and Video [WMRTP] is indicated in the **SDP** description by preceding the MIME subtypes for MPEG-2 audio and video elementary streams with the "vnd.ms.wm-" prefix.

Thus, the MIME subtype for MPEG-2 audio elementary streams MUST be "vnd.ms.wm-MPA", and the MIME subtype for MPEG-2 video elementary streams MUST be "vnd.ms.wm-MPV".

All other parameters MUST be set according to sections 4.1.17 and 4.2.8 of RFC 3555 [RFC3555] for MPEG-2 audio and video elementary streams.

2.1.1.4 OpenCable Digital Receiver Interface Mappings

This section describes how the WMDRM: Network Devices Protocol is used with the OpenCable Digital Receiver Interface (DRI) transport. The protocol procedures used via DRI are **registration**, **proximity detection**, **license retrieval**, license update, and revocation.

2.1.1.4.1 Registration

Registration is accomplished by following the rules defined here. In addition to parsing the certificate as specified here, the transmitter MUST make sure that the receiver has been properly individualized by looking for the following string.

```
<c:SecurityVersion> in <Keyfile><c:CertificateCollection><c:Certificate><c:Data>
```

If the **SecurityVersion** ends in ".1", the cert is individualized.

Registration MUST fail if the receiver is not individualized, as indicated by **SecurityVersion**.

There are multiple certificates in the **CertificateCollection**. Verifying the **SecurityVersion** data on the leaf certificate is sufficient.

The message format follows the rules documented in section 2.2.1.

See section 2.1.1.1 for Registration DRI **UPnP** mappings.

2.1.1.4.2 Proximity Detection

Proximity detection is accomplished by following the rules specified in section 2.1.1.5.3

2.1.1.4.3 License Retrieval

license retrieval is accomplished by following the rules here, and the message format following the rules documented here. See this UPnP Mapping for license retrieval DRI **UPnP** mappings.

2.1.1.4.4 License Update

License updates are accomplished by following the guidelines in section 2.2.1.4.3 and the message format guidance documented in section 2.2.1.4.3. License update messages are sent via TAG packets.

The transmitter initiates a license update for every event defined in [XMR] and for "License Generation", of the OpenCable Digital Unidirectional Receiver Interface Specification [OCDRIP].

Messages are repeated when transferring encrypted content to a receiver. Every TAG packet carries a license update message containing license derivation data.

2.1.1.4.5 Revocation

Prior to initiating registration, the receiver can send the transmitter **device** a **revocation information version (RIV)** structure and the WMDRM: Network Devices Protocol CRL as defined in section 2.2.1.6. These messages are sent over **UPnP**. See section 2.1.1.1 for revocation DRI UPnP mappings.

In addition, the transmitter MUST ensure that the Issue Time of the RIV is current as defined by [CR-WMDRM].

2.1.1.5 Media Transfer Protocol (MTP) over USB Mappings

The MTP extension defined in this section provides a mapping of WMDRM: Network Devices Protocol messages to the Media Transfer Protocol (MTP). The WMDRM: Network Devices Protocol extension to MTP defines operations that enable DRM-protected content to be securely transmitted from the transmitter to the receiver.

The WMDRM: Network Devices Protocol extension to MTP depends on the Microsoft Advanced Audio/Video Transfer (AAVT) extension to MTP for media session management. As a result of this dependence, the AAVT extension MUST also be implemented in both the transmitter and receiver to take advantage of WMDRM: Network Devices Protocol. See the Media Transfer Protocol Specification [MTP] for details concerning the AAVT extension to MTP.

The terms initiator and responder, used in the context of MTP, indicate the MTP initiator and MTP responder. When comparing information with WMDRM: Network Devices Protocol documentation, an initiator corresponds to a WMDRM: Network Devices Protocol receiver, and a responder corresponds to a WMDRM: Network Devices Protocol transmitter.

The following sections give procedural information specific to MTP over USB in the context of WMDRM: Network Devices Protocol.

2.1.1.5.1 Establishing a Connection

After a USB connection is established between the receiver and the transmitter, the transmitter is identified as an MTP **device** [MTP]. Then the receiver is expected to retrieve the **DeviceInfo** dataset from the transmitter using the **GetDeviceInfo** MTP operation. The **DeviceInfo** dataset includes a string indicating the MTP extensions implemented by the transmitter and the set of all MTP operations supported by the transmitter.

To verify compatibility with the Microsoft AAVT MTP extension, the receiver SHOULD check whether the **DeviceInfo** dataset returned by the transmitter indicates that AAVT operations are supported. The **MTPExtensions** field of the **DeviceInfo** dataset SHOULD contain the string "microsoft.com/AAVT: 1.0; ". Additionally, the **OperationsSupported** field of the **DeviceInfo** dataset that contains an array of operation codes supported by the transmitter SHOULD include operation codes 0x9170 (OpenMediaSession), 0x9171 (CloseMediaSession), and 0x9172 (GetNextDataBlock). The transmitter can optionally indicate support for operation code 0x9173 (SetCurrentTimePosition).

In the same way, if the transmitter supports the WMDRM: Network Devices Protocol extension to MTP, it SHOULD advertise it in the **DeviceInfo** dataset. The **MTPExtensions** field of the **DeviceInfo** dataset SHOULD contain the string "microsoft.com/WMDRMND: 1.0; ", and the **OperationsSupported** array SHOULD contain each of the operation codes defined as part of the WMDRM: Network Devices Protocol extension to MTP. These operations are defined by the operation codes 0x9180 through 0x9185, and are specified in the Media Transfer Protocol Specification [MTP].

2.1.1.5.2 Registration

After the physical connection is established and the receiver has confirmed that the transmitter supports the AAVT and WMDRM: Network Devices Protocol extensions, it SHOULD perform WMDRM: Network Devices Protocol registration.

During WMDRM: Network Devices Protocol registration, the receiver supplies its certificate to the transmitter for validation and for later exchanging content keys. Registration is performed using the **SendRegistrationRequest** and **GetRegistrationResponse** MTP operations as described in the Media Transfer Protocol Specification [MTP].

If **SendRegistrationRequest** and **GetRegistrationResponse** succeed, the receiver is considered to be registered. The receiver's certificate will expire 48 hours after registration. The registration procedure MUST be performed again. After the receiver is registered, it can perform WMDRM: Network Devices Protocol **proximity detection**. Both registration and proximity detection are required before the receiver can acquire a license for DRM-protected content to enable WMDRM: Network Devices Protocol delivery. For more information on the registration, see section 2.2.1.

2.1.1.5.3 Proximity Detection

After the receiver has been registered with the transmitter, the **proximity detection** procedure is performed. The proximity detection procedure is performed using the **GetProximityChallenge** and **SendProximityResponse** MTP operations that are described in the Media Transfer Protocol Specification [MTP].

During WMDRM: Network Devices Protocol proximity detection, the transmitter sends a 128-bit random value to the receiver. This value, called the **nonce**, MUST be encrypted by the initiator using the **content encryption key** issued during registration, and returned to the responder within a fixed time period.

For more information on proximity detection, see section 3.1.1.4.

2.1.1.5.4 Establishing a Media Session and Retrieving a License

After successful **proximity detection**, the receiver can request a license to play DRM-protected content. The operations used to request and receive licenses are **SendWMDRMNDLicenseRequest**

and **GetWMDRMNDLicenseResponse**; these methods are discussed in the Media Transfer Protocol Specification [MTP].

Details concerning the **license retrieval** procedure are discussed here and the message format is explained here. For additional information on content retrieval for circumstances where WMDRM: Network Devices Protocol is not used, see the Media Transfer Protocol Specification [MTP].

2.1.1.5.5 Retrieving Content Data Using a Media Session

After the media session is opened and a license is acquired, the receiver can begin content data retrieval. In the context of WMDRM: Network Devices Protocol, the data will have been encrypted with the **content encryption key** obtained during **license retrieval**.

When retrieving content, the receiver requests blocks of content data using the **GetNextDataBlock** operation defined by the AAVT extension to MTP. The receiver can repeat the **GetNextDataBlock** operation request until the transmitter indicates that the end of the **file** has been reached.

The control capabilities returned during the **OpenMediaSession** operation will indicate whether time-based seeking is supported. If it is supported, the initiator can use the AAVT extension **SetCurrentTimePosition** to set the position of the next read operation at any time while the media session is open.

2.1.1.5.6 Terminating a Session

When the receiver determines that it no longer requires an open media session for the current content selection, it can terminate the session. During session termination, the receiver notifies the transmitter that the content is no longer needed and that the transmitter SHOULD free any **resources** associated with the media session. After a media session is closed, any further AAVT extension MTP operation requests that use the **SessionID** corresponding to the closed session will return an error within the range of errors defined for MTP operation responses in [MTP].

Session Termination (**CloseMediaSession** - operation 0x9171)

To terminate a session, the receiver issues a **CloseMediaSession** operation request (as defined by the AAVT extension to MTP), passing the **MediaSessionID** returned during the **OpenMediaSession** operation as the first parameter. The following diagram illustrates the sequence of operations for terminating a session.

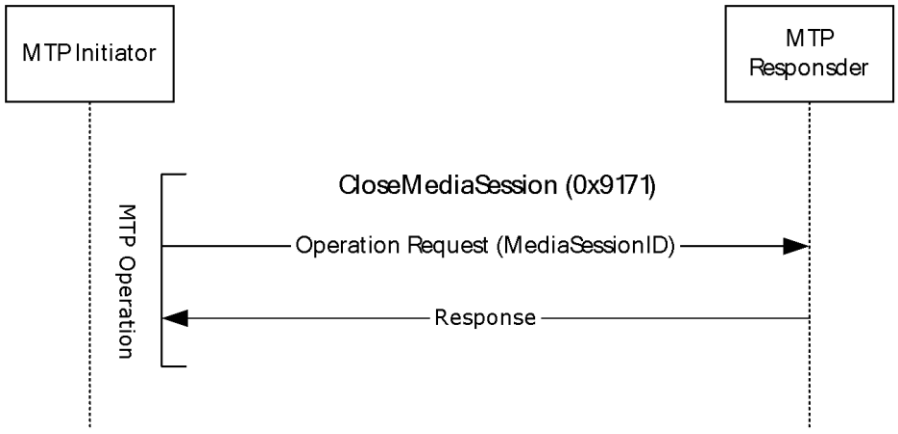


Figure 2: Terminating a Session Operation Sequence

When the transmitter receives the **CloseMediaSession** operation request, it can free any resources allocated for the media session. The transmitter SHOULD then return the result of the operation in the response message.

2.1.1.6 PBDA Mappings

This section describes how the WMDRM: Network Devices Protocol is used with **Protected Broadcast Driver Architecture (PBDA)** tuners. The protocol procedures used with PBDA are registration, **proximity detection**, **authenticated commands**, and revocation.

2.1.1.6.1 Registration

Registration is accomplished by following the rules here. In addition to parsing the certificate as specified here the transmitter MUST make sure that the receiver has been properly individualized by looking for <c:SecurityVersion> in <Keyfile><c:CertificateCollection><c:Certificate><c:Data>.

There are multiple certificates in the **CertificateCollection**. Verifying the **SecurityVersion** data on the leaf certificate is sufficient.

The message format follows the rules in sections 2.2.1.1 and 2.2.1.2 respectively.

See section 2.1.1.1 for registration **PBDA** mappings.

2.1.1.6.2 Proximity Detection

Proximity detection is accomplished by following the rules documented in section 3.1.1.4

2.1.1.6.3 Revocation

Prior to initiating registration, the receiver MUST send the transmitter **device** a **revocation information version (RIV)** structure and the WMDRM: Network Devices Protocol **CRL** as defined in section 2.2.1.6. These messages are sent over **UPnP**. In addition to the policies specified in this document, the transmitter MUST ensure that the Issue Time of the RIV is current as defined by the Compliance and Robustness Rules for Windows Media DRM [CR-WMDRM].

2.1.1.6.4 Authenticated Commands

Use of **authenticated commands** is accomplished by following the rules described in section 2.2.1.7.

2.2 Message Syntax

All integer values are unsigned and are represented in big-endian (**network byte order**) format unless otherwise noted.

All GUID values are in packet format, as specified in [MS-DTYP] section 2.3.4.2.

2.2.1 Registration

The following sections describe the structures of messages used during the registration process.

2.2.1.1 Registration Request Message

The **receiver** sends a registration request message to start the registration procedure. A registration request message contains three sections:

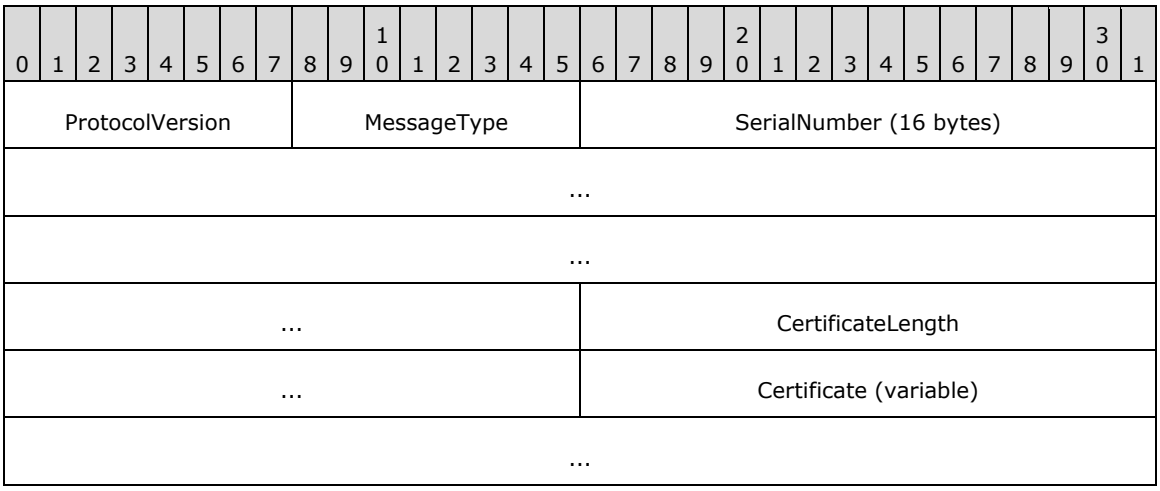
Header: Contains the protocol version supported by the receiver and the message type. The protocol version MUST be 0x03. The message type for registration request messages is 0x01.

Serial number: Contains the serial number used to identify the receiver. Manufacturers MAY use the same certificate for multiple devices as long as every **device** has a unique serial number.

If the certificate is unique, the serial number MUST be zero.

Certificate: Contains a 32-bit value indicating the length, in bytes, of the XML certificate and its **certificate chain**, along with the byte array containing the **XML** certificate and certificate chain.

The certificate MUST NOT be null terminated.



ProtocolVersion (1 byte): Contains the protocol version. The protocol version MUST be set to 0x03.

MessageType (1 byte): Contains the message type. The message type MUST be set to 0x01, which indicates a registration request message.

SerialNumber (16 bytes): Contains the serial number used to identify the receiver. Manufacturers MAY use the same certificate for multiple devices as long as every device has a unique serial number.

CertificateLength (4 bytes): Contains a 32-bit value. This value indicates the length, in bytes, of the XML certificate and its certificate chain.

Certificate (variable): Contains a byte array. This array, **CertificateLength** in size, contains the XML certificate and certificate chain.

2.2.1.2 Registration Response Message

The registration response message is sent by the transmitter and provides the receiver with an encrypted random value. The receiver MAY use it to compute the **content encryption key** and **content integrity key**.

Receivers MUST silently disregard any additional data between the encrypted seed section and signature section of the registration response message. In addition, the receiver MUST silently disregard any additional data after the end of the registration response message.

The registration response message contains the following six sections:

Header: Contains the protocol version supported by the transmitter and the message type. The protocol version MUST be 0x03. The message type for registration response messages is 0x02.

Signature offset: Contains the signature offset, which is a 16-bit value that indicates the number of bytes from the beginning of the message to the location of the signature section. This allows for the extension of the message in future revisions of the protocol while still providing a backward-compatible mechanism for signing the complete message.

Serial number: Contains the 128-bit serial number of the receiver. The receiver MUST check the serial number that is sent by the transmitter against its stored serial number. If there is a mismatch, the receiver MUST quit the operation.

Session ID: Contains the session ID. The transmitter generates the session ID, which is then used by the receiver during **proximity detection**.

Transmitter Identifier: Contains a 16-bit value indicating the length of an ANSI string , in bytes, and the ANSI string itself. The string specifies the communication method for sending and receiving proximity detection messages with the **transmitter**. The string contains the transport protocol as well as any additional data such as IP Address and Port.

Encrypted seed: Contains the seed **encryption** type, encrypted seed length, and the encrypted seed itself. The 8-bit type MUST be set to 0x01 to indicate RSA encryption. The 16-bit length is the length of the encrypted seed, in bytes, which is determined by the length of the receiver's RSA key. For example, because all receivers use 1024-bit **public keys**, the encrypted seed length is 128 bytes. The seed itself is a 128-bit random value that is used to derive the content encryption key and content integrity key. The seed is encrypted using the public key of the receiver. The **private key** of the receiver is used to decrypt the seed.

The seed is encrypted using the RSAES-OAEP encryption scheme that is defined in the **PKCS#1** (version 2.1) standard [PKCS1]. The hash algorithm is SHA-1 [FIPS180-2], which is defined in FIPS 180-2. The mask generation function is MGF1, which is defined in section B.2.1 of the PKCS#1 (version 2.1) standard [PKCS1]. The *label* parameter MUST be set to an empty string.

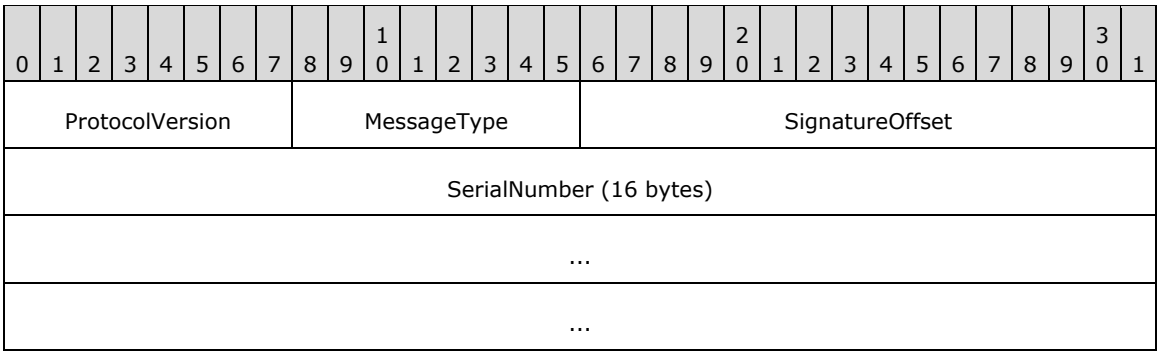
The content encryption key is computed by calculating the SHA-1 [FIPS180-2] hash of the seed concatenated with a constant value. The constant MUST be the 128-bit representation of the number 1. The first 128 bits of the SHA-1 [FIPS180-2] hash constitute the content encryption key.

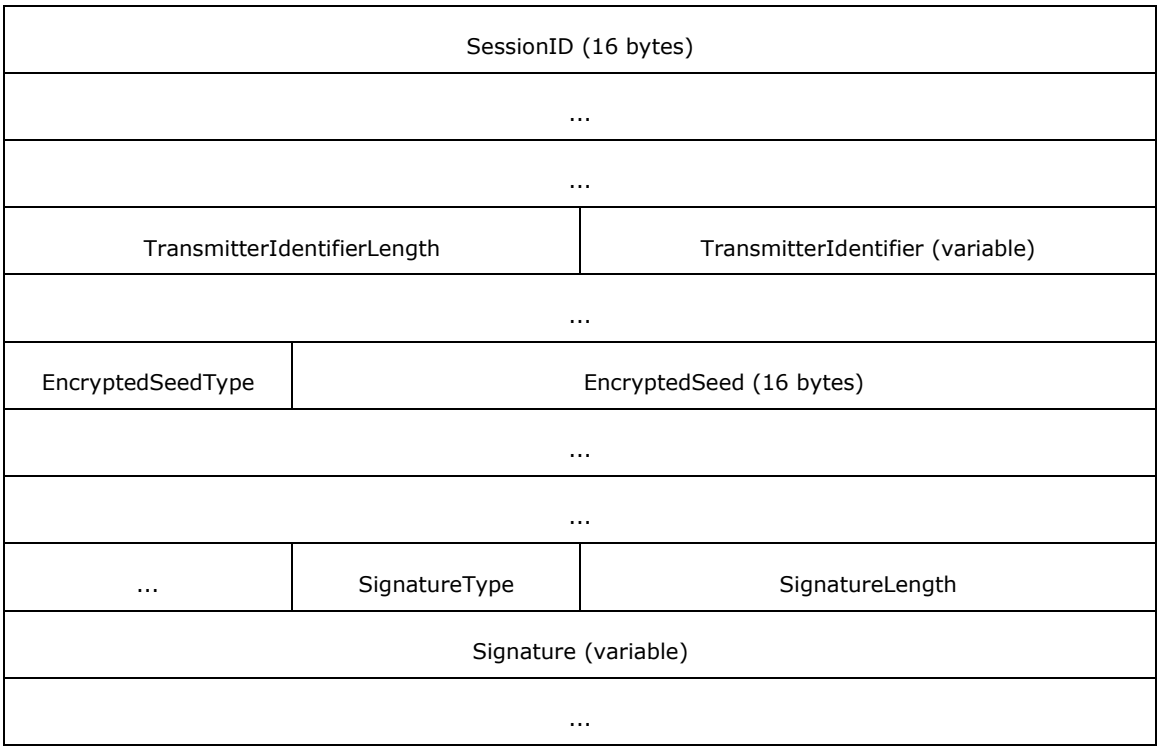
The content integrity key is computed by calculating the SHA-1 [FIPS180-2] hash of the seed concatenated with a constant. The constant MUST be the 128-bit representation of the number 2. The first 128 bits of the SHA-1 [FIPS180-2] hash constitute the content integrity key.

The **authenticated commands** key is computed by calculating the SHA-1 [FIPS180-2] hash of the seed concatenated with a constant. The constant MUST be the 128-bit representation of the number 3. The first 128 bits of the SHA-1 [FIPS180-2] hash constitute the authenticated command key.

Signature: Contains the signature type and length, and the **digital signature** itself. The 8-bit type is set to 0x01 to indicate that it is an **AES** OMAC1 [OMAC]. The length is a 16-bit value containing the length of the digital signature, in bytes, and the digital signature is calculated over all prior sections of the registration request message, including any additional data between the encrypted seed and signature sections.

The OMAC [OMAC] MUST be computed using the content integrity key and a zero initialization vector. Computing the content integrity key is described in the preceding section on encrypted seeds.





ProtocolVersion (1 byte): Contains the protocol version. The protocol version MUST be 0x03.

MessageType (1 byte): Contains the message type for registration response messages. The message type for registration response messages is 0x02.

SignatureOffset (2 bytes): A 16-bit integer value. Indicates the number of bytes from the beginning of the message to the location of the signature section. This allows for the extension of the message in future revisions of the protocol while still providing a backward-compatible mechanism for signing the complete message.

SerialNumber (16 bytes): Contains the 128-bit serial number of the receiver. The receiver MUST check the serial number that is sent by the transmitter against its stored serial number. If there is a mismatch, the receiver MUST quit the operation.

SessionID (16 bytes): Contains the session ID. The transmitter generates the session ID, which is then used by the receiver during proximity detection.

TransmitterIdentifierLength (2 bytes): Contains a 16-bit value indicating the length of an ANSI string, in bytes.

TransmitterIdentifier (variable): Indicates the transport protocol. The string specifies the communication method for sending and receiving proximity detection messages with the transmitter. The string contains the transport protocol as well as any additional data such as IP Address and Port. The format for the string is specified in section 2.2.1.2.1.

EncryptedSeedType (1 byte): An 8-bit integer. Contains an encrypted type. This field MUST be set to 0x01 to indicate RSA encryption.

EncryptedSeed (16 bytes): Contains an encrypted key. The seed itself is a 128-bit random value that is used to derive the content encryption key and content integrity key. The seed is encrypted using the public key of the receiver. The private key of the receiver is used to decrypt the seed.

SignatureType (1 byte): Contains a signature type. The 8-bit type is set to 0x01 to indicate that it is an AES OMAC1 [OMAC].

SignatureLength (2 bytes): Contains the length of the digital signature, in bytes.

Signature (variable): Contains the digital signature, which is calculated over all prior sections of the registration request message, including any additional data between the encrypted seed and signature sections.

2.2.1.2.1 Transport Protocol: Transport-specific Metadata

These are the values for the *Transport Protocol* parameter:

IP4: Denotes an IPv4 address in decimal dot notation.

IP6: Denotes an IPv6 address enclosed in square brackets according to the **URL** syntax for IPv6 addresses defined in [RFC2732].

TA: Denotes transport agnostic for use by receivers implemented with transport agnostic **proximity detection**. Any remaining values in the string will be ignored.

The following are examples of how the string MAY appear. The first example illustrates the syntax for IPv4 addresses, and the second example illustrates the syntax for IPv6 addresses. The third example illustrates the syntax for transport agnostic proximity detection:

```
IP4:192.168.254.2:877
IP6:[FE80::2AA:FF:FE9A:4CA2]:877
TA:
```

This string MUST NOT be null terminated.

2.2.1.3 Proximity Detection

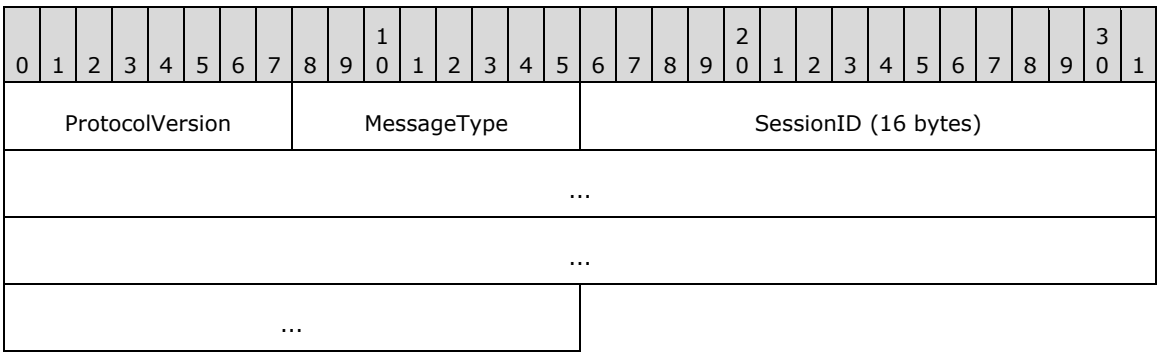
The following sections describe the structure of the messages used during a **proximity detection** session.

2.2.1.3.1 The Proximity Start Message

The proximity start message contains two sections:

Header: Contains the protocol version supported by the receiver and the message type. The protocol version MUST be set to 0x03. The message type for proximity start messages is 0x03.

Session ID: Contains the session ID. The receiver MUST send the same 128-bit session ID that was passed to it in the registration response message.



ProtocolVersion (1 byte): Contains the protocol version supported by the receiver. MUST be set to 0x03.

MessageType (1 byte): Contains the message type for proximity start messages. The message type for proximity start messages is 0x03.

SessionID (16 bytes): Contains the session ID. The receiver MUST send the same 128-bit session ID that was passed to it in the registration response message.

2.2.1.3.2 The Proximity Challenge Message

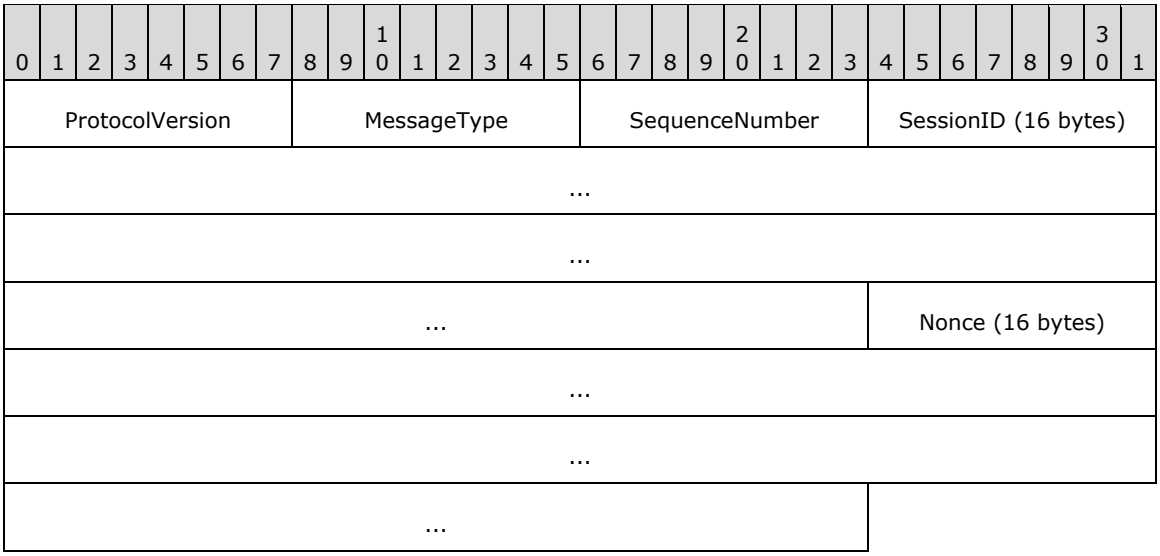
A proximity challenge message contains four sections:

Header: Contains the protocol version supported by the transmitter and the message type. The protocol version MUST be set to 0x03 and the message type for proximity challenge messages is 0x04.

Sequence number: This 8-bit sequential number. This value MUST be incremented by one every time the proximity challenge message is sent to the receiver.

Session ID: Contains the session ID. The transmitter MUST send the same 128-bit session ID that it sent to the receiver in the registration response message.

Challenge value (Nonce): A 128-bit, random **challenge** value selected by the transmitter.



ProtocolVersion (1 byte): Contains the protocol version supported by the transmitter. The protocol version MUST be set to 0x03.

MessageType (1 byte): Contains the message type. The message type for proximity challenge messages is 0x04.

SequenceNumber (1 byte): Contains the sequence number. This 8-bit sequential number MUST be incremented by one every time the proximity challenge message is sent to the receiver.

SessionID (16 bytes): Contains the session ID. The transmitter MUST send the same 128-bit session ID that it sent to the receiver in the registration response message.

Nonce (16 bytes): Contains the nonce. Must be set to a 128-bit random value selected by the transmitter.

2.2.1.3.3 The Proximity Response Message

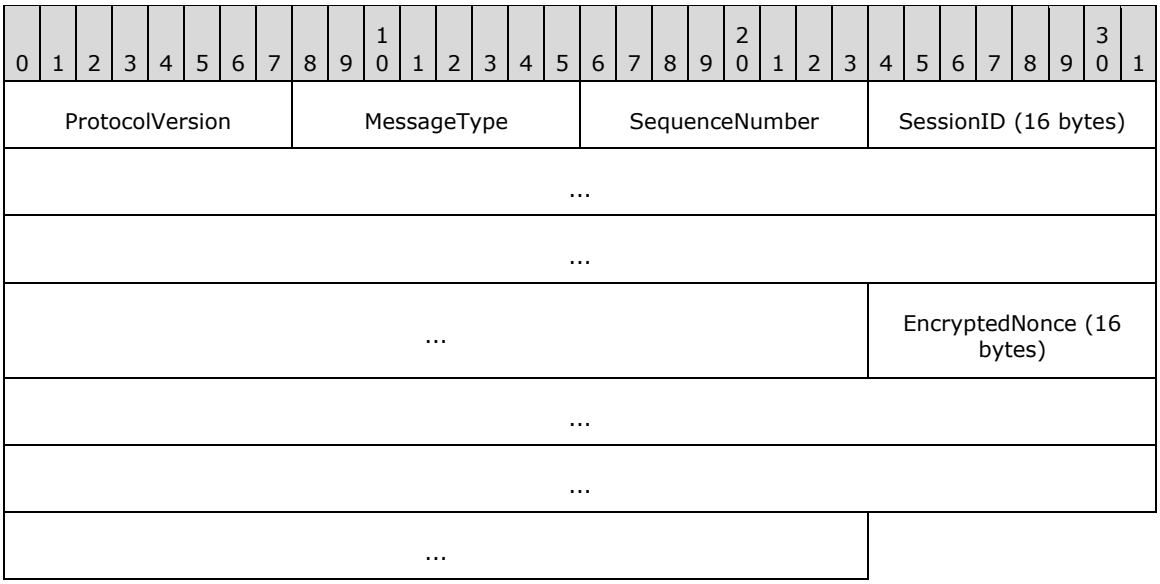
The proximity response message contains four sections:

Header: Contains the protocol version supported by the receiver and the message type. The protocol version MUST be 0x03. The message type for proximity response messages is 0x05.

Sequence number: Contains the sequence number. The sequence number MUST be the same one included in the proximity challenge message.

Session ID: Contains the session ID. The receiver MUST send the same 128-bit session ID that was passed to it in the registration response message.

Encrypted challenge value (nonce): Contains the nonce. The nonce is the 128-bit random value in the proximity challenge message, encrypted using the **AES content encryption key** in ECB mode.



ProtocolVersion (1 byte): Contains the protocol version supported by the receiver. The protocol version MUST be 0x03.

MessageType (1 byte): Contains the message type. The message type for proximity challenge messages is 0x05.

SequenceNumber (1 byte): Contains the sequence number. This 8-bit sequential number MUST be incremented by one every time the proximity challenge message is sent to the transmitter.

SessionID (16 bytes): Contains the session ID. The receiver MUST send the same 128-bit session ID that was passed to it in the registration response message.

EncryptedNonce (16 bytes): Contains the encrypted nonce. This is the 128-bit random value from the proximity challenge message that was encrypted using the content encryption key using AES in ECB mode.

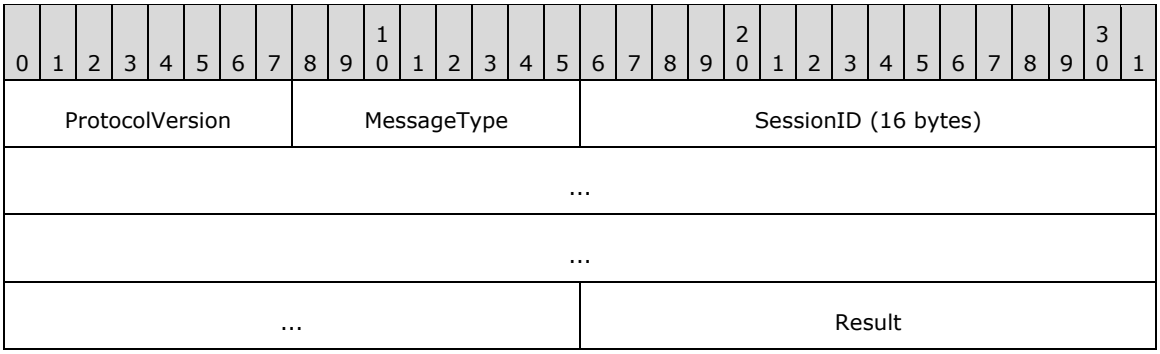
2.2.1.3.4 The Proximity Result Message

The proximity result message contains three sections:

Header: Contains the protocol version supported by the transmitter and the message type. The protocol version MUST be 0x03. The message type for proximity result messages is 0x06.

Session ID: Contains the session ID. The transmitter MUST send the same 128-bit session ID that it sent to the receiver in the registration response message.

Result: A 16-bit integer. The **Result** field is a 16-bit integer and indicates the result of the **proximity detection**. If the proximity detection procedure was a success, then the **Result** field MUST be set to 0x0000. If it was a failure, then the **Result** field MUST be set to error code 106 (Unable to Verify Proximity). The complete list of error codes is defined in section 2.3.



ProtocolVersion (1 byte): Contains the protocol version supported by the transmitter. The protocol version MUST be 3.

MessageType (1 byte): Contains the message type. The message type for proximity result messages is 6.

SessionID (16 bytes): Contains the session ID. The receiver MUST send the same 128-bit session ID that was passed to it in the registration response message.

Result (2 bytes): A 16-bit integer. Indicates the result of the proximity detection. If the proximity detection procedure was a success, then the **Result** field MUST be set to 0x0000. If it was a failure, then the **Result** field MUST be set to error code 0x06a (Unable to Verify Proximity). The complete list of error codes is defined in section 2.3.

2.2.1.4 License Retrieval

The following section describes the messages sent and received to deliver a license to a receiver.

2.2.1.4.1 The License Request Message

The receiver sends a license request message when it requests **content** from the transmitter. The message contains six parts:

Header: Contains the protocol version supported by the receiver and the message type. The protocol version is 0x03 and the message type for license request messages is 0x07.

Rights ID: Contains the Rights ID, which is a random value generated by a cryptographically random number generator that is in accordance with the Compliance Rules [CR-WMDRM] The Rights ID is echoed back by the transmitter during the license response message.

CRL version number: Contains the 32-bit version number of the receiver's **CRL**. This value MUST be zero if the receiver does not have transmitting capabilities.

Serial number: Contains the serial number that identifies the receiver. Manufacturers MAY use the same certificate for multiple devices, as long as every **device** has a unique serial number.

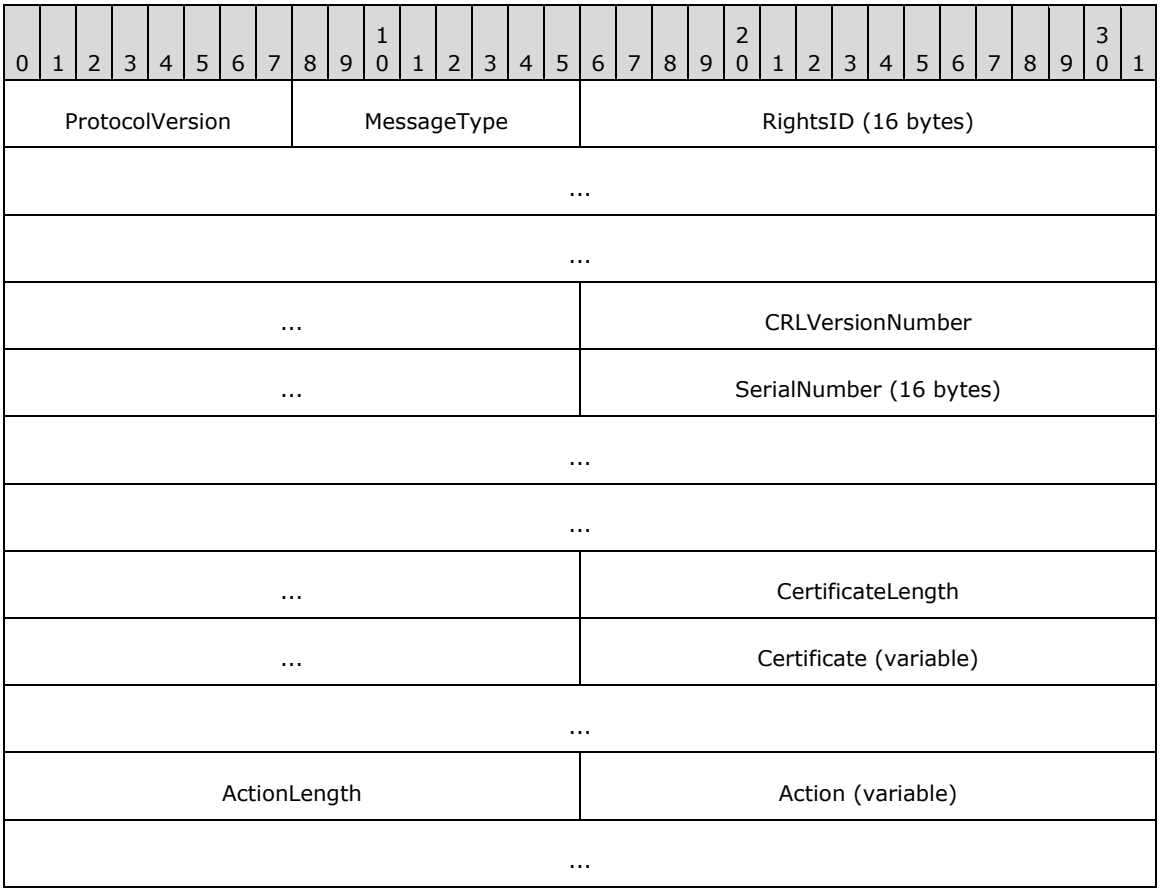
Certificate: Contains a 32-bit value indicating the length (in bytes) of the **XML** certificate and its **certificate chain**, along with the byte array containing the XML certificate and certificate chain.

The certificate MUST NOT be null terminated.

Intended action of the receiver: The action section starts with a 16-bit integer value that specifies the length (in bytes) of the action string, followed by the action string. The action string is encoded in the **ANSI** character set.

The receiver MUST send the action string to the transmitter indicating what it is trying to do with the content. The only action string currently defined is "Play".

The action that is indicated by the receiver is used by the transmitter to locate the appropriate license and does not guarantee that the license returned allows the requested action. The receiver MUST respect the rights in the license. The action string is case sensitive and MUST NOT be null terminated.



ProtocolVersion (1 byte): Contains the protocol version. The protocol version MUST be set to 0x03.

MessageType (1 byte): Contains the message type. The message type for license request messages is 0x07.

RightsID (16 bytes): Contains the rights ID. A random value generated by a cryptographically random number generator that is in accordance with the Compliance and Robustness Rules for Windows Media DRM [CR-WMDRM]. This value is echoed back by the transmitter during the license response message.

CRLVersionNumber (4 bytes): Contains the 32-bit version number of the receiver's CRL. This value MUST be zero if the receiver does not have transmitting capabilities.

SerialNumber (16 bytes): Contains the serial number that identifies the receiver. Manufacturers MAY use the same certificate for multiple devices, as long as every device has a unique serial number.

CertificateLength (4 bytes): Contains a 32-bit value. Indicates the length (in bytes) of the XML certificate and its certificate chain.

Certificate (variable): Contains the XML certificate and certificate chain.

ActionLength (2 bytes): Contains a 16-bit value. Indicates the length (in bytes) of the ANSI action string.

Action (variable): ANSI-encoded string containing the action the receiver will perform with the content.

2.2.1.4.2 The License Response Message

After the transmitter handles the license request message, it sends a license response message to the receiver.

Receivers MUST silently disregard any additional data after the end of the license response message.

The license response message contains three parts:

Header: Contains the protocol version supported by the transmitter and the message type. The protocol version MUST be 0x03. The message type is set to 0x08 for license response messages.

CRL: A 32-bit value. Indicates the length of the variable-length **CRL** section, in bytes, followed by the CRL section itself.

The CRL is sent when the transmitter has a higher CRL version number than the receiver, and the receiver also has transmitting capabilities. The **device** certificate of the receiver specifies whether the receiver has transmitting capabilities.

License: Contains a 32-bit value that specifies the length of the variable-length license, in bytes, followed by the byte array of the license itself. The byte array is represented in XMR format [XMR]. In addition to the usage rights of the license, the license contains but is not limited to the following parts.

Content Encryption Key: The **content encryption key** is used to encrypt the media data using **AES** in **counter mode (CTR)** during data transfer. Encrypted with the **public key** of the receiver.

Content Integrity Key: The **content integrity key** is used to generate MACs using AES-based OMAC1 [OMAC]. Encrypted with the public key of the receiver.

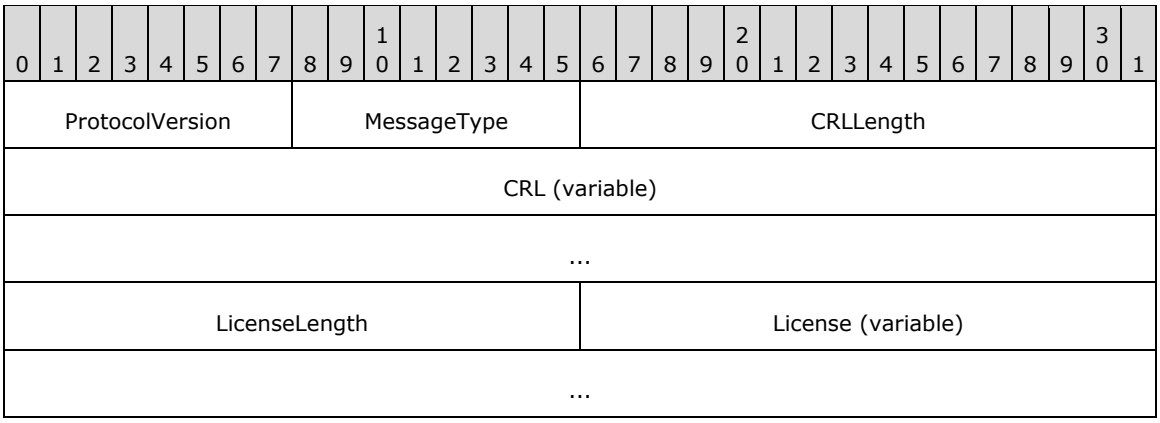
The content encryption key and content integrity key used in the license response message and during data transfer are not related to any keys used for the registration or **proximity detection** procedures.

Rights ID: Contains the same 128-bit random value sent by the receiver in the license request message. The transmitter MUST send this back to the receiver. The receiver MUST confirm that this is the same Rights ID that it sent in the license request message.

CRL version number: A 32-bit integer. Contains the version number of the transmitter's CRL.

Serial number: The serial number is sent by the transmitter so it MAY be checked against the stored serial number on the receiver. If there is a mismatch, the receiver MUST quit the operation.

For more information on licenses, see the XMR specification [XMR].



ProtocolVersion (1 byte): Contains the protocol version supported by the transmitter. The protocol version MUST be set to 0x03.

MessageType (1 byte): Contains the message type. MUST be set to 0x08 for license response message.

CRLLength (2 bytes): Indicates the length of the variable-length CRL section, in bytes.

CRL (variable): This variable-length, optional field is sent when the transmitter has a higher CRL version number than the receiver, and the receiver also has transmitting capabilities. The device certificate of the receiver indicates whether the receiver has transmitting capabilities.

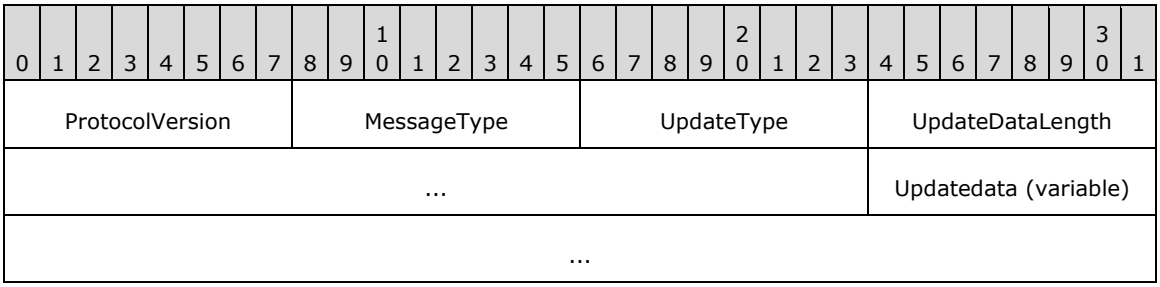
LicenseLength (2 bytes): Indicates the length of the variable-length license, in bytes.

License (variable): Contains the license itself, represented in XMR [XMR] format.

2.2.1.4.3 The License Update Message

Header: Contains the protocol version that is supported by the transmitter and the message type. The protocol version MUST be set to 0x03. The message type for **license update** messages is 0x09.

License Update data: If you are performing license management with license chains, this section contains the root/**leaf license**.



ProtocolVersion (1 byte): Contains the protocol version. The protocol version MUST be set to 0x03.

MessageType (1 byte): Contains the message type for license update messages. The message type for license update messages is 0x09.

UpdateType (1 byte): Contains the update data type. If you are performing license management with license update message, this section contains the update data type, the length of the update data (in bytes) and the byte array that contains the license update data. The 8-bit type is set to 0x01 to indicate that the update data was generated by a version 1.0 Transmitter.

UpdateDataLength (4 bytes): Indicates the length, in bytes, of the Updatedata field.

Updatedata (variable): A byte array **UpdateDataLength** in size. Contains the update data.

When performing license management using license chains, the **Updatedata** contains the root/leaf license. When performing license management using license derivation data, then **Updatedata** contains the following:

- 8-bit license update data type.
- 32-bit license update data length.
- Variable-length license update data.

2.2.1.5 Revocation Lists

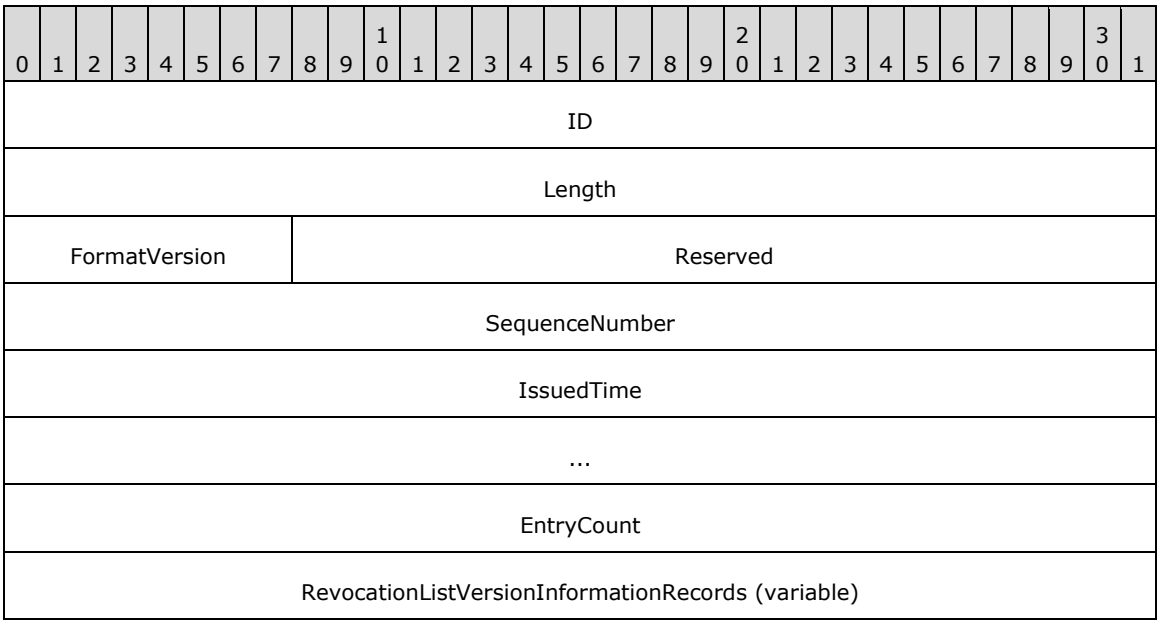
WMDRM: Network Devices Protocol supports distribution of its own **revocation** list and revocation lists for related systems. The list of related systems currently includes the following three types of revocation:

- **Application Revocation**. Application revocation is used to prevent access to Windows Media DRM-protected content by certain compromised or noncompliant applications.
- **Portable Device Revocation**. Device revocation prevents DRM-protected content from being copied or streamed to compromised or noncompliant portable devices.
- **Certified Output Protection Protocol (COPP) Driver Revocation**. **COPP** driver revocation is used to deny playback of content on systems where the **device driver** responsible for enabling output protection technologies is compromised or noncompliant.

The **RIV** structure is specified in section 3.1.5.1.

2.2.1.5.1 Revocation Information Version Structure

The **revocation information version (RIV)** structure contains versioning information about revocation lists. Use of the RIV insures that a revocation list on the receiver is the most recent applicable RIV for a license.



...	
SignatureType	Signature (128 bytes)
...	
...	
...	CertificateChainLength
...	
...	CertificateChain (variable)
...	

ID (4 bytes): Contains the **ASCII** characters 'R', 'L', 'V', and 'I'. Indicates this is a revocation list version information structure.

Length (4 bytes): Contains the length, in bytes, of the signed area of this structure starting from the beginning of the **ID** field up to, but not including, the **SignatureType** field.

FormatVersion (1 byte): Contains the format version. Data structure version MUST be 0x01.

Reserved (3 bytes): Reserved. MUST be set to zero when sent and MUST be ignored on receipt.

SequenceNumber (4 bytes): Contains the sequence number of the revocation list version information. The revision is used to distinguish a new RIV from an older one.

IssuedTime (8 bytes): Contains the date and time at which the revocation list versions contained in this structure were current. Represented as a 64-bit unsigned integer indicating the number of 100-nanosecond intervals since January 1, 1601 (UTC).

EntryCount (4 bytes): Contains the number of revocation list version information records.

RevocationListVersionInformationRecords (variable): As specified in section 2.2.1.5.1.

SignatureType (1 byte): Contains the signature type. A value of 0x01 means that a 1024-bit RSASSA-PSS signature is used with the scheme defined in the PKCS#1 (version 2.1) standard [PKCS1]. The signed section extends from the beginning of the **ID** field up to, but not including, this field.

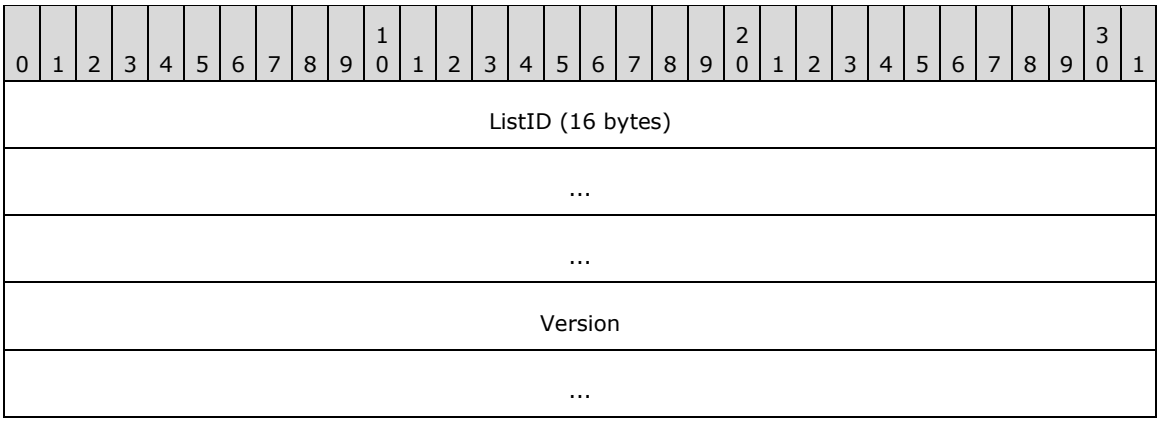
Signature (128 bytes): Contains the digital signature.

CertificateChainLength (8 bytes): Contains the length, in bytes, of the certificate chain field.

CertificateChain (variable): Contains the certificate chain. This certificate chain is the **XML** Certificate Collection for the Microsoft **CRL** signing authority.

2.2.1.5.2 Revocation List Version Information Record

The revocation information version structure is limited to a maximum size of 10 kilobytes.



ListID (16 bytes): Contains the identifier for the revocation list. The valid **GUIDs** for the **ListID** field are:

- {90A37313-0ECF-4CAA-A906-B188F6129300}: Windows Media DRM Application **CRL**. For this list ID, the lower 32 bits of the version MUST be a 32-bit unsigned integer (**little-endian**).
- {3129E375-CEB0-47D5-9CCA-9DB74CFD4332}: Windows Media DRM for Portable Devices CRL. For this list ID, the lower 32 bits of the version MUST be a 32-bit unsigned integer.
- {CD75E604-543D-4A9C-9F09-FE6D24E8BF90}: WMDRM: Network Devices Protocol or **COPP** Driver CRL. For this list ID, the lower 32 bits of the version MUST be a 32-bit unsigned integer.

Additionally, a GUID is specified for representing the **RIV** itself. This GUID is used to represent the RIV in revocation list request and response messages. This allows updated RIVs to be transferred via the **revocation list update** procedure.

- {CCDE5A55-A688-4405-A88B-D13F90D5BA3E}: Identifies the RIV itself.

Version (8 bytes): Indicates the current version number.

The authenticity of the RIV MUST be verified prior to its usage. To verify the RIV signature, a **device** MUST verify that the signing certificate, which is part of RIV, is directly signed by the Microsoft **root certificate** and has the **SignCRL** element value set to the string "1". Optionally, devices can store the well-known Microsoft CRL signing **public key** in secure, nonvolatile storage. For devices that choose this approach, verifying the RIV involves only checking the signature on the RIV itself.

2.2.1.5.3 Certificate Revocation Lists

WMDRM: Network Devices Protocol **certificate revocation lists (CRLs)** contain digests of revoked certificates and can be provided and signed only by Microsoft. They are typically distributed through DRM licenses, and only transmitters or receivers with transmitting capabilities need to maintain CRLs.

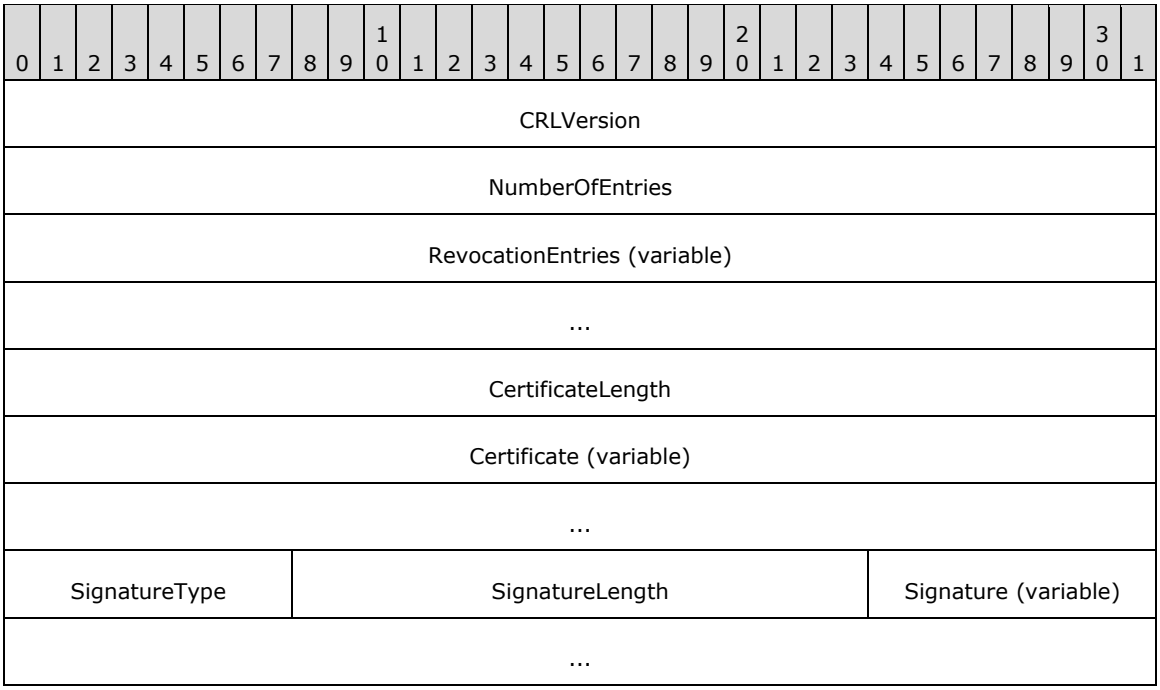
CRLs can revoke any certificate in the **device** certificate's chain. If any certificate in the chain is revoked, then that certificate and all of the certificates below it in the chain are also revoked.

When transmitters receive a CRL, they MUST make sure the CRL is valid. To verify that the CRL is valid, transmitters MUST verify that the CRL certificate, which is part of the CRL, is directly signed by the Microsoft **Root Certificate** and has the **SignCRL** element value set to the string "1". The signature of the CRL MUST also be verified.

After the CRL is verified, transmitters can store it. The CRL version number MUST be checked before storing so that the transmitter always stores the newest version.

Transmitters can also store the well-known Microsoft CRL signing **public key** in secure nonvolatile storage. For devices that choose this approach, verifying the CRL involves only checking the signature on the CRL itself.

A receiver that also has transmitting capabilities MUST compare its currently stored version of the CRL to the transmitter's CRL. If the transmitter's CRL is newer, the transmitter MUST send its CRL the receiver. If the transmitter's CRL is not newer, no CRL is sent to the receiver.



CRLVersion (4 bytes): Contains the version of the CRL.

NumberOfEntries (4 bytes): Indicates the number of revocation entries.

RevocationEntries (variable): The array of revocation entries.

CertificateLength (4 bytes): Indicates the length, in bytes, of the **XML** certificate and its **certificate chain**.

Certificate (variable): The XML certificate.

SignatureType (1 byte): Indicates the type of signature.

SignatureLength (2 bytes): Indicates the length of the signature, in bytes.

Signature (variable): Contains the signature of the CRL.

Header: Contains the version number of the CRL and the number of revocation entries in the CRL. A CRL can contain zero or more entries.

Revocation entries: Contains the revocation entries. Each revocation entry is the 160-bit **digest** of a revoked certificate. The transmitter compares this digest with the **DigestValue** element within the certificate it is checking.

Certificate: A 32-bit integer value. Indicates the length, in bytes, of the XML certificate and its certificate chain, along with the byte array containing the XML certificate of the **certificate authority (CA)** and certificate chain that has Microsoft as the root. The certificate MUST be signed by a CA that has the authority to issue CRLs.

Optionally, devices can store the well-known Microsoft CRL signing public key in secure, nonvolatile storage. For devices that choose this approach, verifying the CRL involves only checking the signature on the CRL itself.

This certificate MUST NOT be null terminated.

Signature: Contains the signature type and length, and the **digital signature** itself. The 8-bit type is set to 0x02 to indicate that it uses SHA-1 [FIPS180-2] with 1024-bit RSA **encryption**. The length is a 16-bit value containing the length of the digital signature, in bytes, and the digital signature is calculated over all prior sections of the CRL.

The signature is calculated using the RSASSA-PSS digital signature scheme defined in (version 2.1) [PKCS1]. The **hash function** is SHA-1 [FIPS180-2], defined in FIPS 180-2—and the mask generation function is MGF1, which is defined in section B.2.1 in **PKCS**#1 (version 2.1) [PKCS1].

2.2.1.6 Revocation List Update

This section describes the messages utilized to update the **revocation lists** on the receiver.

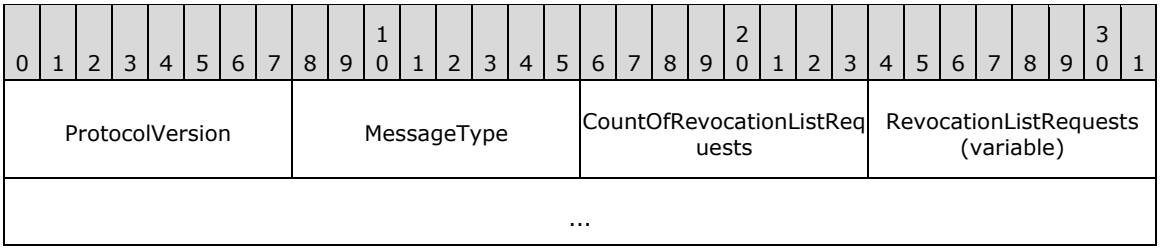
The receiver sends a revocation list request message when it is requesting an updated revocation list from the transmitter. The message contains two parts:

Header: Contains the protocol version that is supported by the transmitter and the message type. The protocol version MUST be set to 0x03. The message type for revocation list is 0x0a.

Revocation List Data: Contains a one-byte count of revocation list entries followed by the revocation list.

2.2.1.6.1 The Revocation List Request Message

This section starts with an 8-bit field specifying the number of list IDs, followed by the array of list IDs itself. A list ID is a **globally unique identifier (GUID)** used to identify which type of revocation list the receiver is requesting. Possible list ID values are listed in Revocation Lists.



ProtocolVersion (1 byte): Contains the protocol version. The protocol version MUST be set to 0x03.

MessageType (1 byte): Contains the message type for revocation list request messages. The message type for revocation list messages MUST be set to 0x0a.

CountOfRevocationListRequests (1 byte): Indicates the number of entries.

RevocationListRequests (variable): A variable-length list of 128-bit GUIDs.

2.2.1.6.2 The Revocation List Response Message

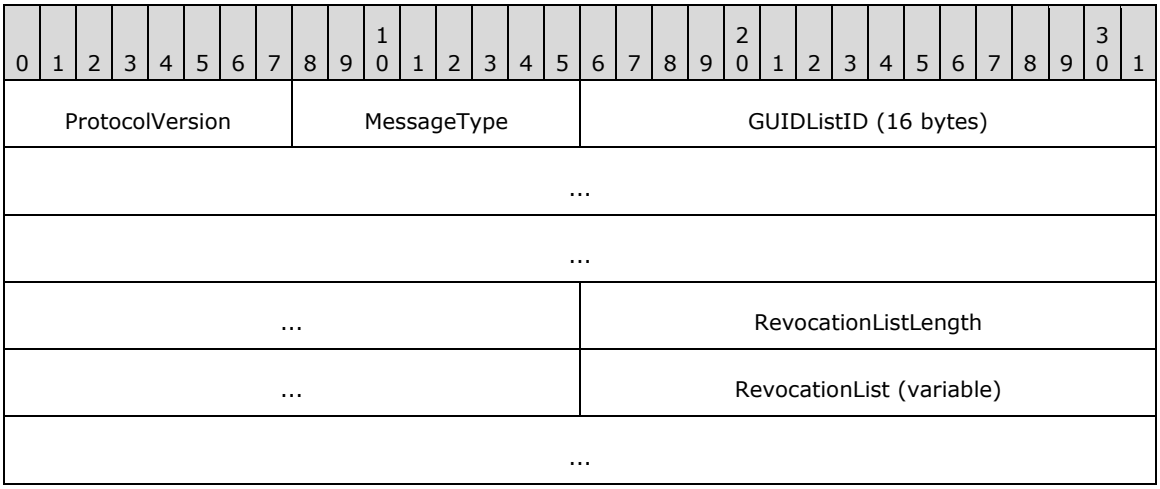
The transmitter sends a revocation list response message to provide the requested revocation list to the receiver. The message contains two parts:

Header: Contains the protocol version supported by the transmitter and the message type. The protocol version MUST be set to 0x03. The message type for revocation list response messages is 0x0b.

Revocation Lists (optional): This field MAY be present. When present, this field contains one or more revocation lists. Each revocation list MUST be placed in this section in the following format.

A 128-bit **GUID** list ID, followed by a 32-bit value that specifies the length of the variable-length revocation list, followed by a byte array containing the variable-length revocation list itself.

In case the transmitter does not store a revocation list for the given list ID, an empty (zero length) revocation list is sent to the receiver. Possible list ID values are included in section 2.2.1.6.



ProtocolVersion (1 byte): Contains the protocol version. The protocol version MUST be set to 0x03.

MessageType (1 byte): Contains the message type. The message type MUST be set to 0x0b for revocation list response message.

GUIDListID (16 bytes): Contains a 16 byte value that identifies the revocation list type that follows.

RevocationListLength (4 bytes): Contains the length of revocation list, in bytes.

RevocationList (variable): Contains possible list ID values. In case the transmitter does not store a revocation list for the given list ID, an empty (zero length) revocation list is sent to the receiver. Possible list ID values are included in revocation list.

Multiple revocation lists MAY be included in this message by sequentially adding a **GUIDListID**, **RevocationListLength** and **RevocationList** for each additional revocation list.

2.2.1.7 Authenticated Commands

This protocol supports sending cryptographically signed commands from the **receiver** to the **transmitter**.

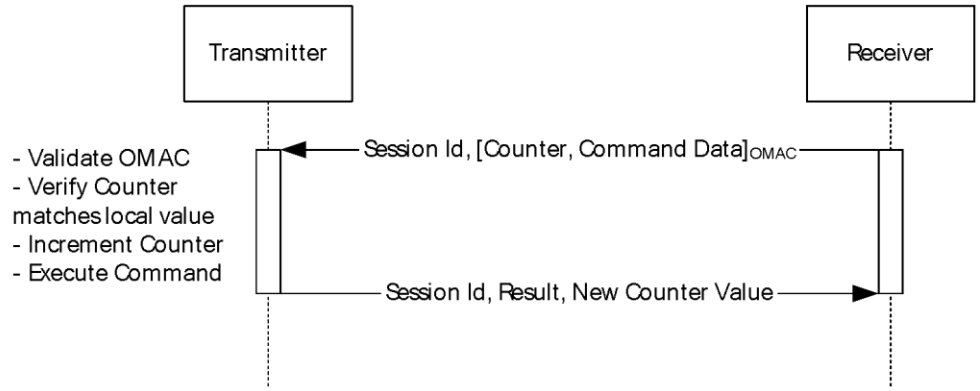


Figure 3: Sending Cryptographically Signed Commands from the Receiver to the Transmitter

The following table describes elements of the previous diagram.

Value	Description
Session ID	128 bit session identifier generated in registration.
Counter	Current value of the counter for sending authenticated messages. This value starts at zero (0).
Command Data	The data structure holding the data for the command. See below for details.
OMAC	OMAC1 [OMAC] of the message using the authenticated command key.
Result	Error code for the operation. 0 means success 0x071 (113) means Bad request 0x072 (114) means Counter Mismatch 0x073 (115) means Invalid Signature

2.2.1.7.1 The Authenticated Command Message

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
ProtocolVersion								MessageType								SessionID (16 bytes)															
...																															
...																															
...																Counter															
...																CommandType															
...																CommandLength															
...																CommandParameters (variable)															

...
OMAC (16 bytes)
...
...

ProtocolVersion (1 byte): Contains the protocol version. The protocol version is 0x03.

MessageType (1 byte): Contains the authenticated command message type. The authenticated command message type is 0x0c.

SessionID (16 bytes): Contains the 128-bit session ID. This value is sent by the transmitter in the Registration Response message.

Counter (4 bytes): Contains a counter for Authenticated Commands. For each Authenticated Command between the receiver and the transmitter, both the **Receiver** and **transmitter** independently increment the counter. The counter MUST be in sync for the transmitter to carry out the Authenticated Command. This mitigates a replay attack.

CommandType (4 bytes): Indicates the type of authenticated command.

CommandLength (4 bytes): Contains the length, in bytes, of the authenticated command.

CommandParameters (variable): Contains command parameters as dictated by **Id** and **length**.

OMAC (16 bytes): Contains the 128-bit **AES**-OMAC1 [OMAC] signature. The signature is generated over the concatenation of the **Counter** field ORed with the **Command Data** field, including **Id** and **length**.

2.2.1.7.2 The Authenticated Command Response Message

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
ProtocolVersion								MessageType								ResultStatus															
Counter																															
ResultDataLength																															
Resultdata (variable)																															
...																															

ProtocolVersion (1 byte): Contains the protocol version. The protocol version MUST be 3.

MessageType (1 byte): Contains the message type. The authenticated command response message type is 0x0d.

ResultStatus (2 bytes): Contains a 16-bit status code indicating success or failure. A value of zero indicates success; any other value is a failure code.

Counter (4 bytes): Contains the new counter value for the next command.

ResultDataLength (4 bytes): Contains the data length, in bytes, of the result. The result SHOULD be zero for all commands currently defined.

Resultdata (variable): Reserved. MUST be set to zero when sent and MUST be ignored on receipt.

The **SetStartCodeProfile** authenticated command controls which start codes are detected, and the number of payload bytes to leave in the clear for each detected start code. Any transport packet that contains a Stream that is using Streaming **AES** Counter Mode is examined, before encryption, for Start Codes.

A Start Code is a 4-byte sequence that contains 0x00, 0x00, 0x01, and 0x*nn*, where *nn* is the start code.

Start Code detection includes the ability to detect Start Codes that span multiple transport stream packets. This detection MAY be done by retaining the last 3 bytes of payload, for the AES encrypted stream, and then using that information in conjunction with the first few bytes of the next packet when it arrives.

If the transmitter is unable to comply with the **SetStartCodeProfile** request, the transmitter MUST return an error and MUST continue streaming with the previously specified Start Code Profiles. Failure to respect the commands MAY interfere or otherwise prohibit the receiver from consuming the content and MAY lead to a broken user experience.

Command ID = 1:

Name of the field	Size in bits	Value	Description
Start Code Enable	256	Varies	A flag that indicates if the **receiver** requires detection of the corresponding start code.
Start Code Cipher Delay	256 * 8	Varies	The number of bytes, starting with the fourth byte of the start code sequence, which MUST be left in the clear, for the matching Contiguous Stream, following the detection of this Start Code. Note that the **transmitter** uses **CipherDelay** to skip encrypting whole transport stream packets for which any byte appears that falls within the **CipherDelay**. A value of 0 for this argument means that no payload bytes are required to be left in the clear. If detection is enabled for such a start code, the transmitter MUST insert a Stream Info Table indicating the presence of an enable start code. By default, no start codes are enabled for detection. If a start code is not enabled for detection, no action taken. A non-zero value means that at least that many transport stream payload bytes, starting with the fourth byte of the start code sequence header, will at least be left in the clear. This will involve rounding out to full transport stream packets. Note that adaptation field bytes MUST NOT be considered payload bytes.

ClearStreamCipher: Contains the authenticated command, which allows the receiver to request that data on a stream not be encrypted. The receiver MUST NOT use this method on any stream that requires content protection. The purpose for this action is to allow the transmitter to stop encrypting content when protection is not required.

An example of where this action is only applicable is when the transmitter is issuing **leaf licenses** with Broadcast Screening Data objects.

Command ID = 2:

Name of the field	Size in bits	Value	Description
Type	32	Varies	Support values are Pid (0x1).
Value	Varies	Varies	Size of the structure 16 bits when the type is Pid.

SetLinkTimer: Contains the authenticated command that sets the link timer, which is the maximum time allowed before the receiver MUST call **SetLinkTimer** again. If the Time Limit has elapsed the transmitter MUST revert all Streams that are in the clear to be encrypted.

The transmitter only needs to support this **interface** if it supports **ClearStreamCipher**.

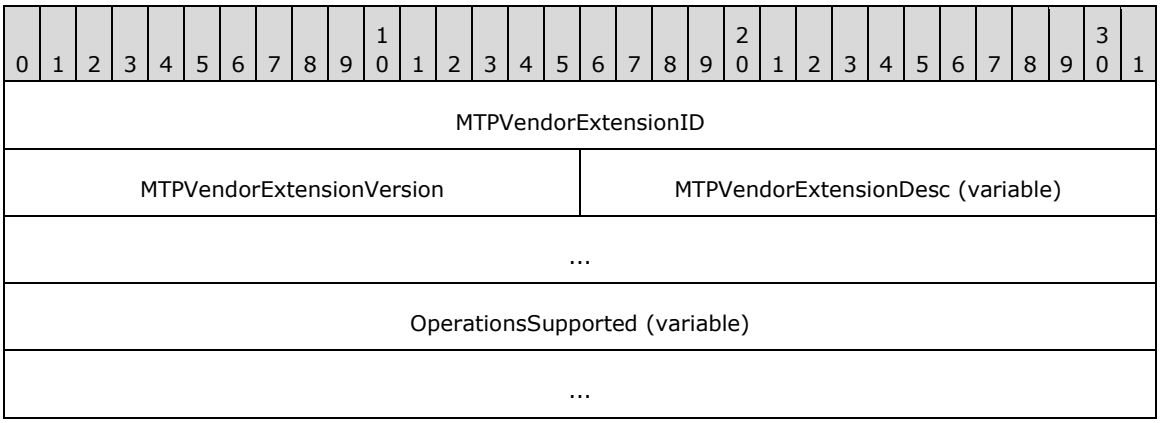
Command ID = 3:

Name of the field	Size in bits	Value	Description
Time Limit	16	Varies	The maximum number of seconds allowed between calls to **SetLinkTimer**.

2.2.2 MTP Vendor Extension Identification Message

When an MTP responder is connected to an MTP initiator over USB, the initiator queries the responder's **DeviceInfo** dataset. Included in this dataset are the MTP extensions supported by the responder.

If an MTP responder supports the AAVT extension to MTP, the dataset returned by the **DeviceInfo** MTP operation MUST contain the settings shown in the following table.



MTPVendorExtensionID (4 bytes): A 32-bit unsigned integer. Indicates the vendor ID of the extension to the Picture Transfer Protocol (PTP) used by the **device**. For MTP devices, this MUST always be 0x00000006.

MTPVendorExtensionVersion (2 bytes): A 16-bit unsigned integer. Indicates the version, in hundredths, of the extension to PTP used by the device. For devices that implement version 1.0 of the MTP specification, this value MUST always be 0x0064 (100). This field is named **MTPVersion** in the **DeviceInfo** dataset definition of the MTP specification.

MTPVendorExtensionDesc (variable): An MTP string that contains the name and version of each MTP extension implemented by the device. This string MUST be in a specific format. The order of extension descriptions in the string is not important. For devices that implement only the MTP

specification, the string is "microsoft.com: 1.0; ". For devices that implement AAVT, this string includes "microsoft.com/AAVT: 1.0; " in addition to "microsoft.com: 1.0; ". This field is named **MTPExtensions** in the **DeviceInfo** dataset definition of the MTP specification. For more information about MTP string types, refer to section Strings in the MTP specification.

OperationsSupported (variable): An array of MTP operation codes that the device supports. This array MUST include the operation codes for all supported extensions to MTP in addition to MTP and PTP core operations. For devices that implement AAVT, this array includes the values 0x9170, 0x9171, and 0x9172. The value 0x9173 can optionally be included to indicate support for time-based seeking. For more information about MTP array types, refer to section Arrays in the MTP specification.

2.3 Protocol Error Codes

The following error codes are currently defined for WMDRM: Network Devices Protocol. These error codes MAY be returned by the transmitter. Section 2.1.1.1.3 specifies how the error codes are mapped to **UPnP**messages, Section 2.1.1.2.6 specifies how they are mapped to HTTP, and section 2.1.1.3.4 specifies how they are mapped to **RTSP**.

Return value/code	Description
0x0064 Invalid Certificate	The receiver's certificate is corrupted or its signature cannot be verified.
0x0065 Certificate Revoked	A hash of the receiver's certificate is listed in the transmitter's CRL.
0x0066 Must Approve	The receiver MUST be approved before it MAY execute the intended operation.
0x0067 License Unavailable	The transmitter does not hold a license for the requested action.
0x0068 Transmitter Failure	The transmitter encountered a generic failure.
0x0069 Device Limit Reached	The transmitter cannot execute the requested action because the number of devices in use has already reached the maximum permitted.
0x006A Unable to Verify Proximity	The proximity detection procedure could not determine that the receiver is near the transmitter.
0x006B Must Register	The receiver MUST be registered before it MAY execute the intended operation.
0x006C Must Revalidate	The receiver MUST be revalidated before it MAY execute the intended operation.
0x006D Invalid Proximity Response	The receiver has responded to the proximity challenge message incorrectly.
0x006E Invalid Session	The session is no longer available.
0x006F Unable to Open File	The transmitter cannot open the requested file.

Return value/code	Description
0x0070 Unsupported Protocol Version	The request received by the transmitter is based on an unsupported protocol version.
0x0071 Bad Request	The format of the request is not valid.

2.4 Common Requirements

This section specifies common cryptographic and protocol requirements for the WMDRM: Network Devices Protocol.

2.4.1 Cryptographic Semantics

The following table describes the cryptographic notation that is used throughout this document.

Symbol	Description
MAC	Message authentication code.
OMAC	Block **cipher**-based message authentication code
|	Concatenation.
{text}M1	Text encrypted with M1's **public key**. Asymmetric key **encryption**.
[text]M1	Text signed with M1's **private key**. Asymmetric key signature.
K{text}	Text encrypted with **secret key** *K*, using **symmetric key** encryption.

2.4.2 Cryptographic Requirements for Receivers

The following security requirements MUST be implemented by **receivers**:

- The block **cipher** is **AES** with 128-bit keys supporting Electronic Code Book (ECB) and **CTR** [FIPS197].
- The **public key** cryptography is 1024-bit RSA [PKCS1]. All receivers are required to have a **private key** and it MUST be stored in accordance with the rules detailed documented in [CR-WMDRM].
- The hashing algorithm is Secure Hashing Algorithm 1 (SHA-1) [FIPS180-2].
- The **Hash-based Message Authentication Code (HMAC)** is AES One-Key Cipher Block Chaining MAC 1 (OMAC1) [OMAC].
- The receiver certificate and its **certificate chain** MUST be stored in a manner consistent with the rules called in [CR-WMDRM].The receiver playback and output control MUST comply with the rules documented in [CR-WMDRM].

2.4.3 Cryptographic Requirements for Transmitters

The following security requirements MUST be implemented by transmitters:

- The block **cipher** is **AES** with 128-bit keys, supporting Electronic Code Book (ECB) and **CTR** [FIPS197].
- The **public key** cryptography is 1024-bit RSA [PKCS1].
- The hashing algorithm is Secure Hashing Algorithm 1 (SHA-1) [FIPS180-2].
- The **HMAC** is AES One-Key Cipher Block Chaining MAC 1 (OMAC1) [OMAC].
- The Microsoft Root Public Key MUST be stored in a manner that is consistent with the [CR-WMDRM] for transmitters.
- The **certificate revocation list (CRL)** MUST be stored in a manner that is consistent with the [CR-WMDRM].
- The random number generator used to generate the encrypted seed MUST be cryptographically random in a manner that is consistent with the [CR-WMDRM].
- A secure and accurate timer MUST be used for **proximity detection**. It MUST be accurate to the millisecond.

2.4.4 Requirements for Receivers

Receivers MUST implement the protocol procedures described in section 3.2.

Receivers SHOULD also implement a **UPnP** control point for the UPnP Connection Manager **service** (ConnectionManager:1) [UPNPCNMGR], and a UPnP control point for the UPnP Content Directory service (ContentDirectory:1) [UPNPCDS1] so that they MAY retrieve **content** information from transmitters.

The ConnectionManager:1 [UPNPCNMGR] and ContentDirectory:1 [UPNPCDS1] services are defined by [UPNPCNMGR] and [UPNPCDS1], respectively. The X_MS_MediaReceiverRegistrar:1 service is defined in section 3.4.

Receivers MUST implement the mapping of the **registration** and revalidation procedures to UPnP, which is defined in section 2.1.1.1. Receivers are recommended to implement the mapping of the authorization procedure to UPnP, which is defined in section 2.1.1.1.1.1.

If a receiver implements WMDRM: Network Devices Protocol for HTTP, the implementation MUST follow the rules that are defined in section 2.1.1.2 for receivers.

If a receiver implements WMDRM: Network Devices Protocol for **RTP** [RFC2250] and **RTSP** [RFC2326], the implementation MUST follow the rules that are defined in section 2.1.1.3 for receivers.

2.4.5 Requirements for Transmitters

Transmitters MUST implement the protocol procedures in section 3.3 and follow all rules in that section that apply to transmitters.

Transmitters SHOULD also implement the **UPnP** Connection Manager **service** (ConnectionManager:1) and UPnP Content Directory service (ContentDirectory:1) so that receivers MAY retrieve **content** information from transmitters.

The ConnectionManager:1 and ContentDirectory:1 services are defined by [UPNPCNMGR] and [UPNPCDS1], respectively. The X_MS_MediaReceiverRegistrar:1 service is defined in section 3.4

Transmitters MUST implement the mapping of the registration and **revalidation** procedures to UPnP, which are defined in section 2.1.1.1. Transmitters are recommended to implement the mapping of the authorization procedure to UPnP, which is defined in section 2.1.1.1.1.2.

If a transmitter implements WMDRM: Network Devices Protocol for HTTP [RFC2616] the implementation MUST follow the rules that are defined in section 2.1.1.2.

If a transmitter implements WMDRM: Network Devices Protocol for **RTP** [RFC3550] and RTSP [RFC2326], the implementation MUST follow the rules that are defined in section 2.1.1.3.

3 Protocol Details

The following components are an integral part of the WMDRM: Network Devices Protocol and are used during the procedures described in the following sections.

3.1 Common Details

3.1.1 Abstract Data Model

3.1.1.1 Overview

The WMDRM: Network Devices Protocol defines the following procedures:

Authorization: The procedure for granting a **Receiver** access to **content** from a transmitter. This procedure is required before a receiver can perform **registration** and access protected content.

Registration and Revalidation: Allows the transmitter to uniquely identify a receiver that is located nearby. Revalidation is the procedure for automatically reregistering a receiver with the transmitter after a certain amount of time has elapsed. Registration and **revalidation** are very similar, so for the purposes of this document, they will be grouped together in one category. Differences will be pointed out as necessary.

Proximity Detection: Measures the latency between the transmitter and a receiver. If the latency is short enough (less than 7 milliseconds), then the receiver is considered to be **near** the transmitter. The transmitter will not send protected content to a receiver unless it is near.

License Retrieval: The procedure for sending a **license** to a registered Receiver when it requests content from a transmitter.

Data Transfer: Performed between a transmitter and a receiver after a license has been sent by the transmitter and enforced by the receiver. The content is encrypted and delivered using HTTP [RFC2616] or **RTP** [RFC3550].

License Management: For this procedure, the transmitter sends an updated license to the receiver to change the receiver's rights to a **resource**.

Revocation List Update: This procedure is permitted only after the registration procedure has successfully completed. The receiver asks the transmitter for an updated **revocation list**. This procedure MAY be initiated at any time after the registration procedure has occurred.

Authorization is performed after a receiver is discovered by a transmitter. Registration/revalidation is performed every 48 hours. The **license retrieval** and data transfer procedures occur every time a **file** is requested and played.

All of the procedures rely on messages--or sets of parameters--that are passed back and forth between a transmitter and a receiver. Messages are mapped to the HTTP,**RTSP**, and **UPnP** protocols. The following conceptual sections describe these procedures in further detail.

Transmitters can enforce a minimum revocation list when transferring content to receivers. These revocation lists can be for any protection system, including the following:

- Windows Media DRM for Network Devices **CRLs**
- Windows Media DRM for Portable Devices CRLs
- **Certified Output Protection Protocol (COPP)** driver CRLs

If the receiver does not yet have the required version of the revocation list, it can request that the transmitter send it. This occurs through the revocation list request and revocation list response messages. These messages can be initiated at any time by the receiver; however, transmitters are not required to support this.

3.1.1.2 Authorization

Authorization is the procedure by which a receiver is granted access to content from a transmitter. This procedure is required before a receiver can perform registration and access protected content. The transmitter MUST only start data transfer of content to receivers that have been authorized.

A transmitter SHOULD perform the authorization procedure when the transmitter discovers a new receiver that supports WMDRM: Network Devices Protocol. For example, authorization can occur when the transmitter detects a receiver sending **SSDP** messages (**ssdp:alive**); authorization can also occur when an unknown new receiver attempts to perform the registration or **license retrieval** procedures. It is recommended that transmitters maintain a record of authorized devices. A transmitter SHOULD NOT attempt authorization with receivers that do not support WMDRM: Network Devices Protocol.

The authorization procedure is implementation-specific and not defined by this specification. For example, the authorization procedure can be implemented by requiring explicit approval from the user of the transmitter to grant each receiver access to content and metadata.

As another example, authorization can be defined by the registration and **revalidation** procedure in conjunction with **proximity detection**.

The user MAY also de-authorize a receiver. This will remove the receiver's access to content and metadata on the transmitter.

UPnP authorization specifies how the authorization procedure is mapped to UPnP. When WMDRM: Network Devices Protocol is used over IP networks, transmitters and receivers SHOULD implement the mapping of authorization to UPnP.

3.1.1.3 Registration and Revalidation

The registration and **revalidation** procedures are essential parts of WMDRM: Network Devices Protocol. If a receiver successfully completes registration or revalidation, it is said to be registered.

Registration and revalidation are required before receivers can retrieve content from the transmitter. Registration and revalidation allow the transmitter to identify a receiver through a unique combination of an **XML** certificate and a 128-bit serial number. This combination can be unique in two ways: either multiple receivers have different serial numbers but the same certificate, or multiple receivers have the same serial number but different certificates. For example, a manufacturer might want all of its receivers to have the same certificate. In this case, each Receiver would then need a unique serial number so that the combination of serial number and certificate would uniquely identify it. And, if the receiver uses a unique certificate, its serial number MUST be zero.

If a receiver's certificate has been revoked, it will not be allowed to register, revalidate, or start a data transfer. For more information on certificates, see the Machine Certificate Specification [XMR]. For more information on **CRLs**, see section 2.2.1.5.3.

Revalidation differs from registration in that the receiver is already registered with the transmitter. During revalidation, the registration and **proximity detection** procedures are performed again, and the last revalidation time is updated. The transmitter MUST enforce revalidation by ensuring it occurs at least once every 48 hours.

If a receiver has not successfully revalidated itself in 48 hours and is receiving content when that limit is reached, the transmitter MUST stop the data transfer to the receiver. Receivers SHOULD revalidate themselves at least once every 48 hours to avoid a possible interruption during playback.

The following diagram shows the registration procedure:

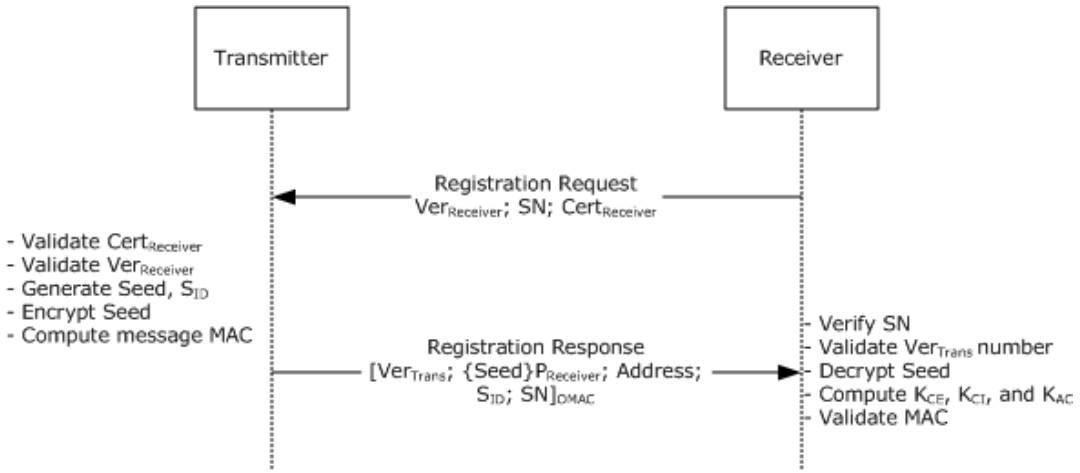


Figure 4: Registration Procedure Flow

The following table describes elements of the preceding diagram:

Value	Description
SN	128-bit serial number.
CertReceiver	Receiver's **device** certificate.
VerReceiver	WMDRM-ND protocol version that is implemented by the receiver.
VerTrans	WMDRM-ND protocol version that is implemented by the transmitter.
PReceiver	**Public key** of the receiver.
{Seed}PReceiver	128-bit seed that is encrypted with the public key of the receiver. The seed is used to derive the **content encryption key**, **content integrity key**, and **authenticated commands** key.
KCE	128-bit content encryption key that is derived from the encrypted seed. This key is different from the one used in data transfer.
KCI	128-bit content integrity key that is derived from the encrypted seed. This key is different from the one used in data transfer.
KAC	128-bit authenticated commands key that is derived from the encrypted seed. This key is used to sign the authenticated commands message.
Address	IP address of the transmitter's incoming and outgoing proximity packets socket.
Session ID	128-bit random session identifier.
OMAC	OMAC1 [OMAC] of the message using the content integrity key.

The syntax of the registration messages is defined in section 2.1.1.

The mapping of registration to **UPnP** is specified in section 2.1.1.1.2.

3.1.1.4 Proximity Detection

In **proximity detection**, the transmitter and receiver **exchange** messages in an attempt to determine the latency between them. If the latency is low enough (less than 7 milliseconds), the

receiver is considered **near** the transmitter. The transmitter transfers licenses only to receivers that are near.

The proximity detection procedure is considered a part of the overall registration procedure. It is always performed after the registration response message is handled by the receiver. Proximity detection MUST be initiated after the successful completion of the registration procedure and after each **revalidation** procedure. The proximity detection procedure MUST have successfully completed before **license retrieval** can occur.

The following diagram illustrates proximity detection, which is part of the registration procedure:

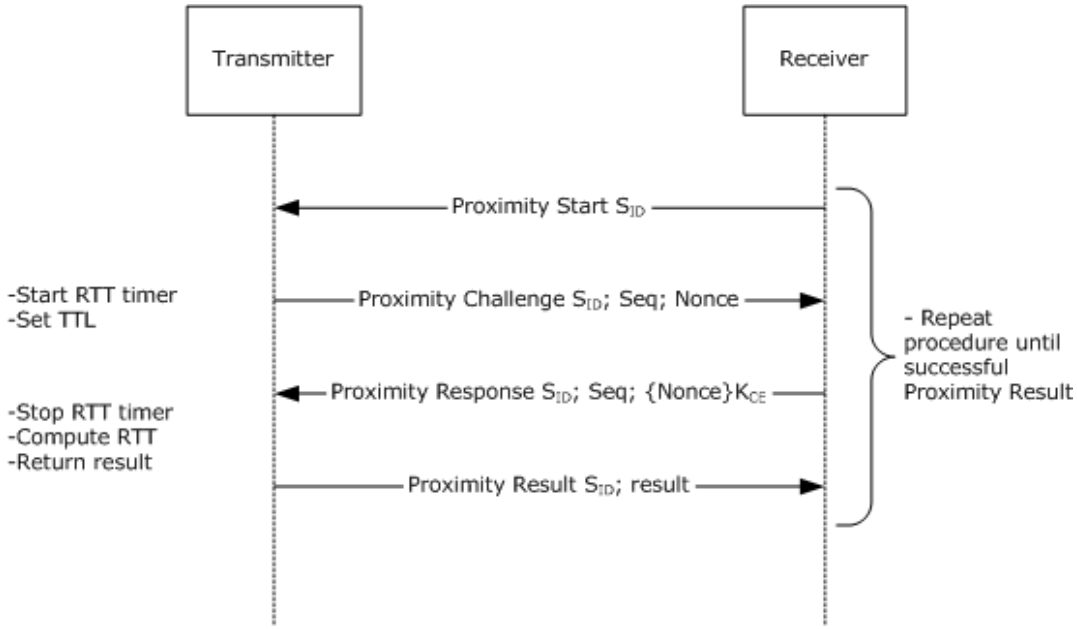


Figure 5: Proximity Detection Flow

The following table describes elements of the preceding diagram.

Value	Description
Session ID	128-bit random session ID.
Seq	8-bit incremental sequence number.
Nonce	128-bit random value.
{Nonce}KCE	**Nonce** encrypted using the **content encryption key**.

The format of the proximity detection messages is defined in section 2.1.1.4.2.

3.1.1.5 License Retrieval

After authorization and registration have been completed and before a receiver can begin receiving data, the receiver MUST receive a license from the transmitter. This is called the **license retrieval** procedure. The license retrieval procedure MUST occur every time a new session is initiated by the receiver. The concept of a session in this context is specific to the control protocol for data transfer (such as HTTP or **RTSP**).

To perform license retrieval, the receiver MUST send the transmitter a license request message. The message notifies the transmitter that the receiver is trying to play the content.

A license response message from the transmitter MUST contain a license in XMR format [XMR], which is defined in [XMR]. In most cases, this license will be either a **root license** or a **base license**. If the license is for delivery of **ASF** files over HTTP, a **standard license** is used. Whenever a root license is used, subsequent **leaf licenses** MUST be delivered to the receiver through the **license update** procedure defined in section 2.2.1.4.3.

If the transmitter and receiver are using license chains, the transmitter will send a root license to the receiver. Root licenses are not intended to impose restrictions, so at least one leaf license also MUST be sent by the transmitter. The leaf license is sent as part of a license update during the data transfer procedure. The root and leaf licenses form a license chain that contains the current set of rights and restrictions. The receiver MUST adhere to the most restrictive policies contained in the chained licenses, whether those policies come from the root license or the leaf license. For more information on root and leaf licenses, see 3.2.5.5 and 3.2.5.5.

If the transmitter and receiver are performing license management using license derivation data, a base license MUST be generated. A base license is a convenient, secure way to establish protected content **flow** between the transmitter and receiver. A base license is an optimization that allows a single license to be generated by the transmitter per registration or **revalidation** interval.

Base licenses are not used to encrypt or decrypt content. Instead, base licenses can be used only to derive other licenses. Base licenses are not intended to impose restrictions, so at least one license derivation data message MUST be sent by the transmitter as part of a license update prior to the data transfer procedure.

A base license will be generated and transferred each time a DRM Pairing process takes place. This means that a base license is generated at least every 48 hours during content flow.

After the receiver successfully completes the license retrieval procedure, it is ready to begin receiving data from the transmitter.

Unlike the registration request and registration response messages, the license request and license response messages MUST NOT be Base64 encoded or decoded [RFC3548].

The following diagram shows the license retrieval procedure:

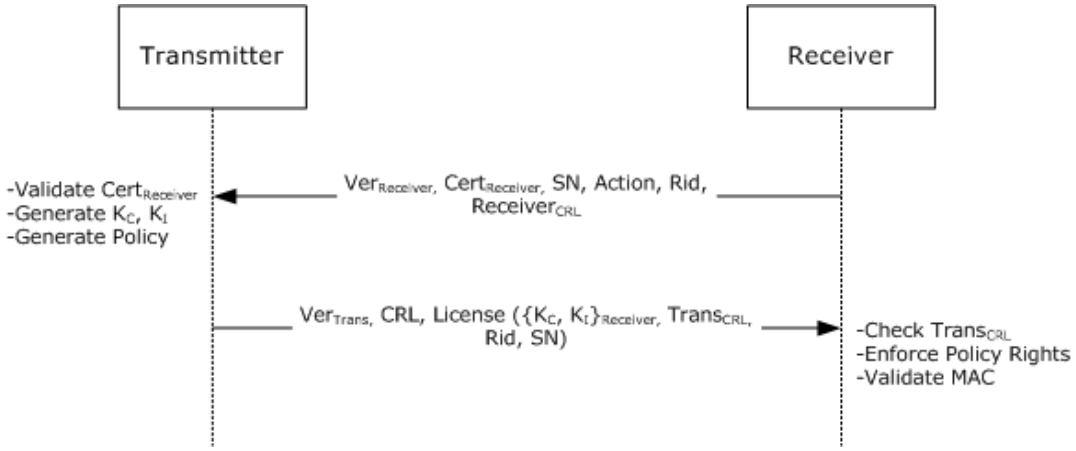


Figure 6: License Retrieval Flow

The following table describes elements of the preceding diagram.

Value	Description
SN	128-bit serial number.

Value	Description
CertReceiver	Receiver's **device** certificate.
VerReceiver	WMDRM-ND protocol version implemented by the receiver.
Action	The action requested by the receiver (i.e., Play).
VerTrans	DRM protocol version implemented by the transmitter.
License (...)	XMR license that contains **policy (2)** rights for the requested content.
{KCE, KCI}PReceiver	128-bit content encryption and **content integrity Key**s encrypted with Receiver's **public key**.
PReceiver	Public key of the receiver.
TransCRL	32-bit **CRL** version number held by the transmitter.
RId	128-bit random Rights ID.
ReceiverCRL	32-bit CRL version number held by the receiver.
CRL	The CRL of the transmitter.

The structure and flow of the license retrieval messages is defined in section 2.2.1.4.

The mappings of the license retrieval procedure to HTTP and RTSP are specified in sections 2.1.1.2.2 and 2.1.1.3.1 respectively.

The mappings of the license retrieval procedure to the OpenCable Digital Receiver Interface [OCDRIP] are specified in section 2.1.1.4.3.

3.1.1.6 Data Transfer

After the receiver has completed the **license retrieval** procedure, it is ready to begin receiving content from the transmitter. This is referred to as the data transfer procedure.

If the license obtained during the license retrieval procedure is a **root license**, the receiver MUST also receive a **leaf license** before it can process the content. The leaf license is delivered at the start of the data transfer procedure, through the license update mechanism described in section 2.2.1.4.3.

The transmitter MUST encrypt the content using **AES** in counter mode with the **content encryption key** before it is delivered to the receiver using HTTP or **RTP**. Section 2.1.1 defines how various types of content MUST be encrypted and delivered over HTTP and RTP.

The mappings of the data transfer procedure to HTTP and **RTSP** are specified in HTTP Data Transfer and section 2.1.1.3.5.

3.1.1.7 License Management

If the transmitter and receiver are performing license management using license chains, the transmitter will send the receiver a **root license**. If the transmitter sends the receiver a root license during the **license retrieval** procedure, then it MUST send a **leaf license** to the receiver. The process of delivering a leaf license to the receiver is called license management using license chains, because the rights and restrictions are formed through the chained **policy (1)**.

The root and leaf licenses form a license chain that contains the current set of rights and restrictions. The receiver MUST adhere to the most restrictive policies contained in the licenses, whether those policies come from the root license or the leaf license.

Sections 3.2.5.5.1 and 3.3.5.5.1 specify rules and requirements for root and leaf licenses that MUST be followed during license management using license chains.

If the transmitter and receiver are performing license management using license derivation data, a **base license** MUST be created.

Sections 3.2.5.5.2 and 3.3.5.5.2 specify rules and requirements for base licenses that MUST be followed during license management using license derivation data.

The transmitter MAY perform the license management procedure multiple times. This allows the policies that are associated with the content to be updated dynamically.

If content is being transferred over HTTP and the content is not encapsulated in **ASF** [ASF], then the transmitter MUST implement license management as specified in this section

When **RTSP** is used, license management is implemented by delivering the license update messages using ANNOUNCE requests, as specified in section 2.1.1.3 RTSP License Management.

When MPEG TS is used, license management is implemented by delivering the license update message using TAG packets.

Revocation List Update

A revocation list identifies software or hardware components to which protected content MUST no longer **flow** . Different content protection systems have different formats for revocation lists. WMDRM: Network Devices Protocol supports distributing revocation lists for different content protection systems through the **revocation list update** procedure. The receiver MAY use this procedure to obtain the latest version of the **RIV** structure. The receiver MAY use this procedure to obtain the latest revocation list for different content protection systems.

Section 2.2.1.6 includes the various content protection systems for which revocation lists might be available. The RIV GUIDS section specifies the **GUID** used to identify revocation lists for each system. RIV Structure also defines a GUID for the RIV structure. The receiver MUST use that GUID when it sends a revocation list request message to obtain the latest version of the RIV structure.

Revocation list update is not supported for transmitters acting as an **OpenCable Unidirectional Receiver (OCUR)**.

Authenticated Commands: The **authenticated commands** procedure allows sending cryptographically signed commands from the receiver to the transmitter. The authenticated commands are needed to control the **PBDA** hardware tuner.

3.1.2 Timers

None.

3.1.3 Initialization

None.

3.1.4 Higher-Layer Triggered Events

None.

3.1.5 Message Processing Events and Sequencing Rules

3.1.5.1 Data Encoding and Decoding

Base64 Encoding and Decoding: The standard base64 encoding algorithm (as specified in [RFC4648]).

The characters used in base64 encoding do not include any of the special characters of the **Simple Mail Transfer Protocol (SMTP)** [RFC2821], or the hyphen used with Multipurpose Internet Mail Exchange (MIME) boundary strings, as specified in [RFC2045].

Base64 Mapping Table: The base-64 mapping table is defined as follows.

0 A	17 R	34 i	51 z
1 B	18 S	35 j	52 0
2 C	19 T	36 k	53 1
3 D	20 U	37 l	54 2
4 E	21 V	38 m	55 3
5 F	22 W	39 n	56 4
6 G	23 X	40 o	57 5
7 H	24 Y	41 p	58 6
8 I	25 Z	42 q	59 7
9 J	26 a	43 r	60 8
10 K	27 b	44 s	61 9
11 L	28 c	45 t	62 +
12 M	29 d	46 u	63 /
13 N	30 e	47 v	
14 O	31 f	48 w	
15 P	32 g	49 x	
16 Q	33 h	50 y	

3.1.5.2 Content Encryption

This section describes how content is encrypted for various content types.

3.1.5.2.1 Content Encryption for AES in Counter Mode

The **AES** in counter mode [FIPS197] encryption is applied to separate segments of content data. The definitions for data segment, data segment ID, and block ID MAY vary for different content types and are further discussed in later sections.

Each data segment MUST be encrypted using AES in counter mode [FIPS197]. The following diagram illustrates the procedure for encrypting a data segment using this technique.

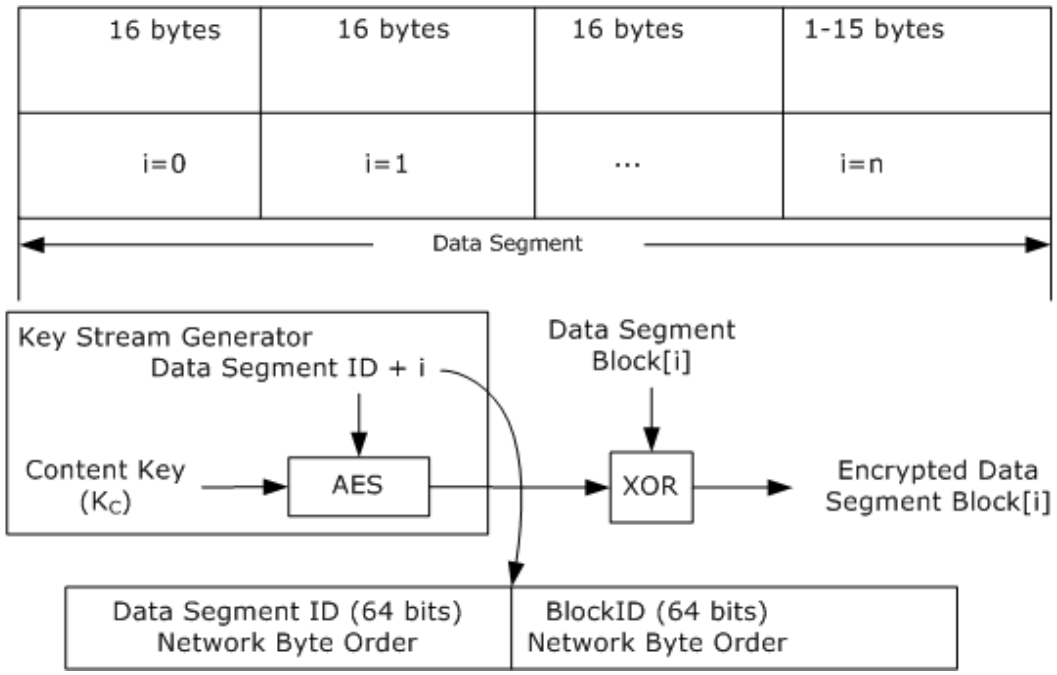


Figure 7: Data Segment Encryption Using AES Counter Mode

AES in counter mode creates a stream of bytes that MUST be XOR'd with the clear text bytes of the content to create the encrypted content. The key stream generator MUST use an AES round to generate 16-byte blocks of key stream at a time. The inputs to the AES round MUST be the **content encryption key** (KC) and the 128-bit concatenation of a data segment ID and the block ID within the data segment.

The output of the key stream generator MUST be XOR'd byte by byte with the data from the corresponding block (i) of the data segment. In case the data segment is not evenly divisible by 16 bytes, only the remaining bytes of the data segment from the last block MUST be XOR'd with the key stream and retained for the encrypted data segment.

The data segment ID values MUST be unique for a given KC.

3.1.5.2.2 Bulk AES Counter Mode

Bulk **AES** counter node is a simple counter encryption of the payload that uses a stream global AES counter. The transmitter MAY default to this mode on any given **packet identifier (PID)** if the receiver has not specified a mode.

With bulk AES counter mode, transport stream packets are encrypted by encrypting the 184 bytes after the first fixed 4 bytes of the transport stream header. If the transport stream packet has an adaptation field, that information is encrypted as well. The 184 bytes will use 11.5 AES encryption blocks. For bulk mode, the extra trailing 8 bytes of the AES counter mask that is not used in the XOR combining operation is to be retained and used at the beginning of the next transport stream packet, so that the processing of two contiguous packets will use 23 AES encryption blocks, as shown in the figure below.

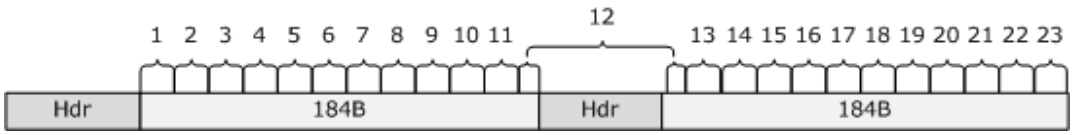


Figure 8: AES Encryption Block

Content encrypted using bulk AES counter mode is less efficient for PVR style usage. It requires that the receiver decrypt the content and potentially re-encrypt it at capture time, in addition to decrypting it at playback time. Streaming AES counter mode on audio/video PIDs saves the capture time decrypt and re-encrypt, but it requires more resources within the tuner to support, as such encryption needs to track per-PID state that is enabled for Streaming AES counter mode.

As such, bulk AES mode is used whenever Streaming AES counter mode is fully consumed, or for any PID that requires significant decryption at capture time.

As a general rule the receiver SHOULD prioritize the use of Streaming AES counter mode to PIDs that contain audio/video content. In addition, if the transmitter performs sufficient content screening it MAY also make this decision and automatically put audio/video content into Streaming AES counter mode.

In any case the transmitter is required to support at least 8 Stream AES counter mode PIDs.

3.1.5.2.3 The Data Segment Descriptor

The data segment descriptor is an auxiliary data structure that is used to represent encryption parameters. Since it is not utilized with all content types, its usage will be noted in the appropriate sections.

The following table describes the format of the data segment descriptor.

0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1
Flags								ExtensionType								ExtensionLength															
Extension (variable)																															
...																															

Flags (1 byte): The **Flags** section indicates the attributes of the media data associated with the data segment descriptor. The following values are currently defined:

0x01 (Encrypted Data): Indicates that the media data is encrypted.

0x00 (Unencrypted Data): Indicates that the media data is unencrypted.

ExtensionType (1 byte): The type of extension. There are two support extension types:

Key ID: The extension type MUST be set to 0x01. The length MUST be set to 16. This means the length is 16 bytes, or 128 bits. The extension MUST contain the key ID value of the license for the encrypted data that is delivered with the data segment descriptor. This extension MUST only be used when the encrypted data flag is set.

AES128 Initialization Vector: The extension type MUST be set to 0x02. The length can be set to 8 or 16. This means the length is 8 bytes or 16 bytes (64 bits or 128 bits, respectively).

If the length is 64 bits, then the extension contains only the data segment ID value for the encrypted data that is delivered with the data segment descriptor. In this case, the block ID value is 0 and is not included in the extension.

If the length is 128 bits, then the extension contains both the data segment ID and the block ID values, in this order.

This extension MUST only be used when the encrypted data flag is set.

The receiver MUST silently disregard and skip any extensions that it does not understand.

ExtensionLength (2 bytes): The total length of the extension, in bytes. This value does not include the size of the content data associated with this extension.

Extension (variable): The extension data. The type of data is variable depending on the extension type.

This extension section contains the number of extensions included in the data segment descriptor followed by multiple extension sections with the following format for each extension: starting with an 8-bit extension type, followed by a 16-bit extension length, and finally the variable-length extension.

3.1.5.2.4 Transport Stream

The **transmitter** supplies the following transport stream headers bits as described below. The full mpeg2 transport stream header is as defined in ISO/IEC 13818-1 [ISO/IEC-13818-1]. This section clarifies the usage of certain bits in the context of **PBDA**.

The transport stream header is never encrypted (first four bytes).

The two **transport_scrambling_control** bits MUST be set in one of the following ways:

Value	Meaning
0x00	The transport stream packet is not encrypted.
0x01	The trailing 184 bytes of the transport stream packet is encrypted using Bulk **AES** Counter Mode.
0x02	The Transport Stream Payload is byte encrypted with Streaming AES Counter Mode. The payload bytes are the bytes in the transport stream packet after the adaptation field.
0x03	Reserved. MUST NOT be used.

3.1.5.2.5 Transmitter Bulk Mode AES Block usage

The **transmitter** MUST mark all transport stream packets encrypted this way with **transport_scrambling_control** set to the value 0x1.

If the transmitter is protecting any PID using Bulk **AES** Counter Mode, the transmitter periodically inserts a **Stream Info Table** that supplies the **KeyId** and the AES Counter Value synchronization. It is recommended this be at least 1 re-synchronization per second.

3.1.5.2.6 Streaming AES Counter Mode

The **transmitter** is required to leave some transport stream packets in the clear on PIDs that are protected via Streaming **AES** Counter Mode and for which contain either the 4th byte of an enabled start code sequence or one or more bytes of a start code payload. The payload size is configured by a **CipherDelay** count on each enabled start code. In addition, the transmitter is required to include a **Stream Info Table** in front on every packet containing one or more detected start codes that are enabled.

The transmitter MUST ensure that no counter / offset is reused when streaming from any given content key. To accommodate this, the transmitter MAY increment the AES counter with each transition from unencrypted to encrypted content, and clear the lower 64 bits every time it puts a new count into the upper 64 bits.

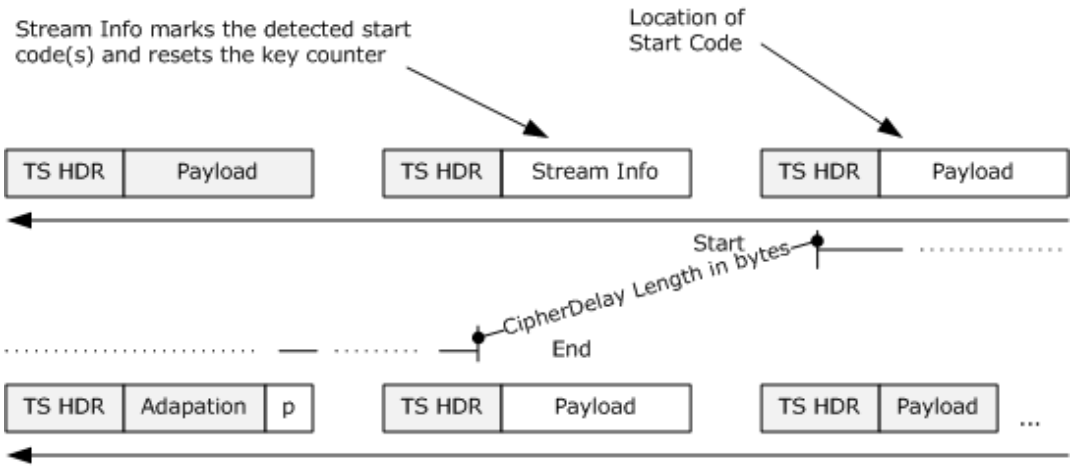


Figure 9: Streaming AES Counter Mode

Start Code Detection and Cipher Delay: The transmitter MUST mark all transport stream packets encrypted using streaming AES counter mode with the **transport_scrambling_control** set to the value 2.

The transmitter MUST use AES Counters that leaves bit 127, of the 128 bit counter, as zero.

3.1.5.2.6.1 AES Payloads

For PIDs that are configured to use Streaming **AES** Counter Mode encryption, the **transmitter** encrypts only the payload bytes of transport stream packets. The payload bytes start after the adaptation field if present; otherwise, after the transport stream header.

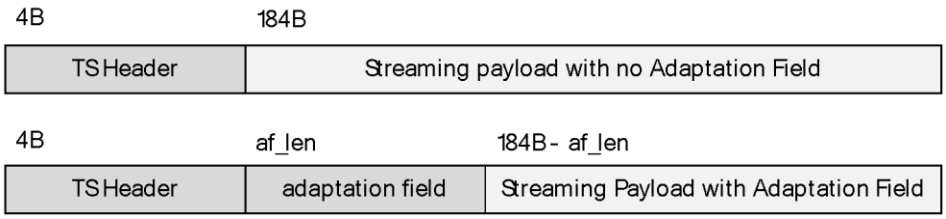


Figure 10: AES Payload Bytes

The transmitter MUST also preserve the partial AES mask such that for any 1 PID in Streaming AES Counter Mode the payload bytes are sequentially byte encryption. The following diagram illustrates the basic flow. First the transmitter sends an unencrypted **Stream Info (TAG) Table** that the **receiver** uses to synchronize which key is being used and what the current counter value is. As following transport stream packets arrive the payloads can be concatenated together into one buffer by the receiver and decrypted via a normal AES Counter Mode decrypt operation. As transport stream packets are not multiples of the AES key size, the transmitter preserves the unused parts of the **cipher** mask between transport stream packets on a per PID basis.

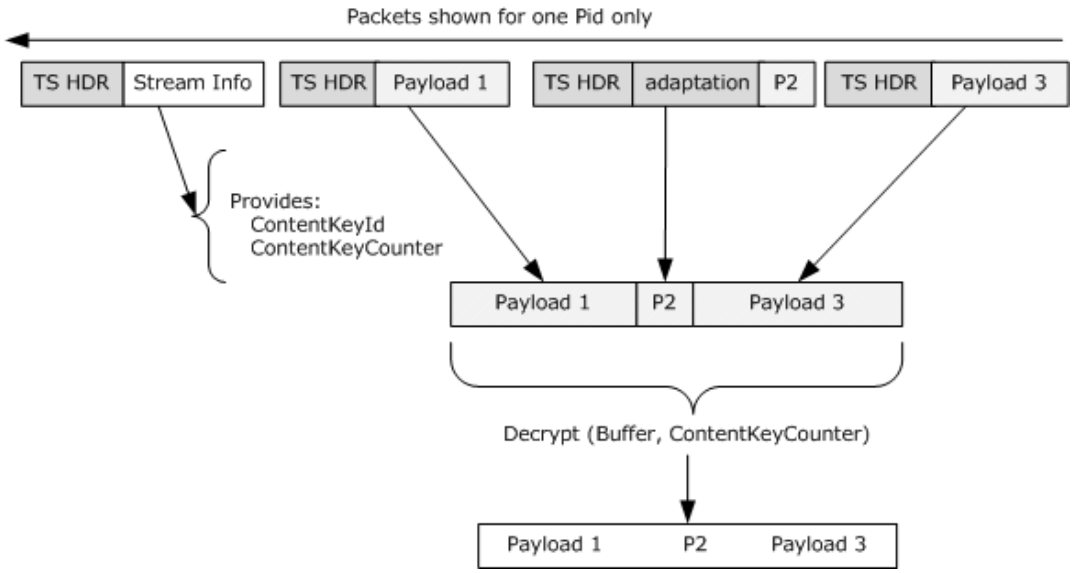


Figure 11: Single PID Structure

3.1.5.2.7 MPEG-2 Transport Stream Content

Encryption of content that is encapsulated in MPEG-2 Transport Stream (TS) [WMRTP] packets is achieved by encrypting the TS packet payloads with **AES** in counter mode, as defined in Counter Encryption for AES in Counter Mode [FIPS197], and using TAG packets that are inserted in the MPEG-2 TS.

A TAG packet is a single MPEG-2 TS packet with a Key Identifier (KID) that is inserted in front of each protected PES unit. Only PES units can be protected. The TAG packet is necessary to retrieve the matching DRM license when the content is delivered to the receiver.

The content protection layer consists of an AES 128-bit key in counter mode, where all the following requirements apply:

- The 128-bit counter is divided in two 64-bit fields: The **base_counter** (MSB) and the **minor_counter (LSB)**.
- The **base_counter** and **minor_counter** are equivalent to the data segment ID and block ID, as shown in the following diagram.

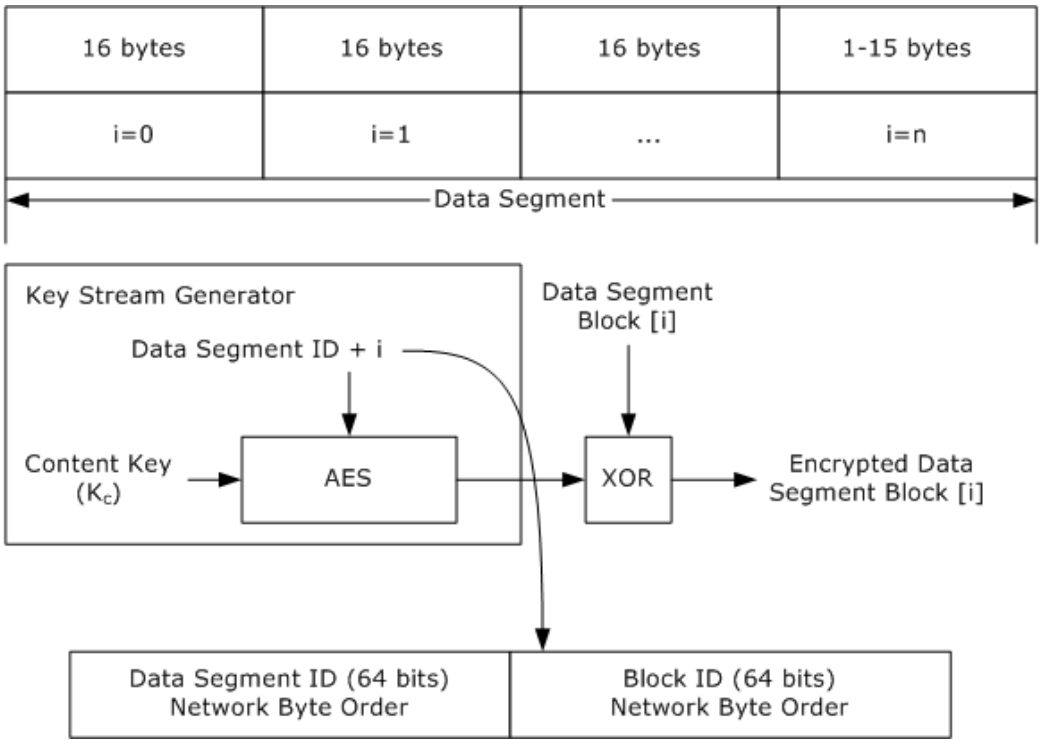


Figure 12: Base and Minor Counter Encoding

The transmitter MUST scramble the selected program PID according to the following flowchart:

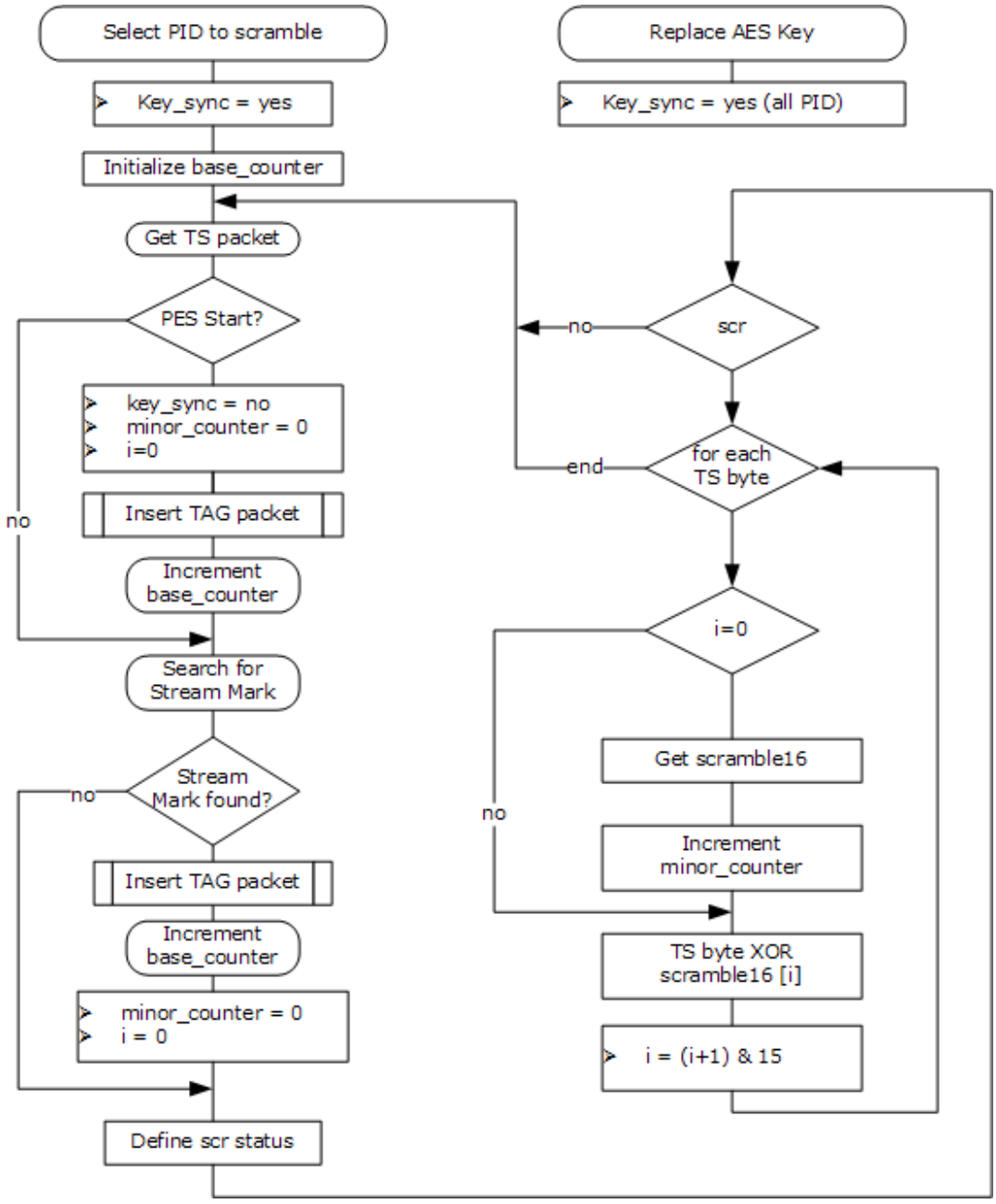


Figure 13: PID Obfuscation Flow

The following list describes the parts of the previous flowchart in more detail.

scr: This variable is set to "yes" if the current TS packet needs to be scrambled, or to "no" otherwise. The TS packet cannot be scrambled if any of the following conditions apply:

- The **key_sync** field is set to "yes".
- The TS packet includes whole or part of a PES header, which includes the 64 payload bytes that follow the detection of each start code, from 0xBC through 0xFF.
- The TS packet includes all or part of one or more of the stream marks listed in the following table. A stream mark is composed of an MPEG start code and its subsequent data payload.

Stream mark	Start code	Byte sequence	Maximum data payload length
Sequence header	B3	00 00 01 B3	12 bytes
GOP header	B8	00 00 01 B8	8 bytes
Picture header	00	00 00 01 00	6 bytes
Private data	B2	00 00 01 B2	107 bytes

Note Each MPEG-2 TS packet can only be scrambled or in the clear, but not partially scrambled.

The last byte of the start code MUST also be in the clear.

key_sync: This variable is set to "yes" if the transmitter is renewing the AES key, or to "no" otherwise.

base_counter: This 64-bit field is uniquely defined by the transmitter throughout the lifetime of the transmitter, as shown in the following diagram.

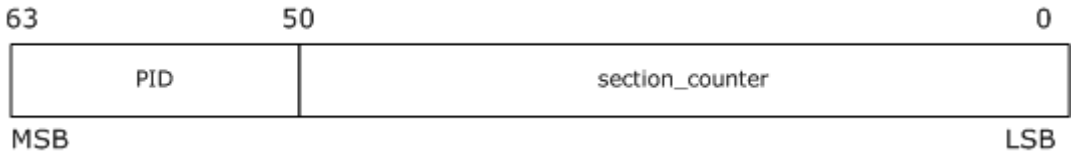


Figure 14: PID Structure

PID (13-bit): The PID value of the selected elementary stream.

section_counter (51-bit): A cyclic counter that is incremented for each no-to-yes transition of the scr state variable.

minor_counter: A 64-bit counter that is incremented for each block of 16 scrambled bytes.

i: A 4-bit counter that is incremented for each scrambled byte.

scramble16: AESKEY [base_counter | minor_counter].

The transmitter MUST insert a TAG packet in front of any TS packet left in the clear. Two possible scenarios MAY occur, as detailed in the following diagram.

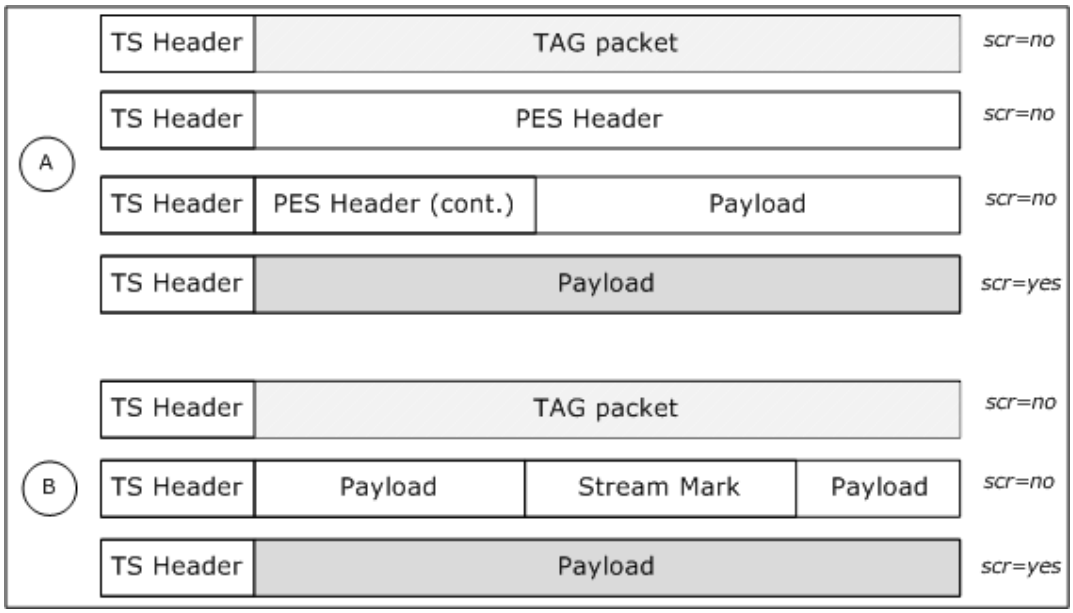


Figure 15: TAG Packet Scenarios

Case A: A TAG packet is inserted in front of a packet containing all or part of a PES header.

Case B: A TAG packet is inserted in front of a packet containing all or part of a stream mark.

A TAG packet has the following structure.

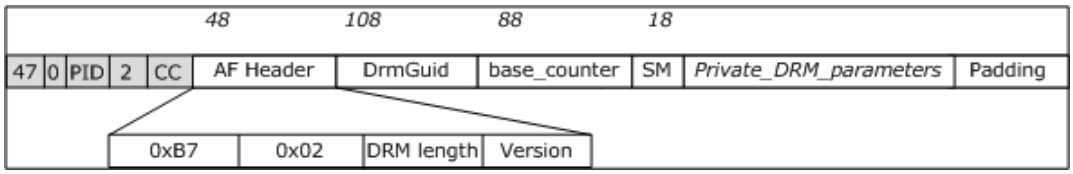


Figure 16: TAG Packet Structure

A TAG packet is a single MPEG-2 Transport Stream packet with a key identifier (KID) that is inserted in front of each protected PES unit. Only PES units can be protected. The TAG packet is necessary to retrieve the matching DRM license when the content is delivered to the receiver.

The **adaptation_field_control** bits (bits 27 and 28 in the TAG packet) are set to 10b; this indicates that the rest of the entirety of the TAG packet is for the adaptation field only, and includes no payload), so there is no requirement to increment the continuity counter.

- The AF Header consists of four bytes to be compliant with MPEG specification:
  - 1st Byte = Adaptation Field length
  - 2nd Byte = Adaptation Field presence flag (Private data = 0x02)
  - 3rd Byte = Private data length (Length, in bytes, of the **Private_DRM_parameters** portion of the packet. Values are variable based on the required length.)
  - 4th Byte = Version number (currently 0x00)
- **DrmGuid** contains the **GUID** that MUST be set to {B0AA4966-3B39-400A-AC35-44F41B46C96B}.
- The **base_counter** resynchronizes the AES counter for the encrypted packet that follows.

- SM byte (stream mark) indicates that the following packet includes the beginning of a stream mark, from which the first few bytes might be missing.
  - SM = 0 -- Next packet carries the beginning of a PES header or an entire PES header.
  - SM = 1 -- Next packet includes the beginning of a stream mark.
  - SM = 2 -- Next packet includes the beginning of a stream mark, from which the first byte (00) is missing.
  - SM = 3 -- Next packet includes the beginning of a stream mark, from which the first two bytes (00 00) are missing.
  - SM = 4 -- Next packet includes the beginning of a stream mark, from which the first three bytes (00 00 01) are missing.
  - SM = other -- Reserved.
- The **Private_DRM**_parameters MUST contain a license derivation data segment descriptor. The data segment descriptor MUST contain a Key ID extension set with the corresponding Key ID value. The AES128 initialization vector extension MUST not be present, since the data segment ID is already indicated in the base_counter section of the TAG packet.
- The remaining transport stream packet is padded with bytes with the value of 0xFF.

3.1.5.2.8 MPEG-2 Elementary Stream Content

The process of encrypting and decrypting MPEG-2 elementary stream (ES) content follows the procedure that is defined here. The data segment corresponds to a single MPEG-2 ES media access unit (MAU) as defined here.

Optionally, the transmitter MAY also leave portions of the data in the clear to facilitate the analysis of the content by the receiver. For a portion to be left in the clear, it MUST contain one of the following stream marks.

Stream mark	Start code	Byte sequence	Maximum data payload length
Sequence header	B3	00 00 01 B3	12 bytes
GOP header	B8	00 00 01 B8	8 bytes
Picture header	00	00 00 01 00	6 bytes
Private data	B2	00 00 01 B2	107 bytes

The amount of data to be left in the clear corresponds, in the table above, to the length of the stream mark plus the maximum data payload length. Notice, however, that the clear section MAY start prior to the stream mark and end after the combined length of the stream mark and maximum data payload length. This MAY occur as long as the combined length does not exceed 368 bytes (equivalent to the length of two consecutive MPEG-2 TS packet payloads).

For example, a transmitter MAY leave between 16 and 368 bytes in the clear for a stream mark that denotes a sequence header (4 bytes for the stream mark plus 12 bytes for the maximum data payload length).

It is also possible to have some amount of data from the previous MAU left in the clear, in case the stream mark appears **near** the beginning of the current MAU. That is allowed, as long as the length of the clear section does not exceed 368 bytes, as previously indicated.

3.1.5.2.9 ASF Sample Encryption Mode

This section specifies how receivers MUST decrypt content that is encapsulated in **Advanced Systems Format (ASF)** and has been encrypted with ASF sample encryption mode. The information in this section can also be used to implement a transmitter that creates such content. See [ASF] for details on the concepts and objects referred to in this section and subsections.

3.1.5.2.10 ASF Header Parsing

Receivers MUST identify the WMDRM: Network Devices Protocol encryption when parsing and examining the **ASF** header.

The Advanced Content Encryption object is used in the ASF header to communicate information about the WMDRM: Network Devices Protocol encryption in the ASF **file**. A content encryption record (in the Advanced Content Encryption Object) with a System ID of ASF_ContentEncryptionSystem_WMDRM_ND_AES indicates the presence of WMDRM: Network Devices Protocol encryption.

Receivers MUST perform the following steps to parse the ASF header:

1. **Identify encrypted streams:** A receiver SHOULD examine the stream properties objects present in the ASF header. For each stream that is marked with the encrypted content flag (in the stream properties object), the receiver will need to decrypt the stream. This document discusses only the **decryption** available in WMDRM: Network Devices Protocol; implementations that have other decryption available SHOULD be able to apply different mechanisms.

2. **Identify WMDRM: Network Devices Protocol content encryption records:** After the receiver has identified one or more streams that need decryption, and located stream numbers (in the stream properties objects), it MUST search for content encryption records in the advanced content encryption object. Applicable content encryption objects will have the system ID set to `ASF_ContentEncryptionSystem_WMDRM_ND_AES = {7A079BB6-DAA4-4e12-A5CA-91D38DC11A8D}` and the system version set to 1. A different system ID indicates that a different type of encryption is in place, and the content encryption record can be ignored. A higher system version indicates that the receiver is not compatible with the WMDRM: Network Devices Protocol encryption in this file, and MUST be treated as a failure case.

   For each suitable content encryption record, the receiver SHOULD attempt to match the record with the streams in the ASF file. Content encryption records contain encrypted object records, which identify what is encrypted. An encrypted object record with encrypted object ID type set to 0x0001 indicates that a stream is encrypted; other encrypted object records SHOULD be ignored. If the encrypted object ID type is 1, the **encrypted object ID** field contains either the special value 0x0000, or an array of 2-byte stream numbers. The special value 0x0000 indicates that every stream in the file is encrypted; otherwise the receiver SHOULD match the stream numbers with the stream numbers from the stream properties objects.

3. **Match streams and WMDRM: Network Devices Protocol records:** The receiver SHOULD have a match between content encryption records (in the Advanced Content Encryption object) and streams (per the Stream Properties objects). If there are any encrypted streams that are not matched, the receiver cannot decrypt this file and SHOULD take appropriate action. If a stream cannot be decrypted, the receiver SHOULD provide an informative error indicating that the receiver cannot access this content.

ASF Sample Parsing: After the ASF header has been parsed and a content key has been identified for each encrypted stream, the ASF packets in the data object of the ASF file MAY be parsed. The media object data in the ASF packets MUST be decrypted. All of the other ASF packet properties are not encrypted and can be parsed as usual.

The receiver MUST perform the following steps to parse the ASF packets:

1. **Reassemble the media objects:** With WMDRM: Network Devices Protocol, the complete media objects are encrypted, not individual ASF payloads. Thus, for fragmented media objects, the media objects MUST be reassembled into complete objects before applying the **AES** decryption.

2. **Retrieve the sample ID:** For each media object, the sample ID MUST be retrieved. The sample ID is stored as an ASF payload extension system, ID ASF_Payload_Extension_Encryption_SampleID = {6698B84E-0AFA-4330-AEB2-1C0A98D7A44D}. If this payload extension does not exist on a WMDRM: Network Devices Protocol encrypted stream, an error has occurred during authoring and the file SHOULD be treated as corrupted, and an appropriate error SHOULD be generated.

   The sample ID is an 8-byte value stored in **network byte order**. The sample ID MAY be stored as a fixed-size extension (8-bytes), or a variable size extension.

Decrypt the sample (media object): Each sample (complete media object) MUST be decrypted using AES in counter mode as defined here. Note that the decryption is applied per sample (referred to as a media object in the ASF specification [ASF]), rather than per ASF payload.

The process of encrypting and decrypting ASF samples follows the procedure that is defined in section 3.1.5.2.9. The data segment corresponds to the sample and the data segment ID corresponds to the sample ID from the ASF payload extension.

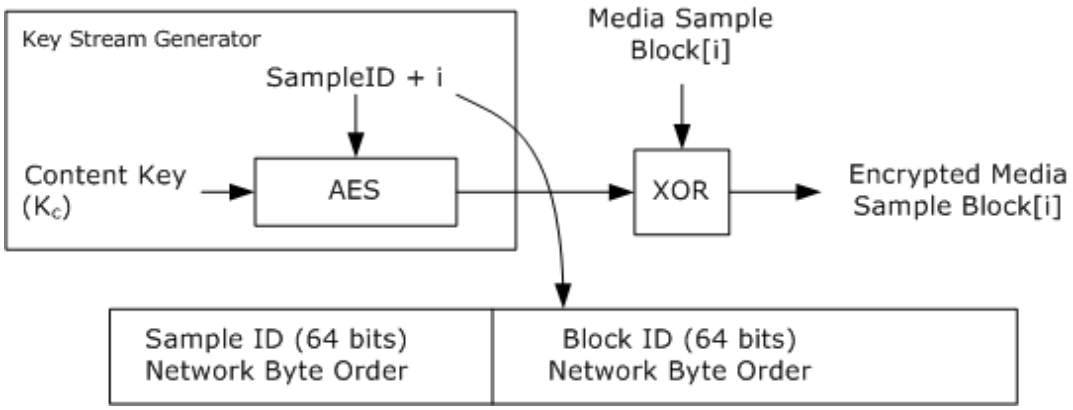


Figure 17: Encrypting and Decrypting ASF Samples

The pseudo-code is as follows:

```
SET SizeOfBlock to 16
SET SizeOfMediaSample to the number of bytes in the Media Sample
SET NumberOfBlocks to the size of the SizeOfMediaSample DIVIDED by SizeOfBlock
SET RemainderBlockSize = SizeOfMediaSample MODULO SizeOfBlock

FOR EACH Block in NumberOfBlocks
CALL DecryptBlock with Block and BlockSize RETURNING DecryptedBlock
ENDFOR

IF RemainderBlockSize NOT EQUAL to 0
CALL DecryptBlock with Block and RemainderBlockSize RETURNING DecryptedBlock
ENDIF
```

3.1.5.2.11 Link Encryption Mode

Link encryption mode is a generic content encryption mode for any content that is not explicitly described in this document. Link encryption mode is defined as a mode of operation which is

independent of the content type and specific to the data transport protocol (for example, HTTP or **RTP**).

The link encryption mode makes use of the data segment descriptor (section 3.1.5.2.3), which is attached to an encrypted or unencrypted data segment during data transfer. The data segment descriptor contains **flags** and extensions that notify the receiver when the media data is encrypted. The data segment descriptor contains ID values that are needed to decrypt the encrypted media data. A data segment descriptor is associated with a portion of the transmitted content to which, if encrypted, a single **policy (1)** and **content encryption key** applies. In other words, the content encryption key and policies cannot be changed within a data segment.

The data segment descriptor can be associated with content that is encrypted or unencrypted. The encrypted data flag in the descriptor indicates whether the content is encrypted.

3.2 Receiver Details

3.2.1 Abstract Data Model

3.2.1.1 Overview

Authorization: The procedure for granting a **Receiver** access to **content** from a transmitter. This procedure is required before a receiver can perform **registration** and access protected content.

Registration and Revalidation: Allows the transmitter to uniquely identify a receiver that is located nearby. Revalidation is the procedure for automatically reregistering a receiver with the transmitter after a certain amount of time has elapsed. Registration and **revalidation** are very similar, so for the purposes of this document, they will be grouped together in one category. Differences will be pointed out as necessary.

Proximity Detection: Measures the latency between the transmitter and a receiver. If the latency is short enough, then the receiver is considered to be **near** the transmitter. The transmitter will not send protected content to a receiver unless it is near.

License Retrieval: The procedure for sending a **license** to a registered receiver when it requests content from a transmitter.

Data Transfer: Performed between a transmitter and a receiver after a license has been sent by the transmitter and enforced by the receiver. The content is encrypted and delivered using HTTP [RFC2616] or **RTP** [RFC3550].

License Management: For this procedure, the transmitter sends an updated license to the receiver to change the receiver's rights to a **resource**.

Revocation List Update: This procedure is permitted only after the registration procedure has successfully completed. The receiver asks the transmitter for an updated **revocation list**. This procedure MAY be initiated at any time after the registration procedure has occurred.

Authorization is performed after a receiver is discovered by a transmitter. Registration/revalidation is performed every 48 hours. The **license retrieval** and data transfer procedures occur every time a **file** is requested and played.

All of the procedures rely on messages--or sets of parameters--that are passed back and forth between a transmitter and a receiver. Messages are mapped to the HTTP, **RTSP**, and **UPnP** protocols. The following conceptual sections describe these procedures in further detail.

Authenticated Commands: The **Authenticated Commands** procedure allows sending cryptographically signed commands from the WMDRM: Network Devices Protocol receiver to the

WMDRM-ND transmitter. The Authenticated Commands are needed to control the **PBDA** hardware tuner.

3.2.2 Timers

None.

3.2.3 Initialization

None.

3.2.4 Higher-Layer Triggered Events

None.

3.2.5 Message Processing Events and Sequencing Rules

3.2.5.1 Registration Details

After the receiver has successfully completed the authorization procedure, it MUST perform the registration procedure. This is not required if the receiver can determine that it successfully completed the registration procedure less than 48 hours ago. The receiver MUST perform registration using the following steps:

1. The receiver MUST send a registration request message to the transmitter. This message contains: the WMDRM: Network Devices Protocol version number that is supported by the receiver, its **device** certificate that MUST be signed by Microsoft (which includes a 1024-bit RSA **public key**), and a serial number.

2. The receiver MUST receive the registration response message.

3. The receiver MUST verify that the serial number in the registration response message matches the serial number assigned to the device. The receiver MUST also verify that the Windows Media DRM-ND protocol version number of the transmitter is compatible with the one that is used by the receiver. The receiver MUST decrypt the seed to generate the content **encryption**, content integrity, and **authenticated commands** keys. If the transmitter is a **PBDA** tuner, the receiver MUST maintain the authenticated commands key in order to utilize the authenticated commands feature. Then, the receiver MUST validate the OMAC [OMAC] of the registration response message.

4. If the protocol version numbers of the receiver and the transmitter are not the same, the receiver and transmitter MUST use the lowest common protocol version in subsequent messages.

5. After the receiver has successfully parsed and validated the registration response message, the receiver MUST initiate the **proximity detection** procedure.

3.2.5.2 Proximity Detection Details

The receiver MUST perform **proximity detection** using the following steps:

1. The receiver MUST send the transmitter a proximity start message that contains the session ID that it received in the registration response message.

2. After the receiver receives the proximity challenge message from the transmitter, it MUST verify the session ID. Then the receiver MUST send a proximity response message containing the

following: the session ID, the sequence number, and the **nonce** encrypted using the **content encryption key** that was computed from the registration response message.

3. Because of possible packet loss on the network, the receiver SHOULD react to all of the following:

   - Every proximity challenge message it receives.
   - Missing proximity challenge messages. The receiver MAY choose to send a new proximity start message.
   - Not receiving a proximity challenge message at all, which MAY be caused by a transient or persistent problem. A transient problem can be a temporary failure in the transmitter, while a persistent problem can be an invalid network **topology** (for example, the receiver is further than two routers from the transmitter).

4. After receiving a proximity result message, the receiver MUST check whether the proximity detection procedure was completed successfully. If proximity detection was a success, then the result code is 0. If it was a failure, then result code is 106 (Unable to Verify Proximity). The error codes are defined in section 2.3.

5. After the receiver successfully completes proximity detection, it is registered and can request a license for any protected content that the user wants to play.

Because of possible packet loss on the network, the receiver SHOULD react to a missing proximity result message. In this case, the receiver SHOULD send a new proximity start message, which can result in the transmitter responding directly with a proximity result message, in case the proximity detection had already succeeded.

If the round trip time is greater than 7 milliseconds (ms) or if the receiver sends a proximity response message with a nonce that is not encrypted correctly, the transmitter MUST fail the proximity detection procedure and report an error code in the proximity result message. If proximity detection has failed, the receiver MAY repeat the proximity detection procedure until it confirms that the proximity detection succeeded. It is recommended that receivers wait 30 to 50 ms between retries, and the receiver MUST send the messages from the same IP address and port number as the original proximity start message. The session ID that the transmitter provided in the registration response message is valid for only a limited period of time--typically two minutes--so the receiver MAY need to restart the registration procedure after that to obtain a new session ID.

3.2.5.3 License Retrieval Details

The receiver MUST perform **license retrieval** using the following steps:

1. The receiver MUST send a license request message to the transmitter that contains the receiver's version number of the WMDRM: Network Devices Protocol, its **device** certificate, serial number, a 128-bit Rights ID value, the intended action of the receiver (such as "Play"), and its version of the **CRL**, if it has a CRL.

2. If the receiver has transmitting capabilities, the license response message MAY include a WMDRM-ND CRL. If a WMDRM-ND CRL is included in the response, it MUST be an updated version of the WMDRM-ND CRL. In this case, the receiver MUST compare its own version of the CRL to the version of the CRL that the transmitter included in the license response message. If the transmitter's CRL version number is higher than the receiver's CRL and the transmitter's CRL is valid, then the receiver MUST install the new CRL.

3. If the license contains a **RIV** object (defined in the XMR specification [XMR]), the receiver MUST compare the RIV specified in the license against the version of its own RIV structure. If the transmitter's RIV version number is higher than the receiver's RIV, or if the receiver does not have a RIV structure, the receiver MUST successfully complete the **revocation list update** procedure

according to the rules in revocation list before it is allowed to request content from the transmitter.

3.2.5.4 Data Transfer Details

A receiver MUST NOT request the content if it cannot enforce the restrictions specified in the license. For example, if the license requires High-bandwidth Digital Content Protection (HDCP) for a digital output, but the receiver does not support HDCP, it MUST NOT request the content.

3.2.5.5 License Management

3.2.5.5.1 Using License Chains

The receiver MUST be able to buffer two **root licenses** and two **leaf licenses** per logical stream. This is required because the transmitter MUST deliver a subsequent root and or leaf license before they are needed.

The receiver MUST process a **license update** as described in the following steps:

1. The receiver MUST ensure that the rights ID in the license response message matches the rights ID in the license request message that the receiver sent during the **license retrieval** procedure.

2. If the receiver has transmitting capabilities, the license response message MAY include an updated version of the WMDRM-ND **CRL**. In this case, the receiver MUST compare its own version of the CRL to the version in the license response message. If the response message CRL version number is higher than the receiver's CRL, and the transmitter's CRL is valid, then the receiver MUST install the new CRL.

3.2.5.5.2 License Management using License Derivation Data

License derivation data contains information from which a receiver can derive a new set of content restrictions. Upon receipt of license derivation data, the receiver generates a new license with updated restrictions to the current **base license**.

3.2.5.6 Revocation List Update Details

When the receiver obtains a new license through the **license retrieval** procedure, the receiver MUST determine whether the license contains a **RIV** object (defined in the XMR specification [XMR]). This object specifies the minimum version of the RIV that the receiver MUST have.

If an RIV object is present in the license, then the receiver MUST compare the version of its RIV structure, against the RIV version specified in the RIV object.

If the license specifies a version of the RIV structure that is more recent than the receiver has, then the receiver MUST obtain the latest RIV structure. Also, if the receiver does not have a RIV structure, and a RIV object is present in the license, then the receiver MUST obtain the latest RIV structure.

The RIV structure specifies the latest version of revocation lists for different protection systems. The receiver MUST compare the version numbers of the revocation lists in the RIV structure against the revocation lists that it has.

If the RIV structure indicates that a newer version of a revocation list is available, the receiver MUST obtain the new revocation list for the protection system. The new revocation list MUST have a version number that is equal to, or higher than the version number specified in the RIV structure.

The receiver MAY use the **revocation list update** procedure to obtain new revocation lists. Alternatively, the receiver MAY also obtain revocation lists in an implementation-specific manner.

The receiver initiates the revocation list update procedure by transmitting a revocation list request message to the transmitter. The message MUST specify one or more content protection systems for which the receiver wants a revocation list.

The revocation list for each content protection system is identified by a **GUID**. The RIV structure is also identified by a GUID and is treated the same way as a content protection system in the revocation list request message. To obtain the latest RIV structure, the receiver MUST specify the GUID assigned to the RIV structure in the revocation list request message.

The receiver MAY perform a revocation list update at any time.

If the receiver MUST perform a revocation list update and it fails, then the receiver MUST NOT request data transfer of content from the transmitter. Likewise, if the receiver MUST obtain a newer revocation list for some content protection system and it fails to do so, the receiver MUST NOT request content from the transmitter.

The format of the revocation list update messages is specified in section 2.2.1.6.

3.2.6 Timer Events

None.

3.2.7 Other Local Events

None.

3.3 Transmitter Details

3.3.1 Abstract Data Model

3.3.1.1 Overview

Authorization: The procedure for granting a **receiver** access to **content** from a transmitter. This procedure is required before a receiver can perform **registration** and access protected content.

Registration and Revalidation: Allows the transmitter to uniquely identify a receiver that is located nearby. Revalidation is the procedure for automatically reregistering a receiver with the transmitter after a certain amount of time has elapsed. Registration and **revalidation** are very similar, so for the purposes of this document, they will be grouped together in one category. Differences will be pointed out as necessary.

Proximity Detection: Measures the latency between the transmitter and a receiver. If the latency is short enough, then the receiver is considered to be **near** the transmitter. The transmitter will not send protected content to a receiver unless it is near.

License Retrieval: The procedure for sending a **license** to a registered Receiver when it requests content from a transmitter.

Data Transfer: Performed between a transmitter and a receiver after a license has been sent by the transmitter and enforced by the receiver. The content is encrypted and delivered using HTTP [RFC2616] or **RTP** [RFC3550].

License Management: For this procedure, the transmitter sends an updated license to the receiver to change the receiver's rights to a **resource**.

Revocation List Update: This procedure is permitted only after the registration procedure has successfully completed. The receiver asks the transmitter for an updated **revocation list**. This procedure can be initiated at any time after the registration procedure has occurred.

Authorization is performed after a receiver is discovered by a transmitter. Registration/revalidation is performed every 48 hours. The **license retrieval** and data transfer procedures occur every time a **file** is requested and played.

All of the procedures rely on messages--or sets of parameters--that are passed back and forth between a transmitter and a receiver. Messages are mapped to the HTTP, **RTSP**, and **UPnP** protocols. The following conceptual sections describe these procedures in further detail.

Authenticated Commands: The **authenticated commands** procedure allows sending cryptographically signed commands from the WMDRM: Network Devices Protocol receiver to the WMDRM: Network Devices Protocol transmitter. The authenticated commands are needed to control the **PBDA** hardware tuner.

3.3.2 Timers

None.

3.3.3 Initialization

None.

3.3.4 Higher-Layer Triggered Events

None.

3.3.5 Message Processing Events and Sequencing Rules

3.3.5.1 Registration Details

The transmitter MUST maintain a record of all registered receivers. This collection of records is referred to as the **device** registration database. Any implementation-specific data storage can be used to store this collection of records.

The transmitter performs registration through the following steps:

1. The transmitter MUST receive the registration request message that is sent by the receiver. The registration request message contains the following: the WMDRM: Network Devices Protocol version number that is supported by the receiver, its device certificate signed by Microsoft (which includes a 1024-bit RSA **public key**), and a serial number.

2. The transmitter MUST parse the registration request message and validate the receiver's device certificate. The certificate MUST be validated according to the rules specified in the Machine Certificate Specification [XMR].

3. The transmitter MUST verify that the device certificate has not been revoked. The transmitter checks each certificate in the chain against the **CRL**. If a receiver's certificate has been revoked, it MUST NOT be allowed to register, revalidate, or start a data transfer. For more information on CRLs, see section 2.2.1.5.3.

4. The transmitter MUST generate a seed and use it to generate three 128-bit numbers: the **content encryption key**, the **content integrity key**, and the **authenticated commands** key.

5. The transmitter MUST add the device to its device registration database. The transmitter MUST NOT transfer content to more than 10 different receivers simultaneously.

6. The transmitter MUST send the registration response message to the receiver. The message contains the following: a random session identifier, the seed (encrypted with the public key of the receiver), the serial number of the receiver, the WMDRM: Network Devices Protocol version number supported by the transmitter, the IP address and port number of the transmitter (for **proximity detection**), and the OMAC [OMAC] of the message, which is computed using the content integrity key. A new random session identifier MUST be generated for every registration response message.

3.3.5.2 Proximity Detection

The transmitter MUST perform **proximity detection** using the following steps:

1. After the transmitter receives the proximity start message, it MUST generate a sequence number and a **nonce**. (The nonce is a 128-bit random value that MUST be generated by a pseudo-random number generator that is cryptographically random in a manner consistent with the compliance rules [CR-WMDRM] for transmitter.)

2. Then the transmitter MUST send the receiver a proximity challenge message. The message MUST contain the sequence number, the session ID, and the nonce. If IPv4 is used, the **time-to-live (TTL)** field in the IPv4 header of the IP **datagram** carrying the proximity challenge message MUST be set to 3. If IPv6 is used, the **Hop Count** field in the IPv6 header MUST be set to 3. The sequence number is incremented by one for each proximity challenge message that is sent to the receiver.

3. The transmitter MUST start a counter and measure the time it takes for the receiver to reply with the proximity response message.

4. The transmitter MUST stop the counter when it receives the Proximity Response message. The transmitter MUST then validate the encrypted Nonce. The transmitter determines, based on the computed **round-trip time (RTT)**, whether the receiver is in proximity. If the RTT is 7 milliseconds or less, proximity detection succeeds. The proximity detection procedure MUST fail if the receiver is not in proximity.

5. Finally, the transmitter MUST send a proximity result message to the receiver indicating whether the proximity detection procedure was successful. An error code of 0 MUST be used in case of success, while a failure is indicated via error code 106 (Unable to Verify Proximity). If IPv4 is used, the **TTL** field in the IPv4 header of the IP datagram carrying the proximity result message MUST be set to 3. If IPv6 is used, the **Hop Count** field in the IPv6 header MUST be set to 3.

If the proximity result message is lost by the network, the receiver can send a new proximity start message. If the proximity detection procedure already succeeded, the transmitter MAY immediately reply with a proximity result message after it receives the new proximity start message from the receiver.

The transmitter MUST securely store the validation states of all receivers in its database, and the transmitter MUST require proximity detection every 48 hours as part of the **revalidation** procedure. If more than 48 hours has elapsed since a receiver performed a successful proximity detection procedure, then the transmitter MUST NOT deliver content to that receiver until the receiver has been revalidated.

3.3.5.3 License Retrieval Details

The transmitter MUST respond to a license request message through the following steps:

1. After the transmitter receives the license request message, it MUST verify that the **device** certificate is valid and has not been revoked by following the certificate validation rules specified in

the Machine Certificate Specification [XMR]. The transmitter MUST also verify that the device certificate has a security level equal to or greater than the license of the content being requested, and that the **EncryptKey** element value in the device certificate is set to the string "1".

2. After that, the transmitter MUST verify that the receiver has revalidated itself within the last 48 hours and is currently registered in the transmitter's device registration database. If the device is not in the database, the transmitter MUST return the error code 107 (MUST Register) to the device. If the device has not been revalidated, the transmitter MUST return the error code 108 (MUST Revalidate) to the device.

3. The transmitter MUST verify that the action requested by the receiver is permissible for the requested content. The only action currently supported is the "Play" action. The transmitter MUST verify that the license for the requested content allows that action.

4. The transmitter MUST send a license response message back to the receiver that contains the following: the version number of the WMDRM: Network Devices Protocol supported by the transmitter, the **CRL** of the transmitter (sent only if the receiver has transmitting capabilities and the transmitter has a copy of the CRL with a higher version number), and the license. The license is XMR [XMR] based and contains the Rights ID, the serial number of the receiver, 128-bit content encryption and **content integrity key**s used during content transfer, and an OMAC [OMAC] of the message, which is computed using the content integrity key.

5. A transmitter MUST check whether a receiver has transmitting capabilities by checking whether the WMDRM: Network Devices Protocol transmitter node is present in the receiver's device certificate.

If the protocol version numbers of the receiver and the transmitter are not the same, the receiver and transmitter MUST use the lowest common protocol version in subsequent messages.

3.3.5.4 Data Transfer Details

In accordance with the Compliance Rules [CR-WMDRM], the transmitter MUST protect any unencrypted, compressed content during any transcription procedure.

If the content has limited usage, the transmitter MUST monitor any state associated with the content and update it as needed before the content is sent to the receiver. For example, if the license only allows the content to be played a certain number of times, the transmitter MUST monitor that usage and deny data transfer to the receiver if the play count has exceeded the maximum allowed value.

3.3.5.5 License Management

3.3.5.5.1 License Management using License Chains

To perform a **license update**, the transmitter MUST send a license response message that contains a **leaf license** to the receiver.

The transmitter will deliver a leaf license before it starts delivering content that depends on that leaf license. However, the transmitter MUST NOT deliver more than one leaf license per logical stream in advance. Receivers are not required to be able to buffer more than one new leaf license in addition to the leaf license that is currently in use.

After a transmitter starts delivering content that depends on a new leaf license, it MUST NOT send content that depends on a leaf license that was previously delivered, because receivers MAY no longer have a copy of the previous leaf license.

The transmitter MUST NOT specify a higher version of the **RIV** in the leaf license than it specified in the **root license**.

3.3.5.5.2 License Management using License Derivation Data

When a license update occurs, the transmitter sends the license derivation data to the receiver. This data conveys the updated **policy (1)** for the content. Updating the license involves changing the keys and policies. The receiver derives the new license from this data.

3.3.5.5.3 License Management for Copy-Never Content

In some cases, the desired **policy (1)** is that content be usable by a receiver only for a limited time. In these cases, the transmitter MUST use a license that expires. Accordingly, access to the license will expire at the expiration time. This has the same effect as a policy in which the content expires after it is delivered to the receiver. For example, if a license is set to expire in 90 minutes, any content sent 30 minutes after the license has been sent will not be viewable for the full length of the content (90 minutes).

To address this issue, transmitters MUST create new licenses so that content being transmitted will not prematurely expire due to license expiration. For example, if a transmitter sent 90 minutes of Copy-Never broadcast content, it would issue a new license set to expire in 90 minutes to the receiver. License derivation data messages with new base counters will then be sent to the receiver every 9 minutes.

3.3.5.6 Revocation List Updates

The transmitter responds to the revocation list request message with a revocation list response message. The response message MUST contain the latest revocation list that is available to the transmitter. The response message MUST contain the latest revocation list available for each content protection system that was specified in the revocation list request. If the transmitter does not have a revocation list for a requested content protection system, the revocation list response MUST include a zero-length revocation list for that system.

The receiver MAY perform a **revocation list update** at any time, and this procedure MUST NOT alter the transmitter's state in any way.

The format of the revocation list update messages is specified in section 2.2.1.6.

3.3.6 Timer Events

None.

3.3.7 Other Local Events

None.

3.4 schemas-opencable-com:1 Service Model Details

The schemas-opencable-com:1 **service** model is a set of services, schemas, specifications, actions and rules applicable to all entities pertinent to the functionality of a specific framework.

3.4.1 Abstract Data Model

3.4.1.1 State Variables

The following table lists the state variables used by the X_MS_MediaReceiverRegistrar:1 **service**.

Variable Name	Required or Optional	Data Type	Allowed Value
A_ARG_TYPE_Result	Required	boolean	0/1 (True/False)
AuthorizationGrantedUpdateID	Required	ui4	Any random integer
AuthorizationDeniedUpdateID	Required	ui4	Any random integer
A_ARG_TYPE_UpdateID	Required	ui4	Any random integer
A_ARG_TYPE_GenericData	Required	bin.base64	Generic Base64 data block containing **registration** information

Values listed in this table are required. To specify standard optional values or to delegate assignment of values to the vendor, you MUST reference a specific instance of an appropriate table below.

A_ARG_TYPE_Result: A_ARG_TYPE_Result is used to pass back the result of an action. The variable is represented as a binary TRUE or FALSE value.

AuthorizationGrantedUpdateID: This variable changes whenever a **device** has been authorized by a user. (Authorized implies the user has consented to using this device.) The actual value of **AuthorizationGrantedUpdateID** is unspecified. However, the value MUST be of type ui4 and change in some way when devices are authorized.

AuthorizationDeniedUpdateID: This variable changes whenever a device has been unauthorized by a user. (Unauthorized implies that the user does not want to use this device.) The actual value of **AuthorizationDeniedUpdateID** is unspecified. However, the value MUST be of type ui4 and change in some fashion when devices are unauthorized.

A_ARG_TYPE_UpdateID: This variable is used in conjunction with any action that includes an updated parameter. The value of parameter *UpdateID* will either be the **AuthorizationGrantedUpdateID** or the **AuthorizationDeniedUpdateID**.

A_ARG_TYPE_GenericData: This variable is used whenever Base64-encoded data is required by an action. The variable contains bin.base64 (Base64-encoded **BLOB**) encoded registration information. For information on Base64, see [RFC3548].

3.4.2 Timers

None.

3.4.3 Initialization

None.

3.4.4 Message Processing Events and Sequencing Rules

This section provides applicable information about OpenCable **services**, rules and schemas

3.4.4.1 Service Model Definitions

3.4.4.1.1 Service Type

The following **service** type identifies a service that is compliant with this template.

```
urn:schemas-opencable-com:service:Security:1
```

3.4.4.1.2 State Variables

The following table shows the state variables used by the schemas-opencable-com **service**.

Variable Name	Required or Optional	Data Type	Allowed Value
A_ARG_TYPE_ LicenseChallenge	Required	bin.base64	Generic Base64 data block that contains license request information.
A_ARG_TYPE_ LicenseResponse	Required	bin.base64	Generic Base64 data block that contains license response information.
A_ARG_TYPE_ LicenseResponseAck	Required	bin.base64	Generic Base64 data block that contains an acknowledgement that a license response has been received.
A_ARG_TYPE_ RegistrationChallenge	Required	bin.base64	Generic Base64 data block that contains registration request information.
A_ARG_TYPE_ RegistrationResponse	Required	bin.base64	Generic Base64 data block that contains registration response information.
A_ARG_TYPE_ RevocationInformationVersion	Required	bin.base64	Generic Base64 data block that contains the **revocation** information version and **revocation list** information version records.
A_ARG_TYPE_ RevocationList	Required	bin.base64	Generic Base64 data block that contains the revocation list information version record.

A_ARG_TYPE_LicenseChallenge: This variable is a Base64-encoded **BLOB** that is used when passing a license request message to a DRI. For more information on the license request message, see section 2.2.1.4.1. For information on Base64, see [RFC3548].

A_ARG_TYPE_LicenseResponse: This variable is a Base64-encoded BLOB that is used when passing a license response message to a home media server (HMS). For more information, see section 2.2.1.4.2.

A_ARG_TYPE_LicenseResponseAck: This variable is a Base64-encoded BLOB that is used when passing an acknowledgement that a license response message has been received by an HMS.

A_ARG_TYPE_RegistrationChallenge: This variable is a Base64-encoded BLOB that is used when passing a registration request message to a DRI. For more information, see section 2.2.1.1.

A_ARG_TYPE_RegistrationResponse: This variable is a Base64-encoded BLOB that is used when passing a registration response message to an HMS. For more information, see section 2.2.1.2.

A_ARG_TYPE_RevocationInformationVersion: This variable is a Base64-encoded BLOB that is used when passing the revocation information version to an HMS. For more information, see section 2.2.1.5.1.

A_ARG_TYPE_RevocationList: This variable is a Base64-encoded BLOB that corresponds to the revocation list version information record for a particular type of revocation (application revocation, **device** revocation, **COPP** driver revocation). For more information, see Revocation.

3.4.4.2 Actions

This section lists the actions supported by the schemas-opencable-com **service**. The focus of the schemas-opencable-com service is on handling the data associated to WMDRM: Network Devices operations.

3.4.4.2.1 ProcessRegistrationChallenge

This action accepts a registration **challenge**, checks the validity of the registration challenge, and generates a registration response.

The following table shows the arguments for the **ProcessRegistrationChallenge** action.

Argument	Direction	relatedStateVariable
RegistrationChallenge	IN	A_ARG_TYPE_ RegistrationChallenge
RegistrationResponse	OUT	A_ARG_TYPE_ RegistrationResponse

3.4.4.2.2 ProcessLicenseChallenge

This action accepts a license request, checks the validity of the request, and returns the **base license** in a license response.

The following table shows the arguments for the **ProcessLicenseChallenge** action.

Argument	Direction	relatedStateVariable
LicenseChallenge	IN	A_ARG_TYPE_ LicenseChallenge
LicenseResponse	OUT	A_ARG_TYPE_ LicenseResponse

3.4.4.2.3 AcknowledgeLicense

This action acknowledges that a license response has been received by an HMS.

The following table shows the arguments for the **AcknowledgeLicense** action.

Argument	Direction	relatedStateVariable
LicenseResponseAck	IN	A_ARG_TYPE_ LicenseResponseAck

3.4.4.2.4 SetRevocationData

This action accepts the revocation information version and the revocation list version information record.

The following table shows the arguments for the SetRevocationData action.

Argument	Direction	relatedStateVariable
RevocationInformationVersion	IN	A_ARG_TYPE_

Argument	Direction	relatedStateVariable
		RevocationInformationVersion
RevocationList	IN	A_ARG_TYPE_ RevocationList

3.4.5 Timer Events

None.

3.4.6 Other Local Events

None.

3.4.6.1 Eventing and Moderation

The eventing and moderation variables for the X_MS_MediaReceiverRegistrar:1 **service** are described in the following table.

Variable Name	Evented	Moderated Event	Max Event Rate	Logical Combination	Min Delta per Event
AuthorizationGrantedUpdateID	Yes	No	n/a	n/a	1
AuthorizationDeniedUpdateID	Yes	No	n/a	n/a	1

Max Event Rate and the Min Delta per Event are determined by *n*, where the Max Event Rate = (Event/*n* seconds) and the Min Delta per Event = *n**(allowedValueRange Step).

3.4.6.2 Event Model

The two events listed in the previous table are not moderated. The events will be sent whenever a **device** is authorized or unauthorized by a user. There is no device ID associated with these events and each device MUST call the **IsAuthorized** action after an event to determine whether it is the device that has been authorized or unauthorized. Devices that are already authorized or unauthorized do not need to respond to an event that implies no change in state.

3.4.6.3 Actions

This section describes all actions associated with the X_MS_MediaReceiverRegistrar:1 **service** . The focus of this service and its actions are to allow devices to register with a transmitter implementing WMDRM: Network Devices Protocol.

Immediately following this table is detailed information about these actions, including short descriptions of the actions, the effects of the actions on state variables, and error codes defined by the actions.

Name	Required or Optional
IsAuthorized	Required
RegisterDevice	Optional (Required for WMDRM-ND))

Name	Required or Optional
IsValidated	Optional (Required for WMDRM-ND)

3.4.6.3.1 IsAuthorized

A **receiver** can use this action to check whether it is authorized. If a receiver is a **UPnP** media renderer, it can pass a **DeviceID** to the transmitter. The transmitter returns a result of TRUE if the receiver has been authorized and FALSE otherwise. The **DeviceID** SHOULD equal the unique **device** name (UDN). For control points, an empty string SHOULD be passed in and the MAC address will be used for determining authorization. If a receiver calls **IsAuthorized** for itself, the action will return with a result of TRUE or FALSE. If a control point is requesting status for another receiver, the receiver MUST be authorized to receive a result; otherwise the transmitter will return a UPnP 801 error (access denied).

The DeviceID used in the **IsAuthorized** action and the serial number described in section 3.1.1.3 are two separate concepts, although receivers can use the same value.

The following table shows the arguments for the IsAuthorized action.

Argument	Direction	relatedStateVariable
DeviceID	IN	A_ARG_TYPE_DeviceID
Result	OUT	A_ARG_TYPE_Result

Whenever a receiver is authorized or unauthorized by a user, an event triggered by a change in the **AuthorizationGrantedUpdateID** variable or the **AuthorizationDeniedUpdatedID** variable is sent by the transmitter to all receivers. Each Receiver MUST call the **IsAuthorized** action after the event is received to determine whether that receiver is the device that has been authorized or unauthorized.

Dependency on State: This action SHOULD only be called when either the **AuthorizationGrantedUpdateID** or **AuthorizationDeniedUpdateID** variable has been evented and has changed from the last number the device knows.

Devices MAY also call this action on power-up.

3.4.6.3.2 RegisterDevice

The **RegisterDevice** action registers a **device** with the WMDRM: Network Devices Protocol **service**. The action takes one input variable: a bin.base64-encoded registration request message as defined in section 2.2.1.1. After the transmitter receives the **RegisterDevice** action, it will respond with a bin.base64-encoded registration response message as defined in section 2.2.1.2. If invoked by a device that is not authorized, **RegisterDevice** will return a **UPnP** 801 error (access denied).

The following table shows the arguments for the **RegisterDevice** action.

Argument	Direction	relatedStateVariable
Generic data block	IN	A_ARG_TYPE_GenericData
Generic data block	OUT	A_ARG_TYPE_GenericData

The Base64 encoding, which is used in the registration request messages and registration response messages, is defined in [RFC4648].

Dependency on State: Devices SHOULD call this action on power-up.

3.4.6.3.3 IsValidated

The **IsValidated** action is used to determine whether a **device** is registered with the WMDRM: Network Devices Protocol **service**. Like the **IsAuthorized** action, the **IsValidated** action takes a **DeviceID**. For **UPnP** media renderers, the **DeviceID** SHOULD equal the UDN. For control points, the **DeviceID** SHOULD equal an empty string. If a device calls **IsValidated** for itself, the action will return with a result of TRUE or FALSE. If a control point is requesting status for another media renderer, the renderer MUST be authorized to receive a result, otherwise the transmitter will return a UPnP 801 error (access denied). The following table shows the arguments for the **IsValidated** action.

Argument	Direction	relatedStateVariable
DeviceID	IN	A_ARG_TYPE_DeviceID
Result	OUT	A_ARG_TYPE_Result

Dependency on State: Devices can call this action on power-up.

3.4.6.4 Common Error Codes

The following table lists error codes common to actions for this **service** type. If an action results in multiple errors, the most specific error MUST be returned.

errorCode	errorDescription	Description
401	Invalid Action	See section 3.2.2 of the **UPnP** Device Architecture specification [UPNPARCH1.1].
402	Invalid Args	See section 3.2.2 of the UPnP Device Architecture specification [UPNPARCH1.1].
404	Invalid Var	See section 3.2.2 of the UPnP Device Architecture specification [UPNPARCH1.1].
501	Action Failed	See section 3.2.2 of the UPnP Device Architecture specification [UPNPARCH1.1].
600-699	TBD	Common action errors defined by UPnP Forum Technical Committee.
700-799	TBD	Common action errors defined by UPnP Forum working committees.
800-899	TBD	Errors specified by UPnP vendors.

4 Protocol Examples

4.1 UPnP Authorization

The following example shows the authorization procedure between a **receiver** and a transmitter.

Receiver to Transmitter

```
POST /upnphost/udhisapi.dll?control=uuid:a7e63d3b-0a14-4143-b997-
3cd79e20766b+urn:microsoft.com:serviceId:X_MS_MediaReceiverRegistrar HTTP/1.1
Content-Type: text/xml; charset="utf-8"
SOAPAction: "urn:microsoft.com:service:X MS MediaReceiverRegistrar:1#IsAuthorized"
User-Agent: Mozilla/4.0 (compatible; UPnP/1.0; Windows 9x)
Host: 192.168.3.106:2869
Content-Length: 401
Connection: Keep-Alive
Cache-Control: no-cache
Pragma: no-cache

<?xml version="1.0"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"><SOAP-ENV:Body><m:IsAuthorized
xmlns:m="urn:microsoft.com:service:X_MS_MediaReceiverRegistrar:1"><DeviceID
xmlns:dt="urn:schemas-microsoft-com:datatypes"
dt:dt="string"></DeviceID></m:IsAuthorized></SOAP-ENV:Body></SOAP-ENV:Envelope>
```

The receiver sends its **DeviceID** to the transmitter and the transmitter responds to the receiver with a 1 if the receiver is currently authorized, or a 0 if the receiver is not authorized.

Transmitter to Receiver

```
HTTP/1.1 200 OK
Content-Length: 411
Content-Type: text/xml; charset="utf-8"
Server: Microsoft-Windows-NT/5.1 UPnP/1.0 UPnP-Device-Host/1.0 Microsoft-HTTPAPI/1.0
Date: Wed, 07 Jul 2004 00:38:23 GMT

<?xml version="1.0"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"><SOAP-
ENV:Body><m:IsAuthorizedResponse
xmlns:m="urn:microsoft.com:service:X_MS_MediaReceiverRegistrar:1"><Result
xmlns:dt="urn:schemas-microsoft-com:datatypes"
dt:dt="int">1</Result></m:IsAuthorizedResponse></SOAP-ENV:Body></SOAP-ENV:Envelope>
```

For more information on IsAuthorized, see section 3.4.6.3.1.

4.2 Registration

The following example shows the receiver attempting to register itself with the transmitter by calling the **RegisterDevice** action.

Receiver to Transmitter

```
POST /upnphost/udhisapi.dll?control=uuid:a7e63d3b-0a14-4143-b997-
3cd79e20766b+urn:microsoft.com:serviceId:X MS MediaReceiverRegistrar HTTP/1.1
Content-Type: text/xml; charset="utf-8"
SOAPAction: "urn:microsoft.com:service:X MS MediaReceiverRegistrar:1#RegisterDevice"
User-Agent: Mozilla/4.0 (compatible; UPnP/1.0; Windows 9x)
Host: 192.168.3.106:2869
```

```
Content-Length: 5903
Connection: Keep-Alive
Cache-Control: no-cache
Pragma: no-cache

HTTP/1.1 100 Continue

<?xml version="1.0"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"><SOAP-
ENV:Body><m:RegisterDevice
xmlns:m="urn:microsoft.com:service:X_MS_MediaReceiverRegistrar:1"><RegistrationReqMsg
xmlns:dt="urn:schemas-microsoft-com:datatypes" dt:dt="bin.Base64">Base64.Encoded Windows
Media DRM for Network Devices Registration Request
Message</RegistrationReqMsg></m:RegisterDevice></SOAP-ENV:Body></SOAP-ENV:Envelope>
```

The receiver sends a Base64-encoded data block [RFC3548] that contains the **registration** request message to the transmitter. The transmitter then responds by sending a Base64-encoded registration response message back to the receiver. The following example shows a registration response message.

Transmitter to Receiver

```
HTTP/1.1 200 OK
Content-Length: 729
Content-Type: text/xml; charset="utf-8"
Server: Microsoft-Windows-NT/5.1 UPnP/1.0 UPnP-Device-Host/1.0 Microsoft-HTTPAPI/1.0
Date: Wed, 07 Jul 2004 00:38:23 GMT

<?xml version="1.0"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"><SOAP-
ENV:Body><m:RegisterDeviceResponse
xmlns:m="urn:microsoft.com:service:X_MS_MediaReceiverRegistrar:1"><RegistrationRespMsg
xmlns:dt="urn:schemas-microsoft-com:datatypes" dt:dt="bin.Base64">Base64.Encoded Windows
Media DRM for Network Devices Registration Response Message
</RegistrationRespMsg></m:RegisterDeviceResponse></SOAP-ENV:Body></SOAP-ENV:Envelope>
```

A receiver can also call the **IsValidated** action to determine whether or not it is registered with the transmitter. The following example shows a receiver checking its registration status.

Receiver to Transmitter

```
POST /upnphost/udhisapi.dll?control=uuid:a7e63d3b-0a14-4143-b997-
3cd79e20766b+urn:microsoft.com:serviceId:X_MS_MediaReceiverRegistrar HTTP/1.1
Content-Type: text/xml; charset="utf-8"
SOAPAction: "urn:microsoft.com:service:X_MS_MediaReceiverRegistrar:1#IsValidated"
User-Agent: Mozilla/4.0 (compatible; UPnP/1.0; Windows 9x)
Host: 192.168.3.106:2869
Content-Length: 401
Connection: Keep-Alive
Cache-Control: no-cache
Pragma: no-cache

<?xml version="1.0"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"><SOAP-ENV:Body><m:IsValidated
xmlns:m="urn:microsoft.com:service:X MS MediaReceiverRegistrar:1"><DeviceID
xmlns:dt="urn:schemas-microsoft-com:datatypes"
dt:dt="string"></DeviceID></m:IsValidated></SOAP-ENV:Body></SOAP-ENV:Envelope>
```

The receiver sends its **device** ID to the transmitter, and the transmitter responds to the receiver with either a 1 indicating that the receiver is currently registered or a 0 if the receiver is not registered.

Transmitter to Receiver

```
HTTP/1.1 200 OK
Content-Length: 411
Content-Type: text/xml; charset="utf-8"
Server: Microsoft-Windows-NT/5.1 UPnP/1.0 UPnP-Device-Host/1.0 Microsoft-HTTPAPI/1.0
Date: Wed, 07 Jul 2004 00:38:23 GMT

<?xml version="1.0"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"><SOAP-
ENV:Body><m:IsValidatedResponse
xmlns:m="urn:microsoft.com:service:X_MS_MediaReceiverRegistrar:1"><Result
xmlns:dt="urn:schemas-microsoft-com:datatypes"
dt:dt="int">1</Result></m:IsValidatedResponse></SOAP-ENV:Body></SOAP-ENV:Envelope>
```

For more information on the RegisterDevice and IsValidated actions and their arguments, see section 6.

4.3 HTTP License Retrieval

The following example illustrates how **license retrieval** is performed using HTTP.

Receiver to Transmitter

```
POST /2cfa8.wmv HTTP/1.0
Accept: */*
Supported: com.microsoft.wmdrm-nd
Content-Type: application/vnd.ms-wmdrm-license-request
Content-Length: 1078
```

License Request message-Transmitter to Receiver

```
HTTP/1.0 200 OK
Supported: com.microsoft.wmdrm-nd
WMDRM-ND: SessionId="E97625B8950CA0B658C98AEAD7DE7796"
Content-Type: application/vnd.ms-wmdrm-license-response
Content-Length: 924
```

License Response message- Receiver to Transmitter

The following example shows the error returned for a **device** certificate that has been revoked.

```
POST /file.wmv HTTP/1.0
Accept: */*
Supported: com.microsoft.wmdrm-nd
Content-Type: application/vnd.ms-wmdrm-license-request
Content-Length: 1078
```

License Request message-Transmitter to Receiver

```
HTTP/1.0 500 Internal Server Error
WMDRM-ND-Status: 101 "Certificate Revoked"
Supported: com.microsoft.wmdrm-nd
```

4.4 Retrieving ASF-Encapsulated Content

Receiver to Transmitter

```
GET /file.wmv HTTP/1.0
Accept: */*
Supported: com.microsoft.wmdrm-nd
WMDRM-ND: SessionId="E97625B8950CA0B658C98AEAD7DE7796"
```

Transmitter to Receiver

```
HTTP/1.0 200 OK
Supported: com.microsoft.wmdrm-nd
Content-Type: video/x-ms-asf
Content-Length: 3908480
```

The following example shows how to retrieve **content** from the transmitter using link **encryption** mode.

Receiver to Transmitter

```
GET /file.avi HTTP/1.0
Accept: */*
Supported: com.microsoft.wmdrm-nd
X-WMDRM-ND: SessionId="E97625B8950CA0B658C98AEAD7DE7796"
```

Transmitter to Receiver

```
HTTP/1.0 200 OK
Supported: com.microsoft.wmdrm-nd
Content-Type: application/vnd.ms-wmdrm-data-transfer; media="video/avi"

CONTROL_AND_DATA_BLOCKS
```

4.5 Using HTTP for a Revocation List Update

The following example illustrates how to perform a **revocation list update** using HTTP.

Receiver to Transmitter

```
POST /file.wmv HTTP/1.0
Accept: */*
Supported: com.microsoft.wmdrm-nd
Content-Type: application/vnd.ms-wmdrm-extended-operation-request
Content-Length: 68
```

Revocation List Request message-Transmitter to Receiver

```
HTTP/1.0 200 OK
Supported: com.microsoft.wmdrm-nd
Content-Type: application/vnd.ms-wmdrm-extended-operation-response
Content-Length: 364
```

Revocation List Response message

4.6 Retrieving a Root License

The following example illustrates how to retrieve a **root license** using **RTSP**.

Receiver to Transmitter

```
DESCRIBE rtsp://servername/file.wmv RTSP/1.0
Accept: application/sdp
CSeq: 1
Supported: com.microsoft.wmdrm-nd, com.microsoft.wm.eosmsg, method.announce
Require: com.microsoft.wmdrm-nd
Content-Type: application/vnd.ms-wmdrm-license-request
Content-Length: 1078
```

License Request message-Transmitter to Receiver

```
RTSP/1.0 200 OK
Content-Length: 1891
Content-Type: application/sdp
CSeq: 1
Supported: com.microsoft.wmdrm-nd, com.microsoft.wm.eosmsg, method.announce
```

SDP Description

Example of data URL containing Base64-encoded License Response. For information on Base64 see [RFC3548].

```
data:application/vnd.ms-wmdrm-license-response;base64,
AggAAAAAAAABOFhNUgAAAAAB+TTbzXCRw1s+/jA4fQQY0wADAAEAAAEgAAMAAgAAADwAAQADAAAAEgBkAAAAAAAAAAAAA
QAMAAAAGKRuHVtxsJ1Lk7WPrQPe5X0AAQANAAAACgABAAMABAAAABoAAQAFAAAAEgBkAGQAZABkAGQAAwAJAAAApgABAA
oAAACeajiAiUBMGrAGUAOIqMGBggABAAEAgC7V1QF54EzuYbTYKPbgBEK6nDXGtbV+bJKF+Cn2yd/FUaC4vTIOxkF/eQL
x+FqvLCUMtxvRSw01dns9Ejt021se2T+IROiZA0t5pRuNl3gq7JK9JKs+ZX8hKsEJFW0V7cyp9wdaCMh2esJ97r9agH1S
xf0mAqcQ0j1Q5dtXlWx/AAEACwAAABwAAQAQZZaX5nGEUAV8w6p6BQr++Q==
```

4.7 Using RTSP for a License Request

This example describes a server requesting a license from a receiver by using RTSP.

Transmitter to Receiver

```
ANNOUNCE rtsp://servername/file.wmv RTSP/1.0
CSeq: 27
Session: 8322364901836665746
Supported: com.microsoft.wmdrm-ndmethod.announce
Require: method.announce
Content-Type: application/vnd.ms-wmdrm-license-response
Content-Length: 924
```

License Response message- Receiver to Transmitter

```
RTSP/1.0 200 OK
CSeq: 27
Session: 8322364901836665746
Supported: com.microsoft.wmdrm-ndmethod.announce
```

4.8 Using RTSP for a Revocation List Update

The following example illustrates how to perform a **revocation list update** using **RTSP**.

Receiver to Transmitter

```
GET PARAMETER rtsp://servername/file.wmv RTSP/1.0
Cseq: 1
Supported: com.microsoft.wmdrm-nd
Content-Type: application/vnd.ms-wmdrm-extended-operation-request
Content-Length: 68
```

Revocation List Request message-Transmitter to Receiver

```
RTSP/1.0 200 OK
Cseq: 1
Supported: com.microsoft.wmdrm-nd
Content-Type: application/vnd.ms-wmdrm-extended-operation-response
Content-Length: 364
```

Revocation List Response message

4.9 Processing an Error

This section is an example of an error returned when there license request is made using a revoked **device** certificate.

Receiver to Transmitter

```
DESCRIBE rtsp://servername/file.wmv RTSP/1.0
Accept: application/sdp
CSeq: 1
Supported: com.microsoft.wm.eosmsg, com.microsoft.wmdrm-nd
Content-Type: application/vnd.ms-wmdrm-license-request
Content-Length: 1078
```

License Request message-Transmitter to Receiver

```
RTSP/1.0 500 Internal Server Error
CSeq: 1
WMDRM-ND-Status: 101 "Certificate Revoked"
Supported: com.microsoft.wm.eosmsg, com.microsoft.wmdrm-nd
```

5 Security

5.1 Security Considerations for Implementers

SHA-1 [FIPS180-2] Hashing has been shown to be theoretically vulnerable to collision, but is still considered a secure form of one-way encoding of data.

5.2 Index of Security Parameters

Security parameter	Section
AES	Section 2.4.3 Cryptographic Requirements for Transmitters
Authentication Command key	Section 2.2.1.2 Registration Response Message
AuthorizationDeniedUpdateID	Section 3.4.1.1 State Variables
AuthorizationGrantedUpdateID	Section 3.4.1.1 State Variables
Content Encryption key	Section 2.2.1.2 Registration Response Message
Certificate - **License** Request	Section 2.2.1.4.1 The License Request Message
Certificate - Registration	Section 2.2.1.2 Registration Response Message
Challenge Value **Nonce**	Section 2.2.1.3.2 The Proximity Challenge Message
Content Integrity key	Section 2.2.1.2 Registration Response Message
CTR	Section 2.4.3 Cryptographic Requirements for Transmitters
ECB	Section 2.4.3 Cryptographic Requirements for Transmitters
Encrypted **challenge** value (nonce)	Section 2.2.1.3.3 The Proximity Response Message
EncryptedNonce	Section 2.2.1.3.4 The Proximity Result Message
Encrypted Seed	Section 2.2.1.2 Registration Response Message
List ID field **GUIDs**	Section 2.2.1.5.1 Revocation List Version Information
MAC	Section 2.4.3 Cryptographic Requirements for Transmitters
Microsoft CTRCRL signing **public key**	Section 2.2.1.5.1 Revocation List Version Information Record
Microsoft Root Public Key	Section 2.4.3 Cryptographic Requirements for Transmitters
OMAC	Section 2.2.1.2 Registration Response Message
Proximity Detection NONCE	Section 2.2.1.3 Proximity Detection
Public Key - Receiver	Section 2.4.2 Common Cryptographic Requirements for Receivers
Rights ID	Section 2.2.1.4.1 The License Request Message
RNG - Random Number Generator	Section 2.4.3 Cryptographic Requirements for Transmitters
Seed Encryption Type	Section 2.2.1.2 Registration Response Message
Sequence number	Section 2.2.1.3.2 The Proximity Challenge Message
Serial Number - Registration Response	Section 2.2.1.2 Registration Response Message

Security parameter	Section
Serial Number - License Request	Section 2.2.1.4.1 The License Request Message
Session ID - Proximity Challenge	Section 2.2.1.3.2 The Proximity Challenge Message
Session ID - Proximity Detection	Section 2.2.1.2 Registration Response Message
SHA-1	Section 2.4.3 Cryptographic Requirements for Transmitters
Session ID	Section 3.1.1.3 Registration and Revalidation
Signature Section	Section 2.2.1.2 Registration Response Message
Transmitter Identifier	Section 2.2.1.2 Registration Response Message

6 Appendix A: Full XML Service Descriptions

6.1 Schemas-Opencable-Com Service

The following code snippet contains the **XML** Web **service** description for the schemas-opencable-com service:

```
<?xml version="1.0" ?>
<scpd xmlns="urn:schemas-upnp-org:service-1-0">
    <specVersion>
        <major>1</major>
        <minor>0</minor>
    </specVersion>
    <actionList>
        <action>
            <name>IsAuthorized</name>
            <argumentList>
                <argument>
                    <name>DeviceID</name>
                    <direction>in</direction>
                    <relatedStateVariable>A_ARG_TYPE_DeviceID</relatedStateVariable>
                </argument>
                <argument>
                    <name>Result</name>
                    <direction>out</direction>
                    <relatedStateVariable>A_ARG_TYPE_Result</relatedStateVariable>
                </argument>
            </argumentList>
        </action>
        <action>
            <name>RegisterDevice</name>
            <argumentList>
                <argument>
                    <name>RegistrationReqMsg</name>
                    <direction>in</direction>

<relatedStateVariable>A_ARG_TYPE_RegistrationReqMsg</relatedStateVariable>
                </argument>
                <argument>
                    <name>RegistrationRespMsg</name>
                    <direction>out</direction>

<relatedStateVariable>A_ARG_TYPE_RegistrationRespMsg</relatedStateVariable>
                </argument>
            </argumentList>
        </action>
        <action>
            <name>IsValidated</name>
            <argumentList>
                <argument>
                    <name>DeviceID</name>
                    <direction>in</direction>
                    <relatedStateVariable>A ARG TYPE DeviceID</relatedStateVariable>
                </argument>
                <argument>
                    <name>Result</name>
                    <direction>out</direction>
                    <relatedStateVariable>A_ARG_TYPE_Result</relatedStateVariable>
                </argument>
            </argumentList>
        </action>
    </actionList>
    <serviceStateTable>
        <stateVariable sendEvents="no">
            <name>A ARG TYPE DeviceID</name>
            <dataType>string</dataType>
```

```
        </stateVariable>
        <stateVariable sendEvents="no">
            <name>A_ARG_TYPE_Result</name>
            <dataType>int</dataType>
        </stateVariable>
        <stateVariable sendEvents="no">
            <name>A_ARG_TYPE_RegistrationReqMsg</name>
            <dataType>bin.base64</dataType>
        </stateVariable>
        <stateVariable sendEvents="no">
            <name>A ARG TYPE RegistrationRespMsg</name>
            <dataType>bin.base64</dataType>
        </stateVariable>
        <stateVariable sendEvents="yes">
            <name>AuthorizationGrantedUpdateID</name>
            <dataType>ui4</dataType>
        </stateVariable>
        <stateVariable sendEvents="yes">
            <name>AuthorizationDeniedUpdateID</name>
            <dataType>ui4</dataType>
        </stateVariable>
        <stateVariable sendEvents="yes">
            <name>ValidationSucceededUpdateID</name>
            <dataType>ui4</dataType>
        </stateVariable>
        <stateVariable sendEvents="yes">
            <name>ValidationRevokedUpdateID</name>
            <dataType>ui4</dataType>
        </stateVariable>
    </serviceStateTable>
</scpd>
```

6.2 X_MS_MediaReceiverRegistrar:1 Service

This section contains the **XML service** description for the X_MS_MediaReceiverRegistrar:1 service.

```
<?xml version="1.0" ?>
<scpd xmlns="urn:schemas-upnp-org:service-1-0">
    <specVersion>
        <major>1</major>
        <minor>0</minor>
    </specVersion>
    <actionList>
        <action>
            <name>IsAuthorized</name>
            <argumentList>
                <argument>
                    <name>DeviceID</name>
                    <direction>in</direction>
                    <relatedStateVariable>A_ARG_TYPE_DeviceID</relatedStateVariable>
                </argument>
                <argument>
                    <name>Result</name>
                    <direction>out</direction>
                    <relatedStateVariable>A ARG TYPE Result</relatedStateVariable>
                </argument>
            </argumentList>
        </action>
        <action>
            <name>RegisterDevice</name>
            <argumentList>
                <argument>
                    <name>RegistrationReqMsg</name>
                    <direction>in</direction>

<relatedStateVariable>A_ARG_TYPE_RegistrationReqMsg</relatedStateVariable>
```

```
                </argument>
                <argument>
                    <name>RegistrationRespMsg</name>
                    <direction>out</direction>

<relatedStateVariable>A ARG TYPE RegistrationRespMsg</relatedStateVariable>
                </argument>
            </argumentList>
        </action>
        <action>
            <name>IsValidated</name>
            <argumentList>
                <argument>
                    <name>DeviceID</name>
                    <direction>in</direction>
                    <relatedStateVariable>A_ARG_TYPE_DeviceID</relatedStateVariable>
                </argument>
                <argument>
                    <name>Result</name>
                    <direction>out</direction>
                    <relatedStateVariable>A_ARG_TYPE_Result</relatedStateVariable>
                </argument>
            </argumentList>
        </action>
    </actionList>
    <serviceStateTable>
        <stateVariable sendEvents="no">
            <name>A_ARG_TYPE_DeviceID</name>
            <dataType>string</dataType>
        </stateVariable>
        <stateVariable sendEvents="no">
            <name>A_ARG_TYPE_Result</name>
            <dataType>int</dataType>
        </stateVariable>
        <stateVariable sendEvents="no">
            <name>A ARG TYPE RegistrationReqMsg</name>
            <dataType>bin.base64</dataType>
        </stateVariable>
        <stateVariable sendEvents="no">
            <name>A ARG TYPE RegistrationRespMsg</name>
            <dataType>bin.base64</dataType>
        </stateVariable>
        <stateVariable sendEvents="yes">
            <name>AuthorizationGrantedUpdateID</name>
            <dataType>ui4</dataType>
        </stateVariable>
        <stateVariable sendEvents="yes">
            <name>AuthorizationDeniedUpdateID</name>
            <dataType>ui4</dataType>
        </stateVariable>
        <stateVariable sendEvents="yes">
            <name>ValidationSucceededUpdateID</name>
            <dataType>ui4</dataType>
        </stateVariable>
        <stateVariable sendEvents="yes">
            <name>ValidationRevokedUpdateID</name>
            <dataType>ui4</dataType>
        </stateVariable>
    </serviceStateTable>
</scpd>
```

7 Appendix B: Product Behavior

The information in this specification is applicable to the following Microsoft products or supplemental software. References to product versions include updates to those products.

- Windows 2000 operating system
- Windows XP operating system
- Windows Vista operating system
- Windows 7 operating system
- Windows Server 2008 R2 operating system
- Windows 8 operating system
- Windows Server 2012 operating system
- Windows 8.1 operating system
- Windows Server 2012 R2 operating system
- Windows 10 v1511 operating system

Exceptions, if any, are noted in this section. If an update version, service pack or Knowledge Base (KB) number appears with a product name, the behavior changed in that update. The new behavior also applies to subsequent updates unless otherwise specified. If a product edition appears with the product version, behavior is different in that product edition.

Unless otherwise specified, any statement of optional behavior in this specification that is prescribed using the terms "SHOULD" or "SHOULD NOT" implies product behavior in accordance with the SHOULD or SHOULD NOT prescription. Unless otherwise specified, the term "MAY" implies that the product does not follow the prescription.

8 Change Tracking

No table of changes is available. The document is either new or has had no changes since its last release.

9 Index

N

O

P

R

S