OMA DRM v2.0 Extensions for Broadcast Support

Draft Version 1.0 – 10 September 2005

Open Mobile Alliance

OMA-TS-DRM-XBS-V1_0-20050910-D

Contents

Figures

Tables

1. Scope

Open Mobile Alliance (OMA) specifications are the result of continuous work to define industry-wide interoperable mechanisms for developing applications and services that are deployed over wireless communication networks.

The scope of OMA "Digital Rights Management" [DRM-v2] is to enable the consumption of digital content in a controlled manner. The content is consumed on authenticated devices per the usage rights expressed by the content owners. The OMA DRM work addresses the various technical aspects of this system by providing appropriate specifications for content formats, protocols, and the rights expression language.

The scope for this specification is the application of the OMA "Digital Rights Management" specifications in a typical broadcast environment in which devices might only be capable of receiving information broadcast over a shared medium. It refers to the general OMA "Digital Rights Management" [DRM-v2] documents as its foundation. The causes defined in this document take precedence over those specified by the foundation documents, thus creating a broadcast interpretation of the OMA Digital Rights Management standard.

2. References

2.1 Normative References

[AES_WRAP] NIST Key Wrap, National Institute of Standards and Technology, 16 November 2001

[EUROCRYPT] EN 50094:1992 - CLC/TC 206 Access control system for the MAC/packet family: EUROCRYPT, 1992

[FIPS 197] National Institute of Standards and Technology, Specification for the Advanced Encryption Standard (AES) FIPS 197. November 26, 2001

[FIPS 198] The Keyed-Hash Message Authentication Code (HMAC), Information Technology Laboratory National Institute of Standards and Technology, Gaithersburg, MD 20899-8900, Issued March 6, 2002

[IOPPROC] "OMA Interoperability Policy and Process", Version 1.1, Open Mobile Alliance™, OMA-IOP-Process-V1_1, URL:http://www.openmobilealliance.org/

[PKCS#1] PKCS #1 v2.1: RSA Cryptography Standard, RSA Laboratories, June 14, 2002

[RFC 1738] RFC 1738, Uniform Resource Locators (URL), T. Berners-Lee - CERN, L. Masinter - Xerox Corporation, M. McCahill - University of Minnesota, December 1994

[RFC 2104] RFC 2104, HMAC: Keyed-Hashing for Message Authentication. H. Krawczyk, M. Bellare, R. Canetti. February 1997

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

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

[RFC 2406] RFC 2406, IP Encapsulating Security Payload (ESP). S. Kent, R. Atkinson. November 1998

[RFC 3566] RFC 3566, The AES-XCBC-MAC-96 Algorithm and Its Use With IPsec, S. Frankel (NIST) H. Herbert (Intel), September 2003

[RFC 3629] RFC 3629, UTF-8, a transformation format of ISO 10646. F. Yergeau, November 2003.

[RFC 3664] RFC 3664, The AES-XCBC-PRF-128 Algorithm for the Internet Key Exchange Protocol (IKE), P. Hoffman VPN Consortium, January 2004

[VERHOEF_1969] Verhoef J, Error detecting decimal codes, Mathematical Centre Tract 29, The mathematical Centre, Amsterdam, 1969.

2.2 Informative References

[DRM-v2] "Digital Rights Management", Open Mobile Alliance™, OMA-DRM-DRM-V2_0, URL:http://www.openmobilealliance.org/

[DRMARCH-v2] "OMA DRM Architecture Overview", Open Mobile Alliance™, OMA-DRM-ARCH-V2-0, URL:http://www.openmobilealliance.org/

[DRMCF-v2] "DRM Content Format", Open Mobile Alliance™, OMA-DRM-DCF-V2_0, URL:http://www.openmobilealliance.org/

3. Terminology and Conventions

3.1 Conventions

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in [RFC2119].

All sections and appendixes, except “Scope” and “Introduction”, are normative, unless they are explicitly indicated to be informative.

3.2 Definitions

Receiver Device	A OMA DRM Device without explicit return channel, capable only of receiving broadcast material. Note that a receiver device can still have an implicit return channel: it may present information, triggers and dialogs to the user who may “implement” the return channel in various ways (e.g. telephone, web portal, service desk).
Enhanced Device	A OMA DRM Device with bi-directional communications channel, but also suited to receive information via the broadcast channel.

3.3 Abbreviations

OMA	Open Mobile Alliance
AES	Advanced Encryption Standard
BAK	BCRO Authentication Key
BCD	Binary Coded Decimal
BCRO	Broadcast Rights Object
DRD	Device Registration Data
DRM	Digital Rights Management
ESP	Encapsulating Security Payload
HMAC	Hashed Message Authentication Code
IPsec	IP Security
LDK	Local Domain Key
LLDF	Long-form Local Domain Filter (a.k.a. longform_domain_id)
MAC	Message Authentication Code
MJD	Modified Julian Date
NDD	Notification of Detailed Data
OCSP	Online Certificate Status Protocol
PAK	Programme Authentication Key
PAS	Programme Authentication Seed
PDR	Push Device Registration
PEAK	Programme Encryption / Authentication Key
PKC	Public Key Certificate

PKC-ID	PKC Identifier: the hash of the Public Key Certificate
PKI	Public Key Infrastructure
PRF	Pseudo Random Function
RI	Rights Issuer
RIAK	Right Issuer Authentication Key
RO	Rights Object
ROT	Root Of Trust
RSA	Rivest-Shamir-Adelman public key algorithm
SAK	Service Authentication Key
SAS	Service Authentication Seed
SEAK	Service Encryption / Authentication Key
SGK	Subscriber Group Key
SHA-1	Secure Hash Algorithm
SLDF	Short-form Local Domain Filter (a.k.a. shortform_domain_id)
TAK	Traffic Authentication Key
TAS	Traffic Authentication Seed
TDK	Token Delivery Key
TEK	Traffic Encryption Key
UDF	Unique Device Filter
UDK	Unique Device Key
UDN	Unique Device Number
UDP	User Datagram Protocol
UGK	Unique Group Key

3.4 Notations

E{K}(M)	Encryption of message ‘M’ using key ‘K’
D{K}(M)	Decryption of message ‘M’ using key ‘K’
A || B	Concatenation of A and B
LSBm(X)	The bit string consisting of the *m* least significant bits of the bit string *X*.
MSBm(X)	The bit string consisting of the *m* most significant bits of the bit string *X*.

4. Introduction

Digital Rights Management [DRM-v2] defines the mechanisms to deliver DRM Content and Rights Objects to a consuming device. In the existing specification suite, devices are assumed to be capable of two-way interaction with other entities, such as a Rights Issuer. In a typical broadcast environment, this may not be the case and devices may exists that can only receive information broadcast over a shared medium.

In general the need for adaptations, extensions and guidelines has been identified for the following OMA Digital Rights Management [DRM-v2] items:

- ROAP Protocol

  The ROAP protocol is specified assuming a bi-directional communication mechanism between Device and Rights Issuer. A broadcast (i.e. uni-directional) equivalent for the functionality provided by the ROAP protocol is required. Bandwidth usage is very important in broadcast and protocol messages should be optimised for size.

- Rights Expression Language

  There is a need for additional types of usage that are typical to the broadcast model, e.g. time-shift, record, edit. These may also have non-standard constraints such as impulse-pay-per-view, prepaid.

- Subscription Group Addressing

  This is a feature that allows – per instance of content protection – to define the exact group of broadcast receivers that will be capable of accessing the protected content. It is required for fine-grained management of broadcast subscription services.

- Authentication of broadcast Rights Objects and broadcast content

  The bandwidth efficiency requirements of broadcast systems may necessitate a broadcast specific authentication scheme for rights objects and content.

- Broadcast Service Support

- Usage Metering

This specification is not stand-alone; it must be interpreted in the context of the existing OMA DRM v2.0 suite of specifications. Its goal is to provide alternative mechanisms for those parts of the standard that do not comply to the specific constraints of broadcast systems: one-way communication and bandwidth efficiency. Next to that, it also defines support for additional broadcast concepts such as ‘broadcast service’, (frequent) re-keying of broadcast content protection and broadcast usage models.

5. Authentication

5.1 Theory of operation

Following picture explains the authentication "hierarchy" of the system.

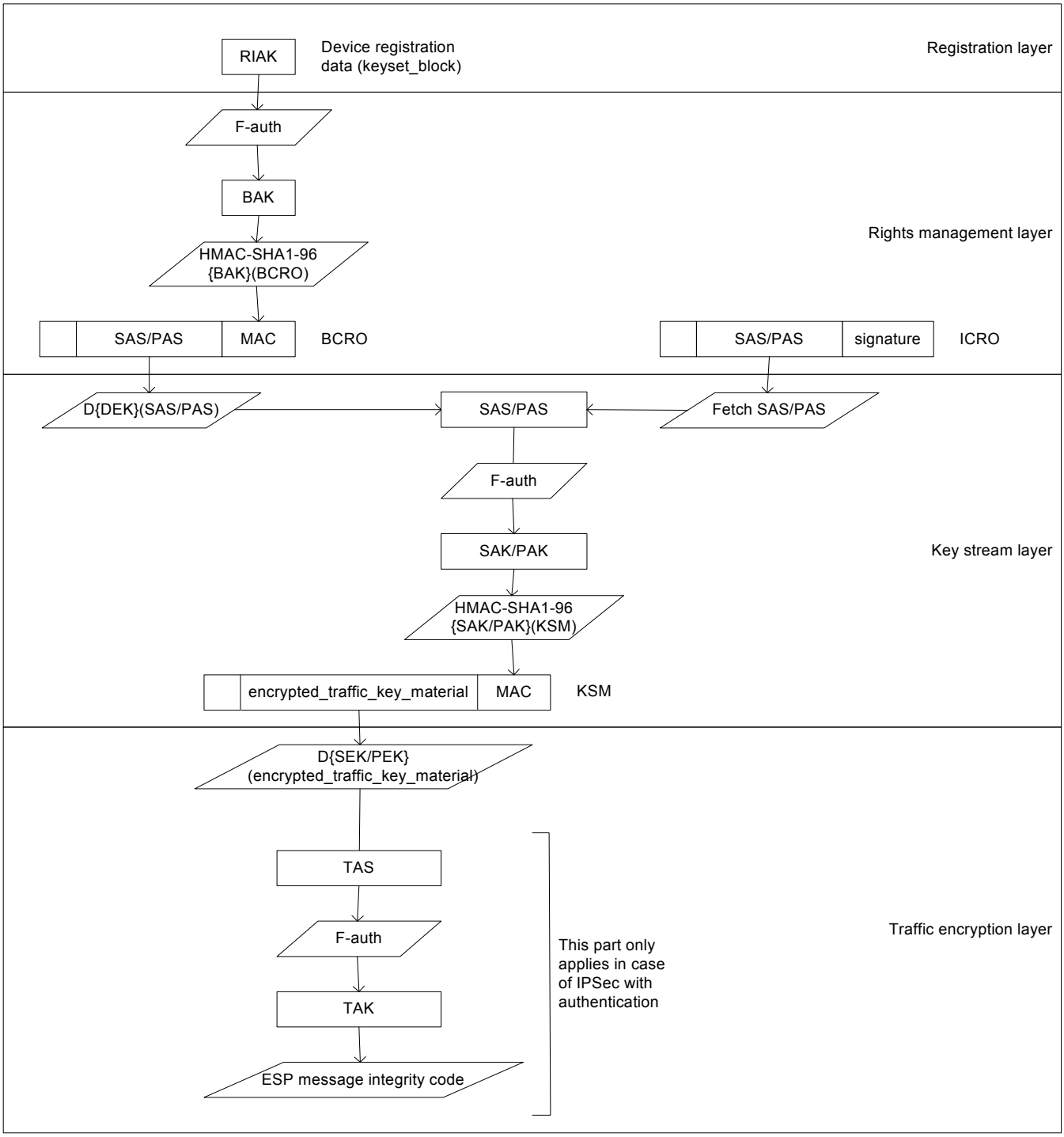


Figure 1: Authentication hierarchy

Key: F-auth is authentication key generation function.

5.1.1.1 Authentication keys on traffic layer

When IPsec is used with authentication, the message SHALL be verifiable by the ESP integrity code. This SHALL be done by means of the BCRO Authentication Key (BAK) which is derived from the RI Authentication Key (RIAK) which is delivered during registration.

5.1.1.2 Authentication keys on key stream layer

The KSM SHALL be authenticated and the integrity of the message SHALL be verified. This SHALL be done by means of the Program Authentication Key (PAK) and/or the service authentication key (SAK), which are derived from the Program Authentication Seed (PAS) and the Service Authentication Seed (SAS), which are delivered as part of the RO.

5.1.1.3 Authentication keys on rights management layer (broadcast mode)

The BCRO SHALL be authenticated and the integrity of the message SHALL be verified. This SHALL be done by means of the BCRO Authentication Key (BAK), which is derived from the RI Authentication Key (RIAK), which is delivered during registration.

5.1.1.4 Authentication keys on registration layer (broadcast mode)

The RI Authentication Key (RIAK) is delivered during registration as part of the device_registration_response().

6. Broadcast Device and Domain Management

6.1 Device Registration

6.1.1Offline notification of detailed device data

To register the device data has to be notified to the RI. There are two cases for the notification of device data to the RI:

Case 1: The device has never been registered before and is activated by the user.

There are two possibilities in which the device has no direct communication back channel to contact the RI but needs to report device data to the RI:

- The device has no interactivity channel or the interactivity channel is not able to make a connection to the RI, but the device is able to create an other connection to a connected OMA device. This device is called an unconnected device, and is covered in [OMA_DMR2] section 14.
- The device has no interactivity channel and is unable to make a connection to an interactive device. This device is called a broadcast (only) device. In this case the 1-pass binary push registered device protocol is used, as is specified in this document.

Case 2: The device has been registered at the RI before and needs to be re-registered.

- In this case the RI uses the 1-pass binary inform registered device protocol to send a message ordering the device to re-register, as is specified in this document.

Following sequence chart explains the registration for broadcast only mode of operation.

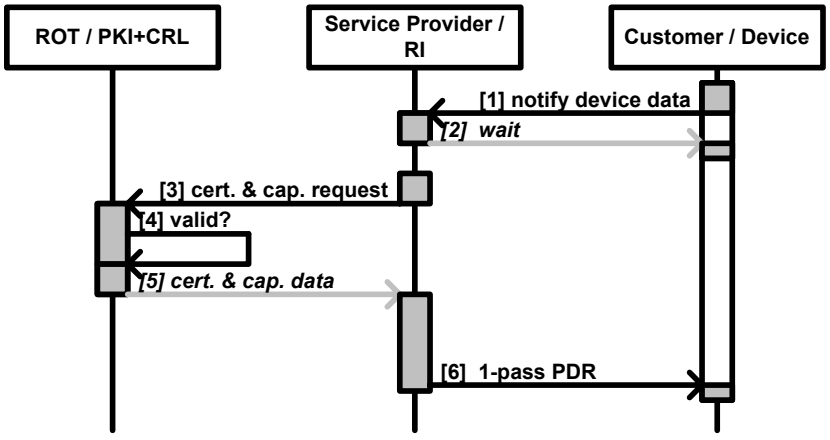


Figure 1: Registration for broadcast mode of operation with one ROT

Note: Notification of device data to the Rights Issuer is performed off-line. Transmission of the registration data from the RI to the device is performed on-line via the broadcast channel.

Explanation of the protocol:

- Once the Rights Issuer (RI) has the device data from the device [1] via the protocol described in this CR section **Error! Reference source not found.**, the RI contacts the Root of Trust (ROT) [3], while the device is entered into registration mode and awaits the registration data [2].

- The Root of Trust decides whether the requested device data is valid or not and whether or not the requested certificate and capabilities data can be passed to the RI.

- If the RI received the requested certificate and capabilities from the ROT [5], the RI SHALL send back a registration data message to the device [6].

- The RI uses the 1-pass binary Push Device Registration data (a.k.a. PDR) protocol to send the registration data over the broadcast network. The PDR protocol is described in section ~~5.1.3.1~~6.1.3, together with the registration data (in the format of the device_registration_response() message). The RI MAY decide to send an error status with the message or send valid registration data containing the data required to create an RI context.

- A device listening for device_registration_response() messages will look for messages with the corresponding message_tag. On every message with a matching message_tag the device will check the longform_udn() parameter. If this matches (any of) the device's local UDN(s), the device will process the message and will start trying to decrypt the secret data in it.

- If the device does not receive registration data within a timeout, the device leaves the registration mode and stops listening for device_registration_response() messages.

- Subsequent distribution of Right Objects at regular intervals is done with a message send as an inform message using the 1-pass Inform Registered Device protocol.

6.1.1 Offline notification of detailed device data

6.1.1.1 Theory of operation

offline Notification of Detailed Devicedata protocol

Note: This protocol is also known as the "offline NDD protocol", short for offline Notification of Detailed Data protocol.

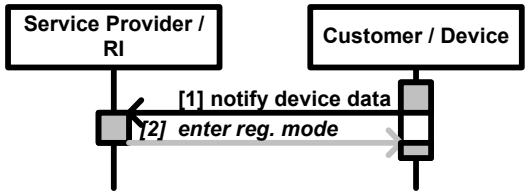


Figure 2: offline NDD protocol

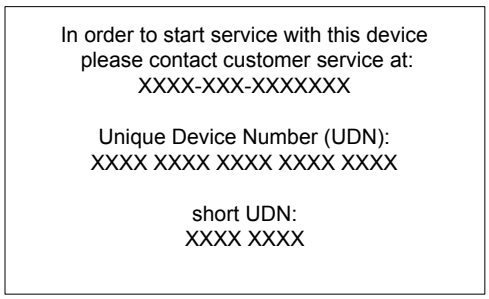
N.B: notification of device data is performed off-line. The device data (device_data_inform() message) is defined in section 6.1.1.3.1~~6.4~~.

Explanation of the protocol:

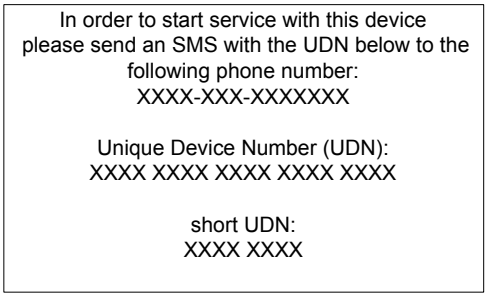
- The purpose of this protocol is to transfer device data somehow to the RI, in case the device does not support a return channel to the RI. After the user has let the device know that he wants to register at an RI, the device produces the device_data_inform() message (refer to section ~~6.4~~6.1.1.3.1 for details) and make this data available to the user.

- The data of the device_data_inform() message consists of a several series of decimal digits and possibly an alphanumeric character. The user needs to transfer these series somehow to the RI. In order to aid the user in this, the device MAY display a dialogue with instructions. Notifying the device data can be done in various ways, for example by showing the user of the device a dialogue on the screen of the device, displaying the device data and a telephone number to call for vocal notification of the device data. Another example is to display instructions to send an SMS message via a mobile phone to the RI, or else.

An example of a displayed message follows, where the following information is reported back to the RI. Please note that when using displays like in the examples, it is useful to present the numeric fields in the order shown[1]:



An example dialogue showing instructions for vocal notification of UDN to callcenter



An example dialogue showing instructions for notification of UDN per SMS to callcenter

Figure 3: examples of notification displays

- If the device does not support a return channel to the RI, the device data (device_data_inform() message) SHALL be notified off-line, using the offline Notification of Detailed Devicedata protocol. The device data to notify SHALL be reduced by a special protocol (refer to section ~~5.1.1.2~~ 6.1.1.2).
- After the notification of the device data, user needs to put the device into registration mode [2]. When put into registration mode, device SHOULD start to listen for the device registration data for a limited time.

6.1.1.2 Unique Device Number (UDN) protocol

To reduce the amount of data that is to be notified to the RI, the device data protocol takes care of data reduction. To ease the detection of errors during the registration process, the device data protocol will also allow detection of errors in the notified device data.

Following data format SHALL be used to construct a Unique Device Number (a.k.a. UDN):

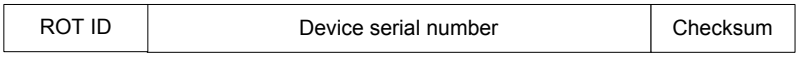


Figure 4: Unique Device Number

[1] Note: It is the sequence of the defined values that is specified. The use of dashes as the delimiter is shown with an example placement to be consistent with the examples used elsewhere in this specification. The text portion of this screen is shown as an example only; there is no implied requirement to duplicate the exact wording or formatting shown. ~~The numeric fields SHALL be included as defined above P(p~~lease note: the short UDN will only be displayed after the first registration, when that data MAY be available for display~~)~~.

Table 1: UDN explanation

Field	Length (digits)	supporting up to
rot_id	3	1000 ROT
device_serial_number	14	10,000 Billion devices
checksum	3	

This totals to 20 digits. The fields are explained below:

rot_id - The first 3 digits in the UDN identify the ROT. Every ROT has an own unique ID.

device_serial_number - There are 10,000 billion (10^{14}) possible device_serial_numbers. This range MAY be subdivided in subranges from which separate entities may issue device serial numbers independently.

checksum - The final digits of the device ID number are check digits, akin to a checksum. The 3 digits allow 1 out of 10^3 possible errors to remain undetected. Please refer to appendix ~~A.7~~ A.2.2 for an explanation of the algorithm.

6.1.1.2.1 Message syntax

The 20 digits of the UDN are encoded in BCD format into the longform_udn(). The message syntax is specified below:

Table 2: longform_udn

fields	length	type
`longform udn(){`		
`  rot_id`	`12`	`bslbf`
`  device serial number`	`56`	`bslbf`
`  checksum`	`12`	`bslbf`
`}`		

Note: The UDN SHALL be constructed according to the abovementioned message syntax. When the UDN is displayed or in other ways presented to the end user, a(ny) checksum digit with value "10" SHALL be represented by an alphanumeric character different from {0..9}, for example X or Z. This ensures the RI will always receive 20 "characters" from the end user notification, providing an easy way to count if the information is complete.

6.1.1.3 Device data – device_data_inform message

6.1.1.3.1 device data inform m~~M~~essage description

The Device data SHALL prove that it is unique. In a one way case the device notifies this device data, yet the length of the unique device data SHOULD remain concise.

Because devices can be uniquely identified by the PKI, it is not needed to incorporate unique data like the device certificate into the (device specific) registration data. The OMA DRM 2.0 certificate is global and the link between the manufacturer and the device can be requested from the PKI, based on the device ID.

Table 3: Notify device data message parameters

Device_Data_Inform()		
parameter	(M)andatory / (O)ptional[2]	Remark
version	M	
contact_nr	O	
longform_udn()	M	

version - is a <major> representation of the highest ROAP version number supported by the Device. For this version of the protocol, the *version* field SHALL be set to value "**1**".

contact_nr - is the number to be contacted in order to register the device. It can be a phone number or an SMS number. This number MAY have been entered into the device at production time and if so MAY be shown in the registration display (refer to section 5.1.1.1for an example). This number could also be provided in human readable form in other ways.

longform_udn() - identifies the unique_device_number to the RI. The UDN SHALL be part of the credentials entered into the device, like the private key and the certificate. Refer to section ~~5.1.1.1~~6.1.1.2 for details.

6.1.1.3.2 Message syntax

Since this is an offline protocol the device data is not really formed into a message that can be transmitted. The device data is decimal and formatted as follows:

Table 3: Device data

Parameter	Format and length	Description
version	1 byte	
contact_number	15 bytes	dependent on target telco network
longform_udn	21 bytes	UDN protocol

6.1.2 Offline notification of short device data~~Broadcast Registration~~

~~{EN: It seems that the title of this section has been changed from 'Offline notification of short device data' by accident. It should be considered to correct this via a clerical CR, or by a joint meeting decision.}~~

The end user of a device might wish to formulate a particular request to the RI. He uses following specified behaviour:

[2] key: (O)ptional means that the user of the message MAY include the parameter in the message, but the device SHALL support the interpretation of the parameter. (M)andatory means that the user of the message SHALL include the parameter in the message.

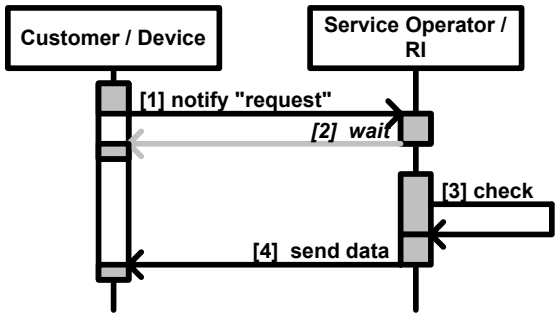


Figure 2: action request round trip

Explanation of the protocol:

- The end user of the device formulates a request and notifies this request to the user [1] as specified in subsequent sections.
- The end user waits after the request has been notified to the Customer Operations Centre in a successful way [2].
- The RI might execute additional checks and composes the data [3].
- The RI MAY send a data message to the device to update data in the device, start the execution of a particular action to produce a desired result or to inform an error status. [4].

6.1.2.1 Theory of operation

Note: This protocol is also known as the "offline NSD protocol", short for offline Notification of Short Data protocol.

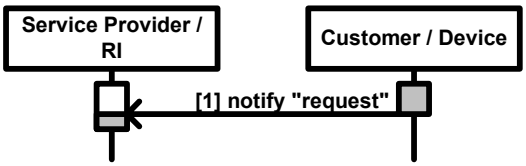


Figure 3: offline NSD protocol

Note: Notification of device data is performed off-line. Refer to Table 2Table 2Table 2 for an overview of the possible "requests".

Explanation of the protocol:

- The user may notify a short decimal code called the action request code (ARC) to the RI via offline methods (e.g. telephone call or SMS or else). The code SHALL be constructed as follows:

Short_udn	Action_code	Checksum

Figure 4: Action Request Code (ARC)

Note that for some of the ARCs (e.g. the ARC token_consumption_report), the user MAY have to notify more digits to the RI than the ones of the ARC.

Table 1: NSD action request code fields

ARC fields	Length (digits)	supporting up to
short_udn	8	100 Million devices
action_code	2	99 action codes
checksum	2	refer to A.1

This totals to 12 digits. The fields are explained below:

short_udn. The offline notification can be performed faster if the long form UDN is not used, but a shorter form instead. After first time notification of the device data to the RI, the RI MAY issue a short version of the full UDN (called short_form_udn) that is carried in the device_registration_response() message. The short_form_udn number is used to speed up the offline interaction with the RI. If this number is stored into the device, subsequent "requests" by the user of the device can be notified offline much quicker by using the short_form_udn number concatenated by a standardised action code.

Please note: In cases where the device needs to be identified uniquely in another network than it's home network where it was registered, the short_udn cannot be used because the (new / different) RI does not have the short_udn in it's database. In this case the only possibility for the hosting RI to identify the device uniquely would be via the long_udn. It is the responsibility of the device to decide when it is appropriate to use the long_udn instead, for example by comparing the Service Operations Centre (SOC) ID received with the SOC ID remembered from registration.

action_code. Following the short_udn the user of the device can notify an action code to the RI. The NSD protocol defined in this specification SHALL use following action_codes to construct the ARC:

Table 2: NSD action types

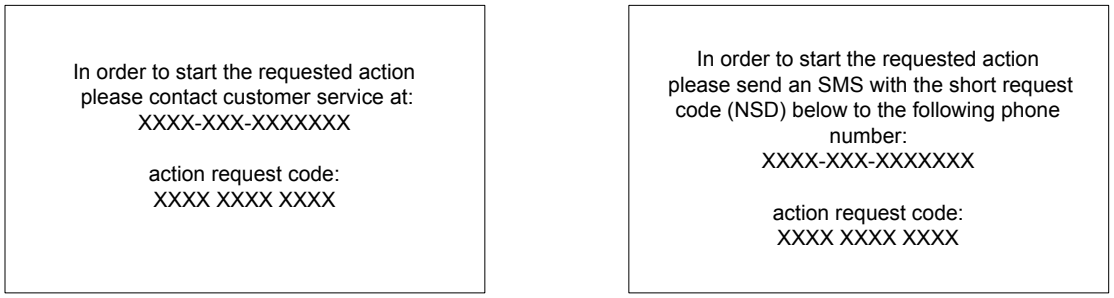
action type	action code (d)	described in section
re-registration (only at same RI)	{0d01}	(*1)6.1.2.1.1
resend BCRO	{0d02}	(*1)11.9
reserved for future use	{0d03,..,0d09}	(*1)
join domain	{0d10,..,0d19}	(*1)6.1.2.1.2
leave domain	{0d20,..,0d29}	(*1)6.1.2.1.3
purchase	{0d30}, whereas content identification is supplied by ESG.	(*1)
token_consumption_report	{0d31,..,0d39}	(*1)6.1.2.1.4
metering	{0d40,..,0d49}	(*1)
token_request	{0d50,..,0d59}	(*1)
notify DRM time drift	{{0d7}+{0d0,..,0d9},..,{0d8}+{0d0,..,0d9}}	(*1)6.1.2.1.6
reserved for future use	{0d90,..,0d99}	(*1)

(*1): Note to Editor: to be defined later in actual XBS document.

checksum. The constructed short_udn and action_code is appended by checksum digits. Please refer to A.2.1 for an explanation of the algorithm.

An example: In order to request to re-register, a sample NSD action request code could look like: "1660 8731 0112". An example of a displayed message follows, where the following information is reported back to the RI[3]:

[3] Note: It is the sequence of the defined values that is specified. The use of dashes as the delimiter is shown with an example placement to be consistent with the examples used elsewhere in this specification. The text portion of this screen is shown as an example only; there is no implied requirement to duplicate the exact wording or formatting shown. The numeric fields SHALL be included as defined above (please note: the short UDN will only be displayed after the first registration, when that data MAY available for display



An example dialogue showing instructions for vocal notification of ARC to callcenter

An example dialogue showing instructions for notification of ARC per SMS to callcenter

Figure 5: samples of notification displays showing an ARC message

6.1.2.1.1 Request re-registration (only at same RI)

After sending this ARC the user will wait until he receives the confirmation of the RI in the form of a device_registration_response() message. Refer to 6.1.3.2

6.1.2.1.2 Request join domain

The Action Request Code (ARC) for the NSD protocol is formed according to following rules:

- the first digit is used to notify the join domain action.
- the second digit is used as a device_nonce to help the device to keep track of join domain requests.

After notifying the ARC to the RI the user MAY notify a particular domain group number identifying a domain where the device is to be entered. The RI SHALL incorporate the device_nonce from the request in the response message.

6.1.2.1.3 Request leave domain

The Action Request Code (ARC) for the NSD protocol is formed according to following rules:

- the first digit is used to notify the join domain action.
- the second digit is used as a device_nonce to help the device to keep track of leave domain requests.

After notifying the ARC to the RI, the user needs to notify a particular domain group number identifying a domain where the device is to be removed from. The device SHALL display a domain ID. The RI SHALL incorporate the device_nonce from the request in the response message.

6.1.2.1.4 Token consumption report

The Action Request Code (ARC) for the NSD protocol is formed according to following rules:

- the first digit is used to notify the token consumption report.
- the second digit is used as a device_nonce to help the device to keep track of token consumption reports.

After notifying the ARC to the RI the user should notify the token consumption data. The device SHALL display the token consumption data e.g. to the left of or below the digits of the ARC for the token consumption report. The RI SHALL incorporate the device_nonce from the request in the response message.

An example of a displayed message follows, where the following information is reported back to the RI[4]:

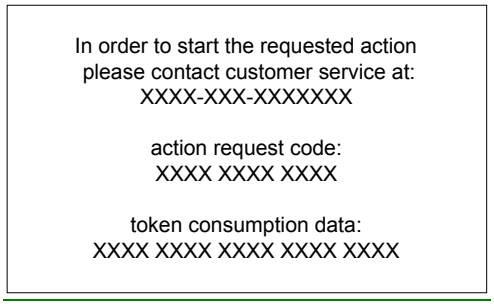


Figure 5: sample of token consumption reporting notification display

6.1.2.1.5 Token consumption data definition

The token consumption data are defined below.

Table 4: token consumption data

Field	Length (digits)	supporting up to
tokens_consumed	4	9999 tokens to be reported
report_authentication_code	13	
checksum	3	

This totals to 20digits. The fields are explained below:

tokens_consumed – This field contains the amount of tokens the device wished to report as consumed to the RI. See section A.11 for more information.

report_authentication_code – This field contains the authentication code for the value in the tokens_consumed field and the value of the device_nonce (second digit of the action_code of the ARC of this message). See A.10 for the computation of the report authentication code.

checksum - The final digits of the device ID number are check digits, akin to a checksum. The 3 digits allow 1 out of 10^3 possible errors to remain undetected. The checksum algorithm used is the UDN checksum, see section A.2.2.

6.1.2.1.6 Notify DRM time drift

Time drift is expressed in minutes and rounded up to next multiple of 5 minutes. The range is 0..100 minutes, whereas value 69 will decode as timedrift >= 100. Some examples of valid ARC values are given below:

E.g.1: Device notifies 4 minutes timedrift from newly received DRM time message: action code is 70.

E.g.2: Device notifies 38 minutes timedrift from newly received DRM time message: action code is 78.

[4] Note: It is the sequence of the defined values that is specified. The use of dashes as the delimiter is shown with an example placement to be consistent with the examples used elsewhere in this specification. The text portion of this screen is shown as an example only; there is no implied requirement to duplicate the exact wording or formatting shown. The numeric fields MUST be included as defined above (please note: the short UDN will only be displayed after the first registration, when that data MAY available for display

E.g.3: Device notifies 235 minutes timedrift from newly received DRM time message: action code is 89.

6.1.3 Broadcast registration

To register the device data has to be notified to the RI. There are two cases for the notification of device data to the RI:

Case 1: The device has never been registered before and is activated by the user.

There are two possibilities in which the device has no direct communication back channel to contact the RI but needs to report device data to the RI:

- The device has no interactivity channel or the interactivity channel is not able to make a connection to the RI, but the device is able to create an other connection to a connected OMA device. This device is called an unconnected interactive device, and is covered in OMA DRM.

- The device has no interactivity channel and is unable to make a connection to an interactive device. This device is called a broadcast (only) device. In this case the 1-pass binary push registered device protocol is used, as is specified in this document.

Case 2: The device has been registered at the RI before and needs to be re-registered.

- In this case the RI uses the 1-pass binary inform registered device protocol to send a message ordering the device to re-register, as is specified in this document.

Following sequence chart explains the registration for broadcast only mode of operation.

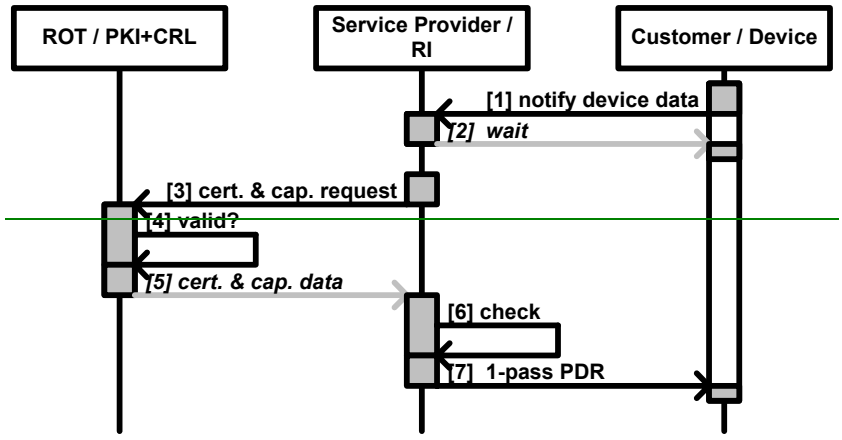


Figure 5: Registration for broadcast mode of operation with one ROT

Note: Notification of device data to the Rights Issuer is performed off-line. Transmission of the registration data from the RI to the device is performed on-line via the broadcast channel.

Explanation of the protocol:

- Once the RI has the device data from the device [1] via the protocol described in section 6.1.1.3, the RI contacts the ROT [3], while the device is entered into registration mode and awaits the registration data [2].

- The ROT implements a Public Key Infrastructure (a.k.a. PKI). The PKI looks up the certificate and capabilities belonging to the device data in question [4]. The ROT should have a Certificate Revocation List (a.k.a. CRL). In any case it is the responsibility of the ROT to decide whether the requested device data is valid or not and whether or not the requested certificate and capabilities data can be passed to the RI.

- Assuming the RI received the requested certificate and capabilities from the ROT [5], the RI will perform some last checks [6] and SHALL send back a registration data message to the device [7].

- The RI uses the 1-pass binary Push Device Registration data (a.k.a. PDR) protocol to send the registration data over the broadcast network. The PDR protocol is described in section 5.1.3.1. The registration data (in the format of the device_registration_response() message) is specified in section 5.1.3.2. The RI MAY decide to send an error status with the message or send valid registration data containing the data required to create an RI context.

- A device listening for device_registration_response() messages will look for messages with the corresponding message_tag. On every message with a matching message_tag the device will check the long_form_udn parameter. If this matches (any of) the device's local UDN(s), the device will process the message and will start trying to decrypt the secret data in it.

- If the device does not receive registration data within a timeout, the device leaves the registration mode and stops listening for device_registration_response() messages.

- Subsequent distribution of Right Objects at regular intervals is done with a message send as an inform message using the 1-pass Inform Registered Device protocol.

6.1.3.1 Theory of operation

Note: This protocol is also known as the "1-pass PDR protocol", short for Push Device Registration protocol.

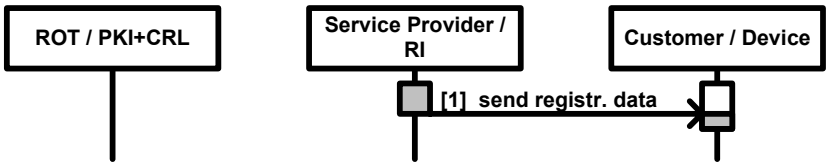


Figure 776: 1-pass PDR protocol – (first) device registration

Note: Transmission of registration data is performed on-line via the broadcast channel. The registration data (device_registration_response() message) is specified in section 6.1.3.2

Explanation of the protocol:

- The RI SHALL use the 1-pass binary Push Device Registration data (a.k.a. PDR) protocol to send registration data over the network [1]. The registration data can be the device_registration_response() message (refer to section 5.1.3.26.1.3.2) or the domain_registration_response() message (Note to Editor: refer to appropriate XBS section). (refer to section 6.4.3.1). The RI SHALL use the RI mechanisms described in section (Note to Editor: refer to appropriate XBS section) 11 to address the message to a device. The RI SHALL include a valid keyset in the message.

- A device listening for device_registration_response() (or domain_registration_response()) messages SHALL look for messages with the corresponding message_tag. On every message with a matching message_tag the device SHALL check the long_form_udn parameter. If this matches (any of) the devices local UDN(s) the device SHALL start validating the signature and check the RI certificate (chain.). If both (UDN and signature) are valid the device detects this message is really addressed to it. The device SHALL start processing the message and SHALL start trying to decrypt the secret data in it. If the message is correct, the device SHALL store the new keyset with key(s). The devise SHALL delete the old keyset (if applicable).

- After a timeout the device SHALL leave the registration mode and stops listening for device_registration_response() messages.

6.1.3.2 Registration data – device_registration_response message

6.1.3.2.1 Device registration response m~~M~~essage description

Using the 1-pass PDR protocol the RI SHALL send a device_registration_response() message with the registration data to the device as specified below:

Table 5~~5~~4: device registration response message description

Device_Registration_Response()		
Parameter name	(M)andatory / (O)ptional[5]	remark
message_tag	M	global, not encrypted
protocol_version	M	global, not encrypted
longform_udn()	M	global, not encrypted
status	M	device specific, not encrypted
certificate_version	M	global, not encrypted
ri_certificate_counter	M	global, not encrypted
c_length	M	global, not encrypted
ri_certificate	M	global, not encrypted
ocsp_response_counter	M	global, not encrypted
r_length	M	global, not encrypted
ocsp_response	M	global, not encrypted
local_time_offset_flag	M	device specific, not encrypted
time_stamp_flag	M	device specific, not encrypted
subscriber_group_key_flag	M	device specific, not encrypted
signature_type_flag	M	global, not encrypted
short_udn_flag	M	device specific, not encrypted
surplus_block_flag	M	device specific, not encrypted
keyset_block_length	M	device specific, not encrypted
unique_group_key	O	device specific, encrypted
subscriber_group_key	O	device specific, encrypted
unique_device_key	O	device specific, encrypted
unique_device_filter	M	device specific, encrypted
ri_authentication_key	M	device specific, encrypted
local_domain_key	O	device specific, encrypted
shortform_domain_id	M	device specific, encrypted
drm_time	M	device specific, not encrypted
local_time_offset	O	device specific, not encrypted
registration_timestamp_start	O	device specific, not encrypted
registration_timestamp_end	O	device specific, not encrypted
shortform_udn	O	device specific, not encrypted
signature_block	M	device specific, not encrypted

message_tag - This parameter identifies the type of the message. Refer to section ~~Error! Reference source not found.~~A.8 for the value of the message_tag.

protocol_version - This parameter indicates the protocol_version of this message. The device SHALL ignore messages that have a protocol_version number it doesn't support. The value of the protocol_version of this message is set to 0x0 (i.e. the original format).

[5] key: (O)ptional means that the user of the message MAY include the parameter in the message, but the device MUST support the interpretation of the parameter. (M)andatory means that the user of the message SHALL include the parameter in the message.

If set to 0x0 the format specified in the this version of the specification is used. If set to anything else than 0x0, then the format is beyond the scope of this version of the specification

longform_udn() - The long form of the UDN. Refer to section ~~6.1.1.2~~ 6.1.1.2.1 for details.

status - The status parameter SHALL indicate one of the values explained in the following table. The device SHALL ignore messages with other error values.

Table 6~~6~~5: Status values

status value	meaning
Success	The registration request was executed successfully and the RI completed all data. The device SHALL process the message.
UnknownError	The RI encountered an unknown error after receiving the registration request. The device MAY put forward a subsequent registration request to the RI (context).
NotSupported	The RI does not support the registration request.
AccessDenied	The RI decided that the device will not be granted access to the service and stops the registration. The RI will stop listening to future registration requests of this device. The device is forced to refrain from future registration and SHALL **suppress** broadcast and/or mixed-mode registration requests to the particular RI (context).
NotFound	The RI decided that the device could not be found (offline UDN and/or UaProf). The device MAY put forward a subsequent registration request to the RI (context).
MalformedRequest	The RI decided that the registration request was malformed and will **force** the device to execute a (re)-registration at once. The device SHALL enter (re)registration mode.

Note: Refer to A.3 for the value of the error codes.

certificate_version - is a numerical representation of the version of the RI certificate. Using the certificate_version parameter the device can decide if it is needed to update the RI certificate (if it was stored before).

Table 7~~7~~6: description of certificate_version parameter

Parameter Fieldname	Field Value ($_h$)	supports
major_version_number	0x0,..,0xA	MSB_4(certificate_version)
minor_version_number	0x0,..,0xA	LSB_4(certificate_version)

The parameter is divided 2 fields of 4 bits, whereas the first 4 bits (MSB left) express the Major number and the last four bits (LSB right) express the Minor version. The major and minor number encode in bslbf format. 16 Major and 16 Minor versions are supported. For example: Major.Minor version <1.2> is expressed as $0001\ 0010_b$.

ri_certificate_counter - This parameter indicates the depth of the RI certificate chain.

number of certificate in chain	Value ($_h$)	remark
0	0x0	will signal absence of ri_certificate e.g. on error status to save bandwidth.
1	0x1	
2	0x2	
3	0x3	
4	0x4	
5	0x5	
6	0x6	
7	0x7	

Note: The certificate chain can have a depth of up to 7 RI certificates.

c_length - This parameter indicates the length in bytes of the ri_certificate.

ri_certificate() - This parameter SHALL be present. When present, the value of a *ri_certificate* parameter SHALL be a certificate chain including the RI's certificate. The chain SHALL NOT include the root certificate. The RI certificate SHALL come first in the list. Each following certificate SHALL directly certify the one preceding it.

The Device MAY store RI certificate verification data indicating that an RI certificate chain has been verified. The purpose of this is to avoid repeated verification of the same certificate chain. The RI certificate verification data stored in this way SHALL uniquely identify the RI certificate and SHALL be integrity protected. The Device SHOULD check if the RI certificate chain received in this parameter corresponds to the stored certificate verification data for this RI. If so, the Device NEED NOT verify the RI certificate chain again, otherwise the Device SHALL verify the RI certificate chain.

If an RI certificate is received that is not in the stored certificate verification data for this RI, and if the Device can determine (in the case of Broadcast Devices that support DRM Time) that the expiry time of the received RI certificate is later than the RI Context for this RI, and the certificate status of the RI certificate as indicated in the OCSP response is good (see [OCSP-MP]), then the Device SHALL verify the complete chain and SHOULD replace the stored RI certificate verification data with the received RI certificate data and set the RI context expiry time to that of the received RI certificate expiry time.

However, if the Device does store RI certificate verification data in this way it SHALL store the expiry period of the RI's certificate (as indicated by the notAfter field within the certificate) and SHALL compare the Device's current DRM Time with the stored RI certificate expiry time whenever verifying the signature on signed messages from the RI. If the Device's current DRM Time is after the stored RI certificate expiry time then the Device SHALL abandon processing the RI message and SHALL initiate the registration protocol.

ocsp_response_counter - This parameter indicates the depth of the OCSP response chain.

number of responses in chain	Value ($_h$)	remark
0	0x0	will signal absence of ocsp_response e.g. on error status to save bandwidth.
1	0x1	
2	0x2	
3	0x3	
4	0x4	
5	0x5	
6	0x6	
7	0x7	

Note: The certificate chain can have a depth of up to 7 OCSP responses.

r_length - This parameter indicates the length in bytes of the ocsp_response.

ocsp_response() - This parameter, when present, SHALL be a complete set of valid OCSP responses for the RI's certificate chain. The Device SHALL NOT fail due to the presence of more than one OCSP response element. A Device SHALL check that an OCSP response is present in the received message. If no OCSP response is present in the device_registration_response() message, then the Device SHALL abort the registration protocol.

local_time_offset_flag - Binary flag to signal presence of the parameter it describes:

local_time_offset_flag	Value ($_h$)	remark
data absent	0x0	
data present	0x1	

time_stamp_flag - Binary flag to signal presence of the parameter it describes:

time_stamp_flag	Value ($_h$)	remark
data absent	0x0	
data present	0x1	

subscriber_group_key_flag - The flag expresses how many subscriber_group_keys (a.k.a. SGK) are delivered with the registration data. When zero message broadcast is used, a set of 8 keys will support a group size of 256. A set of 9 keys will support a group size of 512. Other values or larger group sizes are not supported. A value larger than zero indicates that the registration data message delivers a set of zero message subscriber_group_key (s) to the device and that the device needs to use zero message broadcast style encryption to deduce the decryption key to decrypt the SEK.

subscriber_group_key_flag	Value ($_h$)	remark
data absent	0x0	will signal absence of keyset_block e.g. on error status to save bandwidth.
reserved for future use	0x1-0x7	not used in this version of the specification
set of (8) subscriber_group_key	0x8	
set of (9) subscriber_group_key	0x9	
reserved for future use	0xA-0xF	not used in this version of the specification

signature_type_flag - A flag to signal type of signature algorithm used:

signature_type_flag	Value ($_h$)	remark
RSA 1024	0x0	
RSA 2048	0x1	CMLA requirement (2004-2007)
RSA 4096	0x2	
reserved for future use	0x3	not used in this version of the specification

short_udn_flag - Binary flag to signal presence of the parameter it describes:

short_udn_flag	Value ($_h$)	remark
data absent	0x0	
data present	0x1	

surplus_block_flag - Binary flag to signal the presence of the parameter it describes:

surplus_block_flag	Value ($_h$)	remark
data absent	0x0	
data present	0x1	

keyset_block_length - This parameter indicates the length in bits of the total keyset_block. That is the part in the sessionkey_block() plus the optional second part from the surplus_block().

unique_group_key - An symmetric AES encryption key to address a unique group. This key is also known as UGK. The key length SHALL be 128 bit.

Note: This key is wrapped into the keyset_block. (Refer to 6.1.3.2.2 ~~5.1.3.2.2~~).

subscriber_group_key - An (set of) AES symmetric encryption key(s) which are used for the zero message subscriber_group_key deduction of the key needed to decrypt the SEK and/or PEK. These subscriber_group_key is also known as SGK. The key length SHALL be 128 bit.

Note: This key is wrapped into the keyset_block. (Refer to 6.1.3.2.2 ~~5.1.3.2.2~~).

unique_device_key - An AES symmetric key to address a unique device. This key is also known as UDK. The key length SHALL be 128 bit.

Note: This key is wrapped into the keyset_block. (Refer to 6.1.3.2.2 ~~5.1.3.2.2~~).

unique_device_filter - A [EUROCRYPT] style addressing scheme used to filter for messages like BCROs. A device address consists of 5 bytes and is unique within an operation. The shared address is defined as the 4 most significant bytes of the unique address. The least significant byte (byte 5) defines the position (0….255) in the group that shares an address. This means that each group consists of 256 members. An access mask, in an entitlement, is used to identify individual members. So if for a particular group only member 5 and 100 are allowed to have access to a service then their corresponding bits are set in the access mask. Take the device_id_mask equal to 252 ($1111\ 1100_b$) then the least significant byte of the device_id is masked and thereby creating a shared address. This address is also known as UDF.

Note: This address is wrapped into the keyset_block. (Refer to 6.1.3.2.2 ~~5.1.3.2.2~~).

ri_authentication_key - An AES symmetric key to verify MACs on BCRO and KSM messages. This key is also known as RIAK. The key length SHALL be 128 bit.

Note: This key is wrapped into the keyset_block. (Refer to 6.1.3.2.2 ~~5.1.3.2.2~~).

local_domain_key - An AES symmetric key to address a unique device. This key is also known as LDK. The key length SHALL be 128 bit.

Note: This key is wrapped into the keyset_block. (Refer to 6.1.3.2.2 ~~5.1.3.2.2~~).

longform_domain_id() – This parameter is also known as the Longform Local Domain Filter (LLDF). Please refer to ~~Error! Reference source not found.~~ section A.7.3. for the definition. The longform_domain_id() is used for mixed-mode operation. Note: This address is wrapped into the keyset_block. (Refer to 6.1.3.2.2).

shortform_domain_id – This parameter is also known as the Shortform Local Domain Filter (SLDF). Please refer to 6.1.3.2.2. An addressing scheme used to filter for messages like BCROs. The shortform_domain_id is used for broadcast mode of operation.

Note: This address is wrapped into the keyset_block. (Refer to 6.1.3.2.2 ~~5.1.3.2.2~~).

drm_time - This parameter defines the time in Universal Time Coordinated (UTC). This 40-bit field contains the current time and date in UTC and MJD. Refer to A.4 for calculation of the UTC and Modified Julian Date (MJD). This field is coded as 16 bits giving the 16 LSBs of MJD followed by 24 bits coded as 6 digits in 4-bit BCD.

EXAMPLE: 93/10/13 12:45:00 is coded as “0xC079124500”.

local_time_offset - This parameter indicates the local time offset from the (UTC) drm_time as explained in Annex A.4.

registration_timestamp_start - Indicates from what time onwards the registration data is valid. This is an extra mechanism above the expiration date of the RI certificate. (Note: please note that this parameter can also be used against replay attacks.)

registration_timestamp_end - Indicates from what time onwards the registration data is expires. This is an extra mechanism above the expiration date of the RI certificate. (Note: please note that this parameter can also be used against replay attacks.)

shortform_udn - This parameter allows the RI to give an own defined short number identifying the device. This number can be used as a shorter alternative to the UDN during offline notifications. The shortform_udn is coded in BCD format.

signature_block - The signature SHALL enable a single source authenticity check on the message. The algorithm used for the signature is RSA-1024 or RSA-2048 or RSA-4096. The signature will apply to the implementation guidelines of PCKS#1, as outlined in ~~A.11~~ A.5.

Note Message result:

The stored RI Context SHALL at a minimum contain:

- RI ID, Unique device filter (UDF).
- following keys:

- UGK, BGK1..n and/or UDK
- RIAK.
- SK

If domain addressing via an OMA DRM 2.0 domain is required the keyset SHALL (additionally to the standard addressing above) include following keys:

- LDK.
- Shortform Local Domain Filter (SLDF). A.k.a. "shortform_domain_id". Refer to ~~A.13.1~~A.7.1.

- For mixed-mode devices domain context SHALL additionally contain:
  - Longform Local Domain Filter (LLDF). A.k.a. "longform_domain_id()". Refer to ~~Error! Reference source not found.~~ A.7.3.
- A Device MAY have several Domain Contexts with an RI.
- The RI Context SHALL also contain an RI Context Expiry Time, which is defined to be the timestamp of the registration data if that was send and otherwise the expiration of the RI certificate.
- The RI Context MAY also contain RI certificate validation data.
- If the RI Context has expired, the Device SHALL NOT execute any other protocol than the 1-pass binary device data registration protocol with the associated RI (context), and upon detection of RI Context expiry the Device SHOULD initiate the offline notification of detailed device data protocol using the RI_ID stored in the RI Context. Depending on the implementation a dialogue will be shown to the user and the offline NDD protocol will be executed.
  - Accessing an ESG for purchase is still allowed, as this will require a registration first.
  - The device SHALL be rendered inoperable for any purchase protocol or playback of future content. The device MAY use stored BCROs to play old content for which the device obtained ROs, but SHALL NOT use these BCROs for new content received after the re-registration request until the device is re-registered with the RI.

Requirements:

- The Device SHALL have at most one RI Context with each RI. An existing RI Context SHALL be replaced with a newly established RI Context after successful re-registration with the same RI.
- The device SHALL support at least 6 RI context for broadcast mode of operation.
- For standard addressing the keyset SHALL include a valid set of :
  - UGK, BGK1..n and/or UDK keys
  - RIAK key. A single RIAK key is bound to a single Subscriber Group (e.g. 256 or 512 members).
  - Unique device filter (UDF).
- If domain addressing via an OMA DRM 2.0 domain is required the keyset SHALL (additionally to the standard addressing above) include a valid set of :
  - LDK key.
  - Shortform Local Domain Filter (SLDF). A.k.a. "shortform_domain_id". Refer to ~~A.13.1~~A.7.1.

And in case of mixed-mode operation devices the keyset SHALL contain:

- A Longform Local Domain Filter (LLDF, a.k.a. "longform_domain_id()") that matches the SLDF. Refer to Error! Reference source not found. A.7.3.

6.1.3.2.2 Protection of the (device registration) keyset

The device_registration_response() message is split in two parts: device specific (time bound) data and global (not time bound) data.

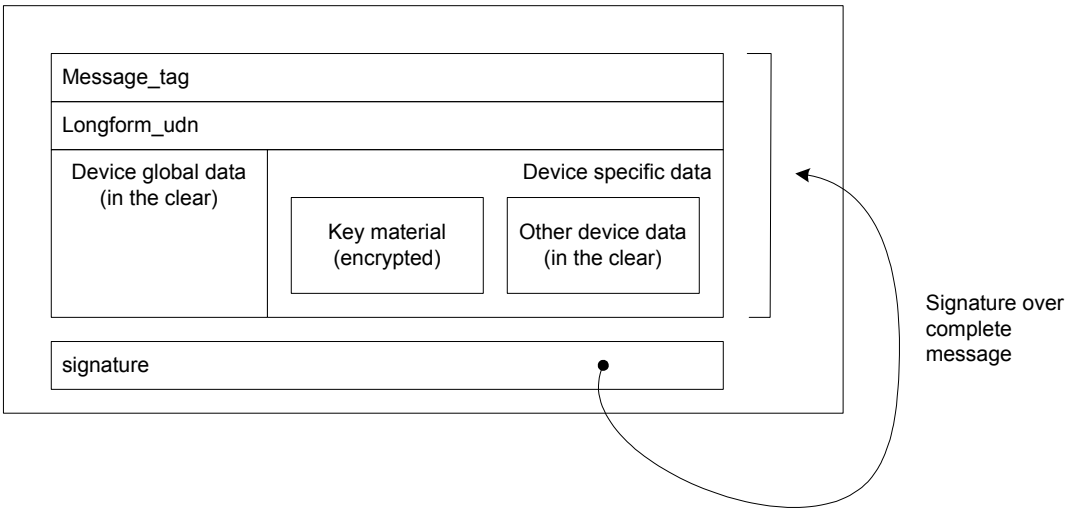


Figure 887: device_registration_response() message

The device global data SHALL be in the clear. The device specific data contains the keyset for the device. This key material SHALL be encrypted, whereas the rest of the device specific data SHALL be in the clear. The key material SHALL be protected by encryption. The RI SHALL use the device's public key to encrypt all key material in the device specific data part of the message.

The RI SHALL use his private key to sign the complete message data. Upon reception the device SHALL verify the RI signature, by using the issuer's public key from the RI certificate. The device SHALL make sure that this message is correct by using a valid and correct RI certificate.

The complete message SHALL be authenticated by a signature from the RI.

Creation of the encrypted message SHALL adhere to the following rules:

1. Generate a (128 or 192 or 256) bit AES key to be used as session key (SK) for the device_registration_response() message.
2. Concatenate the keyset (UGK, BGK1..n, UDK, RIAK, UDF and/or LDK, SLDF plus optional LLDF if applicable) under rules of [FIPS_197] and the Tag Length Format described in section A.13A.7.
3. Encrypt the keyset using [AES_WRAP] using the generated SK as (AES-WRAP style) KEK. This will produce the *keyset_block*.
4. Calculate the part of the keyblock that would fit into the RSA block (depending on the size of RSA used, be that 1024, 2048 or 4096), including the SK and under implementation rules of the PCKS#1. If the keyset_block fits into one RSA block continue at step 5. Else continue at step 4.
5. If the SK plus keyset_block including PCKS#1 header, aligning, etc did not fit into one RSA block, then keep the remainder part as surplus_block().

6. Encrypt SK plus the (part of the)keyset_block that fits into the RSA block with the public key of the target device using RSA (1024 or 2048 or 4096) under implementation guidelines of [PKCS#1]. This will produce the *sessionkey_block()*.

7. Concatenate the (non encrypted) parameters that were not used in the key_block and create the message "header" from this. Refer to ~~6.3.1~~6.1.3.2.3 for details. (for reason of completeness: of course the sessionkey_block(), the (optional) surplus_block() and the signature_block are not part of the message header)

8. Concatenate the message "header" and the sessionkey_block() . If the SK plus keyset_block including PCKS#1 header, aligning, etc did not fit into one RSA block, then also concatenate surplus_block() part. Hash the result under implementation guidelines of [PKCS#1]. Please refer to section ~~A.11~~A.5. This will produce the signature_input_data.

9. Sign the signature_input_data with RSA (1024 or 2048 or 4096) using the private key of the RI. The signature will apply to the implementation guidelines of PCKS#1, as outlined in ~~A.11~~A.5. This will produce the *signature_block*.

10. The device_registration_response() message comprises of the message "header" plus sessionkey_block(), optionally the surplus_block() and the signature_block.

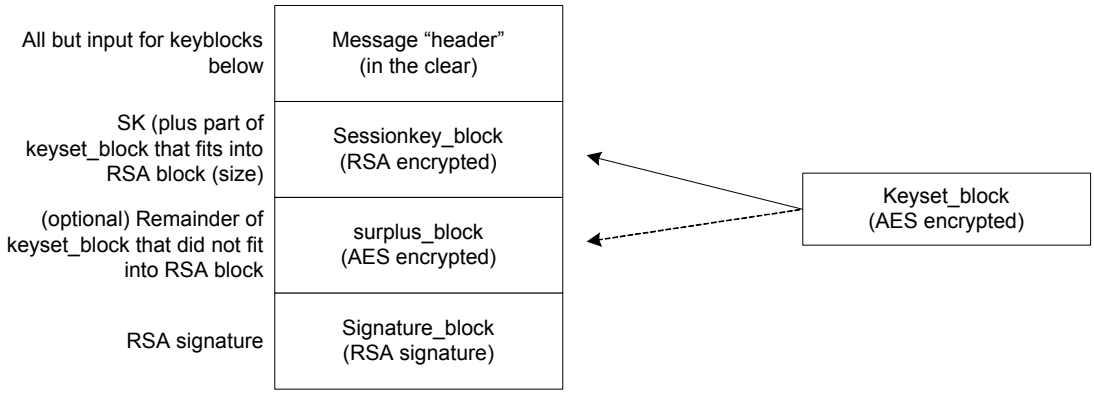


Figure 9~~9~~~~8~~: structure of device_registration_response() message.

Concluding: The number of RSA blocks used should be kept to a minimum. The AES surplus_block() is present if and when the keyset does not completely fit into the sessionkey_block() given the RSA block size used. If present the AES surplus_block() contains those keys that did not fit into one RSA block (i.e. the sessionkey_block()). The complete keyset needed for operation after registration is included in the encrypted keyset_block, which is concatenated from the first part in the sessionkey_block() and optionally the surplus_block(). Refer to appendix ~~A.7~~ for calculations on the surplus_block_size.

Decryption of the encrypted message SHALL adhere to the following rules:

1. Locate the message via message_tag

2. Verify if the message is intended for this device by comparing the long_form_udn with the UDN stored in the device.

3. Verify the signature_block of the message by using the public key from the RI.

4. Locate the sessionkey_block() and decrypt the block with the private key of the local device. Locate the session key (SK) from the header and (eventual) padding (according to PCKS#1). Then locate the keyset_block part from the header and (eventual) padding (according to PCKS#1).

5. (Optionally) If there is a surplus_block() concatenate this part to the keyset_block. This will complete the keyset_block.

6. Use the SK to decrypt the keyset_block.

7. Allocate the individual keyset_items from the keyset_block according to [AES_WRAP] and the Tag Length Format described in section ~~A.13~~A.7.

Note: The SK SHALL be stored into protected storage. The AES encrypted keyset_block MAY be stored as is into unprotected storage and decrypted by the device upon use. If the keyset_block is not stored but the decrypted keys from that block are stored instead, the device SHALL store all key data safely. The keys SHALL NOT leak outside the device.

6.1.3.2.3 Device registration response m~~M~~essage syntax

Table 8~~8~~7: Device registration response message syntax

fields	length	type
device_registration_response(){		
/* signature protected part starts here */		
/* message header starts here */		
message_tag	8	bslbf
protocol_version	4	bslbf
reserved_for_future_use	4	bslbf
longform_udn()	80	bslbf
status	8	bslbf
flags {		
ri_certificate_counter	3	bslbf
ocsp_response_counter	3	bslbf
local_time_offset_flag	1	bslbf
time_stamp_flag	1	bslbf
subscriber_group_key_flag	4	bslbf
short_udn_flag	1	bslbf
signature_type_flag	2	bslbf
surplus_block_flag	1	bslbf
keyset_block_length	16	uimsbf
}		
certificate_version	8	bslbf
for(cnt1=0; cnt1 < ri_certificate_counter ;cnt1++){		
c_length	16	uimsbf
ri_certificate()	8*c_length	bslbf
}		
for(cnt2=0; cnt2 < ocsp_response_counter ;cnt2++){		
r_length	16	uimsbf
ocsp_response()	8*r_length	bslbf
}		
drm_time	40	mjdutc
if (local_time_offset_flag == 0x1) {		
local_time_offset	16	bslbf
}		
if (time_stamp_flag == 0x1) {		
registration_timestamp_start	40	mjdutc
registration_timestamp_end	40	mjdutc
}		
if (short_udn_flag == 0x1) {		
short_udn	32	bslbf
}		
/* message header ends here */		
if (signature_type_flag == 0x0){		
sessionkey_block()	1024	bslbf
} else if (signature_type_flag == 0x1)		
sessionkey_block()	2048	bslbf
} else if (signature_type_flag == 0x2)		
sessionkey_block()	4096	bslbf
}		
if (surplus_block_flag == 0x1){		
surplus_block()	(*1)	bslbf
padding_bits	(*2)	bslbf
}		
/* signature protected part ends here */		
if (signature_type_flag == 0x0){		
signature_block	1024	bslbf

} else if (signature type flag == 0x1)		
signature_block	2048	bslbf
} else if (signature type flag == 0x2)		
signature_block	4096	bslbf
}		
}		

key:

(*1) for details please refer to section A.7

(*2) (surplus_block() length) mod 8

6.1.4 On-line Registration

A broadcast enabled device may register using the ROAP protocol, either directly in case it is a connected device, or via a connected device that acts as a proxy.

Extensions to the ROAP are required to allow transfer of all subscriber group key material and the authentication key for broadcast rights objects.

6.1.4.1 Registration Request

Rights issuers can derive from the device capabilities in the device certificate the modes of operation supported by the registering device. From this information it should be possible to determine whether to include the extensions (defined in the next section) in the registration response or not. To avoid possible confusion, an extension is defined for the **<roap:RegistrationRequest>** to allows a rights issuer to determine directly whether or not to include the broadcast extensions in **<roap:RegistrationResponse>**.

Extensions: The following extensions are defined for the ROAP-RegistrationRequest message in addition to the extensions already defined.

Broadcast Registration Request: This extension allows a device to indicate to a broadcast enabled Rights Issuer to use the broadcast extensions in the registration response.

The following schema fragment defines the *Broadcast Registration* extension to the ROAP schema:

```
<complexType name="roap:BroadcastRegistrationRequest">
  <complexContent>
    <extension base="roap:Extension">
    </extension>
  </complexContent>
</complexType>
```

When included in a **<roap:RegistrationRequest>**, this extension MUST be marked as critical.

6.1.4.2 Registration Response

A Rights Issuer that receives a **<roap:RegistrationRequest>** including the **<roap:BroadcastRegistrationRequest>** extension and that does not support the broadcast extensions MUST abort the registration procedure and respond accordingly. A Rights Issuer that does support broadcast extensions MUST respond with a **<roap:RegistrationRequest>** including the following defined **<roap:BroadcastRegistration>** extension.

Extensions: The following extensions are defined for the ROAP-RegistrationResponse message in addition to the extensions already defined.

Broadcast Registration: This extension allows an RI to securely transfer broadcast group key material and addressing information as well as the authentication key to use to verify authenticity of broadcast rights objects.

The following schema fragment defines the *Broadcast Registration* extension to the ROAP schema:

```
<complexType name=”roap:SubscriberGroupKey”>
  <complexContent>
    <extension base=”ds:KeyInfo”/>
    <attribute name=”node” type=”hexBinary”/>
  </complexContent>
</complexType>

<simpleType name=”roap:ShortUniqueDeviceNumber”>
  <restriction base=”string”>
    <pattern value=”\d{8}”
  </restriction>
</simpleType>

<complexType name="roap:SubscriberGroupRegistration">
  <complexContent>
     <sequence>
       <element name=”subscriberGroupAddress” type=”roap:SubscriberGroupIdentifier”/>
       <element name=”uniqueGroupKey” type=”ds:KeyInfo”/>
       <element name=”uniqueDeviceKey” type=”ds:KeyInfo” minOccurs=”0”/>
       <element name=”subscriberGroupKey” type=”roap:SubscriberGroupKey” minOccurs=”0”
maxOccurs=”unbounded”/>
       <element name=”shortUniqueDeviceNumber” type=”roap:ShortUniqueDeviceNumber”/>
    </sequence>
  </complexContent>
</complexType>

<complexType name="roap:BroadcastRegistration">
  <complexContent>
    <extension base="roap:Extension">
      <sequence>
        <element name=”subscriberGroupregistration” type=”roap:SubscriberGroupRegistration”
minOccurs=”0”/>
       <element name=”rightsIssuerAuthenticationKey” type=”ds:KeyInfo” minOccurs=”0”/>
       <element name="encKey" type="xenc:EncryptedKeyType"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

Subscriber Group Registration

The optional **<subscriberGroupRegistration>** element holds all information regarding the subscriber group feature: subscriber group address, device position and key material.

The **<subscriberGroupAddress>** element MUST contain the subscriber group base address and the device position. It SHALL NOT contain an access mask.

The **<uniqueGroupKey**> element holds an **<xenc:EncryptedKey>** element. This MUST hold a **<ds:KeyInfo>** element, an empty **<xenc:EncryptionMethod>** element and an **<xenc:CipherData>** element. The **<ds:KeyInfo>** element MUST contains a **<ds:RetrievalMethod>** element of which the **URI** attribute references the key used to encrypt the subscriber group's unique group key (UGK). **The <xenc:EncryptedKey>** element MUST also hold an empty **<xenc:EncryptionMethod>** element of which the **Algorithm** attribute identified the algorithm used to protect the UGK. This algorithm MUST be AES-128 Key Wrap, and the value of the **Algorithm** attribute MUST be "http://www.w3.org/2001/04/xmlenc#kw-aes128". The **<xenc:CipherData>** element contains the **<xenc:CipherValue>** element that holds the base64 encoded value of the encrypted UGK.

The optional **<uniqueDeviceKey**> element holds an **<xenc:EncryptedKey>** element. This MUST hold a **<ds:KeyInfo>** element, an empty **<xenc:EncryptionMethod>** element and an **<xenc:CipherData>** element. The **<ds:KeyInfo>** element MUST contains a **<ds:RetrievalMethod>** element of which the **URI** attribute references the key used to encrypt the subscriber group's unique device key (UDK). **The <xenc:EncryptedKey>** element MUST also hold an empty **<xenc:EncryptionMethod>** element of which the **Algorithm** attribute identified the algorithm used to protect the UDK. This algorithm MUST be AES-128 Key Wrap, and the value of the **Algorithm** attribute MUST be "http://www.w3.org/2001/04/xmlenc#kw-aes128". The **<xenc:CipherData>** element contains the **<xenc:CipherValue>** element that holds the base64 encoded value of the encrypted UDK.

The optional **<subscriberGroupKey**> elements each hold one key associated with the binary tree of key nodes from the subscriber group. Each **<subscriberGroupKey>** is of type **<roap:DerivationKey>** which extends the **<ds:KeyInfo>** type with a single **node** attribute. The value of the **node** attribute is the hexBinary encoded node number of the node associated with the derivation key contained by the **<subscriberGroupKey>** element. Each **<subscriberGroupKey>** element MUST hold a **<ds:KeyInfo>** element, an empty **<xenc:EncryptionMethod>** element and an **<xenc:CipherData>** element. The **<ds:KeyInfo>** element MUST contains a **<ds:RetrievalMethod>** element of which the **URI** attribute references the key used to encrypt the subscriber group's node key of node i (NK_i). **The <xenc:EncryptedKey>** element MUST also hold an empty **<xenc:EncryptionMethod>** element of which the **Algorithm** attribute identified the algorithm used to protect the NK_i. This algorithm MUST be AES-128 Key Wrap, and the value of the **Algorithm** attribute MUST be "http://www.w3.org/2001/04/xmlenc#kw-aes128". The **<xenc:CipherData>** element contains the **<xenc:CipherValue>** element that holds the base64 encoded value of the encrypted NK_i.

The device MUST check the consistency relations between the node keys and its subscriber position as defined by the broadcast extension.

The **<shortDeviceUniqueNumber>** MUST be included in the RI Context, and MAY be used at a later moment to receive binary push (re)registration messages over the broadcast interface.

Authentication Key

The **<rightsIssuerAuthenticationKey>** holds an **<xenc:EncryptedKey>** element. This MUST hold a **<ds:KeyInfo>** element, an empty **<xenc:EncryptionMethod>** element and an **<xenc:CipherData>** element. The **<ds:KeyInfo>** element MUST contains a **<ds:RetrievalMethod>** element of which the **URI** attribute references the key used to encrypt the rights issuer's authentication key (RIAK). **The <xenc:EncryptedKey>** element MUST also hold an empty **<xenc:EncryptionMethod>** element of which the **Algorithm** attribute identified the algorithm used to protect the RIAK. This algorithm MUST be AES-128 Key Wrap, and the value of the **Algorithm** attribute MUST be "http://www.w3.org/2001/04/xmlenc#kw-aes128". The **<xenc:CipherData>** element contains the **<xenc:CipherValue>** element that holds the base64 encoded value of the encrypted RIAK.

The **<encKey>** element is of type **xenc:EncryptedKeyType** from [XMLEnc]. It consists of a wrapped broadcast registration encryption key, K_{BRK}. The **Id** attribute of this element SHALL be present and SHALL have the same value as the value of the **URI** attribute of the **<ds:RetrievalMethod>** element in any **<ds:KeyInfo>** elements inside the subscriber group registration extension. The **<ds:KeyInfo>** child element of the **<encKey>** element SHALL identify the wrapping key. The child of the **<ds:KeyInfo>** element SHALL be of type **roap:X509SPKIHash**, identifying a particular DRM Agent's public key through the (SHA-1) hash of the DER-encoded subjectPublicKeyInfo value in its certificate.

6.2 Inform Registered Device Protocol

6.2.1 Theory of Operation

Note: This protocol is also known as the "1-pass IRD protocol", short for Inform Registered Device protocol.

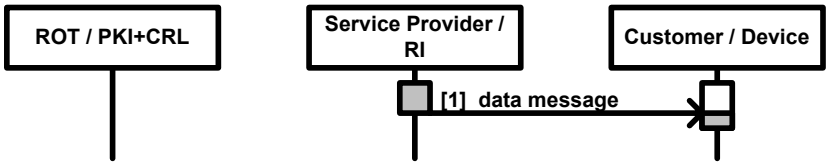


Figure 10109: 1-pass IRD protocol – RI initiated message to device.

Explanation of the protocol:

- The 1-pass IRD protocol is designed to meet the messaging push case. Its successful execution assumes the device to have an existing RI context with the sending RI.
- Several messages are defined for the IRD protocol.

Table 998: Messages of the 1-pass IRD protocol

message name	for msg syntax refer to section	remark
force to re-register		
update RI certificate	6.2.1 (or to be defined by main editor)6.2.2	
update DRM Time	6.2.3	in BCRO carousel
update contact number	6.2.4	in BCRO carousel
update domain	6.4.3.2	
force to join domain	6.4.3.3	
force to leave domain	6.4.3.4	
token delivery	6.3.4.1	

Note: The processing of each message will be discussed in following sections.

6.2.2 Update RI certificate

The RI can use this message to update the RI certificate in one or more devices.

- The RI SHALL enter a valid RI certificate in the message.
- The RI MAY enter a rooted RI certificate chain in the message. The root certificate is to be excluded.
- The RI SHALL use the mechanisms described in section 6.2.111.6 to address the message to a device.
- The device SHALL filter on the message_tag to identify the message. Then it SHALL filter for the UDN and compare it to the local UDN (see **Error! Reference source not found.**) of the device. If those match the device SHALL start validating the signature and check the RI certificate (chain.). If both are valid the device detects this message is really addressed to it, and the device SHALL start to perform the intended action.
- If the message is correct, the device SHALL save the new RI certificate in the message after the signature of the message has been verified correctly. The old RI certificate SHALL be made obsolete.

6.2.2.1 Update RI certificate - update_ri_certificate_msg() message

Using the 1-pass IRD protocol (refer to 6.2) the RI sends a update_ri_certificate_msg() message, forcing the device to update his RI certificate chain.

This update_ri_certificate_msg() trigger is almost identical to the re_register_msg() message described in section 6.2.5 , with the following adaptations:

- being that the message_tag is different. Refer to A.8 for the value of the message_tag.
- Status/Error code is Succes or NotSupported. Refer to A.3 for the value of the error codes.

6.2.3 Update DRM time

The RI can use this message to update the DRM time.

- The RI SHALL enter a valid DRM time in the message.
- The RI MAY put a time offset in the message. The timeoffset SHALL be valid.
- The RI SHALL use the mechanisms described in section ~~6.2.1~~11.6 ~~-~~to address the message to a device.
- The device SHALL filter on the message_tag to identify the message. Then the device SHALL start validating the signature and check the RI certificate (chain.). If both are valid the device detects this message is really addressed to it, and the device SHALL start to perform the intended action.
- If the message successfully validated and the RI certificate is valid, the device SHALL save the new DRM time into the device.

6.2.3.1 Updating the DRM time - update_drmtime_msg() message

6.2.3.1.1 Update drmtime m~~M~~essage description

Using the 1-pass IRD protocol (refer to ~~6.2.1~~6.2) the RI sends a update_drmtime trigger message with the drmtime to the device as specified below:

Table 3: Update drmtime message description

update_drmtime_msg( )		
Parameter name	(M)andatory / (O)ptional[6]	remark
message_tag	M	
protocol_version	M	
status	M	
signature_type_flag	M	
local_time_offset_flag	M	
drm_time	M	
local_time_offset	O	
signature_block	M	

message_tag - This parameter identifies the type of the message. Refer to A.8 for the value of the message_tag.

[6] key: (O)ptional means that the user of the message MAY include the parameter in the message, but the device MUST support the interpretation of the parameter. (M)andatory means that the user of the message SHALL include the parameter in the message.

protocol_version - This parameter indicates the protocol_version of this message. The device SHALL ignore messages that have a protocol_version number it doesn't support. The value of the protocol_version of this message is set to 0x0 (i.e. the original format).

If set to 0x0 the format specified in the this version of the specification is used. If set to anything else than 0x0, then the format is beyond the scope of this version of the specification.

status - The status parameter SHALL indicate one of the values explained in the following table. The device SHALL ignore messages with other error values.

Table 4: Status values

status value	meaning
Success	The message contains valid DRM time RI.
NotSupported	The RI does not support the sending of DRM time request. The device will use other means to update DRM time.
DeviceTimeError	The RI concluded that the DeviceTime might be false and forces the device to update it's time. As an extra result the device will determine the eventual clock drift and notify this to the RI per ARC (offline notification of short device data; refer to ~~Error! Reference source not found.~~ 6.1.2).Please note: this capability should be used with great care.)

Note: Refer to A.3 for the value of the error codes.

local_time_offset_flag - Binary flag to signal presence of the parameter it describes:

local_time_offset_flag	Value ($_h$)	remark
data absent	0x0	
data present	0x1	

signature_type_flag - A flag to signal type of signature algorithm used:

signature_type_flag	Value ($_h$)	remark
RSA 1024	0x0	
RSA 2048	0x1	CMLA requirement (2004-2007)
RSA 4096	0x2	
reserved for future use	0x3	not used in this version of the specification

drm_time - This parameter defines the time in Universal Time Coordinated (UTC). This 40-bit field contains the current time and date in UTC and MJD. Refer to Appendix A.4 for calculation of the UTC and Modified Julian Date (MJD). This field is coded as 16 bits giving the 16 LSBs of MJD followed by 24 bits coded as 6 digits in 4-bit BCD.

EXAMPLE: 93/10/13 12:45:00 is coded as "0xC079124500".

local_time_offset - This parameter indicates the local time offset from the (UTC) drm_time as explained in Annex A.4.

signature_block - The signature SHALL enable a single source authenticity check on the message. The algorithm used for the signature is RSA-1024 or RSA-2048 or RSA-4096. The signature will apply to the rules of PCKS#1, as outlined in ~~A.11~~A.5.

6.2.3.1.2 Update drmtime m~~M~~essage syntax

Table 5: update drmtime message syntax

fields	length	type
`update_drmtime_msg(){`		
`  /* signature protected part starts here */`		
`  message_tag`	8	bslbf
`  protocol version`	4	bslbf
`  reserved_for_future_use`	4	bslbf

status	8	bslbf
flags {		
local time offset flag	1	bslbf
signature type flag	1	bslbf
reserved for future use	6	bslbf
}		
drm time	40	mjdutc
if (local_time_offset_flag == 0x1) {		
local time offset	16	bslbf
}		
/* signature protected part ends here */		
if (signature_type_flag == 0x0){		
signature block	1024	bslbf
} else if (signature_type_flag == 0x1)		
signature block	2048	bslbf
} else if (signature_type_flag == 0x2)		
signature block	4096	bslbf
}		
}		

6.2.4 Update contact number~~DRM time~~

~~{EN: The CR originally called this section 'update DRM time', but that is likely to be a cut/paste error. The section would be more appropriately labeled 'Update contact number'.}~~

The RI can use this message to update the contact number that the device should contact during the offline notification processes (both for use with the NDD or NSD protocols):

- The message SHALL contain (a) valid telephone number(s) to contact.
- The RI SHALL use the mechanisms described in section ~~6.2.1~~11.6 ~~-~~to address the message to a device.
- The device SHALL filter on the message_tag to identify the message. Then the device SHALL start validating the signature and check the RI certificate (chain.). If both are valid the device detects this message is really addressed to it, and the device SHALL start to perform the intended action.
- If the message is correct, the device SHALL store the new contact number(s) and delete the old one(s).

6.2.4.1 Update the contact number – update_contact_number_msg() message

6.2.4.1.1 Update contactnumber m~~M~~essage description

Using the 1-pass IRD protocol (refer to 6.2.1) the RI sends a update_contact_number_msg() message with a (set of) contact number(s) to the device as specified below:

Table 6: update contactnumber message description

update_contact_number_msg( )		
Parameter name	(M)andatory / (O)ptional[7]	Remark
message_tag	M	
protocol_version	M	
status	M	
signature_type_flag	M	
ri_certificate_counter	M	
c_length	M	
ri_certificate	M	
ocsp_response_counter	M	
r_length	M	
ocsp_response	M	
contact_counter	M	
contact	O	
signature_block	M	

message_tag - This parameter identifies the type of the message. Refer to section A.8 for the value of the message_tag.

protocol_version - This parameter indicates the protocol_version of this message. The device SHALL ignore messages that have a protocol_version number it doesn't support. The value of the protocol_version of this message is set to 0x0 (i.e. the original format).

If set to 0x0 the format specified in the this version of the specification is used. If set to anything else than 0x0, then the format is beyond the scope of this version of the specification.

status - The status parameter SHALL indicate one of the values explained in the following table. The device SHALL ignore messages with other error values.

Table 7: Status values

status value	meaning
Success	The message contains valid contact numbers from the RI.
NotSupported	The RI does not support the sending of contact numbers. The device will use other means to use contact numbers (e.g. via ESG).

Note: Refer to A.3 for the value of the error codes.

signature_type_flag - A flag to signal type of signature algorithm used:

signature_type_flag	Value ($_h$)	remark
RSA 1024	0x0	
RSA 2048	0x1	CMLA requirement (2004-2007)
RSA 4096	0x2	
reserved for future use	0x3	not used in this version of the specification

certificate_version - is a numerical representation of the version of the RI certificate. Using the certificate_version parameter the device can decide if it is needed to update the RI certificate (if it was stored before).

[7] key: (O)ptional means that the user of the message MAY include the parameter in the message, but the device MUST support the interpretation of the parameter. (M)andatory means that the user of the message SHALL include the parameter in the message.

Table 8: description of certificate_version parameter

Parameter Fieldname	Field Value ($_h$)	supports
major_version_number	0x0,..,0xA	MSB_4(certificate_version)
minor_version_number	0x0,..,0xA	LSB_4(certificate_version)

The parameter is divided 2 fields of 4 bits, whereas the first 4 bits (MSB left) express the Major number and the last four bits (LSB right) express the Minor version. The major and minor number encode in bslbf format. 16 Major and 16 Minor versions are supported. For example: Major.Minor version <1.2> is expressed as $0001\ 0010_b$.

ri_certificate_counter - This parameter indicates the depth of the RI certificate chain.

number of certificate in chain	Value ($_h$)	remark
0	0x0	will signal absence of ri_certificate e.g. on error status to save bandwidth.
1	0x1	
2	0x2	
3	0x3	
4	0x4	
5	0x5	
6	0x6	
7	0x7	

Note: The certificate chain can have a depth of up to 7 RI certificates.

c_length **-** This parameter indicates the length in bytes of the ri_certificate.

ri_certificate() **-** This parameter SHALL be present. When present, the value of a *ri_certificate* parameter SHALL be a certificate chain including the RI's certificate. The chain SHALL NOT include the root certificate. The RI certificate SHALL come first in the list. Each following certificate SHALL directly certify the one preceding it.

If an RI certificate is received that is not in the stored certificate verification data for this RI, and if the Device can determine (in the case of Broadcast Devices that support DRM Time) that the expiry time of the received RI certificate is later than the RI Context for this RI, and the certificate status of the RI certificate as indicated in the OCSP response is good (see [OCSP-MP]), then the Device SHALL verify the complete chain.

ocsp_response_counter - This parameter indicates the depth of the OCSP response chain.

number of responses in chain	Value ($_h$)	remark
0	0x0	will signal absence of ocsp_response e.g. on error status to save bandwidth.
1	0x1	
2	0x2	
3	0x3	
4	0x4	
5	0x5	
6	0x6	
7	0x7	

Note: The certificate chain can have a depth of up to 7 OCSP responses.

r_length - This parameter indicates the length in bytes of the ocsp_response.

ocsp_response() **-** This parameter, when present, SHALL be a complete set of valid OCSP responses for the RI's certificate chain. The Device SHALL NOT fail due to the presence of more than one OCSP response element. A Device SHALL check

that an OCSP response is present in the received message. If no OCSP response is present in the device_registration_response() message, then the Device SHALL abort the registration protocol.

contacts_counter - This parameter indicates the number of contacts carried in the message.

contact – This object specifies the contact. Please refer to 6.2.4.1.3.

signature_block - The signature SHALL enable a single source authenticity check on the message. The algorithm used for the signature is RSA-1024 or RSA-2048 or RSA-4096. The signature will apply to the implementation guidelines of PCKS#1, as outlined in A.5.

6.2.4.1.2 Update contact number message syntax

Table 9: Update contact number message syntax

fields	length	type
update_contact_number_msg() {		
/* signature protected part starts here */		
message_tag	8	bslbf
protocol_version	4	bslbf
reserved_for_future_use	4	bslbf
status	8	bslbf
flags {		
contacts_counter	4	bslbf
signature_type_flag	1	bslbf
ri_certificate_counter	3	bslbf
ocsp_response_counter	3	bslbf
reserved_for_future_use	5	bslbf
}		
certificate_version	8	bslbf
for(cnt1=0; cnt1 < ri_certificate_counter ;cnt1++){		
c_length	16	uimsbf
ri_certificate()	8*c_length	bslbf
}		
for(cnt2=0; cnt2 < ocsp_response_counter ;cnt2++){		
r_length	16	uimsbf
ocsp_response()	8*r_length	bslbf
}		
for(cnt3=0; cnt3 < contacts_counter ;cnt3++){		
contact()		
}		
/* signature_protected_part_ends_here */		
if (signature_type_flag == 0x0){		
signature_block	1024	bslbf
} else if (signature_type_flag == 0x1)		
signature_block	2048	bslbf
} else if (signature_type_flag == 0x2)		
signature_block	4096	bslbf
}		
}		

6.2.4.1.3 Format of the contact object

Table 10: content object format

Field	length	type
contact(){		
contact_type	4	uimsbf
reserved_for_future_use	4	bslbf
contact_length	8	uimsbf
contactdata	8*contact_length	bslbf
}		

contact_type: This field specifies the type of action as listed in Table 11Table 11Table 11.

Table 11: contact type

contact_type	description	comments	max length (chars)
0x00	local_ri_phone_number	The number the user of the device needs to contact to start service provision.	20
0x01	int_ri_phone_number	The number the user of the device needs to contact to start service provision when he would call from abroad.	20
0x02	ri_sms_number	The SMS number the user of the device needs to contact to start service provision.	20
0x03	ri_url	The URL address the user of the device needs to contact to start service provision.	30
0x04	local_home_coc_phone_number	The number the user of the device needs to contact to start service provision.	20
0x05	int_home_coc_phone_number	The number the user of the device needs to contact to start service provision when he would call from abroad.	20
0x06	home_coc_sms_number	The SMS number the user of the device needs to contact to start service provision.	20
0x07	home_coc_url	The URL address the user of the device needs to contact start service provision.	30
0x08	local_reporting_phone_number	The number the user of the device needs to contact to report token consumption.	20
0x09	int_reporting_phone_number	The number the user of the device needs to contact to report token consumption when he would call from abroad.	20
0x0A	reporting_sms_number	The SMS number the user of the device needs to contact to report token consumption.	20
0x0B	reporting_url	The URL address the user of the device needs to contact to report token consumption.	30
0x0C-0x0F	reserved for future use		

contact_length - This parameter indicates the length in bytes of the contact field. Maximum length of the contacts is specified in Table 11Table 11Table 11.

UTF-8 [RFC 3629] character encoding for ASCII characters is 'efficient' with 1 byte per character. On the other hand, there are characters that are encoded using 6 bytes (Asian languages).

For example: a URL is limited to 30 characters. The 30 URL UTF-8 characters are translated into bytes as follows:

E.g.: "Western" languages - character is 1 byte - Longest URL encoded as bytes is 1*30 characters = 30 bytes.

E.g.: Asian languages - character is 6 bytes - Longest URL encoded as bytes is 6*30 characters = 180 bytes.

contactdata - The value in this field specifies any of the contact_type possibilities the user of the device needs to contact (via other means) to start service provision.

contact types	contactdata encoding rules
phone numbers	The phone number is encoded as alphabetic, supporting telephone numbers like: “0800-123456789” but also for example: “0800-philips”. The string that forms the phone number is encoded using UTF-8.
SMS numbers	The SMS number is encoded as hexadecimal, supporting telephone numbers like: “0800-123456789” but also for example: “philips+subscribe”. The string that forms the SMS number is encoded using UTF-8.
URLs	The URL is encoded as hexadecimal, according to [RFC 1738], supporting URLs like: www.philips.com/start. The string that forms the URL is encoded using UTF-8.

6.2.5 Force re-registration

In this case the RI is sending a message to the device to get it into registration mode.

- The RI SHALL use the mechanisms described in section ~~6.2.1~~11.6 ~~-~~to address the message to a device.
- The device SHALL filter on the message_tag to identify the message. Then it SHALL filter for the UDN and compare it to the local UDN ~~(**Error! Reference source not found.)**~~ of the device. If those match the device SHALL start validating the signature and check the RI certificate (chain.). If both (UDN and signature) are valid the device detects this message is really addressed to it, and the device SHALL start to perform the intended action.
- If the message is correct, the reception of this message SHALL start the (re-) registration process. The device will be rendered inoperable, but only in relation with the associated RI (context) as described below:
    - Accessing an ESG for purchase is still allowed, as this will require a registration first.
    - The device SHALL be rendered inoperable for any purchase protocol or playback of future content. The device MAY use stored BCROs to play old content for which the device obtained ROs, but SHALL NOT use these BCROs for new content received after the re-registration request until the device is re-registered with the RI.
- Depending on the implementation a dialogue will be shown to the user and the offline NDD protocol will be executed, using the RI_ID stored in the RI Context. Depending on the implementation a dialogue MAY be shown to the user and the offline NDD protocol SHALL be executed.

6.2.5.1 (Force to) Re-register - re_register_msg() message

6.2.5.1.1 Re-register message description

Using the 1-pass IRD protocol (refer to 6.2.1) the RI sends a register_msg message, indirectly triggering a (re)registration . The message is specified as follows:

Table 12: Re-register message description

re_register_msg( )		
Parameter name	(M)andatory / (O)ptional[8]	Remark
message_tag	M	
protocol_version	M	
longform_udn	M	
status	M	
signature_type_flag	M	
certificate_version	M	
ri_certificate_counter	M	
c_length	M	
ri_certificate	M	
ocsp_response_counter	M	
r_length	M	
ocsp_response	M	
signature_block	M	

message_tag - This parameter identifies the type of the message. Refer to A.8 for the value of the message_tag.

protocol_version - This parameter indicates the protocol_version of this message. The device SHALL ignore messages that have a protocol_version number it doesn't support. The value of the protocol_version of this message is set to 0x0 (i.e. the original format).

If set to 0x0 the format specified in the this version of the specification is used. If set to anything else than 0x0, then the format is beyond the scope of this version of the specification.

longform_udn() - The long form of the UDN. Refer to section ~~**Error! Reference source not found.**~~ 6.1.1.2.1 for details.

status - The status parameter SHALL indicate one of the values explained in the following table. The device SHALL ignore messages with other error values.

Table 13: Status values

status value	meaning
Success	The message contains valid reregistration message and cancels any preceding forced channel usage restrictions.
ForceInteractiveChannel	If the device is a mixed mode device the (re)registration will be possible via OOB and/or the interactive channel. By using this status code the RI can indicate to the device that the device SHALL direct subsequent (re)registrations to the RI over the device's interactive channel only. When the device receives this status code it will also exclusively use the interaction channel for all other messages. When the interactive channel of the device is not able to connect to the RI the mixed mode device MAY revert back to the OOB re-registration dialogue. Please note that a mixed mode device will remain to have full broadcast reception capabilities after receiving this status code.
ForceOobChannel	If the device is a mixed mode device the (re)registration will be possible via OOB and/or the interactive channel. By using this status code the RI can indicate to the device that the device SHALL direct subsequent (re)registrations to the RI over the device's OOB channel. When the device receives this status code it will also exclusively use the OOB channel for all other messages. Please note that a mixed mode device will remain to have full interactive channel capabilities after receiving this status code, but will not use the interactive channel.

Note: Refer to A.3 for the value of the error codes.

[8] key: (O)ptional means that the user of the message MAY include the parameter in the message, but the device MUST support the interpretation of the parameter. (M)andatory means that the user of the message SHALL include the parameter in the message.

signature_type_flag - A flag to signal type of signature algorithm used:

signature_type_flag	Value ($_h$)	remark
RSA 1024	0x0	
RSA 2048	0x1	CMLA requirement (2004-2007)
RSA 4096	0x2	
reserved for future use	0x3	not used in this version of the specification

certificate_version **-** is a numerical representation of the version of the RI certificate. Using the certificate_version parameter the device can decide if it is needed to update the RI certificate (if it was stored before).

Table 10~~10~~~~9~~: description of certificate_version parameter

Parameter Fieldname	Field Value ($_h$)	supports
major_version_number	0x0,..,0xA	MSB_4(certificate_version)
minor_version_number	0x0,..,0xA	LSB_4(certificate_version)

The parameter is divided 2 fields of 4 bits, whereas the first 4 bits (MSB left) express the Major number and the last four bits (LSB right) express the Minor version. The major and minor number encode in bslbf format. 16 Major and 16 Minor versions are supported. For example: Major.Minor version <1.2> is expressed as 0001 0010_b.

ri_certificate_counter - This parameter indicates the depth of the RI certificate chain.

number of certificate in chain	Value ($_h$)	remark
0	0x0	will signal absence of ri_certificate e.g. on error status to save bandwidth.
1	0x1	
2	0x2	
3	0x3	
4	0x4	
5	0x5	
6	0x6	
7	0x7	

Note: The certificate chain can have a depth of up to 7 RI certificates.

c_length **-** This parameter indicates the length in bytes of the ri_certificate.

ri_certificate() **-** This parameter SHALL be present. When present, the value of a *ri_certificate* parameter SHALL be a certificate chain including the RI's certificate. The chain SHALL NOT include the root certificate. The RI certificate SHALL come first in the list. Each following certificate SHALL directly certify the one preceding it.

If an RI certificate is received that is not in the stored certificate verification data for this RI, and if the Device can determine (in the case of Broadcast Devices that support DRM Time) that the expiry time of the received RI certificate is later than the RI Context for this RI, and the certificate status of the RI certificate as indicated in the OCSP response is good (see [OCSP-MP]), then the Device SHALL verify the complete chain.

ocsp_response_counter - This parameter indicates the depth of the OCSP response chain.

number of responses in chain	Value ($_h$)	remark
0	0x0	will signal absence of ocsp_response e.g. on error status to save bandwidth.
1	0x1	
2	0x2	
3	0x3	
4	0x4	
5	0x5	
6	0x6	
7	0x7	

Note: The certificate chain can have a depth of up to 7 OCSP responses.

r_length - This parameter indicates the length in bytes of the ocsp_response.

ocsp_response() - This parameter, when present, SHALL be a complete set of valid OCSP responses for the RI's certificate chain. The Device SHALL NOT fail due to the presence of more than one OCSP response element. A Device SHALL check that an OCSP response is present in the received message. If no OCSP response is present in the device_registration_response() message, then the Device SHALL abort the registration protocol.

signature_block - The signature SHALL enable a single source authenticity check on the message. The algorithm used for the signature is RSA-1024 or RSA-2048 or RSA-4096. The signature will apply to the implementation guidelines of PCKS#1, as outlined in A.5.

6.2.5.1.2 Re-register message syntax

Table 14: re-register message syntax

fields	length	Type
re_register_msg() {		
/* signature protected part starts here */		
message_tag	8	bslbf
protocol_version	4	bslbf
longform_udn()	80	bslbf
flags {		
signature_type_flag	1	bslbf
ri_certificate_counter	3	bslbf
ocsp_response_counter	3	bslbf
reserved_for_future_use	5	bslbf
}		
certificate_version	8	bslbf
for(cnt1=0; cnt1 < ri_certificate_counter ;cnt1++){		
c_length	16	uimsbf
ri_certificate()	8*c_length	bslbf
}		
for(cnt2=0; cnt2 < ocsp_response_counter ;cnt2++){		
r_length	16	uimsbf
ocsp_response()	8*r_length	bslbf
}		
/* signature protected part ends here */		
if (signature_type_flag == 0x0){		
signature_block	1024	bslbf
} else if (signature_type_flag == 0x1)		
signature_block	2048	bslbf
} else if (signature_type_flag == 0x2)		
signature_block	4096	bslbf
}		
}		

6.3 Token handling

6.3.1 Protocol overview

The theory of operation (refer to section ~~Error! Reference source not found.~~A.11) results in the specification of several protocols:

- offline protocols (from device to RI)

protocol	section	purpose
token request protocol	6.3.2	request to purchase tokens
token reporting protocol	6.3.3	protocol to report the consumption of tokens

- 1-pass protocols (from RI to device)

protocol	section	purpose
1-pass binary Push Device Registration protocol	6.1.3	transmit registration data to device
1-pass binary Inform Registered Device protocol	6.2	inform device via messages.

The protocols interrelate in following way (roundtrip):

kicking off action…	…results in
token request protocol (request to purchase tokens)	token delivery response message (transmit tokens to device)
token reporting protocol (report the consumption of tokens)	token delivery response message (transmit tokens to device)

6.3.2 Token request protocol

When the user of a device wants to obtain tokens, he uses the NSD protocol with the token_request action type. (refer to section ~~Error! Reference source not found.~~ 6.1.2.1.4).

6.3.3 Token reporting protocol

When the user of a device is instructed by his device to report token consumption, he uses the NSD protocol with the token_consumption_message action type in order to send a token consumption report. (refer to section ~~Error! Reference source not found.~~ 6.1.2.1.4).

6.3.4 Binary messages

6.3.4.1 delivering tokens – token_delivery_response() message

6.3.4.1.1 Token delivery~~t~~ response message description

Using the 1-pass ~~IDR~~IRD protocol (refer to section ~~Error! Reference source not found.~~ 6.2.1) the RI sends a token_delivery_response() message, informing the device of the delivery of new tokens. The message is specified below:

Table 15: token delivery response message description

token_delivery_response()		
Parameter name	(M)andatory / (O)ptional[9]	remark
message_tag	M	not encrypted
protocol_version	M	not encrypted
message_length	M	not encrypted
group_size_flag	M	not encrypted
address_mode	M	not encrypted
One	M	not encrypted
Address	M	not encrypted
bit_access_mask	not used in this version of the specification	not encrypted
position_in_group	M	not encrypted
domain_id_extension	not used in this version of the specification	not encrypted
domain_generation	not used in this version of the specification	not encrypted
rights_issuer_id	M	not encrypted
Status	M	not encrypted
device_nonce	M	not encrypted
response_flag	M	not encrypted
token_reporting_flag	M	not encrypted
earliest_reporting_time_flag	M	not encrypted
latest_reporting_time_flag	M	not encrypted
token_quantity_flag	M	not encrypted
token_delivery_response_id	M	not encrypted
latest_consumption_time	O	not encrypted
earliest_reporting_time	O	not encrypted
latest_reporting_time_flag	O	not encrypted
encrypted_token_quantity	O	encrypted
encrypted_report_authentication_key	O	encrypted
MAC	M	not encrypted

message_tag - This parameter identifies the type of the message. Refer to A.8 for the value of the message_tag.

protocol_version - This parameter indicates the protocol_version of this message. The device SHALL ignore messages that have a protocol_version number it doesn't support. The value of the protocol_version of this message is set to 0x0 (i.e. the original format).

If set to 0x0 the format specified in the this version of the specification is used. If set to anything else than 0x0, then the format is beyond the scope of this version of the specification.

message_length - 12-bit field indicating the length in bytes of the message starting immediately after this field.

group_size_flag - 1-bit field indicating the group size used. 0 – a maximum group size 256 is used, 1 – a maximum group size of 512 is used

address_mode - 3-bit field indicating the addressing mode used by this message. Table 16Table 16Table 16 lists all possible address modes. Not that not all modes are used in this version of the specification for the token delivery response message.

[9] key: (O)ptional means that the user of the message MAY include the parameter in the message, but the device MUST support the interpretation of the parameter. (M)andatory means that the user of the message SHALL include the parameter in the message.

Table 16: address_mode for token delivery response message

address_mode	description
0x0	addressing whole of unique group This addressing mode is not used in this version of the specification for the token delivery response message
0x1	addressing of broadcast group using a bit_mask size of 256 or 512 bit depending on group_size_flag (subset of unique group) This addressing mode is not used in this version of the specification for the token delivery response message
0x2-0x3	addressing of unique device
0x4	addressing of OMA DRM 2.0 domain. Address field concatenated with the domain_id_extension will be the domain id in this case This addressing mode is not used in this version of the specification for the token delivery response message
0x5-0x7	reserved for future use

one **-** 1-bit flag which SHALL have the value 0x1 in this version of the specification. This field MAY have value 0x0 in future versions of the specification

address - 4-byte group address. Each rights issuer has its own address space. If the group_size is 512 then the group address is made of the first 31 bit of the address field. If this message is addressed to a unique device in a group then the LSB of the address field is the MSB of the group position.

bit_access_mask - This field is not used in this version of the specification and MAY be used in future versions. It is indicated here, so devices according to this version of the specification know its size. All bits of the field SHALL be set to 0, when the field is not used.

position_in_group - If this message addresses a unique device then this field specifies the position of the unique device in the given broadcast group. If group_size_flag is 0 than the position in the group is directly given by the position_in_group field. If group_size_flag is 1 then 9 bit are used to identify the position in the group. If group_size_flag is 1 then the LSB (bit 0) from the address field is used as the 9th bit, the MSB. The real position in the group is then given by:

```
int real_position_in_group;
if(address_mode&0x6==0x2){
            if(group_size_flag == 0){
            //maximum size of 256 devices in group.
            real_position_in_group = position_in_group;
            }else{
                        //maximum size of 512 devices in group;
                        real_position_in_group =
((address&0x1)<<8)||position_in_group;
            }
}
```

domain_id_extension - This field is not used in this version of the specification and MAY be used in future versions. It is indicated here, so devices according to this version of the specification know its size. All bits of the field SHALL be set to 0, when the field is not used.

domain_generation **-** This field is not used in this version of the specification and MAY be used in future versions. It is indicated here, so devices according to this version of the specification know its size. All bits of the field SHALL be set to 0, when the field is not used.

rights_issuer_id() **-** The ID of the rights issuer. This is the 160-bit SHA-1 hash of the public key of the RI. See X509PKISHash in [~~OMA-DRM-DRM~~DRM-v2].

status - The status parameter SHALL indicate one of the values explained in the following table. The device SHALL ignore messages with other error values.

Table 17: message error codes

status value	meaning
Success	The message contains valid token delivery data from the RI.
NotSupported	The RI does not support the sending of tokens from the RI. In this message, the RI SHALL set the value of token_quantity to zero or SHALL set the token_quantity_flag to 0x0.
TokenConsumptionMessageError	The RI did receive a token consumption message, but that it was erroneous and that the device should redo the last token consumption message. In this token delivery response message, the RI SHALL set the value of token_quantity to zero or SHALL set the token_quantity_flag to 0x0.. The RI SHALL use a token_reporting_flag of value 0x1. The RI SHALL use the device_nonce of the last token consumption message that the RI successfully processed or set the response_flag to 0x0 in case no token consumption messages have been successfully processed. The device SHALL generate a token consumption message, reporting on the token consumption from the time of the generation of the token consumption message with the same device_nonce as the device_nonce in this token delivery response message, or from first start-up in case the response_flag was set to 0x0.
NoTokenConsumptionMessage	The RI did not receive a token consumption message yet, but was expecting one, because the present date/time is later than the last latest_token_consumption_time sent to the device in a token delivery response message. In this token delivery response message, the RI SHALL set the value of token_quantity to zero or SHALL set the token_quantity_flag to 0x0. The RI SHALL use a token_reporting_flag of value 0x1. The RI SHALL use the device_nonce of the last token consumption message that the RI successfully processed or set the response_flag to 0x0 in case no token consumption messages have been successfully processed. The device SHALL generate a token consumption message, reporting on the token consumption from the time of the generation of the token consumption message with the same device_nonce as the device_nonce in this token delivery response message, or from first start-up in case the response_flag was set to 0x0.

Note: Refer to A.3 for the value of the error codes.

device_nonce – If the response_flag equals 0x1, the device_nonce is the nonce present in the request (using the offline NSD protocol) to which this token delivery response message is a response. If the response_flag field equals 0x0, this token delivery response message does not refer to any request from the device to the RI and the device_nonce MAY be ignored. The nonce is encoded in BCD.

response_flag – If this flag equals 0x1, this token delivery response message is a response to a message from the device to the RI and the device_nonce in this token delivery response message is taken from that message. If this flag equals 0x0, this token delivery response message does not refer to any message from the device to the RI and the device_nonce can be any value.

token_reporting_flag – If this flag equals 0x1, the device has to report to the RI the consumption of the tokens received with this token delivery response message. If this flag equals 0x0, the device can consume all tokens delivered with this token delivery response message, as well as any other previously delivered tokens which are still not consumed, without ever having to report their consumption.

earliest_reporting_time_flag - Binary flag to signal presence of the parameter it describes:

earliest_reporting_time field	Value ($_h$) of earliest_reporting_time_flag	remark
data absent	0x0	
data present	0x1	

latest_reporting_time_flag - Binary flag to signal presence of the parameter it describes:

latest_reporting_time field	Value ($_h$) of latest_reporting_time_flag	remark
data absent	0x0	
data present	0x1	

token_quantity_flag - Binary flag to signal presence of the parameter it describes:

token_quantity field	Value ($_h$) of token_quantity_flag	remark
data absent	0x0	
data present	0x1	

token_delivery_response_id – This is the ID of the token delivery response message. The RI SHALL use the same token_delivery_response_id when retransmitting a token delivery response message. The RI SHALL generate a random number using a sufficiently good pseudo random number generator for every new token delivery response message. Devices SHALL discard token delivery response messages with a token_delivery_response_id identical to the one in an already received token delivery response message.

latest_token_consumption_time – After the date/time indicated in the latest_token_consumption_time field, the device SHALL NOT use any tokens, which have been received after the last token delivery response message that had the token_reporting_flag set to 0x0, for the consumption of protected content controlled by the RI. The device SHALL use the date/time in the latest_token_consumption_time field, if present, of the last received token delivery response message, regardless of the value of the field status.

earliest_reporting_time - If the device reports the consumption of tokens before the date/time indicated in the earliest_reporting_time field, the RI NEED NOT change the latest_token_consumption_time in its subsequent token delivery response message.

latest_reporting_time – The purpose of this field is to make uninterrupted token consumption possible. If the device reports the token consumption before the date/time indicated in the latest_reporting_time field, the RI SHALL send the next token delivery response message before the latest_token_consumption_time, unless the RI wishes to interrupt or disable the token consumption.

encrypted_token_quantity – A 4-byte field, containing the encrypted token_quantity. Token_quantity is a signed, two's complement 32-bit number. If the value of token_quantity is positive, it specifies the number of tokens the device receives from the RI. If the value of token_quantity is negative, it specifies how many tokens the RI removes from the device. If the field encrypted_token_quantity is not present, no tokens are received from the RI and no tokens are removed from the device by this token delivery response message. The token_quantity is encrypted using AES-128-CBC, with fixed IV 0 and with 0 padding in the last block if needed. The encryption key used depends on the addressing mode of the token delivery response message, see the next table.

Table 18: Mapping of address_mode to keys for the token delivery response message

address_mode	Key(s) used to decrypt field
0x0 (unique group)	This addressing mode is not used in this version of the specification for the token delivery response message
0x1 (broadcast group)	This addressing mode is not used in this version of the specification for the token delivery response message
0x2-0x3 (unique device)	Token Delivery Key
0x4 (domain)	This addressing mode is not used in this version of the specification for the token delivery response message
0x5-0x7	reserved for future use

encrypted_report_authentication_key - This field contains the encrypted Report Authentication Key. The Report Authentication Key a 128 bit key to authenticate the reported number of tokens with in the next token consumption message. The encrypted_report_authentication_key field is only present if the token_reporting_flag has the value 0x1. The RI SHALL generate a random number using a sufficiently good pseudo random number generator for the value of every newly required Report Authentication Key. The Report Authentication Key is encrypted using AES-128-CBC, with fixed IV 0 and with 0 padding in the last block if needed. The encryption key used depends on the addressing mode of the token delivery response message, see the next table.

Table 19: Mapping of address_mode to keys for the token delivery response message

address_mode	Key(s) used to decrypt field
0x0 (unique group)	This addressing mode is not used in this version of the specification for the token delivery response message
0x1 (broadcast group)	This addressing mode is not used in this version of the specification for the token delivery response message
0x2-0x3 (unique device)	Token Delivery Key
0x4 (domain)	This addressing mode is not used in this version of the specification for the token delivery response message
0x5-0x7	reserved for future use

MAC: This is the authentication code calculated over all bytes before this field in this message using HMAC-SHA-1-96 (see [RFC 2104]). The MAC is used for integrity check of this message. The key used to create the MAC is the token delivery response message authentication key TDRMAK as defined in A.9.5. Devices SHALL NOT use token delivery response messages with an invalid MAC.

Note Message result:

- More information on device actions after the reception of this message can be found in section **Error! Reference source not found.** {EN: I don't know to which section to refer.}section A.11.2

6.3.4.1.2 Token delivery response message syntax

Table 20: token delivery response message syntax

fields	length	type
`token_delivery_response(){`		
`/* MAC protected part starts here */`		
`message_tag`	8	bslbf
`protocol version`	4	bslbf
`message_length`	12	uimsbf
`group size flag`	1	bslbf
`reserved for future use`	3	bslbf
`address mode`	3	uimsbf

one	1	bslbf
address	32	uimsbf
if(address_mode == 0x1 && group_size_flag == 0){[10]		
bit_access_mask	256	bslbf
}else if(address_mode == 0x1 && group_size_flag == 1){		
bit_access_mask10109	512	bslbf
}else if (address_mode&0x6 == 0x2){		
position_in_group	8	uimsbf
}else if (address_mode == 0x4){10109		
domain_id_extension	6	bslbf
domain_generation	10	uimsbf
}		
rights_issuer_id()	160	bslbf
status	8	bslbf
device_nonce	4	bslbf
flags {		
response_flag	1	bslbf
token_reporting_flag	1	bslbf
earliest_reporting_time_flag	1	bslbf
latest_reporting_time_flag	1	bslbf
token_quantity_flag	1	bslbf
reserved_for_future_use	7	bslbf
}		
token_delivery_response_id	96	bslbf
if (token_reporting_flag == 0x1) {		
latest_token_consumption_time	40	mjdutc
if (earliest_reporting_time_flag == 0x1) {		
earliest_reporting_time	40	mjdutc
}		
if (latest_reporting_time_flag == 0x1) {		
latest_reporting_time	40	mjdutc
}		
}		
/* encrypted_part_starts_here		
if(token_quantity_flag == 1){		
encrypted_token_quantity	32	bslbf
}		
encrypted_report_authentication_key	128	bslbf
/* encrypted_part_ends_here		
/* MAC_protected_part_ends_here */		
MAC	96	bslbf
}		

Note that all reserved for future use fields SHALL have the value 0 for token delivery response messages created according to this version of the specification.

6.4 Domain Management

6.4.1 Domain joining and leaving

Interactive devices will adhere to [DRM-v2].

- Interactive devices will therefore use OMA DRM 2.0 domain ID.

Broadcast devices will adhere to the mechanisms as described in this section.

[10] Although this addressing mode is indicated here to facilitate future upgrades, this addressing mode is NOT used in this version of the specification. for the token delivery response message

- Broadcast devices will use “shortform_domain_id” a.k.a. SLDF.

Mixed-mode SHALL have the "interoperability" requirement to support both domain ID formats of interactive and broadcast devices:

- Mixed-mode device will receive:
  - “longform_domain_id()”, a.k.a. LLDF, which is a translation of OMA DRM 2.0 domain ID.
  - “shortform_domain_id” a.k.a. SLDF.
- mixed-mode devices registered for both interactive and broadcast operations MAY pass either domain ID format to other mixed-mode devices in the domain.
- interactive only devices SHALL pass longform_domain_id() format to other devices in the domain. The mixed-mode device will understand this, while broadcast does not understand.
- broadcast devices SHALL pass shortform_domain_id format to other devices in the domain. The mixed-mode device will understand this, while interactive does not understand.

6.4.2 protocol overview

The theory of operation results in the specification of several protocols:

- offline protocols (from device to RI)

protocol	section	purpose
offline Domain Join Request protocol	6.4.2.1	request to join a domain
offline Domain Leave Request protocol	6.4.2.2	request to leave a domain

- 1-pass protocols (from RI to device)

protocol	section	purpose
1-pass binary Push Device Registration protocol	6.1.3	transmit registration data to device
1-pass binary Inform Registered Device protocol	6.2	inform device via messages.

The protocols interrelate in following way (roundtrip):

kicking off action…	…results in
offline domain join request. (request to join a domain).	domain_registration_response() message. (transmit registration data to device).
offline domain leave request (request to leave a domain)	domain_update_response() message. (inform device via messages)
join_domain_msg() (inform device via messages)	offline domain join request, which on it’s turn may result in domain_registration_response() as listed above.
leave_domain_msg() (inform device via messages)	offline domain leave request, which on it’s turn may result in domain_update_response() as listed above.

6.4.2.1 Offline Domain Join Request

When the user of a device might want to join a particular domain, he uses the NSD protocol with the destined action code range. (refer to ~~Error! Reference source not found.~~ 6.1.2.1).

6.4.2.2 Offline Domain Leave Request

When the user of a device might want to leave a particular domain, he uses the NSD protocol with the destined action code range. (refer to ~~Error! Reference source not found.~~ 6.1.2.1).

6.4.3 Binary messages

6.4.3.1 Domain data - domain_registration_response() message

6.4.3.1.1 Domain registration response mMessage description

Using the 1-pass PDR protocol (see 6.1.36.1.3.1) the RI sends a domain_registration_response() message, informing the device of a new domain keyset. The message is specified below:

Table 21: message description

domain_registration_response()		
Parameter name	(M)andatory / (O)ptional[11]	remark
message_tag	M	global, not encrypted
protocol_version	M	global, not encrypted
longform_udn	M	global, not encrypted
device_nonce	M	device specific, not encrypted
status	M	device specific, not encrypted
time_stamp_flag	M	device specific, not encrypted
certificate_version	M	global, not encrypted
ri_certificate_counter	M	global, not encrypted
c_length	M	global, not encrypted
ri_certificate	M	global, not encrypted
ocsp_response_counter	M	global, not encrypted
r_length	M	global, not encrypted
ocsp_response	M	global, not encrypted
domain_timestamp_start	O	device specific, not encrypted
domain_timestamp_end	O	device specific, not encrypted
signature_type_flag	M	global, not encrypted
keyset_block_length	M	device specific, not encrypted
local_domain_key	M	device specific, encrypted
longform_domain_id()	O	device specific, encrypted
shortform_domain_id	M	device specific, encrypted
signature_block	M	device specific, not encrypted

message_tag - This parameter identifies the type of the message. Refer to A.8 for the value of the message_tag.

protocol_version - This parameter indicates the protocol_version of this message. The device SHALL ignore messages that have a protocol_version number it doesn't support. The value of the protocol_version of this message is set to 0x0 (i.e. the original format).

If set to 0x0 the format specified in the this version of the specification is used. If set to anything else than 0x0, then the format is beyond the scope of this version of the specification.

longform_udn() - The long form of the UDN. Refer to section Error! Reference source not found. 6.1.1.2 for details.

status - The status parameter SHALL indicate one of the values explained in the following table. The device SHALL ignore messages with other error values.

[11] key: (O)ptional means that the user of the message MAY include the parameter in the message, but the device MUST support the interpretation of the parameter. (M)andatory means that the user of the message SHALL include the parameter in the message.

Table 22: Status values

status value	meaning
Success	The message contains valid domain registration data from the RI.
NotSupported	The RI does not support the sending of domain registration data from the RI. The RI SHALL NOT include any valid keyset in the message. The device will use other means to obtain valid domain registration data from the RI.
InvalidDomain	The RI could not recognize the domain identifier that was used in the join domain request or decided that the domain identifier is invalid. The RI SHALL NOT include any valid keyset in the message.
DomainFull	The RI indicates that no more devices are allowed to join the domain. The RI SHALL NOT include any valid keyset in the message.

Note: Refer to A.3 for the value of the error codes.

device_nonce - The device_nonce is the nonce which was present in the request (using the offline NSD protocol) to which this message is a response. This nonce is a encoded in BCD.

time_stamp_flag - Binary flag to signal presence of the parameter it describes:

time_stamp_flag	Value ($_h$)	remark
data absent	0x0	
data present	0x1	

certificate_version **-** is a numerical representation of the version of the RI certificate. Using the certificate_version parameter the device can decide if it is needed to update the RI certificate (if it was stored before).

Table 23: description of certificate_version parameter

Parameter Fieldname	Field Value ($_h$)	supports
major_version_number	0x0,..,0xA	MSB_4(certificate_version)
minor_version_number	0x0,..,0xA	LSB_4(certificate_version)

The parameter is divided 2 fields of 4 bits, whereas the first 4 bits (MSB left) express the Major number and the last four bits (LSB right) express the Minor version. The major and minor number encode in bslbf format. 16 Major and 16 Minor versions are supported. For example: Major.Minor version <1.2> is expressed as $0001\ 0010_b$.

ri_certificate_counter - This parameter indicates the depth of the RI certificate chain.

number of certificate in chain	Value ($_h$)	remark
0	0x0	will signal absence of ri_certificate e.g. on error status to save bandwidth.
1	0x1	
2	0x2	
3	0x3	
4	0x4	
5	0x5	
6	0x6	
7	0x7	

Note: The certificate chain can have a depth of up to 7 RI certificates.

c_length **-** This parameter indicates the length in bytes of the ri_certificate.

ri_certificate() - This parameter SHALL be present. When present, the value of a *ri_certificate* parameter SHALL be a certificate chain including the RI's certificate. The chain SHALL NOT include the root certificate. The RI certificate SHALL come first in the list. Each following certificate SHALL directly certify the one preceding it.

The Device MAY store RI certificate verification data indicating that an RI certificate chain has been verified. The purpose of this is to avoid repeated verification of the same certificate chain. The RI certificate verification data stored in this way SHALL uniquely identify the RI certificate and SHALL be integrity protected. The Device SHOULD check if the RI certificate chain received in this parameter corresponds to the stored certificate verification data for this RI. If so, the Device NEED NOT verify the RI certificate chain again, otherwise the Device SHALL verify the RI certificate chain.

If an RI certificate is received that is not in the stored certificate verification data for this RI, and if the Device can determine (in the case of Broadcast Devices that support DRM Time) that the expiry time of the received RI certificate is later than the RI Context for this RI, and the certificate status of the RI certificate as indicated in the OCSP response is good (see [OCSP-MP]), then the Device SHALL verify the complete chain and SHOULD replace the stored RI certificate verification data with the received RI certificate data and set the RI context expiry time to that of the received RI certificate expiry time.

However, if the Device does store RI certificate verification data in this way it SHALL store the expiry period of the RI's certificate (as indicated by the notAfter field within the certificate) and SHALL compare the Device's current DRM Time with the stored RI certificate expiry time whenever verifying the signature on signed messages from the RI. If the Device's current DRM Time is after the stored RI certificate expiry time then the Device SHALL abandon processing the RI message and SHALL initiate the registration protocol.

ocsp_response_counter - This parameter indicates the depth of the OCSP response chain.

number of responses in chain	Value ($_h$)	remark
0	0x0	will signal absence of ocsp_response e.g. on error status to save bandwidth.
1	0x1	
2	0x2	
3	0x3	
4	0x4	
5	0x5	
6	0x6	
7	0x7	

Note: The certificate chain can have a depth of up to 7 OCSP responses.

r_length - This parameter indicates the length in bytes of the ocsp_response.

ocsp_response() - This parameter, when present, SHALL be a complete set of valid OCSP responses for the RI's certificate chain. The Device SHALL NOT fail due to the presence of more than one OCSP response element. A Device SHALL check that an OCSP response is present in the received message. If no OCSP response is present in the domain_registration_response() message, then the Device SHALL abort the registration protocol.

domain_timestamp_start - Indicates from what time onwards the registration data for the domain is valid. This is an extra mechanism above the expiration date of the RI certificate. (Note: please note that this parameter can also be used against replay attacks.)

domain_timestamp_end - Indicates from what time onwards the registration data for the domain expires. This is an extra mechanism above the expiration date of the RI certificate. (Note: please note that this parameter can also be used against replay attacks.)

signature_type_flag - A flag to signal type of signature algorithm used:

signature_type_flag	Value ($_h$)	remark
RSA 1024	0x0	
RSA 2048	0x1	CMLA requirement (2004-2007)
RSA 4096	0x2	
reserved for future use	0x3	not used in this version of the specification

keyset_block_length - This parameter indicates the length in bits of the total keyset_block. That is the part in the sessionkey_block().

local_domain_key - An AES symmetric key to address a unique device. This key is also known as LDK. The key length SHALL be 128 bit.

Note: This key is wrapped into the keyset_block. (Refer to 6.4.3.1.2).

longform_domain_id() – This parameter is also known as the Longform Local Domain Filter (LLDF). Please refer to A.7.3 for the definition. The longform_domain_id() is used for mixed-mode operation. Note: This address is wrapped into the keyset_block. (Refer to 6.4.3.1.2).

shortform_domain_id – This parameter is also known as the Shortform Local Domain Filter (SLDF). Please refer to A.7.1. An addressing scheme used to filter for messages like BCROs. The shortform_domain_id is used for broadcast mode of operation.

Note: This address is wrapped into the keyset_block. (Refer to 6.4.3.1.2).

signature_block - The signature SHALL enable a single source authenticity check on the message. The algorithm used for the signature is RSA-1024 or RSA-2048 or RSA-4096. The signature will apply to the implementation guidelines of PCKS#1, as outlined in ~~A.7~~ A.5.

Note Message result:

The stored domain context SHALL at a minimum contain:

- Following keys:
  - LDK.
  - Shortform Local Domain Filter (SLDF). A.k.a. “shortform_domain_id”. Refer to A.7.1.
- For mixed-mode operation, devices’ domain context SHALL additionally contain:
  - Longform Local Domain Filter (LLDF). A.k.a. “longform_domain_id()”. Refer to A.7.3.
- A Device MAY have several Domain Contexts with an RI.
- If the domain context has expired, the Device SHALL NOT execute any other protocol than the 1-pass binary device data registration protocol with the associated RI (context), and upon detection of domain context expiry the Device SHOULD initiate the offline notification of short device data protocol using the correct ARC. Depending on the implementation a dialogue will be shown to the user and the offline NSD protocol will be executed.
  - Accessing an ESG for purchase is still allowed, as this will require a (domain) registration first.
  - The device SHALL be rendered inoperable for any purchase protocol or playback of future content. The device MAY use stored BCROs to play old content for which the device obtained ROs, but SHALL NOT use these BCROs for new content received after the re-registration request until the device is re-registered with the RI.

Requirements:

- If domain addressing via an OMA DRM 2.0 domain is required the keyset SHALL include a valid set of :
  - LDK key.
  - Shortform Local Domain Filter (SLDF). A.k.a. “shortform_domain_id”. Refer to A.7.1.

  And in case of mixed-mode operation devices the keyset SHALL contain:
  - A Longform Local Domain Filter (LLDF, a.k.a. “longform_domain_id()”) that matches the SLDF. Refer to A.7.3.

6.4.3.1.2 Protection of the (domain registration) keyset

The domain_registration_response() message is split in two parts: device specific (time bound) data and global (not time bound) data.

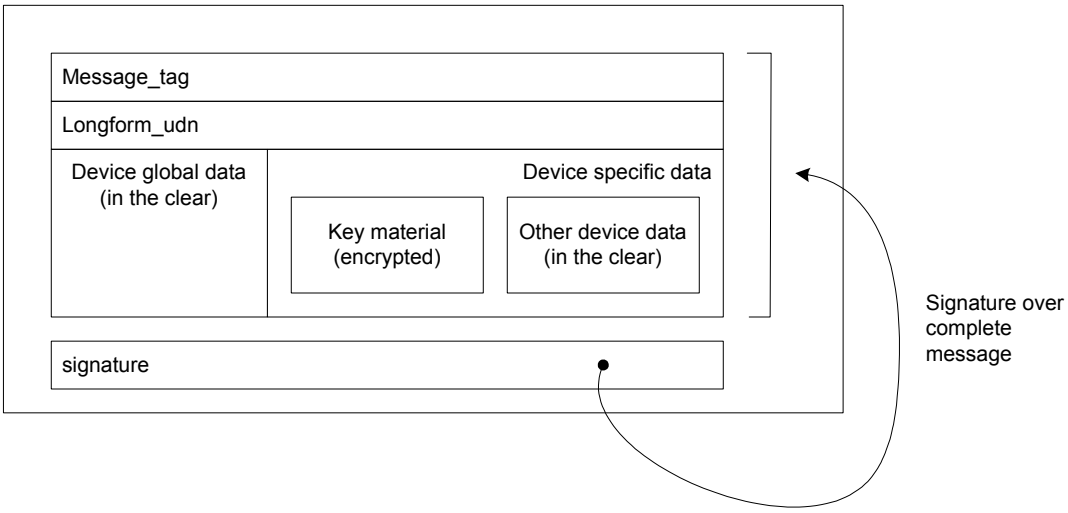


Figure 6: domain_registration_response() message

The device global data SHALL be in the clear. The device specific data contains the keyset for the device. This key material SHALL be encrypted, whereas the rest of the device specific data SHALL be in the clear. The key material SHALL be protected by encryption. The RI SHALL use the device’s public key to encrypt all key material in the device specific data part of the message.

The RI SHALL use his private key to sign the complete message data. Upon reception the device SHALL verify the RI signature, by using the issuer’s public key from the RI certificate. The device SHALL make sure that this message is correct by using a valid and correct RI certificate.

The complete message SHALL be authenticated by a signature from the RI.

Creation of the encrypted message SHALL adhere to the following rules:

1. Generate a (128 or 192 or 256) bit AES key to be used as session key (SK) for the domain_registration_response() message.
2. Concatenate the keyset (LDK, SLDF plus optional LLDF if applicable) under rules of [FIPS_197] and the Tag Length Format described in section A.7A.13. More than one context is allowed up to the RSA blocksize.
3. Encrypt the keyset using [AES_WRAP] using the generated SK as (AES-WRAP style) KEK. This will produce the *keyset_block*.

4. Calculate the part of the keyblock that would fit into the RSA block (depending on the size of RSA used, be that 1024, 2048 or 4096), including the SK and under implementation rules of the PCKS#1.

5. Encrypt SK plus the keyset_block with the public key of the target device using RSA (1024 or 2048 or 4096) under implementation guidelines of [PKCS#1]. This will produce the *sessionkey_block()*.

6. Concatenate the (non encrypted) parameters that were not used in the key_block and create the message "header" from this. Refer to 6.4.3.1.3 for details. (for reason of completeness: of course the sessionkey_block() and the signature_block are not part of the message header)

7. Concatenate the message "header" and the sessionkey_block() . Hash the result under implementation guidelines of [PKCS#1]. Please refer to section ~~A.11~~A.5. This will produce the signature_input_data.

8. Sign the signature_input_data with RSA (1024 or 2048 or 4096) using the private key of the RI. The signature will apply to the implementation guidelines of PCKS#1, as outlined in ~~A.11~~A.5. This will produce the *signature_block*.

9. The domain_registration_response() message comprises of the message "header" plus sessionkey_block() and the signature_block.

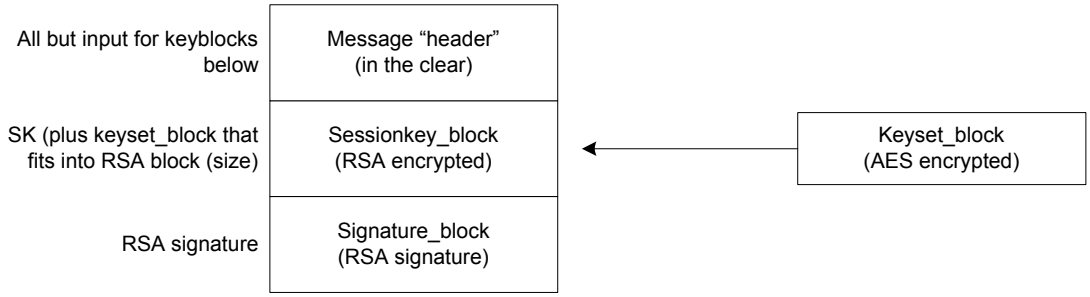


Figure 7: structure of domain_registration_response() message.

Decryption of the encrypted message SHALL adhere to the following rules:

1. Locate the message via message_tag

2. Verify if the message is intended for this device by comparing the long_form_udn with the UDN stored in the device.

3. Verify the signature_block of the message by using the public key from the RI.

4. Locate the sessionkey_block() and decrypt the block with the private key of the local device. Locate the session key (SK) from the header and (eventual) padding (according to PCKS#1). Then locate the keyset_block part from the header and (eventual) padding (according to PCKS#1).

5. Use the SK to decrypt the keyset_block.

6. Allocate the individual keyset_items from the keyset_block according to [AES_WRAP] and the Tag Length Format described in section ~~A.13~~A.7.

Note: The SK SHALL be stored into protected storage. The AES encrypted keyset_block MAY be stored as is into unprotected storage and decrypted by the device upon use. If the keyset_block is not stored but the decrypted keys from that block are stored instead, the device SHALL store all key data safely. The keys SHALL NOT leak outside the device.

6.4.3.1.3 Domain registration response message syntax

Table 24: domain registration response message syntax

fields	length	type
`domain_registration_response(){`		

/* signature protected part starts here */		
/* message header starts here /*		
message_tag	8	bslbf
protocol_version	4	bslbf
reserved_for_future_use	4	bslbf
unique_device_number	80	bslbf
reserved_for_future_use	4	bslbf
device_nonce	4	bslbf
status	8	bslbf
flags {		
ri_certificate_counter	3	bslbf
ocsp_response_counter	3	bslbf
signature_type_flag	2	bslbf
time_stamp_flag	1	bslbf
reserved_for_future_use	7	bslbf
keyset_block_length	16	uimsbf
}		
certificate_version	8	bslbf
for(cnt1=0; cnt1 < ri_certificate_counter ;cnt1++){		
c_length	16	uimsbf
ri_certificate()	8*c_length	bslbf
}		
for(cnt2=0; cnt2 < ocsp_response_counter ;cnt2++){		
r_length	16	uimsbf
ocsp_response()	8*r_length	bslbf
}		
if (time_stamp_flag == 0x1) {		
domain_timestamp_start	40	mjdutc
domain_timestamp_end	40	mjdutc
}		
/* message header ends here /*		
if (signature_type_flag == 0x0){		
sessionkey_block()	1024	bslbf
} else if (signature_type_flag == 0x1)		
sessionkey_block()	2048	bslbf
} else if (signature_type_flag == 0x2)		
sessionkey_block()	4096	bslbf
}		
/* signature protected part ends here */		
if (signature_type_flag == 0x0){		
signature_block	1024	bslbf
} else if (signature_type_flag == 0x1)		
signature_block	2048	bslbf
} else if (signature_type_flag == 0x2)		
signature_block	4096	bslbf
}		
}		

6.4.3.2 Updating a domain - domain_update_response() message

6.4.3.2.1 Domain update response mMessage description

Using the 1-pass IRD protocol (see 6.2), the RI sends a domain_update_response() message, informing the device that it left a particular domain. The message is specified below:

Table 25: domain update response message description

domain_update_response()		
Parameter name	(M)andatory / (O)ptional[12]	remark
message_tag	M	global, not encrypted
protocol_version	M	global, not encrypted
longform_udn	M	global, not encrypted
status	M	device specific, not encrypted
device_nonce	M	device specific, not encrypted
certificate_version	M	global, not encrypted
ri_certificate_counter	M	global, not encrypted
c_length	M	global, not encrypted
ri_certificate	M	global, not encrypted
ocsp_response_counter	M	global, not encrypted
r_length	M	global, not encrypted
ocsp_response	M	global, not encrypted
shortform_domain_id	M	device specific, not encrypted
signature_type_flag	M	global, not encrypted
signature_block	M	device specific, not encrypted

message_tag - This parameter identifies the type of the message. Refer to A.8 for the value of the message_tag.

protocol_version - This parameter indicates the protocol_version of this message. The device SHALL ignore messages that have a protocol_version number it doesn't support. The value of the protocol_version of this message is set to 0x0 (i.e. the original format).

If set to 0x0 the format specified in the this version of the specification is used. If set to anything else than 0x0, then the format is beyond the scope of this version of the specification.

longform_udn() - The long form of the UDN. Refer to section ~~Error! Reference source not found.~~ 6.1.1.2.1 for details.

status - The status parameter SHALL indicate one of the values explained in the following table. The device SHALL ignore messages with other error values.

Table 26: Status values

status value	meaning
Success	The message informs the device that the RI has removed this device from the domain it was registered in. The device SHALL remove the domain keyset that was associated to the particular domain.
NotSupported	The RI does not support the request to leave a domain. The device will use other means to notify the RI that it wants to leave a particular domain.
InvalidDomain	The RI is unable to support the request to leave a domain, because the domain is invalid

Note: Refer to A.3 for the value of the error codes.

device_nonce - The device_nonce is the nonce which was present in the request (using the offline NSD protocol) to which this message is a response. This nonce is a encoded in BCD.

certificate_version - is a numerical representation of the version of the RI certificate. Using the certificate_version parameter the customer device can decide if it is needed to update the RI certificate (if it was stored before).

[12] key: (O)ptional means that the user of the message MAY include the parameter in the message, but the device MUST support the interpretation of the parameter. (M)andatory means that the user of the message SHALL include the parameter in the message.

Table 27: description of certificate_version parameter

Parameter Fieldname	Field Value ($_h$)	supports
major_version_number	0x0,..,0xA	MSB_4(certificate_version)
minor_version_number	0x0,..,0xA	LSB_4(certificate_version)

The parameter is divided 2 fields of 4 bits, whereas the first 4 bits (MSB left) express the Major number and the last four bits (LSB right) express the Minor version. The major and minor number encode in bslbf format. 16 Major and 16 Minor versions are supported. For example: Major.Minor version <1.2> is expressed as $0001\ 0010_b$.

ri_certificate_counter - This parameter indicates the depth of the RI certificate chain.

number of certificate in chain	Value ($_h$)	remark
0	0x0	will signal absence of ri_certificate e.g. on error status to save bandwidth.
1	0x1	
2	0x2	
3	0x3	
4	0x4	
5	0x5	
6	0x6	
7	0x7	

Note: The certificate chain can have a depth of up to 7 RI certificates.

c_length **-** This parameter indicates the length in bytes of the ri_certificate.

ri_certificate() **-** This parameter SHALL be present. When present, the value of a *ri_certificate* parameter SHALL be a certificate chain including the RI's certificate. The chain SHALL NOT include the root certificate. The RI certificate SHALL come first in the list. Each following certificate SHALL directly certify the one preceding it.

The Device MAY store RI certificate verification data indicating that an RI certificate chain has been verified. The purpose of this is to avoid repeated verification of the same certificate chain. The RI certificate verification data stored in this way SHALL uniquely identify the RI certificate and SHALL be integrity protected. The Device SHOULD check if the RI certificate chain received in this parameter corresponds to the stored certificate verification data for this RI. If so, the Device need not verify the RI certificate chain again, otherwise the Device SHALL verify the RI certificate chain.

If an RI certificate is received that is not in the stored certificate verification data for this RI, and if the Device can determine (in the case of Broadcast Devices that support DRM Time) that the expiry time of the received RI certificate is later than the RI Context for this RI, and the certificate status of the RI certificate as indicated in the OCSP response is good (see [OCSP-MP]), then the Device SHALL verify the complete chain and SHOULD replace the stored RI certificate verification data with the received RI certificate data and set the RI context expiry time to that of the received RI certificate expiry time.

However, if the Device does store RI certificate verification data in this way it SHALL store the expiry period of the RI's certificate (as indicated by the notAfter field within the certificate) and SHALL compare the Device's current DRM Time with the stored RI certificate expiry time whenever verifying the signature on signed messages from the RI. If the Device's current DRM Time is after the stored RI certificate expiry time then the Device SHALL abandon processing the RI message and SHALL initiate the registration protocol.

ocsp_response_counter - This parameter indicates the depth of the OCSP response chain.

number of responses in chain	Value ($_h$)	remark
0	0x0	will signal absence of ocsp_response e.g. on error status to save bandwidth.
1	0x1	
2	0x2	
3	0x3	
4	0x4	
5	0x5	
6	0x6	
7	0x7	

Note: The certificate chain can have a depth of up to 7 OCSP responses.

r_length - This parameter indicates the length in bytes of the ocsp_response.

ocsp_response() - This parameter, when present, SHALL be a complete set of valid OCSP responses for the RI's certificate chain. The Device SHALL NOT fail due to the presence of more than one OCSP response element. A Device SHALL check that an OCSP response is present in the received message. If no OCSP response is present in the domain_registration_response() message, then the Device SHALL abort the registration protocol.

shortform_domain_ID – the shortform_domain_id is the SLDF.

signature_type_flag - A flag to signal type of signature algorithm used:

signature_type_flag	Value ($_h$)	remark
RSA 1024	0x0	
RSA 2048	0x1	CMLA requirement (2004-2007)
RSA 4096	0x2	
reserved for future use	0x3	not used in this version of the specification

signature_block - The signature SHALL enable a single source authenticity check on the message. The algorithm used for the signature is RSA-1024 or RSA-2048 or RSA-4096. The signature will apply to the implementation guidelines of PCKS#1, as outlined in ~~A.11~~A.5.

6.4.3.2.2 Domain update response message syntax

Table 28: domain update response message syntax

fields	length	type
domain_update_response(){		
/* signature protected part starts here */		
message_tag	8	bslbf
protocol_version	4	bslbf
reserved_for_future_use	4	bslbf
longform_udn()	80	bslbf
reserved_for_future_use	4	bslbf
device_nonce	4	bslbf
status	8	bslbf
flags {		
ri_certificate_counter	3	bslbf
ocsp_response_counter	3	bslbf
signature_type_flag	2	bslbf
}		
certificate_version	8	bslbf
for(cnt1=0; cnt1 < ri_certificate_counter ;cnt1++){		
c_length	16	uimsbf
ri_certificate()	8*c_length	bslbf
}		

for(cnt2=0; cnt2 < ocsp response counter ;cnt2++){		
r_length	16	uimsbf
ocsp response()	8*r length	bslbf
}		
shortform domain id	48	uimsbf
/* signature protected part ends here */		
if (signature type flag == 0x0){		
signature_block	1024	bslbf
} else if (signature type flag == 0x1)		
signature_block	2048	bslbf
} else if (signature type flag == 0x2)		
signature_block	4096	bslbf
}		
}		

6.4.3.3 (Force to) Join a domain - join_domain_msg() message

Using the 1-pass IRD protocol (see 6.2) the RI sends a join_domain_msg() message, forcing the device to join a particular domain.

This join_domain_msg() trigger is almost identical to the re_register_msg() message described in section 6.2.5, with the only adaptation being that the message_tag is different. Refer to A.8 for the value of the message_tag.

6.4.3.4 (Force to) Leave a domain - leave_domain_msg() message

Using the 1-pass IRD protocol (see 6.2), the RI sends a leave_domain_msg() message, forcing the device to leave a particular domain.

This leave_domain_msg() trigger is almost identical to the re_register_msg() message described in section 6.2.5, with the only adaptations being that :

- the message_tag is different. Refer to A.8 for the value of the message_tag.
- the shortform_domain_id is incorporated, which is the SLDF.

For the message **description** with an explanation of the parameters refer to the re_register_msg() message. For sake of completion the complete leave_domain_msg() message **syntax** is explained below:

6.4.3.4.1 Leave Domain message syntax

Table 29: leave domain message syntax

fields	length	Type
leave domain msg() {		
/* signature protected part starts here */		
message_tag	8	bslbf
protocol version	4	bslbf
reserved_for_future_use	4	bslbf
longform udn()	80	bslbf
flags {		
signature_type_flag	1	bslbf
ri certificate counter	3	bslbf
ocsp response counter	3	bslbf
reserved for future use	1	bslbf
}		
shortform_domain_id	48	uimsbf
certificate version	8	bslbf
for(cnt1=0; cnt1 < ri certificate counter ;cnt1++){		
c_length	16	uimsbf
ri_certificate(0	8*c_length	bslbf

}		
for(cnt2=0; cnt2 < ocsp_response_counter ;cnt2++){		
r_length	16	uimsbf
ocsp_response()	8*r_length	bslbf
}		
/* signature_protected part ends here */		
if (signature_type_flag == 0x0){		
signature_block	1024	bslbf
} else if (signature_type_flag == 0x1)		
signature_block	2048	bslbf
} else if (signature_type_flag == 0x2)		
signature_block	4096	bslbf
}		
}		

7. Broadcast Rights

7.1 Broadcast Rights Objects

7.1.1 Goals and Constraints

The delivery of rights objects over a Broadcast network without return channel necessitates some changes to the current ROAP because of the following reasons:

- the XML encoding according to the ROAP schema is not optimised for size
- the current ROAP does not support a subscription group addressing mechanism
- the current ROAP uses signatures based on the RSA PKI scheme that yield large signatures.

This chapter defines a new format for the delivery of authenticated and integrity protected rights objects, in which content encryption keys are cryptographically protected with either:

- domain key
- subscription group addressing group key
- subscription group addressing subset key (derived key)
- subscription group addressing device key

The primary design goal is to offer the same or equivalent cryptographic protection on Broadcast Rights Objects as is available for Rights Objects obtained via the standard ROAP protocol. This includes authentication, integrity checking and confidentiality of encryption keys.

The secondary design goal is optimisation of message size. This is motivated by the fact that these rights objects may have to be Broadcast repeatedly, as no return path is available to confirm reception. It is assumed that an out-of-band mechanism is available to perform an equivalent of a RORequest, i.e. the initiation of rights object acquisition.

7.1.2 Design Considerations and Decisions

The Broadcast Rights Objects (BCRO) are intended to be Broadcast to receivers in a well-defined repetitive manner. The particular means of delivery is to be defined in the context of the Broadcast system. It is the intention to support devices without a return channel (next to more capable devices), which implies that Broadcast rights object will be transmitted repeatedly to increase the chance of a receiver to capture rights objects addressed to that device.

The key-wrapping technique used in standard ROAP to cryptographically bind a MAC and REK to a device or domain will not be used. Instead the domain key or subscription group key is directly used to protect the content encryption keys in the Broadcast Rights Object. The motivation for this is that a REK adds little or no extra security, but adds significant size to a Broadcast Rights Object.

Because subscription group addressing offers the possibility to address a single unique device, BCROs will offer only addressing subscription groups or domains. Addressing a device using its device ID will not be supported with a BCRO.

RSA signatures on Broadcast rights objects would contribute very significantly to the size of each BCRO. Instead, each BCRO is protected with a MAC, based on an authentication key that is registered in the rights issuer context in a device. At registration, this authentication key is provided along with the subscription group addressing key material.

The broadcast content is protected with a varying encryption key. The encryption keys associated with assets in the BCRO will be applied to decrypt the key stream messages on the key stream layer. Besides decryption, such messages should also be authenticated. To avoid using the rights issuer authentication key for these frequent messages, the BCRO also carries an authentication key to be used for authenticating key stream messages. [This is subject to specifications of the key stream layer in OMA BCAST.]

7.2 Format of the Broadcast Rights Object

7.2.1 Format of the OMADRMBroadcastRightsObject class

The *OMADRMAsset*, *OMADRMPermission* and *OMADRMConstraint* object correspond in their meaning to their counterparts in OMA-DRM-REL-V2_0. The *OMADRMAction* object corresponds to the allowed elements in the permissions element from the same specification.

```
align(8) class OMADRMBroadcastRightsObject
{
      int i;
      // MAC protected part starts here
      bit(8)  message_tag;
      bit(4)  version;
      bit(12) bcro_length;

      bit(1)  group_size_flag;
      bit(1)  timestamp_flag;
      bit(1)  stateful_flag;
      bit(1)  refresh_time_flag;
      bit(3)  address_mode;
      bit(1)  rights_issuer_flag;
      bit(32) address;
      if ( address_mode == 0x1 )
      {
             if ( group_size_flag == 0)
             {
                    bit(256)       bit_access_mask;
             }
             else
             {
                    bit(512)       bit_access_mask;
             }
      }
      else if ( address_mode&0x6 == 0x2 )
      {
             bit(8)  position_in_group;
      }
      else if ( address_mode == 0x4 )
      {
             bit(6)  domain_id_extension;
             bit(10)        domain_generation;
      }
      if ( rights_issuer_flag == 1 )
      {
             bit(160)       rights_issuer_id;
      }
      if ( timestamp_flag == 1 )
      {
             bit(40) bcro_timestamp;
```

```
        }
        if ( refresh_time_flag == 1 )
        {
                bit(40) refresh_time;
        }
        bit(1)  permissions_flag;
        bit(7)  rekeying_period_number;
        bit(32) purchase_item_id;

        bit(8)  number_of_assets;
        for ( i=0; i<number_of_assets; i++ )
        {
                OMADRMAsset    asset[i];
        }
        if ( permissions_flag == 1 )
        {
                bit(8)  number_of_permissions;
                for ( i=0; i<number_of_permissions; i++ )
                {
                        OMADRMPermission permission[i];
                }
        }

        // MAC protected part ends here
        bit(96) MAC;

}
```

message_tag: Tag identifying this message as a BCRO. The value for this filed is defined in A.8.

version: 3-bit flag which indicates the version of the BCRO message format. If set to 0 the original format is used. Devices SHALL ignore BCROs with versions it does not support.

bcro_length: 12-bit field indicating the length in bytes of the BCRO starting immediately after this field (excluding locally added information). The size of an BCRO SHALL NOT exceed 4096 bytes. Note however that other restrictions, e.g. the UDP packet size can restrict the size of an BCRO even more.

Note: the fields up to and including 'length' are not protected by a MAC. All following fields up to but not including the MAC field will be protected by a MAC.

group_size_flag: 1-bit field indicating the group size used. 0 – a maximum group size 256 is used, 1 – a maximum group size of 512 is used

timestamp_flag: 1-bit field indicating that the BCRO is timestamped.

stateful_flag: 1-bit flag indicating that when set to 1 the BCRO contains stateful information.

refresh_time_flag: 1-bit flag indicating that a refresh_time for the BCRO is contained in this BCRO

address_mode: 3-bit field indicating the addressing mode used by this BCRO.

Field: address_mode	Description

0x0	addressing whole of unique group
0x1	addressing of Subscription group using a bit_mask size of 256 or 512 bit depending on group_size_flag (subset of unique group)
0x2-0x3	addressing of unique device
0x4	addressing of OMA domain. Address field concatenated with the domain_id_extension will be the domain id in this case
0x5-0x7	reserved for future use

rights_issuer_flag: 1-bit flag indicating that the rights issuer id is listed in this BCRO. Normally this information is given via a dedicated BCRO stream. This flag will only be set if BCROs from different rights issuers are carried in the same stream.

rights_issuer_id: The ID of the rights issuer. This is the 160-bit SHA1 hash of the DER encoded public key of the RI. See X509PKISHash in OMA.

address: 4-byte group address. Each rights issuer has its own address space. If the group_size is 512 then the group address is made of the first 31 bit of the address field. If the BCRO is addressed to a unique device in a group then the LSB of the address field is the MSB of the group position.

bit_access_mask: If the BCRO addresses a subset of a unique group (address_mode 0x1) than the bit_access_mask defines to which receivers in the group this BCRO is addressed to. Receivers not listed in the bit_access_mask cannot decrypt the key material in this BCRO as zero message Broadcast encryption is used for the encryption of the key material. The size of the bit_access_mask is given by the address mode

position_in_group: If the BCRO addresses a unique device then this field specifies the position of the unique device in the given Subscription group. If group_size_flag is 0 than the position in the group is directly given by the position_in_group field. If group_size_flag is 1 then 9 bit are used to identify the position in the group. If group_size_flag is 1then bit 0 (the LSB) from the address_mode is used as the 9th bit, the MSB. The real position in the group is then given by:

```
int real_position_in_group;
if(address_mode&0x6==0x2)
{
        if(group_size_flag == 0)
        {
                //maximum size of 256 devices in group.
                real_position_in_group = position_in_group;
        }
        else
        {
                //maximum size of 512 devices in group;
                real_position_in_group =
                ((address_mode&0x1)<<8)||position_in_group;
        }
```

```
}
```

domain_id_extension: The domain_id is given by the address field concatenated with the domain_id_extension to form a 38 bit id:

```
domain_id = (address<<6)|domain_id_extension
```

domain_generation: This 10 bit field specifies the generation of the domain.

bcro_timestamp: Field containing a timestamp at the point of issuing of the BCRO. This 40-bit field contains the time and date of the moment of issuing of the BCRO in Universal Time, Co-ordinated (UTC) and Modified Julian Date (MJD). This field is coded as 16 bits giving the 16 LSBs of MJD followed by 24 bits coded as 6 digits in 4-bit Binary Coded Decimal (BCD).

EXAMPLE 1: 93/10/13 12:45:00 is coded as "0xC079124500".

refresh_time: The refresh_time specifies the time when the terminal should acquire a new BCRO. It does not specifies when the keys in the BCRO expire. This field is a hint to a receiver to acquire a new BCRO for the content listed in the BCRO before the keys in the BCRO expires. The encoding is similar to that of the bcro_timestamp field.

permissions_flag: 1-bit flag indicating that the BCRO contains at least 1 permission.

rekeying_period_number: 7-bit counter used to differentiate between different ROs with the same purchase_item_id.

purchase_item_id: 32-bit field specifying the purchase ID this RO is associated with

number_of_assets: This field specifies the number of assets (see below) in this BCRO. Each asset listed in this BCRO has an internal id which is equal to the index of the asset in this BCRO. In other words the first asset listed in this BCRO has the internal asset id (index) of 0, the second of 1 etc. This internal id or index is used by permissions objects (see below) to identify the assets it addresses.

number_of_permissions:This field specifies the number of permissions (see below) in this BCRO.

MAC: This is the authentication code calculated over all bytes before this field with the exception of the first two bytes in the BCRO using HMAC-SHA-1-96 (see [RFC 2104]).

The MAC is used to authenticate and check the integrity of the BCRO. The key used to create the MAC is the BCRO authentication key BAK as described in A.9.3.

7.2.2 Format of the OMADRMAsset class

```
class OMADRMAsset
{
      int i;
      bit(96) BCI;
      bit(1)  key_flag;
      bit(1)  key_type;
      bit(2)  reserved_for_future_use;
      bit(1)  inherit_flag;
      bit(2)  asset_type;
      bit(1)  reserved_for_future_use;
      if ( inherit_flag )
```

```
        {

                bit(32) purchase_item_id;
                bit(1)  reserved_for_future_use;
                bit(7)  rekeying_period_number;


        }
        if ( key_flag == 1 )
        {
                if ( asset_type == 0x0 )
                {

                        if ( key_type == 0x0 )
                        {
                                bit(256)        encrypted_service_encryption_authentication_key;
                        }
                        else if ( key_type == 0x1 )
                        {
                                bit(256)        encrypted_program_encryption_authentication_key;
                        }
                }
                else
                if ( asset_type == 0x1 )
                {
                        bit(128)        encrypted_content_encryption_key;
                }
        }
}
```

BCI: This 96-bit field is the Binary Content ID. [The encoding of this field might be the SHA1-96 hash of the Content ID in 'cid' URI form.]

reserved_for_future_use: all fields reserved_for_future_use SHALL be set to 0 for this version of the specification.

key_flag:1-bit flag indicating that the asset does contain key material.

key_type: 1-bit flag indicating the type of the key material. If set to 0 the key material contains a service encryption key (SEK), when set to 1 it contains a program encryption key (PEK).

inherit_flag: 1-bit flag indicating whether inheritance is used. If set to 1 the asset inherits the rights setting from a parent rights object.

asset_type: 2-bit flag indicating the asset type as defined in the table below. If the asset_type is set to 0 the asset MAY contain either a PEK or a SEK. If the asset_type is set to 0x1 then the asset MAY contain a content encryption key.

Field: asset_type	Description
0x0	Broadcast stream protected IPSec or SRTP as defined in this specification
0x1	downloaded file content as defined by OMA

0x2-0x3	reserved

purchase_item_id: 32-bit field specifying the purchase ID this RO is associated with.

rekeying_period_number: 7-bit field specifying the rekeying_period_number of the parent rights object. The purchase_item_id and rekeying_period_number are used together with the socID and deviceID or domainID to uniquely identify the parent rights object.

encrypted_service_encryption_authentication_key: If key_type is set to 0 then this field contains the encrypted SEAK, the service encryption key (SEK) concatenated with the Service Authentication Seed (SAS). The field itself is protected using AES-128-CBC, with fixed IV 0 and with 0 padding in the last block if needed. The key used to decrypt this field depends on the addressing mode of the BCRO.

Field: address_mode	Keys used
0x0 (subscription group addressing / whole group)	UGK (Unique Group Key)
0x1 (subscription group addressing / derived keys)	Deduced decryption key (based on bit_access_mask and subscription group keys)
0x2 or 0x3 (unique device)	UDK (Unique device key)
0x4 (OMA Domain)	LDK (Local Domain Key)

encrypted_program_encryption_authentication_key: If key_type is set to 1 then this field contains the encrypted PEAK, the program encryption key (PEK) concatenated with the program authentication seed (PAK). The field itself is protected using AES-128-CBC, with fixed IV 0 and with 0 padding in the last block if needed. The key used to decrypt this field is depending on the addressing mode of the BCRO.

Field: address_mode	Keys used
0x0 (subscription group addressing / whole group)	UGK (Unique Group Key)
0x1 (subscription group addressing / derived keys)	Deduced decryption key (based on bit_access_mask and subscription group keys)
0x2 or 0x3 (unique device)	UDK (Unique device key)
0x4 (OMA Domain)	LDK (Local Domain key)

encrypted_content_encryption_key: This field contains the encrypted content encryption key (CEK). The field is protected using AES-128-CBC, with fixed IV 0 and with 0 padding in the last block if needed. The key used to decrypt this field is depending on the addressing mode of the BCRO.

Field: address_mode	Keys used
0x0 (subscription group addressing / whole group)	UGK (Unique Group Key)
0x1 (subscription group addressing / derived keys)	Deduced decryption key (based on bit_access_mask and subscription group keys)
0x2 or 0x3 (unique device)	UDK (Unique device key)

0x4 (OMA Domain)	LDK (Local Domain key)

[Editor's note: The following table was not removed by CR0254, while that CR did produce an alternative table. So the table below probably has to be deleted. However, a clerical CR or consensus agreement to remove this table is required.]

Field: address_mode	**Key(s) used to decrypt field**
0x0 (unique group)	UGK (Unique Group Key)
0x1 or 0x2 (Subscription group)	Deduced decryption key (based on bit_access_mask and subscription group keys)
0x3 (unique device)	UDK (Unique device key)

7.2.3 Format of the OMADRMPermission class

```
class OMADRMPermission
{
      int i;
      bit(6)  number_of_assets;
      bit(1)  constraint_flag;
      bit(1)  actions_flag;
      for ( i=0; i<number_of_assets; i++ )
      {
            bit(8)  asset_index;
      }
      if ( constraint_flag == 1 )
      {
            OMADRMConstraint      constraint;
      }
      if ( actions_flag == 1 )
      {
            bit(8)  number_of_actions;
            for ( i=0; i<number_of_actions; i++)
            {
                  OMADRMAction   action[i];
            }
      }
}
```

number_of_assets: The number of assets this permission object links to. Assets linked to by this permission object are bound by this permission object.

constraint_flag: 1-bit flag which indicates when set to 1 that a constraint object is present in this permissions object. The constraint object applies to all action listed in this permission object.

action_flag: 1-bit flag. When set to 1, 1 or more actions are contained in this permission object.

asset_index: A list of number_of_assets links to assets in this BCRO. Assets are linked to by using the internal asset id (the index of the asset in this BCRO).

number_of_actions: Field specifying the number of actions (see below) contained in this permission object

7.2.4 Format of the OMADRMAction class

```
class OMADRMAction
{
      bit(7)  action_type;
      bit(1)  constraint_flag;
      if ( constraint_flag )
      {
            OMADRMConstraint constraint;
      }
}
```

action_type: 7-bit field specifying the type of action as listed in table below:

Field: action_type	Description
0x00	PLAY_ACTION
0x01	DISPLAY_ACTION
0x02	EXECUTE_ACTION
0x03	PRINT_ACTION
0x04	EXPORT_ACTION
0x05	ACCESS_ACTION
0x06-0x7F	reserved for future use

constraint_flag: 1-bit flag which indicates when set to 1 that a constraint object is present in this action object. The constraint object only applies to the action it is in.

7.2.5 Format of the OMADRMConstraint class

```
abstract class OMADRMConstraintDescriptor : bit(8) constraint_id = 0
{
      bit(8) length;
}

class OMADRMConstraint
{
      int i;
      int j;
      bit(4)  number_of_constraints;
      bit(12) constraint_descriptor_length;
      for ( i=0; i<number_of_constraint; i++ )
      {
            OMADRMConstraintDescriptor constraint[i];
      }
}
```

number_of_constraints: 4-bit number specifying the number of constraint descriptors (see below)

constraints_descriptor_length: length of all constraint descriptors in bytes which follow this field.

constraint_tag: Tag identifying the specific constraint_descriptor as listed below:

Field: constraint_tag	Description
0x00	count constraint
0x01	timed-count constraint
0x02	date time constraint
0x03	interval constraint
0x04	accumulated constraint
0x05	individual constraint
0x06	system constraint
0x07-0xFF	reserved for future use

7.2.5.1 Count constraint descriptor

```
class OMADRMCountConstraintDescriptor extends OMADRMConstraintDescriptor
: bit(8) constraint_id = 0x00
{
      bit(8*length)          count;
}
```

length: The number of bytes used for the count field. Length SHALL NOT exceed 4, hence the maximum size of the count field can be 32 bits.

count: The number of times the content can be played. The field can be of size 8, 16, 24 and 32 bits.

7.2.5.2 Timed count constraint descriptor

```
class OMADRMTimedCountConstraintDescriptor extends OMADRMConstraintDescriptor
: bit(8) constraint_id = 0x01
{
      bit(16)                timer;
      bit(8*(length-2))      count;
}
```

length: The number of bytes following this field. The count field is length-2 bytes long and SHOULD NOT exceed 32 bits.

timer: Specifies the number of seconds after which the count state is reduced starting from beginning to render the content.

count: The number of times the content can be played. The field can be of size 8, 16, 24 and 32 bits.

7.2.5.3 Date-time constraint descriptor

```
class OMADRMDateTimeConstraintDescriptor extends OMADRMConstraintDescriptor
: bit(8) constraint_id = 0x02
```

```
{
      bit(1)                start_flag;
      bit(1)                end_flag;
      bit(6)                reserved;
      if ( start_flag )
      {
            bit(40)         start_date;
      }
      if ( end_flag )
      {
            bit(40)         end_date;
      }
}
```

length: The number of bytes of the descriptor immediately following this field.

start_flag: 1-bit field. When set the descriptor contains a start time.

end_flag: 1-bit field. When set the descriptor contains a end time.

start_time: Time field with the semantics of 'not before' time for a permission. The start_time must be before the end_time if present.

end_time: Time field with the semantics of 'not after' time for a permission. The end_time must be after the start_time if present.

7.2.5.4 Interval constraint descriptor

```
class OMADRMIntervalConstraintDescriptor extends OMADRMConstraintDescriptor
: bit(8) constraint_id = 0x03
{
      bit(8*length)         time_interval;
}
```

length: The number of bytes following this field. Length specifies the size of the time_interval field.

time_interval: Specifies the number of seconds starting from first receiving this BCRO that the permission is valid. The length of the field is given by the length field and SHOULD NOT exceed 32 bit.

7.2.5.5 Accumulated constraint descriptor

The accumulated_constraint_descriptor specifies the maximum period of metered usage time during which the rights can be exercised over the DRM content.

```
class OMADRMAccumulatedConstraintDescriptor extends OMADRMConstraintDescriptor
: bit(8) constraint_id = 0x04
{
      bit(8*length)         accumulated_time;
}
```

length: The number of bytes following this field. Length specifies the size of the accumulated_time field.

accumulated_time: Specifies the maximum period of metered usage time during which the rights can be excercised. The period is given in seconds. The length of the field is given by the length field and SHOULD NOT exceed 32 bit.

7.2.5.6 Individual constraint descriptor

Constraint used to bind content to individuals. If the content should be bound to more than one individual multiple individual_constraint_descriptor(s) can be carried in one constraint object.

```
class OMADRMIndividualConstraintDescriptor extends OMADRMConstraintDescriptor
: bit(8) constraint_id = 0x05
{
      bit(4)                reserved;
      bit(4)                id_type;
      bit(8*(length-1))     individual_id;
}
```

length: The number of bytes following this field. Length-1 specifies the size of the individual_id field.

id_type: Tag identifying format of the individual_id as listed below:

Field: id_type	Description
0x0	The individual_id field contains the IMSI number coded as 16 digit 4-bit BCD. The first digit SHALL be 0 and SHALL be ignored. The length of the individual_id field is 64 bit.
0x1	The individual_id field contains the PKC id of the WIM to which the content is bound.
0x2-0xF	reserved for future use

individual_id: Individual ID. The format and length of this field is identified by the identifier_type and length field see the table above.

7.2.5.7 System constraint descriptor

Constraint used identify systems to which the content and rights objects are allowed to be exported to.

```
class OMADRMSystemConstraintDescriptor extends OMADRMConstraintDescriptor
: bit(8) constraint_id = 0x06
{
      bit(8)                constraint_tag;
      bit(8)                length;
      bit(64)               system_id;
}
```

length: The number of bytes following this field.

system_id: The system id of the system the content and RO can be exported to. This is the SHA1-64 encoded hash of the system name as registered with OMNA. [The values are registered with OMNA (currently only strings), we either use SHA1-64 to hash the strings or OMNA registers numbers for that as well]

7.3 Interaction Channel Rights Objects

Terminals can acquire rights to access broadcast content by retrieving and processing binary Broadcast Rights Objects. In addition, terminals that support an interaction channel next to the broadcast interface can also acquire rights to access broadcast content via the ROAP protocol or the exchange of Domain RO's.

The ROAP protocol via the interaction channel ensures an authenticated delivery of one or more <protectedRO> elements. The exchange of Domain RO's also consists of the exchange of one or more <protectedRO> elements.

If a <protectedRO> is to convey rights to access broadcast content, then the following applies for all assets that encode rights for broadcast content:

- The <o-dd:uid> element in the <o-ex:context> element in the <o-ex:asset> element MUST hold the BCI (binary content identifier) for the broadcast content referred to by this asset.
- The <o-ex:digest> element in the <o-ex:asset> SHALL NOT be present.
- The <xenc:CipherValue> element contained in the <ds:KeyInfo> element MUST hold the wrapped concatenation of the encryption key (SEK or PEK) and the authentication key (SAK or PAK). The concatenation method is equal to that of the MAC key and the RO encryption key REK in the <roap:ROPayload> element of the <roap:ROResponse> message where the SEK or PEK takes the place of the MAC and the SAK or PAK takes the place of the REK.

8. Usage Metering

8.1 Additions to the OMA DRM 2.0 REL

The new *token-based* constraint defines that usage of the DRM content involves consumption of tokens. A device can receive tokens from multiple RIs and use them to consume DRM content whose usage is defined as token-based in the RO associated with the DRM content.

The <token-based> element will have the following 3 required attributes:

- *token-constraint-type*: The type of stateful consumption which is governed by token availability. Currently the only suitable constraints in the REL are count, timed-count and accumulated.
- *token-unit*: The unit of the specified constraint which corresponds to tokens being decremented, e.g. a single count or 30 minutes of time.
- *tokens-consumed*: Tokens consumed per token unit, e.g. 3 tokens consumed for every count.

An example of the usage of this constraint (shown below) instructs the DRM agent to consume two tokens every time that the corresponding content item is played.

```
<o-dd:play>
<o-ex:constraint>
    <oma-dd:token-based>
                                    <oma-dd:token-constraint>count</o-dd:token-constraint>
                                    <oma-dd:token-unit>1</o-dd:token-unit>
                                    <oma-dd:tokens-consumed>2</o-dd:tokens-consumed>
     </oma-ex:token-based>
</o-ex:constraint>
</o-dd:play>
```

Figure 8: Example Usage of Token-based Constraint

8.1.1 REL DTD Additions

Figure 9Figure 9Figure 9 shows the additions required to the existing REL DTD to introduce the metering constraint.

```
<!ELEMENT o-ex:rights (o-ex:context, o-ex:agreement)>
<!ATTLIST o-ex:rights
      xmlns:o-ex CDATA #FIXED "http://odrl.net/1.1/ODRL-EX"
      xmlns:o-dd CDATA #FIXED "http://odrl.net/1.1/ODRL-DD"
      xmlns:oma-dd CDATA #FIXED "http://www.openmobilealliance.com/oma-dd"
      xmlns:ds CDATA #FIXED "http://www.w3.org/2000/09/xmldsig#"
      xmlns:xenc CDATA #FIXED "http://www.w3.org/2001/04/xmlenc#"
>
<!ELEMENT o-ex:context (o-dd:version?, o-dd:uid*)>
<!ELEMENT o-dd:version (#PCDATA)>
```

```
<!ELEMENT o-dd:uid (#PCDATA)>
<!ELEMENT o-ex:agreement (o-ex:asset+, o-ex:permission*)>
<!ELEMENT o-ex:asset (o-ex:context?, o-ex:inherit?, o-ex:digest?, ds:KeyInfo?)>
<!ATTLIST o-ex:asset
   o-ex:id ID #IMPLIED
   o-ex:idref IDREF #IMPLIED
>
<!ELEMENT o-ex:inherit (o-ex:context)>
<!ELEMENT o-ex:digest(ds:DigestMethod, ds:DigestValue)>
<!ELEMENT ds:DigestMethod (#PCDATA)>
<!ATTLIST ds:DigestMethod
      Algorithm CDATA #FIXED "http://www.w3.org/2000/09/xmldsig#sha1"
>
<!ELEMENT ds:DigestValue (#PCDATA)>
<!ELEMENT ds:KeyInfo (xenc:EncryptedKey?, ds:RetrievalMethod?)>
<!ELEMENT xenc:EncryptedKey (ds:KeyInfo?, xenc:EncryptionMethod, xenc:CipherData)>
<!ELEMENT xenc:EncryptionMethod (#PCDATA)>
<!ATTLIST xenc:EncryptionMethod
      Algorithm CDATA #FIXED "http://www.w3.org/2001/04/xmlenc#kw-aes128"
>
<!ELEMENT xenc:CipherData (xenc:CipherValue)>
<!ELEMENT xenc:CipherValue (#PCDATA)>
<!ELEMENT ds:RetrievalMethod (#PCDATA)>
<!ATTLIST ds:RetrievalMethod
      URI CDATA #REQUIRED
>
<!ELEMENT o-ex:permission (o-ex:constraint?, o-ex:asset*, o-dd:play?, o-dd:display?, o-dd:execute?, o-
dd:print?, oma-dd:export?)>
<!ELEMENT o-dd:play (o-ex:constraint?)>
<!ELEMENT o-dd:display (o-ex:constraint?)>
<!ELEMENT o-dd:execute (o-ex:constraint?)>
<!ELEMENT o-dd:print (o-ex:constraint?)>
<!ELEMENT o-ex:constraint (o-dd:count?, oma-dd:timed-count?, o-dd:datetime?, o-dd:interval?, o-
dd:accumulated?, o-dd:individual?, oma-dd:system*, oma-dd:token-based?)>
<!ELEMENT o-dd:count (#PCDATA)>
<!ELEMENT oma-dd:timed-count (#PCDATA)>
<!ATTLIST oma-dd:timed-count
      oma-dd:timer CDATA #IMPLIED
>
<!ELEMENT o-dd:datetime (o-dd:start?, o-dd:end?)>
<!ELEMENT o-dd:start (#PCDATA)>
<!ELEMENT o-dd:end (#PCDATA)>
<!ELEMENT o-dd:interval (#PCDATA)>
<!ELEMENT o-dd:accumulated (#PCDATA)>
<!ELEMENT o-dd:individual (o-ex:context)>
<!ELEMENT oma-dd:export (o-ex:constraint)>
<!ATTLIST oma-dd:export
      oma-dd:mode (move | copy) #REQUIRED
>
<!ELEMENT oma-dd:system (o-ex:context)>
```

```
<!ELEMENT oma-dd:token-based (oma-dd:token-constraint-type)>
<!ELEMENT oma-dd:token-constraint-type (count | timed-count | accumulated)>
<!ATTLIST oma-dd:token-constraint-type
      oma-dd:timer CDATA #IMPLIED
>
<!ATTLIST oma-dd:token-based
      oma-dd:token-unit PCDATA #REQUIRED
      oma-dd:tokens-consumed  PCDATA  #REQUIRED>
>
```

Figure 9: REL DTD Additions

An example of the usage of the metering constraint shown below instructs the DRM agent to consume two tokens every time that the corresponding content item is played once.

```
<o-ex:permission>
                <o-dd:play/>
                <o-ex:constraint>
                    <oma-dd:token-based>
                                                <oma-dd:token-constraint>count</oma-dd:token-constraint>
                                                <oma-dd:token-unit>1</o-dd:tokenunit>
                                                <oma-dd:tokens-consumed>2</o-dd:tokensconsumed>
                     </oma-ex:token-based>
                </o-ex:constraint>
</o-ex:permission>
```

8.1.2 Element <token-based>

Element	<!ELEMENT oma-dd:token-based>
Semantics	The <token-based> element specifies how many tokens are consumed when the DRM content to which the constraint applies is used in a certain way. If the DRM agent detects that no tokens for this RI, or a number of tokens less than that contained in the <tokens-consumed> attribute of the <token-based> element are available, the DRM Agent MUST NOT grant the corresponding permission to the DRM Content.

8.1.3 Element <token-constraint>

Element	<!ELEMENT oma-dd:token-constraint (count | timed-count | accumulated)>

Semantics	The <token-constraint> element must contain either “count”, “timed-count” or “accumulated”. If it contains “count”, then every time the number of counts specified in the corresponding <token-unit> element is consumed, the token store is decremented by the number of tokens in the <tokens-consumed> element. For example, if the <token-unit> element contains “1” and the <tokens-consumed> element contains “2”, then each time the permission is exercised the token store is decremented by 2 tokens. If it contains “timed-count”, then the semantics are as for the “count” value but with the addition of an optional timer attribute. If the timer attribute is omitted or contains an invalid value, or if the device is not able to measure the time passed as required by the semantics of this element, then the device MUST reduce the count state immediately upon beginning to access the content. In this case the semantics of the “timed-count “ value are identical to those of the “count” value. If the value is “accumulated”, this specifies the maximum period of time during which the permission can be exercised over the DRM Content before the token store is decremented by the number of tokens in the <tokens-consumed> element. For example, if the <token-unit> element specifies 900 seconds and the <tokens-consumed> element contains “1”, then each time the permission is exercised for 900 seconds (since the last token decrement) the token store is decremented by 1 token.

8.1.4 Attribute “timer”

Attribute	<!ATTLIST oma-dd:token-constraint oma-dd:timer CDATA #IMPLIED>
Semantics	The attribute contains a positive integer value. It specifies the number of seconds after which the count state is reduced starting from beginning to render the Content. For example, if the timer value is set to “30” (without the quotes) the count state is decremented after the content has been rendered for 30 seconds. When the number of counts specified in the corresponding <token-unit> element is consumed, the token store is decremented by the number of tokens in the <tokens-consumed> element. For example, if the <token-unit> element contains “1”, the timer attribute specifies 30 seconds, and the <tokens-consumed> element contains 2, then each time the permission is exercised for at least 30 seconds, the token store is decremented by 2 tokens. The timer attribute should be defined only when the value of the <token-constraint> element is “timed-count”.

8.1.5 Attribute “token-unit”

Attribute	<!ATTLIST oma-dd:token-based oma-dd:token-unit PCDATA #REQUIRED >

Semantics	The format of the <token-unit> element depends on the value of the <token-constraint-type> element. If the <token-constraint> element contains "count", the corresponding <token-unit> value specifies the number of times permission may be granted over an asset in order for the corresponding number of tokens in the <tokens-consumed> element to be consumed. This must be a positive integer value. If the <token-constraint> element contains "accumulated", then the <token-unit> value specifies the number of seconds a permission may be granted over an asset in order for the corresponding number of tokens in the <tokens-consumed> element to be consumed. The lexical representation of this value MUST use the restricted accumulated format PnDTnHnMnS or any reduced precision and truncated representation version thereof as specified in [XMLSchema]. For example, P15DT10H30M20S represents an accumulated of 15 days, 10 hours, 30 minutes and 20 seconds. The specified period SHOULD be greater than zero. If the specified period is equal to zero, then the permission MUST NOT be granted. [XMLSchema] allows the number of seconds in the period to include decimal digits to arbitrary precision. However, to ensure interoperability, ROs MUST NOT contain fractional seconds in the period.

8.1.6 Attribute "tokens-consumed"

Attribute	<!ATTLIST oma-dd:token-based oma-dd:tokens-consumed PCDATA #REQUIRED >
Semantics	The attribute contains a positive integer value. It specifies the number of tokens by which the token store should be decremented when a single token unit (as specified in the "token-unit" attribute) is consumed when exercising the permission to which the <metered> constraint is attached. For example, if the "token-unit" indicates 900 seconds and "tokens-consumed" contains 1, then the token store will be decremented each time the DRM content is played for a total of 900 accumulated seconds.

8.2 Extensions to ROAP to Issue Tokens

ROAP would be extended to allow tokens to be delivered to a device. Either a 1-pass or 2-pass version can be used.

8.2.1 Token Delivery

The first element of the 2-pass ROAP extension is a ROAP trigger which prompts the device to send a token acquisition request to the RI. The RI then responds with a token acquisition response.

In the 1-pass version, only a ROAP-TokenResponse is delivered by the RI to the device.

8.2.2 TokenAcquisitionTrigger

The full extensions to the ROAP schema for triggers required to add this trigger are shown in **Error! Reference source not found.**. The elements in the token acquisition trigger have the following meanings:

- The RI ID MUST uniquely identify the Rights Issuer.
- The <nonce> element provides a way to couple ROAP triggers with ROAP requests.
- The DRM Agent MUST use the URL specified by the <roapURL> element when initiating the ROAP transaction. When the <roapTrigger> element carries a <tokenRequest> element, the PDU MUST be a ROAP-TokenAcquisitionRequest PDU.
- The Token Delivery ID identifies the token request in a similar way to the way the RO ID identifies an RO.
- If the trigger is signed, the <ds:Reference> element of the <ds:SignedInfo> child element of the trigger <signature> shall reference a ROAPTrigger element by using the same value for the URI attribute as the value for the ROAP trigger element's id attribute.

```
<schema
  targetNamespace="urn:oma:bac:dldrm:roap-trigger-1.0"
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:roap-trigger="urn:oma:bac:dldrm:roap-trigger-1.0"
  xmlns:roap="urn:oma:bac:dldrm:roap-1.0"
  xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
  xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
  elementFormDefault="unqualified"
  attributeFormDefault="unqualified">

<import namespace="urn:oma:bac:dldrm:roap-1.0" schemaLocation="roap.xsd"/>

<import namespace="http://www.w3.org/2000/09/xmldsig#"
      schemaLocation="http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/xmldsig-core-
schema.xsd"/>

<import namespace="http://www.w3.org/2001/04/xmlenc#"
      schemaLocation="http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/xenc-schema.xsd"/>

<complexType name="RegistrationRequestTrigger">
  <sequence>
    <element name="riID" type="roap:Identifier"/>
    <element name="nonce" type="roap:Nonce" minOccurs="0"/>
    <element name="roapURL" type="anyURI"/>
  </sequence>
  <attribute name="id" type="ID"/>
</complexType>

<complexType name="ROAcquisitionTrigger">
  <sequence>
    <element name="riID" type="roap:Identifier"/>
    <element name="nonce" type="roap:Nonce" minOccurs="0"/>
    <element name="roapURL" type="anyURI"/>
    <element name="domainID" type="roap:DomainIdentifier"
          minOccurs="0"/>
    <sequence maxOccurs=”unbounded”>
      <element name="roID" type="ID"/>
      <element name="contentID" type="anyURI" maxOccurs="unbounded"/>
```

```
    </sequence>
  </sequence>
  <attribute name="id" type="ID"/>
</complexType>

<complexType name="DomainTrigger">
  <sequence>
    <element name="riID" type="roap:Identifier"/>
    <element name="nonce" type="roap:Nonce" minOccurs="0"/>
    <element name="roapURL" type="anyURI"/>
    <element name="domainID" type="roap:DomainIdentifier"/>
  </sequence>
  <attribute name="id" type="ID"/>
</complexType>

<complexType name="TokenAcquisitionTrigger">
  <sequence>
    <element name="riID" type="roap:Identifier"/>
    <element name="nonce" type="roap:Nonce" minOccurs="0"/>
    <element name="roapURL" type="anyURI"/>
     <element name="tokenID" type="ID"/>
    <attribute name="id" type="ID"/>
</complexType>


<!-- ROAP trigger -->
<element name="roapTrigger" type="roap-trigger:RoapTrigger"/>
<complexType name="RoapTrigger">
  <annotation>
    <documentation xml:lang="en">
      Message used to trigger the device to initiate a Rights Object Acquisition Protocol.
    </documentation>
  </annotation>
  <sequence>
    <choice>
      <element name="registrationRequest" type="roap-trigger:RegistrationRequestTrigger"/>
      <element name="roAcquisition" type="roap-trigger:ROAcquisitionTrigger"/>
      <element name="joinDomain" type="roap-trigger:DomainTrigger"/>
      <element name="leaveDomain" type="roap-trigger:DomainTrigger"/>
      <element name="tokenAcquisition" type="roap-trigger:TokenAcquisitionTrigger"/>

    </choice>
    <element name="signature" type="ds:SignatureType" minOccurs="0"/>
    <element name="encKey" type="xenc:EncryptedKeyType" minOccurs="0"/>
  </sequence>
  <attribute name="version" type="roap:Version"/>
  <attribute name=”proxy” type=”boolean”/>
</complexType>
</schema>
```

Figure 10: Addition of Token Acquisition Trigger to Schema

8.2.3 ROAP-TokenRequest

A device can create a token request from the device to a rights issuer. This is an extension of the existing ROAP request type.

ROAP-TokenRequest	
Parameter	**Mandatory/Optional**
Device ID	M
RI ID	M
Device Nonce	M
Token Delivery ID	M
Certificate Chain	M
Extensions	O
Signature	O

Figure 11: Token Request Message Description

Device ID identifies the requesting Device.

RI ID identifies the authorizing RI.

Device Nonce is a nonce chosen by the Device.

Token Delivery ID identifies the tokens to be issued to this device in a similar fashion to the way an RO ID identifies a RO.

Certificate Chain: This parameter is sent unless it is indicated in the RI Context that this RI has stored necessary Device certificate information. When present, the parameter value SHALL be as described for the Certificate Chain parameter in the ROAP-RegistrationRequest message.

Extensions: The following extensions are defined for the ROAP-TokenRequest message:

- *Peer Key Identifier*: An identifier for an RI public key stored in the Device. If the identifier matches the RI's current public key, or if the extension is empty, it means the Device has already stored the RI ID and the corresponding RI certificate chain, and the RI need not send down its certificate chain in its response message.
- *No OCSP Response*: Presence of this extension indicates to the RI that there is no need to send any OCSP responses since the Device has cached a complete set of valid OCSP responses for this RI.
- *OCSP Responder Key Identifier*: This extension identifies an OCSP responder key stored in the Device. If the identifier matches the key in the certificate used by the RI's OCSP responder, the RI MAY remove the OCSP Responder certificate chain from the OCSP response before providing the OCSP response to the Device.

The Device MUST send the Peer Key Identifier extension if, and only if, it has stored the RI public key. The Device MUST send the No OCSP Response extension if, and only if, it has a complete set of valid OCSP responses for the RI certificate chain. The Device MUST send the OCSP Responder Key Identifier extension if, and only if, it has stored an OCSP Responder key for this RI.

Signature is a signature on this message (besides the *Signature* element itself). The signature method is as follows:

- The message except the *Signature* element is canonicalized using the exclusive canonicalization method defined in [XC14N].
- The result of the canonicalization, *d*, is considered as input to the signature operation.

- The signature is calculated on *d* in accordance with the rules of the negotiated signature scheme

The RI MUST verify the signature on the ROAP-TokenRequest message.

8.2.3.1 Message syntax

The <tokenRequest> element specifies the ROAP-TokenRequest message. It has complex type roap:TokenRequest, which extends the basic roap:Request type.

```
<element name="tokenRequest" type="roap:TokenRequest"/>

<complexType name="TokenRequest">
 <annotation>
  <documentation xml:lang="en">
   Message sent from Device to RI to request tokens
  </documentation>
 </annotation>
 <complexContent>
  <extension base="roap:Request">
   <sequence>
    <element name="deviceID" type="roap:Identifier"/>
    <element name="riID" type="roap:Identifier"/>
    <element name="nonce" type="roap:Nonce"/>
    <element name ="tokenID" type="ID"/>
    <element name="certificateChain" type="roap:CertificateChain" minOccurs="0"/>
    <element name="extensions" type="roap:Extensions" minOccurs="0"/>
    <element name="signature" type="base64Binary"/>
   </sequence>
  </extension>
 </complexContent>
</complexType>
```

8.2.4 ROAP-TokenDeliveryResponse

The ROAP-TokenResponse is returned to the device by the RI in response to a ROAP-TokenRequest, or can also be used in the 1-pass version without any preceding messages.

Parameter	ROAP-TokenDeliveryResponse

	2-pass *Status = Success*	*2-pass* *Status ≠ Success*	*1-pass*
Status	M	M	M
Device ID	M	-	M
RI ID	M	-	M
Token Delivery ID	M	-	M
Device Nonce	M	-	-
Token Quantity	M	-	M
Token Reporting URL	O	-	O
Latest Token Consumption Time	O	-	O
Earliest Reporting Time	O	-	O
Latest Reporting Time	O	-	O
Certificate Chain	O	-	O
OCSP Response	O	-	M
Extensions	O	-	O
Signature	M	-	M

Figure 12: Token Delivery Response

Status indicates if the request was successfully handled or not. In the latter case, an error code as specified in OMA DRM 2.0 status codes are sent. The following additional status values are defined to support support token-based metering. These values are only valid in a token delivery response.

- TokenConsumptionReportError: The RI did receive a token consumption report but it was erroneous and the device should resend.
- NoTokenConsumptionReport: The RI did not receive a token consumption report yet, but was expecting one as the present date and time is later than the last token consumption time in a previous token delivery message.

Device ID identifies the requesting Device. The value returned here MUST equal the Device ID sent by the Device in the ROAP-TokenRequest message that triggered this response in the 2-pass ROAP. In the 1-pass ROAP, the RI selects the Device ID of the recipient Device. If the Device ID is incorrect, the ROAP-TokenResponse processing will fail and the Device MUST discard the received TokenResponse PDU.

RI ID identifies the RI. In the 2-pass protocol, the value MUST equal the RI ID sent by the Device in the preceding ROAP-TokenRequest message.

Token Delivery ID identifies the tokens to be issued to this device in a similar fashion to the way an RO ID identifies a RO. This ID should match the Token Delivery ID in the preceding *TokenRequest* message. Devices must discard token delivery response messages with a token delivery ID which is identical to the one in any previously processed token delivery response messages.

Device Nonce: If present (2-pass), MUST have the same value as the corresponding parameter value in the preceding ROAP-TokenRequest.

Token Quantity: Contains the number of tokens being issued. If this is a positive number, the device should increment its token store by the given quantity. If it is a negative number the device should decrement the token store by the given quantity.

Token Reporting URL: The presence of this parameter indicates that token consumption from this token delivery must be reported. The parameter defines the URL to which the *ROAPTokenConsumptionReport* message should later be sent.

Earliest Reporting Time: The device should report consumption after this time and before the latest reporting time. If the device reports consumption of tokens before the date/time defined in this parameter, in the subsequent token delivery response the RI may not change the latest token consumption time. In other words the next delivery of tokens is within the same reporting period. The field should only be defined when a token reporting URL is specified.

Latest Reporting Time: The device should report consumption before this time and after the earliest reporting time. If the RI receives the report before this time, it should send a new token delivery message before the latest token consumption time so the device can continue consumption. This field should only be defined when a token reporting URL is specified.

Latest token consumption time: After the date/time indicated in this parameter, the device SHALL NOT use any tokens which have been received after the last token delivery message which includes a token reporting URL. If reports are being made on time by the device, this date is constantly being updated and therefore consumption should never be blocked. This field should only be defined when a token reporting URL is defined.

OCSP Response: This parameter, when present, SHALL be a complete set of valid OCSP responses for the RI's certificate chain. The Device MUST NOT fail due to the presence of more than one OCSP response element. This parameter will not be sent if the Device sent the Extension No OCSP Response in the preceding ROAP-RegistrationRequest (and the RI did not ignore that extension).

Certificate Chain: This parameter MUST be present unless a preceding ROAP-TokenRequest message contained the Peer Key Identifier extension, the extension was not ignored by the RI, and its value identified the RI's current key. When present, the value of a Certificate Chain parameter shall be as described for the Certificate Chain parameter of the ROAP-RegistrationResponse message.

Signature is a signature on data sent in the protocol. The signature is computed using the RI's private key and the current message (besides the Signature element itself). The signature method is as follows:

- All elements except the Signature element are canonicalized using the exclusive canonicalization method defined in [XC14N].
- The resulting data d is considered as input to the signature operation.
- The signature is calculated on d in accordance with the rules of the negotiated signature scheme

The Device MUST verify this signature. A Device MUST NOT accept the token acquisition as successful unless the signature verifies, the RI certificate chain has been successfully verified, and the OCSP response indicates that the RI certificate status is good. If the acquisition protocol failed, the Device MUST NOT install the received tokens.

8.2.4.1 Message syntax

The **<tokenDeliveryResponse>** element specifies the ROAP-TokenDeliveryResponse message. It has complex type **roap:TokenDeliveryResponse**, which extends the basic **roap:Response** type.

```
<element name="TokenDeliveryResponse" type="roap:Response"/>
<complexType name="TokenDeliveryResponse">
 <annotation>
  <documentation xml:lang="en">
   Message sent from RI to Device to deliver tokens
  </documentation>
 </annotation>
 <complexContent>
  <extension base="roap:Response">
   <sequence minOccurs="0">
    <element name="deviceID" type="roap:Identifier"/>
    <element name="riID" type="roap:Identifier"/>
    <element name="tokenDeliveryID" type="roap:Identifier"/>
    <element name="nonce" type="roap:Nonce" minOccurs="0"/>
    <element name="tokenQuantity"  type="nonNegativeInteger"/>
    <element name="tokenReportingURL"  type="anyURI" minOccurs="0"/>
    <element name="earliestReportingTime"  type=" dateTime "  minOccurs="0"/>
    <element name="latestReportingTime"  type=" dateTime"  minOccurs="0"/>
    <element name="latestTokenConsumptionTime"  type="dateTime" minOccurs="0"/>
    <element name="certificateChain" type="roap:CertificateChain"  minOccurs="0"/>
    <element name="ocspResponse" type="base64Binary" minOccurs="0" maxOccurs="unbounded"/>
    <element name="signature" type="base64Binary"/>
   </sequence>
  </extension>
 </complexContent>
</complexType>
```

Figure 13: Message syntax of Token delivery response

The following changes are made to the status type. The **Status** simple type enumerates all possible error messages.

```
<simpleType name="Status">
                <restriction base="string">
                        <enumeration value="Success"/>
                        <enumeration value=”Abort”/>
                        <enumeration value="NotSupported"/>
                        <enumeration value="AccessDenied"/>
                        <enumeration value="NotFound"/>
                        <enumeration value="MalformedRequest"/>
                        <enumeration value="UnknownCriticalExtension"/>
                        <enumeration value="UnsupportedVersion"/>
                        <enumeration value="UnsupportedAlgorithm"/>
                        <enumeration value="NoCertificateChain"/>
                        <enumeration value="InvalidCertificateChain"/>
                        <enumeration value="TrustedRootCertificateNotPresent"/>
                        <enumeration value=”SignatureError”/>
```

```
                    <enumeration value="DeviceTimeError"/>
                    <enumeration value="NotRegistered"/>
                    <enumeration value="InvalidDCFHash"/>
                    <enumeration value="InvalidDomain"/>
                    <enumeration value="DomainFull"/>
                    <enumeration value="TokenConsumptionReportError">
                    <enumeration value="NoTokenConsumptionReport">

            </restriction>
</simpleType>
```

Figure 14: Updates to status type

Upon transmission or receipt of a message for which Status is not "Success", the default behaviour, unless explicitly stated otherwise below, is that both the RI and the Device SHALL immediately close the connection and terminate the protocol. RI systems and Devices are required to delete any session-identifiers, nonces, keys, and/or secrets associated with a failed run of the ROAP protocol.

8.3 Extensions for ROAP for Reporting

Reporting can be done via ROAP by devices with a backchannel. The report from the device is based on the ROAPRequest type. The response expected to such a request is a ROAPDeliveryResponse (1-pass) or ROAPDeliveryTrigger (2-pass).

ROAP-TokenConsumptionReport	
Parameter	**Mandatory/Optional**
Device ID	M
RI ID	M
Report Time	M
Tokens Consumed	M
Certificate Chain	O
Signature	M

Figure 15: ROAP TokenConsumptionReport

Device ID identifies the requesting Device.

RI ID identifies the RI.

Device Nonce: This parameter, if present, MUST have the same value as the corresponding parameter value in the preceding trigger.

Report Time is the current DRM Time, as seen by the Device.

Tokens Consumed: Contains information on how many tokens were consumed since the last report.

Certificate Chain: This parameter MUST be present. The value of a *Certificate Chain* parameter shall be as described for the *Certificate Chain* parameter of the ROAP-TokenConsumptionReport message.

Signature is a signature on this message (besides the Signature element itself). The signature method is as follows:

- The message except the Signature element is canonicalized using the exclusive canonicalization method defined in [XC14N].
- The result of the canonicalization, d, is considered as input to the signature operation.
- The signature is calculated on d in accordance with the rules of the negotiated signature scheme

The RI MUST verify the signature on the ROAP-TokenConsumptionReport message.

Finally, the device must receive and process a ROAP-TokenConsumptionReport. The device should clear all token consumption information for the preceding report period.

8.3.1 Message syntax

The **<tokenConsumptionReport>** element specifies the ROAP-TokenDeliveryResponse message. It has complex type **roap:TokenConsumptionReport**, which extends the basic **roap:Request** type.

```
<element name="tokenConsumptionReport" type="roap:TokenConsumptionReport"/>

<complexType name="TokenConsumptionReport">
  <annotation>
    <documentation xml:lang="en">
      Message sent from Device to RI to report token consumption report
    </documentation>
  </annotation>
  <complexContent>
    <extension base="roap:Request">
      <sequence>
        <element name="deviceID" type="roap:Identifier"/>
        <element name="riID" type="roap:Identifier"/>
        <element name="nonce" type="roap:Nonce"/>
        <element name="time" type="dateTime"/>
        <element name="tokensConsumed"  type="nonNegativeInteger"/>
        <element name="certificateChain" type="roap:CertificateChain" minOccurs="0"/>
        <element name="signature" type="base64Binary"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

Figure 16: Message Syntax of token consumption report

9. Subscriber Groups

9.1 Introduction

A subscriber group is a set of devices that share a group address along with cryptographic key material and algorithms that allow any subset of this group to be associated with a cryptographic key. A subscriber group can be cryptographically secure, which means that it has the additional property that any device from the group cannot deduce the distinct cryptographic keys for subsets that exclude the device.

The capability to address multiple devices using a single message provides for improved efficiency of the communication protocols. In particular it is very beneficial in the distribution of rights objects.

9.2 Addressing

9.2.1 Addressing Modes

Subscriber group addressing allows for three addressing modes, as is explained in figure 1 below.

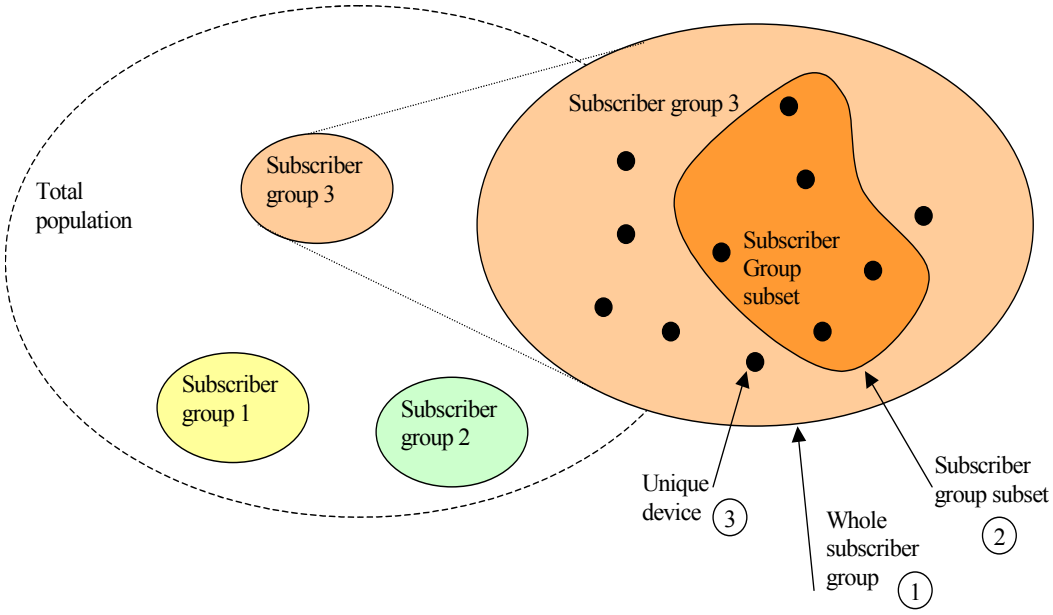


Figure 1: Subscriber group concept

A whole subscriber group contains all devices in a group. A subscriber group subset can be smaller than or as large as the whole group. One or more subscriber groups form the population of devices.

The following sections describe the relation between the registration data and the Broadcast Rights Object (a.k.a. BCRO). The registration data is sent to the device after successful registration of the device. At a later stage the device may receive a BCRO as a means to obtain the content (encryption) key, which in turn is used to decrypt the encrypted AV content. When using subscriber group addressing, the content key is encrypted with a Deduced Encryption Key (DEK) by the RI.

There are three types of addressing possible.

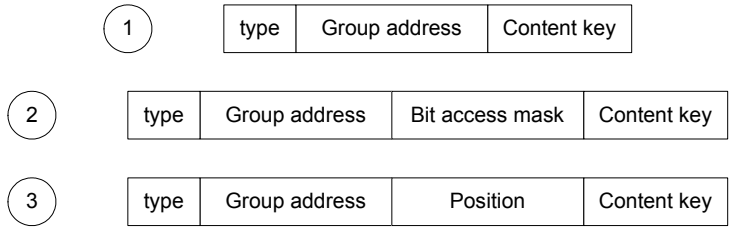


Figure 17: Addressing modes

The first addressing mode addresses the whole subscriber group, each of which has a unique address. The second addressing mode allows the rights issuer to specify exactly which devices in a subscriber group may access the BCRO. This is done by adding a Eurocrypt style bit access mask to the group address. Each device in the subscriber group has a unique position in that group (determined at registration time). The bit in the bit access mask at this position determines whether the BCRo may be processed by a device.

The third addressing mode addresses a single unique device. This is achieved by appending the device's position in the subscriber group to the subscriber group address.

9.2.2 Subscriber Group Identifier

To identify a subscriber group, a subscriber group subset or a subscriber group unique device, a new identifier type is required. The following schema defines the **roap:SubscriberGroupIdentifier** identifier:

```
<complexType name="SubscriberGroupIdentifier">
  <sequence>
    <element name="subscriberGroupBase" type="base64Binary"/>
    <choice minOccurs="0"/>
      <element name="subscriberAccessMask" type="base64Binary"/>
      <element name="subscriberPosition" type="base64Binary"/>
  </sequence>
</complexType>
```

If the **<subscriberAccessMask>** and the **<subcriberPosition>** element are not included in the **roap:SubscriberGroupIdentifier**, then the content of the **<subscriberGroupBase>** identifies the whole subscriber group. If the **<subscriberAccessMask>** is present, then the **<subscriberGroupBase>** identifies the group, and the mask value identifies which devices in that group are addressed. If the **<subscriberPosition>** is present, then the single device with the corresponding number in the group is addressed.

9.3 Confidentiality of Message Content

9.3.1 Introduction

If the subscriber group addressing is cryptographically secure, then it can be used very effectively to distribute a rights object to such a subset, where the content encryption keys in the rights object are protected with the distinct key associated with that particular subset. All devices in the subset can determine this key, and hence can decrypt the content encryption keys in the rights object. All other devices in the group cannot, and therefore cannot access the protected content.

Refer to **Error! Reference source not found.** for a more detailed introduction to confidentiality in the subscriber group addressing concept.

9.3.2 Subscriber Group Key Material

Each subscriber group has a single unique group key that is used to protect the confidentiality of sensitive broadcast information when the subscriber group is addressed as a whole. This unique group key (*UGK*) is transferred to each device in

the subscriber group upon registration with the rights issuer. The *UGK* is shared between all devices in the same subscriber group.

Each device in a subscriber group also has a unique device key that is used to protect the confidentiality of sensitive broadcast information when device addressing is used (subscriber group address and subscriber position), This unique device key (*UDK*) is transferred to the device upon registration with the rights issuer.

Each device in a subscriber group also has a set of node keys NK_i in case two or more, but not all devices in a subscriber group are addressed by a BCRO, and that can be used to compute all device keys DK_j, except its own device key. In a group of maximum 256 devices, each device gets 8 node keys, whereas in a group of maximum 512 devices, each device gets 9 node keys.

9.3.3 Applying the Subscriber Group Keys

To protect the confidentiality of the key material included in an asset in a given BCRO, that key material is encrypted using a key called *DEK*, and its computation depends on the addressing mode used by the BCRO and the content identifier *BCI* of the first asset encoded in the BCRO.

9.3.3.1 Domain Addressing

In case domain addressing is used by the BCRO, then the *DEK* depends on the domain key K_D of the addressed domain:

$$DEK = \text{HMAC-SHA1-128}\{ K_D \} (BCI)$$

9.3.3.2 Unique Device Addressing

In case unique device addressing is used by the BCRO, then the *DEK* is computed using the unique device key UDK:

$$DEK = \text{HMAC-SHA1-128}\{ UDK \} (BCI)$$

9.3.3.3 Group Addressing

In case the whole group is addressed by the BCRO, then the DEK_{BCI} is computed using the unique group key UGK:

$$DEK = \text{HMAC-SHA1-128}\{ UGK \} (BCI)$$

9.3.3.4 Subset Addressing

If the BCRO is addressed to two or more, but not all devices in a subscription group then the *DEK* is computed using the concatenated node keys associated with the revoked devices as key:

$$DEK = \text{HMAC-SHA1-128}\{ NK_a \,||\, NK_b \,||\, NK_c \,||\ldots||\ \} (BCI)$$

Where NK_a, NK_b, NK_c, … are those node keys associated with devices that must not be allowed to access the asset.

Each node key NK_i is associated with a node number. The nodes from the subscriber group key derivation tree are sequentially numbered in a breadth-first manner, starting from the root node with number 0.

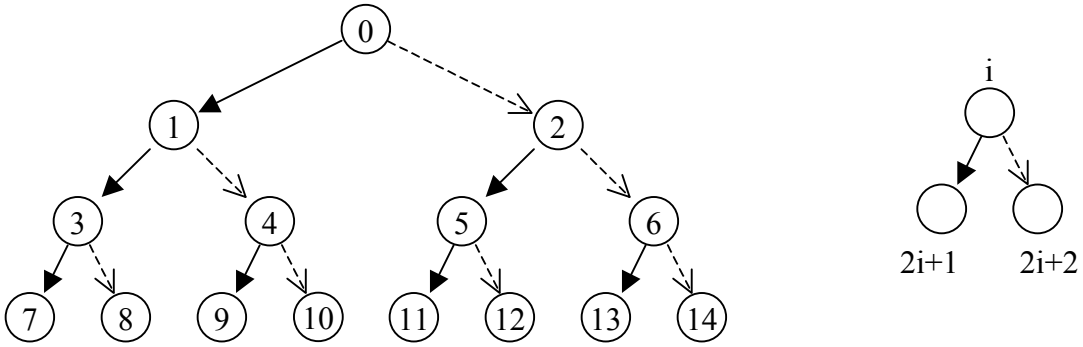


Figure 18 Subscriber Group Node (and Node Key) Numbering

Each device gets a set of node keys such that it can apply the key derivation functions 'left' and 'right' to compute the node keys of all leaf nodes, except of the leaf node that is associated with its own position. The relation between subscriber position and associated leaf node number is:

leaf node number = subscriber position + subscriber group size – 1

Each node in the subscriber group key tree can be associated also with a depth in the tree. The root node has depth 0, its child nodes 1 and 2 have depth 1. In general, the child nodes of a node with depth *d* have depth *d+1*. With this defined, the set of node keys has the following property: all nodes associated with the node keys given to a device have different depth, and the root node is not part of this set.

If NK_i denotes the key associated with node *i*, then the key derivation functions 'left' and 'right' are defined as:

$$NK_{2i+1} := \text{left}(i) := \text{AES-128}\{NK_i\}((2i+\text{LEFT\_CONSTANT}) \bmod 2^{128})$$

$$NK_{2i+2} := \text{right}(i) := \text{AES-128}\{NK_i\}((2i+\text{RIGHT\_CONSTANT}) \bmod 2^{128})$$

Where LEFT_CONSTANT = 0x01010101010101010101010101010101 and RIGHT_CONSTANT = 0x02020202020202020202020202020202.

Example:

The very small subscriber group from Figure 18 consists of 8 devices (numbered 0 to 7, associated with nodes 7 to 14). A Rights Issuer randomly generates a key for the root of the key tree. From that root key, all other keys in the tree are computed using the key derivation functions.

Then:

i	NK_i	*Derivation*
0	0123456789abcdef0123456789abcdef	(not derived, randomly determined by the rights issuer , never send to devices)
1	e50ae5f0c279c65ec332d9bcc1117e92	=AES{0123456789abcdef0123456789abcdef } (01010101010101010101010101010101)
2	1c55d4149103150fc10da6800dd5884a	=AES{0123456789abcdef0123456789abcdef } (02020202020202020202020202020202)
3	4d8249b05af00c67ee7b600927a75eb6	=AES{e50ae5f0c279c65ec332d9bcc1117e92} (01010101010101010101010101010103)
4	a9f5aa423ca8d1efbbcf50014be61b82	=AES{e50ae5f0c279c65ec332d9bcc1117e92} (02020202020202020202020202020204)
5	bad128a946f85174d66ffc326fe5f9e8	=AES{1c55d4149103150fc10da6800dd5884a} (01010101010101010101010101010105)
6	811adb84ab42947df9028444448aa7e4	=AES{1c55d4149103150fc10da6800dd5884a} (02020202020202020202020202020206)
7	**b4bdbf499b8c43e184d270fe198f08df**	=AES{4d8249b05af00c67ee7b600927a75eb6} (01010101010101010101010101010107)
8	**660967ab0c5d5960652b484af71ecba8**	=AES{4d8249b05af00c67ee7b600927a75eb6} (02020202020202020202020202020208)
9	**8e465d379f5cfc324a9c0f3eacb92ee1**	=AES{a9f5aa423ca8d1efbbcf50014be61b82} (01010101010101010101010101010109)
10	**3527bdd7eaccb5c0e6d89a7004d603d8**	=AES{a9f5aa423ca8d1efbbcf50014be61b82} (0202020202020202020202020202020a)

11 **c0d7b5c58b9732b5480dc4c54093c738** =AES{bad128a946f85174d66ffc326fe5f9e8} (01010101010101010101010101010b)

12 **49916d5d931a68ce2e99bf6726098f2e** =AES{bad128a946f85174d66ffc326fe5f9e8} (02020202020202020202020202020c)

13 **3dd317bc38087c3f310c238861958706** =AES{811adb84ab42947df9028444448aa7e4} (01010101010101010101010101010d)

14 **0433bc21b1d5ca5b2b0778475c2ca5ba** =AES{811adb84ab42947df9028444448aa7e4} (02020202020202020202020202020e)

The key NK_0 is the root key from which all other keys are derived. It is randomly selected by the rights issuer and is never distributed to any device. A device that knows NK_0 can compute all device exclusion keys, also its own, and hence circumvent being excluded.

The keys 7 to 14 in bold are the keys associated with the devices 0 to 7 respectively.

To effectively disallow devices 1,6 and 7 to access a certain asset, the rights issuer derives a *DEK* by concatenating the device revocation keys (NK_8, NK_{13} and NK_{14}) , and using this concatenation as key for computing a MAC over the broadcast content identifier *BCI* as retrieved from the rights object:

$$DEK = \text{HMAC-SHA1-128}\{ DK_1 \,||\, DK_6 \,||\, DK_7\}\ (BCI)$$

$$= \text{HMAC-SHA1-128}\{ NK_8 \,||\, NK_{13} \,||\, NK_{14}\}\ (BCI)$$

Device 2 (that is not excluded) has been given the following node keys $\{ NK_{10}, NK_3, NK_2 \}$

$$DK_1 = NK_8$$

$$= \text{right}(NK_3)$$

$$DK_6 = NK_{13}$$

$$= \text{left}(NK_6)$$

$$= \text{left}(\text{right} (NK_2))$$

$$DK_7 = NK_{14}$$

$$= \text{right}(NK_6)$$

$$= \text{right}(\text{right}(NK_2))$$

And note that in computing DK_6 the device already computes NK_6, that is also applied in the computation of DK_7.

An attempt by e.g. device 7 to compute the *DEK* will fail because it will be given the key set $\{ NK_{13}, NK_5, NK_1 \}$, and although that is sufficient to calculate DK_1 and DK_6, it cannot compute its own key.

$$DK_1 = NK_8$$

$$= \text{right}(NK_3)$$

$$= \text{right}(\text{left}(NK_1))$$

$$DK_6 = \underline{NK_{13}}$$

$$DK_7 = NK_{14}$$

$$= \text{right}(NK_6)$$

$$= \text{right}(\text{right}(NK_2))$$

$= \text{right}(\text{right}(\text{right}(NK_0)))$

At that point, there is no more key derivation function available to compute the unknown key NK_0 (which is the root key, and is never distributed to devices!). Because device 7 cannot compute DK_7 it cannot construct the key $DK_1 || DK_6 || DK_7$ that is needed to compute the *DEK* for this BCRO.

9.3.4 Consistency

For any device position, it is easy to derive the node numbers of the key nodes for which the keys must be included in the set of node keys for that device. If $sibling_i$ yields the unique node that has the same parent as node i, $parent_i$ yields the parent node of node i, and NK_i yields the key associated with node i, then the following algorithm yields all the keys to be included in the device's set of derivation keys:

```
KeySet = ∅
while node ≠ root
        node := sibling_node
        KeySet := KeySet ∪ NK_node
        node := parent_node
end
```

With this algorithm it is easy to check the consistency of the key set and the subscriber position given to a device.

10.Broadcast Service Support

10.1 Key Stream Handling

Key stream handling is an ordered sequence of steps that allows refreshing the cryptographic context of the broadcast content transport layer.

The following steps are required to complete this process:

- reception of a key stream message (out of scope of this document)
- linking this key stream message to an appropriate rights object
- using the authentication key from the rights object to authenticate the key stream message
- using the encryption key from the rights object to decrypt the key material in the key stream message
- refresh the cryptographic context of the transport layer using the decrypted key material.

10.1.1 Linking Key Stream Message to Rights Object

To successfully process a key stream message, the terminal MUST find an appropriate rights object that refers to the correct content and holds the appropriate key material. Both normal rights objects (e.g. as delivered via ROAP) as well as broadcast rights object are equally usable in this respect.

A key stream message is linked to a rights object by comparing content identifiers. In a normal rights object, this is the value encoded in the <o-ex:context> element of the <o-ex:asset> elements inside the <o-ex:rights> element in the <ro> element of the <protectedRO> element in the <ROResponse> message. In a broadcast rights object, this is the value of the *BCI* fields in each asset.

The content identifier used is a binary value, which is defined by the key stream layer:

program rights object

content identifier = SHA1-64(*bsdaID* + '#P' + *serviceBaseCID* +'@')+ *program_CID_extension*

service rights object

content identifier = SHA1-64(*bsdaID* + '#S' + *serviceBaseCID* +'@')+ *service_CID_extension*

To process a key stream message, the DRM Agent should be given also the *bsdaID* and the *serviceBaseCID*. These values are defined in the service guide.

In case *program_flag*=1 in the key stream message, the agent would first try to find a rights object with matching content identifiers. The DRM agent will determine if any of the rights objects it has stored governs an asset that has a content identifier equal to:

SHA1-64(*bsdaID* + '#P' + *serviceBaseCID* +'@')+ *program_CID_extension*

Where *program_CID_extension* is found in the key stream message. Alternatively, the agent could be given the whole content identifier in combination with the key stream message to be processed. This requires the agent's environment to compute this content identifier using information from the service guide and the key stream message.

If one or more of such rights objects are found, one is selected among those using the normal OMA procedures. That rights object is now linked to this key stream message.

Otherwise, if service_flag=1 in the key stream message (regardless of P=1 or P=0) then the agent tries to find rights objects with a ContentID equal to:

SHA1-64(*bsdaID* + '#S' + *serviceBaseCID* +'@')+ *service_CID_extension*

If one or more of such rights objects are found, one is selected among those using the normal OMA procedures. That rights object is now linked to this key stream message.

If no suitable rights object is found, then the DRM Agent MUST stop processing this key stream message.

10.1.2 Authentication

Using the suitable and selected rights object, it MUST verify the proper MAC field.

If the rights object is linked to the key stream message using a content id of the form SHA1-64(*bsdaID* + '#P' + *serviceBaseCID* +'@')+ program_*CID_extension* , then it holds a holds a PEK/PAK combination, and the PAK must be used to verify the program_MAC field of the key stream message.

If the rights object is linked to the key stream message using a content id of the form SHA1-64(*bsdaID* + '#S' + *serviceBaseCID* +'@')+ service_*CID_extension,* then it holds a SEK/SAK combination, and the SAK must be used to verify the service_MAC field of the key stream message.

If the verification succeeds it may proceed with decryption of the traffic key material.

When the computed MAC differs from the value encoded in the message, verification fails and the DRM Agent MUST stop processing this key stream message.

10.1.3 Confidentiality

After successful verification, the protected_traffic_key_material field may be decrypted.

There are three possibilities:

1. service_flag=1/program_flag=0

   This is the case in subscriber only access. The content is not available as a pay-per-view item. The protected_traffic_key_material should be decrypted using the SEK in the rights object. Successful verification proves SAK valid, so SEK can be applied.

2. service_flag=0/program_flag=1

   This is the case in pay-per-view only access. The content is not available as a subscription item. The protected_traffic_key_material should be decrypted using the PEK in the rights object. Successful verification proves a valid PAK, so PEK can be applied.

3. service_flag=1/program_flag=1

   This is the case in subscriber access combined with pay-per-view access. The protected_traffic_key_material should be decrypted using the PEK in the rights object, and if a rights object holding the PEK is not available, then an intermediate decryption of the protected_program_key_material is required.

   Based on the selected rights object, two scenarios can be followed.

   If the selected rights object holds a PEK/PAK pair, then PEK can be applied to decrypt the protected_traffic_key_material field.

If the selected rights object holds a SEK/SAK pair, then first the SEK is applied to decrypt the protected_program_key_material field. From the decrypted protected_program_key_material field, the PEK is found. With the PEK now available, the protected_traffic_key_material field is decrypted.

If the DRM Agent encounters any problems during the process of decrypting the traffic key material, it MUST stop processing this key stream message.

10.1.4 Cryptographic Context Update

After successful linking a key stream message to a rights object, verification of the appropriate MAC and decryption of the confidential key material, the cryptographic context of the broadcast content transport layer can be updated.

11.Rights Issuer Services

Rights Issuer Streams are used to carry Registration Layer and Rights Management Layer objects and messages. These include all the messages that are allocated a message tag in A.8.

Within this chapter, the objects and messages to be carried are referred to as “objects”.

The data carried by broadcast systems is logically divided into services. Each Rights Issuer Service consists of one or more IP streams.

A Rights Issuer Stream SHALL be a distinct IP stream within a service.

Rights Issuer Services SHALL carry only Rights Issuer Streams. It is also allowed for other types of service, including media services, to carry RI Streams. The following types of RI Stream are described:

- Ad-hoc RI Stream
- Scheduled RI Stream
- In-Band RI Stream

Additionally, Rights Issuers MAY use Rights Issuer Streams to deliver messages in any way they require. A Rights Issuer Service MAY contain any number of Rights Issuer Streams.

RI Services SHALL be identified as services in the ESG. RI Services SHALL contain only RI Streams.

All RI Streams forming part of any service SHALL be identified as such in the ESG.

An informative schedule MAY be broadcast for RI Services. Where available, this SHALL be provided as part of the ESG. It is used to indicate times at which data for particular sets of devices or Broadcast Groups will be broadcast. This allows devices to listen to RI Services only when necessary, and will also allow Service Operators to make use of spare network capacity when available; for example, at night.

Where a Rights Issuer broadcasts a complete schedule covering all its registered devices, it MAY have any number of Rights Issuer Services. This schedule SHALL indicate, for each device or group of devices, a single Rights Issuer Service which will be used to deliver objects to that set of devices. Otherwise, Rights Issuers SHALL have exactly one Rights Issuer Service. This requirement allows a device to determine exactly one Rights Issuer Service to which it listens.

This specification aims to allow enough flexibility for operators to fulfil their own requirements for message and Rights Object delivery, and to trade off latency against bandwidth, while also allowing devices to minimise power consumption. To support this, there are no restrictions on which messages can be carried in which type of stream, although the expected mode of operation is described in chapter 11.1.

11.1 Expected Mode of Operation [Informative]

It is foreseen that the system will be used in the following way. However, it is noted that considerable variation in actual operation is possible within the scope of this specification, in order to support the needs of Service Operators and Rights Issuers. Any message can be carried in any RI Service, at the discretion of the Service Operator and Rights Issuer.

- Using the ESG, a device can determine which Rights Issuer Service will be used to deliver messages to it. This service will be used to deliver the messages mentioned above.

- An RI Service can contain a number of streams, some of which carry scheduled data while others carry ad-hoc data. When a device is receiving an RI Service, it will receive all the streams within that service.
- A Scheduled RI Stream carries all the messages that an RI wishes to make available.

  -These messages are grouped in time by device or Broadcast Group. A schedule giving the times at which information for particular sets of devices or Broadcast Groups will be carried is made available in the ESG. Devices need only listen to the service at the times relevant to it.

  -It is expected that all BCROs required by any authorised device to receive a protected service will be carried in a carousel format within a Scheduled RI Stream.

  -Future BCROs will also be carried, to prevent breaks in service when services keys are changed.

  -It is also expected that re-registration messages, domain update messages, etc will be carried.

  -RI Certificate Chain updates can be separately scheduled within the ESG. The mechanism specified in this chapter makes it possible for RIs to make these updates available in a scheduled stream alongside other messages, or for a stream to be available which continuously repeats the RI Certificate Chain message.

  -The bandwidth used for individual RI Services can be varied, for example to use any spare capacity that is available at certain times of the day.
- An Ad-hoc RI Stream is used to deliver messages with low latency. In order to receive an Ad-hoc RI Stream, a device will select the relevant RI Service – to do this it could be put into a special mode or have a particular service selected by the user. Examples of messages expected to be carried in an ad-hoc service include:

  -Registration messages, sent directly after a user has registered.

  -BCROs for services that a user has just purchased.

  -Domain control messages.

  -Token delivery messages.
- Additionally, there can also be In-Band RI Streams. These are broadcast as a separate IP stream within, most likely, a media service. These services can be used in whatever way an RI requires, but it is expected that they will carry:

  -BCROs which require immediate delivery, probably to many devices. Examples include BCROs for Free To View services or for free previews.

  -BCROs for content items being delivered within the service.

  -Any message that an RI wishes to make available immediately.

11.2 Scheduled RI Stream

In a Scheduled RI Stream, the timing of message broadcast may be scheduled in some way, according to device or Broadcast Groups.

The schedule describes, for each RI Service, blocks of times at which messages are expected to be available for particular ranges of devices or Broadcast Groups. Where provided, it SHALL be available in the ESG.

Note that although the schedule applies to the whole RI Service, it may be that there are streams within the service that do not follow the schedule – for example, Ad-hoc RI Streams.

It is recommended that a Rights Issuer fulfil the advertised schedule. However, when circumstances require, a Rights Issuer MAY deviate from the schedule that has been broadcast. This MAY cause some devices to miss schedule slots.

A Scheduled RI Service does not have to be available continuously. It could, for example, only be broadcast at night. It is also possible for an RI Service's bandwidth to vary.

11.3 Ad-hoc RI Stream

An Ad-hoc Stream is used to carry messages that a Rights Issuer wishes to be sent spontaneously, i.e. with low latency. It is expected that a device will receive this stream when it is in some special registration mode or when the Rights Issuer Service is specifically selected.

11.4 In-Band RI Streams within a Media Service

Each protected service MAY contain In-Band RI Streams. When receiving a protected service which has associated In-Band RI Streams, a device SHALL listen to the RI Streams for Rights Issuers with which it is registered when receiving the protected service.

It is expected that In-Band RI Streams will contain:

- Messages that need to be delivered immediately to large numbers of devices. Examples include Rights Objects for free previews or free-to-view services; or
- Rights Objects for content being carried by the service.

Devices SHALL be able to identify In-Band RI Streams within protected services from the ESG.

11.5 Broadcast Format of RI Streams

All the objects defined in this specification are carried in Rights Issuer Streams. The format of these streams is defined in this chapter.

These streams SHOULD have the following characteristics:

- The bandwidth overhead of the stream format SHOULD be minimised.
- Objects of varying sizes (smaller than, similar to and larger than the size of an IP packet) SHOULD be efficiently carried.
- Devices SHOULD be able to start interpreting the stream at any packet.
- Where packet reception is unreliable or where packets have been reordered, devices SHOULD be able determine which objects have been correctly and completely received.

Note that it is assumed that the underlying IP stack, and the layers below it, will provide all the necessary error detection, and that IP packets received by the service protection system can be assumed to be as transmitted.

11.5.1 IP Characteristics

Rights Issuer streams are IP streams advertised in the ESG. The ESG SHALL also carry an identifier for the version of this specification used to generate each RI Stream. The format of the IP packets is UDP [RFC 768]. This specification does not specify any limits to the length of these IP packets – this will instead be determined by the underlying network.

Chapter 11.5.1 defines the format of the packets of the RI Stream.

11.5.2 RI Stream Packet Format

The Rights Issuer Stream is made up of RI Stream Packets. There SHALL be at most one RI Stream Packet per UDP packet, and each UDP packet SHALL contain only an RI Stream Packet. The length of the RI Stream Packet is determined by the broadcaster.

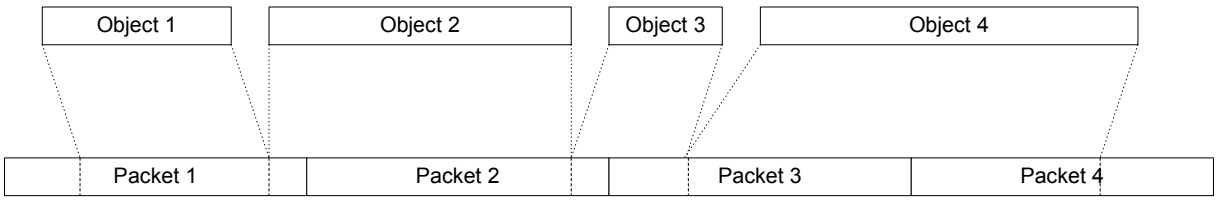


Figure 19: Example mapping of objects to RI Stream Packets

Objects are placed into packets according to the following rules:

- If the length of an object and its RI Stream header is less than or equal to the remaining empty length of a packet, the object is placed in the packet in its entirety and the split_flag is set to zero.
- If the length of an object is greater than the remaining empty length of a packet:
  - The object is allocated an object_id.
  - The number of packets required to carry the object is calculated, including the remaining space in the current packet. The part of an object to be placed in each packet is hereafter referred to as a fragment.
  - The object is split into the appropriate fragments. Note that fragments will be of varying length, for example, if the first fragment of the object begins part way through a packet.
  - While fragments remain to be carried:
    - A header for each fragment is generated, containing the object ID, the number of this fragment within the object and the total number of fragments in the object. The split_flag is set to one.
- Packets SHALL NOT contain any empty space. The end of the last bytes within a packet carrying information SHALL be the end of the packet and the length field of the UDP and IP packet headers will be filled in appropriately. No padding bytes are allowed as part of this protocol.
- The process is repeated with the next object. The size of each packet can be decided by the Rights Issuer, up to the maximum MTU supported by the network.

The format of the packet is as follows.

Table 30: Format of the Rights Issuer Stream

Fields	length	format
`while(bytes left in packet){`		
`  Split flag`	1	bslbf
`  if(split_flag == 1) {`		
`    object id`	7	bslbf
`    fragment_number_within_object`	4	bslbf

total_number_of_fragments_for_object	4	bslbf
if(fragment_number_within_object == total_number_of_fragments_for_object) {		
reserved_for_future_use	4	
remaining_length_in_packet	12	bslbf
}		
bytes_of_object()		
}		
else{		
reserved_for_future_use	3	
length_of_object	12	bslbf
bytes_of_object()		
}		
}		

split_flag: If 1, this object is split over multiple packets. If 0, this object is completely contained in this packet.

object_id: An identifier for this object. All fragments of an object (that are carried in separate packets) have the same object ID. This is only required for objects that are split over multiple packets. For each split object generated, this object ID SHALL be incremented by 1mod(2^7).

fragment_number_within_object: The number of this fragment within the object.

total_number_of_fragments_for_object: The total number of fragments that make up the object.

remaining_length_in_packet: The length of the remaining bytes of the current object in this packet.

length_of_object: The length of this object (which is completely contained in this packet).

bytes_of_object(): The bytes of the object to be carried in this packet.

11.5.3 Implementation Notes

11.5.3.1 Unreliable Delivery

IP networks do not usually offer reliable delivery of packets – this is particularly true of broadcast systems. Devices might not receive all the packets of the RI Stream. Where missing packets cause the device to receive only part of an object, the device SHALL discard this object, although see 11.5.3.2 as apparently missing packets could later be received due to packet reordering.

11.5.3.2 Changes in Packet Order (Informative)

IP packet order can change between the source and destination hosts on some types of IP network. Of course, this cannot happen on a broadcast link, but it could happen within head-end systems or where this service protection scheme is used over other types of link.

At reception time, it is not possible for a device to tell whether an apparently missing packet has been missed due to a reception problem, or whether it will be later received due to some upstream packet reordering. Consider the situation where three packets 1,2,3 are reordered and a device receives them in the order 1,3,2. When the packet processing module receives packet number 3, it will appear as if packet 2 has been missed. However, if the device stores packet 3, and then receives and processes packet 2, it can reconstruct all the objects contained in all three packets. In order to implement this reconstruction scheme, the device buffers partly received objects for some time, and then reconstruct the whole object if the remainder is later received. Incomplete objects are discarded after some period of time. The limit on the use of this technique and the extent of reordering it can cope with is the amount of buffering provided within a device for partly received objects.

The implementation of any scheme of this kind is not required by this specification.

It is recommended that Service Operators and Rights Issuers minimise changes in packet order within their systems.

11.5.3.3 Addressing of Objects

The RI Stream Packet format does not contain addressing information for objects. The format of each object includes addressing information relevant to that object. Devices can determine when an object is addressed to the local device or a group of which the device is a member in the following way:

- The device examines the message tag and the version number of the message to determine what type of message is being broadcast.
- The format of the message contains fields addressing the message to devices in some way. These fields are used to determine whether the local device is being addressed.

11.6 Mapping of Messages to RI Services and Streams

Within a broadcast network, devices discover streams using the ESG and various SI/PSI tables.

- The ESG maps services to IP addresses, allowing a device to discover what services are available, on which IP stream or streams these services are carried and on which IP addresses these streams can be found.
- SI/PSI data describes how a device can receive broadcasts to particular IP addresses, including such information as the PID of the stream carrying the data.

Information about RI Services is carried in the same way as for any other service. RI Services MAY contain any number of IP streams. When receiving a service, a device will receive all the streams that make up that service.

Broadcast systems typically use a number of multicast streams to transmit data to receiving devices. It is not anticipated that devices will be allocated individual IP addresses that will then be used to address streams to single devices.

The following chapters describe how messages are mapped to services and streams.

11.6.1 Rights Issuer Services With Complete Schedule Information

As mentioned above, Rights Issuers MAY provide a schedule for the broadcast of messages to sets of devices. If a Rights Issuer broadcasts a complete schedule of messages to be sent to all devices (excluding ad-hoc streams), that Rights Issuer MAY have any number of RI Services, containing any number of RI Streams.

For each service, the Rights Issuer SHALL broadcast, within the ESG, one or more schedule items containing a list of devices for which messages may be broadcast on that service, and the times at which those messages will be broadcast. Any device registered with the Rights Issuer SHALL be able to locate a single RI Service to listen to at any one time.

11.6.2 Rights Issuer Services Without Complete Schedule Information

If a Right Issuer broadcasts either no schedule information or incomplete schedule information, that Rights Issuer SHALL broadcast only one Rights Issuer Service.

Devices for which schedule information is broadcast SHOULD listen at the appropriate times. Devices for which schedule information is not broadcast SHOULD listen as often to practical, but no requirements are placed on their behaviour.

11.7 Discovery of RI Services, Streams and Schedule Information

The ESG carries information about RI Services, Streams and their associated schedule information.

The ESG requirements are:

- The capability to describe a service as an RI Service, and have an associated Rights Issuer ID.
- The capability to describe a stream within any service as an RI Stream, and, when the stream is not part of an RI Service, have an associated Rights Issuer ID.
- The capability to describe the version of the specification used to construct an RI Stream.
- The capability to include multiple RI schedule blocks, which may be placed within ESG context or also associated with a purchase channel.
- The capability to indicate that a Certificate Chain update will be included in a particular schedule slot.

For Scheduled RI Services, a number of particular "content items" are announced, corresponding to device, group or domain ranges (or no range, corresponding to all devices). One of the existing attributes of the content item is for this purpose defined to have a particular structure and semantics.

The broadcast times of these particular content items will be announced within schedule items, and thereby a device will be able to determine when it has to receive the RI service.

11.8 Certificate Chain Updates

It is important that devices can acquire Certificate Chain updates, which may include an OCSP response, as quickly as possible. A device will not be able to decode services until it has a current certificate chain (although a grace mechanism is defined in [Please refer to the corresponding chapter introduced by CR 223 OMA-DLDRM-2005-0223-OCSP_grace_broadcast, once it is agreed]A.1.2 to make this more user-friendly). The following requirements are made on the broadcast of Certificate Chain updates.

- It is strongly recommended that schedule information for certificate chain updates is made available in the ESG. When such schedule information is carried, devices SHOULD listen to the relevant RI Services when they need to acquire updates. No firm requirement on device behaviour can be made, as a device may not be able to receive a service at a particular time, for example because it has a low battery or is out of range of the broadcast network.
- Furthermore, it is strongly recommended that a reference to at least the next certificate chain update is always carried in the ESG.
- Where such schedule information is not carried, certificate chain updates SHALL be carried, at least, in the RI Service belonging to the relevant Rights Issuer.

Using the mechanisms described in this chapter, two possible schemes for the broadcast of Certificate Chain updates are informatively described below.

- Certificate Chain updates can be broadcast continuously in an RI Stream. A schedule block indicating a certificate chain update, with no device range limit and a time limit of, say, midnight to midnight, is broadcast for this stream, indicating that Certificate Chain updates can always be found on this stream.

- Certificate Chain updates are broadcast periodically on an RI Stream. Schedule blocks indicating a certificate chain update, with no device range limit and the time limit for when the updates will be broadcast, is broadcast for this stream.

11.9 Resending of BCROs

There is no guarantee that a device will receive the BCROs sent to it via the broadcast channel. A device may request that the BCROs be sent once again by the Rights Issuer.

11.9.1 Resending of BCROs to Interactive Devices

For an interactive device, requests to resend BCROs can be made via the interactivity channel. If the BCROs are to be delivered via the broadcast channel, the device will listen to the relevant Rights Issuer Service after sending the request. It is recommended that devices listen to this channel for at least one hour, or until the BCROs are received. It is expected that the BCROs will be delivered in an Ad-hoc RI Stream.

When a Rights Issuer receives a request from an interactive device to resend BCROs over the broadcast channel, it MAY resend the BCROs for that device.

11.9.2 Resending of BCROs to Broadcast Devices

Rights Issuers may allow users of broadcast devices to request that BCROs for that device are resent. If the Rights Issuer does allow this, the device may prompt the user to make such a request, as specified in **Error! Reference source not found.**. The device SHOULD then listen to the relevant Rights Issuer Service, possibly after the user has acknowledged that the request has been made. It is recommended that devices listen to this channel for at least one hour, or until the BCROs are received. It is expected that the BCROs will be delivered in an Ad-hoc RI Stream.

No firm requirement on device behaviour can be made, as a device may not be able to receive a service at a particular time, for example because it has a low battery or is out of range of the broadcast network.

When the Rights Issuer receives such a request, it MAY resend the BCROs for that user.

11.10 Summary of Requirements for Rights Issuers

If a Rights Issuer delivers messages to devices via the broadcast channel, it SHALL use Rights Issuer Services and Streams to do so and SHALL meet the requirements below. If a Rights Issuer does not deliver messages via the broadcast channel, it will not have Rights Issuer Services and Streams, and the remainder of this chapter does not apply.

- Each Rights Issuer SHALL either:
    - Provide a complete schedule for their Rights Issues services, covering all registered devices and allowed any registered device to identify one RI Service to listen to; or
    - Have exactly one Rights Issuer Service.
- Each Rights Issuer Service:
    - MAY contain any number of Scheduled or Ad-hoc RI Streams.
    - SHALL contain only Rights Issuer Streams.

- Rights Issuers SHOULD provide an informative schedule for the broadcast of messages in their RI Service, unless the system is being used in an environment where power consumption of devices is not an issue (as the scheduling of RI Services is primarily intended as a power-saving feature for devices).
- Any other type of service MAY carry, at most, one In-Band Rights Issuer Stream per Rights Issuer.
- Rights Issuers SHOULD broadcast both the current and next Rights Objects required to receive services, to reduce the likelihood of a device not having the Rights Object required to receive a service which it is entitled to receive.

11.11 Summary of Requirements for Devices

The following is a summary of the requirements relating to RI Services for devices which support the Broadcast mode of operation. Note that none of these requirements apply to devices which only use the interactivity channel to communicate with Rights Issuers.

- For each Rights Issuer with which the device is registered, a device SHALL listen to the associated Rights Issuer Service, subject to the following:

  -Where a schedule for the RI Service is available, devices MAY receive that schedule and MAY listen to the RI Service only at the relevant times.

  - Devices that make use of the schedule SHOULD check for new schedule data at least once per day.

  -Otherwise, when a schedule for the RI Service is not available or a device does not listen to it:

  - Mains powered devices or devices under charge SHOULD listen to that service continuously.
  - Battery powered devices SHOULD listen to that service at least when the device is powered on for some purpose.

- When receiving a Rights Issuer Service, devices SHALL listen to all streams within that service.
- It SHALL be possible to put a device into a mode in which it receives the RI Service of a particular Rights Issuer, for some period, in order to receive, for example, registration data, domain messages and recently purchased Rights Objects. These are expected to be delivered in Ad-hoc RI Streams. This does not apply in the case that a device continuously receives Rights Issuer Services.
- When a device is receiving a service containing an In-Band RI Service for a Rights Issuer with which it is registered, the device SHALL listen to that service.

12.PDCF adaptation for Traffic Encryption Key stream

This section allows a Traffic Encryption Key (TEK) stream (transmitted using Layer 3 of the 4-layer model for Service Protection and Content Protection of RTP streams) to be stored within a PDCF.

The PDCF file format as defined in OMA DRM v2.0 [DRMCF-v2.0] allows audio video content to be stored in a file format together with the relevant OMA DRM information. Audio and video tracks can be encrypted as defined in [DRMCF-v2.0] using the appropriate CEK stored in a Rights Object (RO).

In the context of broadcast services, RTP streams can be encrypted at the content level (encrypting Access Units as explained in [DRMCF-v2.0]) using TEKs transmitted using Layer 3 as shown in Figure 9.1. This key is not the traditional CEK stored in a RO. In the broadcast context the CEK is a Service Encryption Key (SEK) or a Program Encryption Key (PEK) delivered using Layer 2 (delivered in a broadcast RO or via a return path). This SEK or PEK allows the TEK delivered in Traffic Encryption Key stream messages delivered in Layer 3 to be decrypted. The TEK is used to encrypt content transmitted in RTP packets. As this key changes regularly, this section explains how the PDCF file format can be adapted to include storage of the relevant TEK stream information.

12.1 Overall PDCF structure

The table below outlines the mandatory and optional ISO boxes and their order. Additional boxes MAY be added after the mandatory boxes have first appeared. Table 31~~Table 31~~~~Table 31~~ shows the nesting order of the mandatory boxes, on the left is the parent and on the right, the child. The first column indicates which fields and boxes MUST be present in PDCF and which boxes MAY appear in the PDCF. The following syntax is adopted:

M	ISO mandatory boxes
MO	mandatory OMA boxes
O	optional boxes

The table 9.1 below includes all boxes defined by ISO in [ISO14496-12] when OMA information is specified per track. The file format structure corresponds to OMA DRM v2.0 [DRMCF-v2.0] and is not modified in this specification. It is fully ISO compliant.

Table 31: Logical PDCF box structure diagram for single protected track

Present in PDCF	Data type/value										Field purpose
M	‘ftyp’										ISO File header (fixed File Type box)
M	‘moov’										ISO movie box
M		‘mvhd’									ISO movie header box
M		‘trak’									ISO track box
M			‘tkhd’								ISO track header
M			‘tref’								ISO track reference
M			‘mdia’								ISO media information box
M				‘mdhd’							ISO media header
M				‘hdlr’							ISO handler

M				‘minf’							ISO media information container
M					‘stbl’						ISO sample table box, container for the time/space map
M						‘stsd’					ISO sample descriptions
M							‘encv or 'enca'				ISO protected sample entry
M								‘sinf’			ISO protection scheme information box (always present)
M									‘frma’		ISO original format (always present)
M									‘schm’		ISO SchemeTypeBox (when used to apply to single track)
M									'schi'		ISO SchemeInformationBox (if applies to this ‘trak’ only)
MO										‘ohdr’	OMA DRM Common Headers box (when used to apply to single track)
MO										'osfm'	OMA Sample format Box (when used to apply to single track)
O		‘odrb’									OMA Rights Object container box

Table 32Table 32Table 32 below shows the PDCF file format when the OMA information is specified at the movie box level, applying to all tracks. This implementation, however, is not ISO compliant.

Table 32: Logical PDCF box structure diagram with all tracks protected

Present in PDCF	Data type/value										Field purpose
M	‘ftyp’										File header (fixed File Type box)
M	‘moov’										ISO movie box
M		‘mvhd’									ISO movie header box
M		‘trak’									ISO track box
M			‘tkhd’								ISO track header
M			‘tref’								ISO track reference
M			‘mdia’								ISO media information box
M				‘mdhd’							ISO media header
M				‘hdlr’							ISO handler
M				‘minf’							ISO media information container
M					‘stbl’						ISO sample table box, container for the time/space map
M						‘stsd’					ISO sample descriptions

M							‘encv or 'enca'				ISO protected sample entry
M								‘sinf’			ISO protection scheme information box (always present)
M									‘frma’		ISO original format (always present)
O		‘sinf’									ISO protection scheme information box (when used to apply to all tracks)
O			‘schm’								ISO SchemeTypeBox (applies to all ‘trak’s)
O			'schi'								ISO SchemeInformationBox (when used to apply to all tracks)
O				‘ohdr’							OMA DRM Common Headers box (when used to apply to all tracks)
O				‘osfm’							OMA SampleFormat Box (when used to apply to all tracks)
O		‘odrb’									OMA Rights Object container box

1

2

Table 33Table 33Table 33 below shows the PDCF file format when on OMA key track is defined, rather than an audio or video track. This format is ISO compliant.

Table 33: Logical PDCF box structure diagram showing OMA key track

Present in PDCF	Data type/value										Field purpose
M	'ftyp'										ISO File header (fixed File Type box)
M	'moov'										ISO movie box
M		'mvhd'									ISO movie header box
M		'trak'									ISO track box
M			'tkhd'								ISO track header
M			'tref'								ISO track reference
M			'mdia'								ISO media information box
M				'mdhd'							ISO media header
M				'hdlr'							ISO handler
M				'minf'							ISO media information container
M					'stbl'						ISO sample table box, container for the time/space map
M						'stsd'					ISO sample descriptions 'okey' for OMA key track
MO							'oksd'				OMA key sample description box

12.2 PDCF adaptation for key stream inclusion

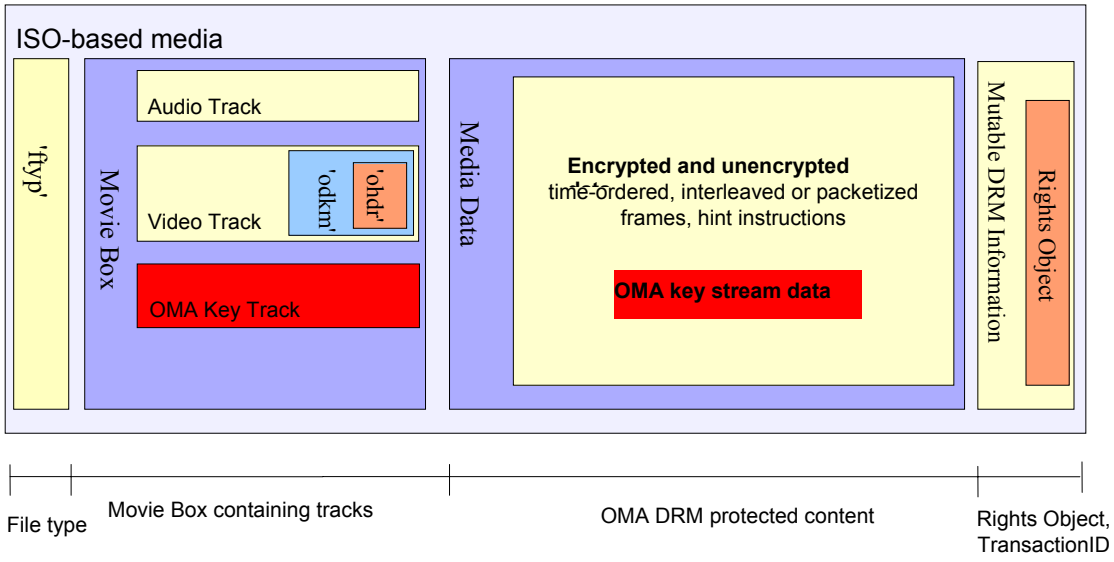
This section details the modifications required in the PDCF file format of OMA DRM v2.0 [DRMCF-v2.0] so as to allow an OMA key stream to be stored in the PDCF.

The adapted PDCF file format is schematically shown in Figure 20Figure 20Figure 20 below in a simplified format, as per OMA DRM v2.0. The only difference between the diagram below and the original PDCF file format is the addition of an OMA key track in the Movie Box and the associated OMA key track data in the Media Data box, as shown in red. Full backward compatibility with the original PDCF file format is thus ensured.

Details on the PDCF file format, key track and details on how to link the key track to appropriate audio / video tracks are given in this specification in the sections below.

Supporting the adapted PDCF format defined in this specification is OPTIONAL for a Device, as is the case for the original PDCF format in OMA DRM v2.0.

Figure 20: Example of adapted PDCF Structure



12.2.1 Movie Box and Tracks

The ISO Movie Box (‘moov’) contains one or more audio or video tracks as defined in the ISO specification [ISO14496-12] and can additionally contain one or more OMA key tracks as defined in this specification.

12.2.1.1 HandlerBox

The ISO HandlerBox (‘hdlr’) within a Media Box declares the process by which the media-data in the track is presented, and thus, the nature of the media in a track. For example, a video track would be handled by a video handler.

The ISO definition is shown below.

```
aligned(8) class HandlerBox extends FullBox('hdlr', version = 0, 0) {
unsigned int(32) pre_defined = 0;
```

```
unsigned int(32) handler_type;
const unsigned int(32)[3] reserved = 0;
string name;
}
```

The associated parameter definitions are shown below, with the addition of a new OMA key track type:

Field name	Type	Purpose
pre_defined	unsigned int (32)	Set to 0
handler_type	unsigned int (32)	Indicates track format of meta box contents: 'vide' = video track 'soun' = audio track 'hint'= hint track 'okey' = OMA key track
reserved	Const unsigned int (32)	Reserved = 0
name	String	Null-terminated string in UTF-8 characters which gives a human readable name for the track

For implementations not recognising the new OMA key track, it will be ignored.

12.2.1.2 SampleDescriptionBox

A new OMA key sample entry (‘okey’) contained in the ISO SampleDescriptionBox (‘stsd’) is used to describe some initialization information for decoding the key track that would be more convenient to put at this level than in the ProtectionSchemInfoBox of the encrypted media track, as defined below:

```
aligned(8) class SampleDescriptionBox (unsigned int(32) handler_type)
      extends FullBox('stsd', 0, 0){
      int i ;
      unsigned int(32) entry_count;
      for (i = 1 ; i • entry_count ; i++){ entry_count ; i++){
             switch (handler_type){
                    case ‘soun’: // for audio tracks
                           AudioSampleEntry();
                    break;
                    case ‘vide’: // for video tracks
                           VisualSampleEntry();
                    break;
                    case ‘hint’: // Hint track
                           HintSampleEntry();
                    break;
                    case ‘okey’: // OMA key track
                           OMAKeySampleEntry();
                    break;
             }
      }
```

```
}
```

A new handler_type, called 'okey' identifies the new OMA key track defined in this specification.

12.2.1.3 OMAKeySampleDescriptionEntry

The new OMA key track is defined by OMAKeySampleDescriptionEntry box as follows:

```
aligned(8) class OMAKeySampleDescriptionEntry extends SampleEntry('oksd') {
unsigned int(8) sample_version;            // sample version
}
```

The OMAKeySampleDescriptionEntry field is defined as follows:

Table 34: OMAKeySampleEntry fields

Field name	Type	Purpose
sample_version	Unsigned int (8)	Identifies OMA key sample version Version = 0x00 for the key track defined in this specification

There is only one OMAKeySampleDescriptionEntry box per key track.

12.2.1.4 Track referencing

The presence of a key track in the PDCF file is insufficient as a link is required between the OMA key track and the audio / video track(s) it applies to. Each declared track is identified by a track_ID, as defined by ISO. Following ISO convention, the key track refers to one or more tracks via their track_IDs.

The ISO Track Reference Box ('tref') is placed inside the track box to indicate references to one or more tracks:

```
aligned(8) class TrackReferenceBox extends Box('tref') {
}
aligned(8) class TrackReferenceTypeBox (unsigned int(32) reference_type) extends
Box(reference_type) {
unsigned int(32) track_IDs[];
}
```

The Track Reference Box of an OMA key track contains track reference type boxes.

Parameters are as defined below:

Table 35: Track Reference Box fields

Field name	Type	Purpose
reference_type	unsigned int(32)	Reference box type identifier 'hint' = the referenced track(s) contain the original media for this hint track 'cdsc = this track describes the referenced

		track 'okey' = OMA DRM key track
track_ID[]	unsigned int(32)	integer that provides a reference from the containing track to another track in the presentation. track_IDs are never re-used and cannot be equal to zero

If the key track applies to more than one audio or video tracks, then the appropriate track_IDs are placed in the reference_type box.

12.2.2 OMA DRM information boxes

12.2.2.1 Sample description transform

In encryption, the samples are transformed – encrypted – so that the underlying media cannot be accessed by readers without the appropriate information (e.g. keys). The format of the encrypted samples is "owned" and documented by the encryption system.

The purpose of the sample description transformation is twofold: The sample description prevents accidental treatment of encrypted data as if it were un-encrypted and documents the transforms applied. The documentation of the encryption scheme and its parameters is supplied in a uniform way. Note that in the following definitions that "n" in bit(n), unsigned int(n) and int(n) is always a bit count.

The transformation of the sample description is described entirely by the following procedure:

1. The 4CC of the sample description is replaced with a 4CC indicating the encryption: e.g. ‘mp4v’ or 's263' are replaced with ‘encv’ for encrypted video and e.g. ‘mp4a’ is replaced with ‘enca’ for encrypted audio.

2. A ProtectionInfoBox (defined below) is appended to the sample description, leaving all other boxes unmodified.

12.2.2.2 Protection Scheme Information

The ISO `ProtectionSchemeInfoBox` ‘sinf’ is used to carry DRM key management system specific information, thus it is only a container box.

It contains the `ISO SchemeTypebox` (‘schm’) adapted for OMA as defined in 12.2.2.3 below, the ISO `SchemeInformationBox` 'schi' as defined in 12.2.2.4 below and the ISO `OriginalFormatBox` 'frma' as defined here.

The ISO Protection Info Box contains all the information required both to understand the encryption transform applied and its parameters, and also to find other information such as the kind and location of the key management system. It also documents the original (unencrypted) format of the media. The Protection Info Box is a container Box.

```
aligned(8) class ProtectionInfoBox(fmt) extends Box('sinf') {
OriginalFormatBox(fmt) original_format;
SchemeTypeBox scheme_type;
SchemeInformationBox info;
}
```

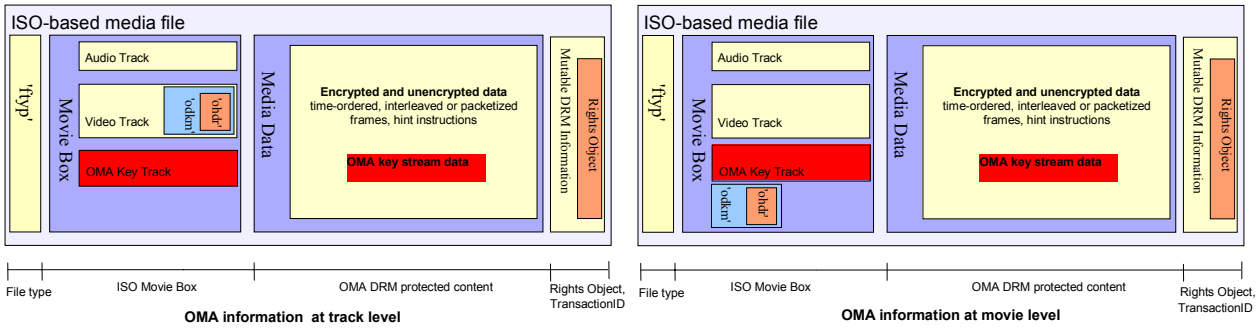
When used in a protected sample entry, the 'sinf' box' must contain the ISO Original Format Box ‘frma’ which holds the 4CC of the unencrypted sample description:

```
aligned(8) class OriginalFormatBox(codingname) extends Box ('frma') {
unsigned int(32) data_format = codingname;
// format of decrypted, encoded data
// could be 'mp4v', 'h263', 'avc1', 'mp4a', etc.
}
```

There MAY be several instances of the `ISO Protection Scheme Information Box` in a PDCF file, and one can appear either at the movie level or exactly one per each protected track. There MUST NOT be ISO Protection Scheme Information boxes containing OMA DRM boxes in both movie level and track level.

The two alternative solutions are shown below:

Figure 21: Possible ProtectionSchemeInfoBox positions within PDCF



If not all tracks are protected by OMA DRM, then the ProtectionSchemeInfoBox is placed with each protected track.

If all tracks are protected using OMA DRM, then the ProtectionSchemeInfoBox can be placed within the Movie Box, rather than within each protected track. Note that this means the file format is NOT ISO compliant. Note that in this case each protected track MUST still contain an ISO ProtectionSchemeInfoBox ‘sinf’ together with an ISO Original Format box ‘frma’, as shown in Table 32~~Table 32~~~~Table 32~~. Note: this is copied from the existing PDCF specification but is NOT ISO compliant, we should consider correcting this and preventing such a use. Other sections in this specification would be corrected accordingly.

Note: as the newly defined OMA key track is part of this specification, it must be interpreted as such if the ProtectionSchemeInfoBox is within the MovieBox, covering all tracks. In this situation, OMA key track samples SHALL NOT be interpreted in the same manner as audio or video tracks.

This box is exactly the same as defined in [DRMCF-v2.0] and is not modified in this specification.

12.2.2.3 DRM Scheme Type

The ISO `SchemeTypeBox` (‘schm’) includes information on which DRM system is being used to manage keys and decryption of the content. As the media file format MAY support also other key management systems than OMA DRM, the key management system in use is indicated by a 4CC in the *SchemeType* field.

Table 36: PDCF Scheme Type for OMA DRM

Scheme_type	Value	Semantics

OMA DRM	'odkm'	OMA DRM is used for key management in the PDCF.

Table 37: PDCF Scheme Version for OMA DRM

Scheme_version	Value	Semantics
TBD	TBD	OMA DRM version is TBD (version 2.0 does not allow the key track)

For PDCF files conforming to this specification, the *SchemeType* MUST be the 4CC 'odkm', and *SchemeVersion* MUST be TBD (version TBD). If OMA DRM key management scheme 'odkm' is indicated, then the file is a PDCF and MUST contain at least one `OMADRMKMSBox`. A PDCF MUST support only OMA DRM for the key management system.

12.2.2.4 DRM Scheme Information

The ISO `ProtectionSchemeInfoBox` is used to carry DRM key management system specific information, thus it is only a container box. For OMA DRM, this box MUST include exactly one OMADRMCommonHeaders box 'ohdr' (see [DRMCF-v2.0]), as the first sub-box and exactly one `OMASampleFormatBox`, as the second sub-box.

```
aligned(8) class ProtectionSchemeInfoBox extends Box('schi') {
Box scheme_specific_data[];
}
```

```
aligned(8) class OMADRMCommonHeaders extends FullBox('ohdr', version, 0) {
      unsigned int(8)       EncryptionMethod;      // Encryption method
      unsigned int(8)       EncryptionPadding;     // Padding type
      unsigned int(64)      PlaintextLength;       // Plaintext content length in bytes
      unsigned int(16)      ContentIDLength;       // Length of ContentID field in bytes
      unsigned int(16)      RightsIssuerURLLength; // Rights Issuer URL field length in bytes
      unsigned int(16)      TextualHeadersLength;  // Length of the TextualHeaders array in bytes
      char                  ContentID[];           // Content ID string
      char                  RightsIssuerURL[];     // Rights Issuer URL string
      string                TextualHeaders[];      // Additional headers as Name:Value pairs
      Box                   ExtendedHeaders[];     // Extended headers boxes
}
```

```
aligned(8) class OMASampleFormatBox extends FullBox('osfm', 0, 0) {
bit(1) SelectiveEncryption;
bit(7) reserved;
unsigned int(8) KeyIndicatorLength;
unsigned int(8) IVLength;
}
```

Parameters are as defined below:

Table 38: OMA Sample Format Box fields

Field name	Type	Purpose
SelectiveEncryption	Bit(1)	Indicate whether selective encryption is used or not

Reserved	Bit(7)	Reserved, SHOULD be set to 0.
KeyIndicatorLength	Unsigned int(8)	Size of the key indicator in bytes
IVLength	unsigned int(8)	Size of the IV in bytes

If the selective encryption bit is set to 0 then all content to which the ISO ProtectionSchemeInformationBox applies is encrypted and no "encrypted" field is present in OMADRMAUHeader.

If the selective encryption bit is set to 1 then the OMADRMAUHeader preceding Access Units indicates whether or not a particular AU is encrypted.

12.2.2.5 OMADRMAUHeader

This header, which MUST precede the codec-specific sample data in each Access Unit, provides the OMA DRM information whose length is specified in the OMA Sample Format box defined above (Table 38~~Table 38~~~~Table 38~~). The OMA DRM AU Header is defined as follows:

```
aligned(8) class OMADRMAUHeader {
      if (SelectiveEncryption == 1) {// from the OMASampleFormatBox
            bit(1)  sample_is_encrypted;          // Encryption indicator
            bit(7)  reserved;                     // Must be zero
      }
      else sample_is_encrypted = 1;
      if (sample_is_encrypted==1) {
            unsigned int(8 * KeyIndicatorLength) KeyIndicator;
            unsigned int(8 * IVLength) IV;
      }
      unsigned int(8) data[]; // encrypted media data, to end of sample
}
```

Table 39: OMA DRM AH Header fields

Field name	Type	Purpose
sample_is_encrypted	bit(1)	Encryption Indicator for the access unit.
KeyIndicator	unsigned int(8)	Key indicator field preceding the access unit payload.
IV	unsigned int(8)	IV preceding the access unit payload.

Table 40: Encryption Indicator values

Encrypted	Value	Semantics
None	0	Access unit is not encrypted.
Encrypted	1	Access unit is encrypted.

When encrypting PDCF Content, the `OMADRMAUHeader` information MUST be added to the processed access unit, even if the EncryptionMethod field in the OMACommonHeaders box is set to NULL.

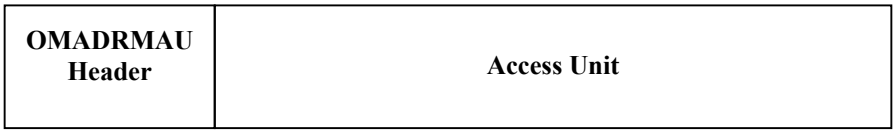
Note: this is copied from the existing PDCF specification but is not actually needed, we should consider correcting the above sentence to say that if the EncryptionMethod is NULL, OMADRMAUHeaders MUST not be used as they are not required.

When encrypting PDCF Content, the OMADRMAUHeader information MUST be added to the processed access unit, even if the *selective_encryption* field in the OMASampleFormat box is set to 0.

A playing Device uses the header information for decryption purposes and is able to extract the actual sample(s).

Figure 22Figure 22Figure 22 shows how the OMADRMAUHeader is placed before each AU.

Figure 22: OMADRMAUHeader and Access Unit



12.3 Traffic Encryption Key stream storage format

The new OMA key track defined in [12.2.1.2] is described by the sample description information defined in [12.2.1.3]. In order to provide maximum flexibility, this information merely declares the key track version and size only. This section defines the actual OMA key sample format stored in the Media Data box containing OMA key track samples.

As needs evolve, new sample formats can be defined as this specification evolves, identifying new formats with new key track sample version numbers. This approach ensures future PDCF specifications will remain fully backward compatible.

```
aligned(8) class OMAKeySample  {
unsigned int(8) KeyIndicator;              // key identifier
unsigned int(8) Key;                       // encrypted key
Further information To be Defined
}
```

Further information To Be Defined in further CRs.

12.4 Recording RTP streams

This section explains how RTP streams can be recorded in a PDCF file.

12.4.1 Content encrypted by a single CEK

The existing PDCF streamable file format can be streamed as defined in [DRMCF-v2.0]. In this case the content is encrypted using a single CEK delivered via a RO. The recording process consists in storing Access Units in the PDCF.

Depending on the RTP transport format, multiple or fragmented AUs may be present in a single RTP packet. This will be taken into account to ensure complete AUs are stored together with the appropriate OMADRMAUHeader.

12.4.2 Content encrypted by a TEK stream

OMA Broadcast extensions provide dynamic keying that can be applied to encryption of content transmitted in RTP streams (content protection of RTP streams). The adapted PDCF file format described in this specification provides the means to record encrypted AUs. The associated TEK key stream can be recorded in the appropriate key track.

For broadcast purposes TEK messages may be repeated several times during the same cryptoperiod (time interval during which the same key is used). These messages are therefore redundant and MAY be ignored. However, if these messages are different in any other way (i.e. due to other information they contain, even though the keys are the same), they SHALL be stored in the PDCF key track.

12.4.3 Change of Rights and Recommendations for Recording

The following rules SHALL be observed when recording streamed content in a PDCF:

1. If the user has a valid Rights and the end of a program / event is reached, a new track MAY be created for the new program / event. Alternatively, a new file MAY be created for the new program / event, rather than using the same file.

2. If the user has a valid Service Rights and PEKs are used to protect TEKs, then new tracks or files MAY be created when PEKs change, rather than using the same track.

3. If a program / event is being recorded for which the user has the appropriate Rights and a new program / event starts for which the user has NO valid Rights, a new track or a new file SHOULD be created, rather than using the same track.

4. If a program / event is being recorded for which the user has no Rights, a new track or file MAY be created for a new program / event, rather than using the same track, if the user still has no valid Rights for the new program / event.

   If the user has valid Rights for the new program / event, a new track or file SHOULD be created, rather than using the same track.

5. In all cases, if a different RO is required, a different track or file SHALL be used.

Appendix A.

A.1 Security Considerations

A.1.1 Handling weak keys

(The responsible components in) the head-end architecture SHALL NOT use weak keys that will be used for messages in the IP Datacast network. At the time of this writing there are no specified weak keys for use in AES. This does not mean to imply that weak keys do not exist. If, at some point, a set of weak keys for AES are identified, the use of these weak keys SHALL be rejected in the head-end architecture followed by a request for replacement key.

A.1.2 Handling OCSP grace period

If a device without a return channel inspects a certificate, because the user wants to consume certain content for which he has acquired the RO, and the device finds out that the OCSP response of the certificate chain has expired, then the device is still allowed to use it for a short period of time during which the user has time to set the process in motion through which the device will receive a new OCSP response. This means that the user can enjoy the content he was entitled to consume straight away, at the expense of a slightly increased security risk of being able to use possibly compromised certificates for a somewhat longer time.

A device in broadcast-only mode SHALL implement the grace period mechanism.

1) The device checks periodically a particular or all RI context for expiration.

2) If a RI context is expired, the device displays an OCSP response expiry reminder for the associated RI context. The reminder notifies the user that the user needs to get a new OCSP response (of course in terms that a user can understand like “Call this number with this message please”)

3) Until this OCSPresponse expiry reminder is invoked the device will be rendered inoperable, but only in relation with the associated RI (context) as described below:

    a. Accessing an ESG for purchase is still allowed.

    b. The device SHALL be rendered inoperable for any purchase protocol or playback of future content. The device MAY use stored BCROs to play old content for which the device obtained ROs, but SHALL NOT use these BCROs for new content received after the re-registration request until the device received a fresh OCSP response or is re-registered with the RI.

4) A device SHALL be allowed to use an expired OCSP response for a pre-defined grace period. The grace period SHALL NOT be more than the OCSP response’s lifetime (the difference between the nextUpdate and thisUpdate fields in the OCSP response), and MUST NOT exceed 48 hours.. During the grace period, the device can use the expired OCSP response.

    a. The grace period is for a one-time use only.

    b. The terminal SHALL support secure DRM time.

    c. Rules in ROs SHALL have precedence over the OCSP response grace period usage.

5) If the secure timer (i.e. grace period) expires and a fresh OCSP response has not been received, the device will be rendered inoperable, but only in relation with the associated RI (context) as described below:

    a. Accessing an ESG for purchase is still allowed.

    b. The device SHALL be rendered inoperable for any purchase protocol or playback of future content. The device MAY use stored BCROs to play old content for which the device obtained ROs, but SHALL NOT use these BCROs for new content received after the re-registration request until the device received a fresh certificate chain or is re-registered with the RI.

A device in broadcast mode MAY implement a mechanism to automatically schedule the certificate chain updates.

1) An update (powerup/powerdown) timeslot is programmed in which the RI will transmit the certificate chain. The timeslot may be obtained from the ESG. The device SHOULD parse the received ESG data to find a time at which it can receive a certificate chain update. Note that it may be the case that certificate chain updates are broadcast continuously. See section ~~(Dear reader: refer to CR2005-0219, Note to Editor: refer to appropriate XBS section)~~11.8 for more details.

2) Upon power down before update the device may display a warning message that the device needs to update it's device chain. An example might look like: "Do not power off device. Device will perform update during xx:yy h".

The device will be powered up and down in timeslot xx:yy h to pick up the message to update the RI certificate chain (notably the OCSP response).

A.2 Checksum algorithms

According to empirical research by [VERHOEF_1969] the likelihood of errors is expressed as:

nr	error	representation	relative likelihood in %
1	single substitution	a => b	60 to 95
2	single adjacent transpositions	ab => ba	10 to 20
3	twin errors	aa => bb	0,5 to 1,5
4	jump transpositions (Longer jumps are even rarer)	acb => bca	0,5 to 1,5
5	phonetic errors (phonetic, because in some languages the two have similar pronunciation, e.g., thirty and thirteen)	a0 => 1a where a={2,..,9}	0,5 to 1,5
6	adding or omitting digits		10 to 20

Key:

a < > b, while c can be any decimal digit.

The most common errors are therefore errors 1, 2 and 6. Error 6 is easily detected. Following sections explain a method to detect other errors.

A.2.1 Checksum on SDN

Definition:

The checksum on the ARC is calculated by F-*SDN*

Take n=12, r=2 and p=11. We consider the code defined by the r=2 following check equations:

```
8*c1 + 8*c2 + 6*c3 +...+ 1*c11 = 0 (modulo 11)
3*c1 + 6*c2 + 4*c3 +...+ 1*c12 = 0 (modulo 11)
```

In other words, a string ($c1$,$c2$,...,$c12$) with elements from $Z_{11}$is a codeword if and only if it has inner product zero (modulo 11) with both rows of the following matrix *H1*:

	n1	n2	n3	n4	n5	n6	n7	n8	n9	n10	n11	n12
H1	8	8	6	5	10	5	6	4	1	4	1	0
	3	6	4	2	6	8	2	1	2	4	0	1

Error detection simply takes place by checking if the received word r = (r1,r2,...,r12) satisfies the two parity check equations.

Encoding can for example be done as follows: choose c1,c2,...,c10 in any way. If we define

```
c11 = - ( 8*c1 + 8*c2 + 6*c3 +...+ 4*c10) modulo 11
c12 = - ( 3*c1 + 6*c2 + 4*c3 +...+ 4*c10) modulo 11
```

then (c1,c2,...,c12) is a codeword. We can view c11 and c12 as parity check digits. Note that we may restrict c1,c2,...,c10 to be any of the numbers 0,1,2. . . ,9. Any of the two parity check digits can be '10'. This '10' can be represented by an alphanumerical character different from 0,1,. . . ,9, for example X or Z.

Decoding is done by:

```
c11 = ( 8*c1 + 8*c2 + 6*c3 +...+ 1*c11) modulo 11
c12 = ( 3*c1 + 6*c2 + 4*c3 +...+ 1*c12) modulo 11
```

From this table, we draw the following conclusions.

- All single and double substitution errors are detected.
- All single and double transposition errors are detected.
- Any combination of a substitution error in position 12, and transposition error in positions not involving position 12 is detected.
- A substitution error not in position 12 "matches" exactly one transposition error. About 1% not detected.

where a transposition is ab => ba and a substitution is a => b.

Example:

Note: following example illustrates the use of the algorithm on valid ARC as input number :

position (n)	1	2	3	4	5	6	7	8	9	10	11	12	
input number	1	6	6	0	8	7	3	1	0	1			choose a digit (0..9)
matrix H1	8	8	6	5	10	5	6	4	1	4	1	0	line for C11 & S11
	3	6	4	2	6	8	2	1	2	4	0	1	line for C12 & S12

coding checkdigit = -sum(n1..n10) mod 11

C11	8	48	36	0	80	35	18	4	0	4		***9***
C12	3	36	24	0	48	56	6	1	0	4		***9***
codeword	1	6	6	0	8	7	3	1	0	1	9	9

decoding checkdigit = +sum(n1..n11 or n12) mod 11

S11	8	48	36	0	80	35	18	4	0	4	9	0	***0***
S12	3	36	24	0	48	56	6	1	0	4	0	9	***0***

A.2.2 Checksum on UDN

Definition

The checksum on the UDN is calculates by F-*UDN*

We use codes over Zp, the integers modulo p,where p=11. That is to say, codewords are strings with entries from for $\{0,1,....p-1\}$. We consider codes of length n defined by r parity equitions: a string $(c1, c2..., cn)$ with elements from Zp is a codeword if and only if it satisfies the following equations:

for $i = 1,2,...r, \sum_{j=1}^{n} aj^{(i)} cj \equiv 0 (\mathrm{mod}\ p)$

We now describe a [20,17] code, that is defined over 20 symbols from Z11 using the three following check equations as described in the matrix H3 below:

Take *n*=17, *r*=3 and *p*=11. We consider the code defined by the *r*=3 following check equations:

```
10*c1 + 1*c2 + 9*c3...+ 8*c17 = 0 (modulo 11)
 0*c1 + 1*c2 + 0*c3...+ 7*c17 = 0 (modulo 11)
 1*c1 + 0*c2 + 1*c3...+ 8*c17 = 0 (modulo 11)
```

In other words, a string (*c*1,*c*2,...,*c*20) with elements from Z11is a codeword if and only if it has inner product zero (modulo 11) with the rows of the following matrix *H3*:

	n1	n2	n3	n4	n5	n6	n7	n8	n9	n10	n11	n12	n13	n14	n15	n16	n17	n18	n19	n20
H3	1	0	1	0	1	0	1	0	1	0	1	2	3	4	5	7	8	1	0	0
	0	1	0	1	0	1	0	1	0	1	0	1	2	3	4	6	7	0	1	0
	10	1	9	2	8	3	7	4	6	5	4	5	7	10	3	2	8	0	0	1

Error detection simply takes place by checking if the received word r = (r1,r2,...,r20) satisfies the three parity check equations. Encoding can for example be done as follows:

Choose c1,c2,...,c17 in any way. If we define

```
c18 = - (10*c1 + 1*c2 + 9*c3 +...+ 8*c17) modulo 11
c19 = - ( 0*c1 + 1*c2 + 0*c3 +...+ 7*c17) modulo 11
c20 = - ( 1*c1 + 0*c2 + 1*c3 +...+ 8*c17) modulo 11
```

then (c1,c2,...,c20) is a codeword. We can view c18, c19 and c20 as parity check digits. Note that we may restrict c1,c2,...,c17 to be any of the numbers 0,1,2. . . ,9. Any of the three parity check digits can be '10'. This '10' can be represented by an alphanumerical character different from 0,1,. . . ,9, for example X or Z.

Decoding is done by:

```
c18 = (10*c1 + 1*c2 + 9*c3 +...+ 1*c20) modulo 11
c19 = ( 0*c1 + 1*c2 + 0*c3 +...+ 1*c19) modulo 11
c20 = ( 1*c1 + 0*c2 + 1*c3 +...+ 1*c18) modulo 11
```

Summarizing, the code defined with H3 detects all errors of any of the following types:

- Single and double substitution errors.
- Single and double transposition errors.
- Any combination of a single substitution error and a single transposition error.
- All three consecutive substitution errors.

where a transposition is ab => ba and a substitution is a => b.

Example:

N.b.: following example illustrates the use of the algorithm on valid UDN as input number :

position (n)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
inputnumber	**8**	**5**	**6**	**2**	**8**	**7**	**0**	**1**	**2**	**1**	**5**	**3**	**2**	**9**	**5**	**6**	**7**				
matrix H3	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**2**	**3**	**4**	**5**	**7**	**8**	**1**	**0**	**0**	line for C20 & S20
	0	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**2**	**3**	**4**	**6**	**7**	**0**	**1**	**0**	line for C19 & S19
	10	**1**	**9**	**2**	**8**	**3**	**7**	**4**	**6**	**5**	**4**	**5**	**7**	**10**	**3**	**2**	**8**	**0**	**0**	**1**	line for C18 & S18

coding checkdigit = -sum(n1..n17) mod 11

C18	8	0	6	0	8	0	0	0	2	0	5	6	6	36	25	42	56			***9***
C19	0	5	0	2	0	7	0	1	0	1	0	3	4	27	20	36	49			***10***
C20	80	5	54	4	64	21	0	4	12	5	20	15	14	90	15	12	56			***2***
codeword	8	5	6	2	8	7	0	1	2	1	5	3	2	9	5	6	7	9	10	2

decoding checkdigit = +sum(n1..n18 or n19 or n20) mod 11

S18	80	5	54	4	64	21	0	4	12	5	20	15	14	90	15	12	56	0	0	2	***0***
S19	0	5	0	2	0	7	0	1	0	1	0	3	4	27	20	36	49	0	10	0	***0***
S20	8	0	6	0	8	0	0	0	2	0	5	6	6	36	25	42	56	9	0	0	***0***

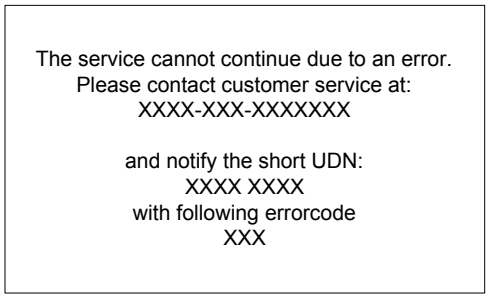
A.3 Status and Error Message Handling

This section describes the status and error values for use in the 1-pass protocols for broadcast devices.

The Status field is a binary value. Upon receipt of a message for which Status is not "Success", the default behaviour, unless explicitly stated otherwise below, is that both the RI and the Device SHALL immediately close the connection and terminate the protocol. RI systems and Devices are required to delete any session-identifiers, nonces, keys, and/or secrets associated with a failed run of the protocol.

When possible, the Device SHOULD present an appropriate error message to the user[13].

[13] Note: It is the sequence of the defined values that is specified. The use of dashes as the delimiter is shown with an example placement to be consistent with the examples used elsewhere in this specification. The text portion of this screen is shown as an example only; there is no implied requirement to duplicate the exact wording or formatting shown. The numeric fields SHALL be included as defined above (please note: the short UDN will only be displayed after the first registration, when that data is available for display).



An example dialogue showing an error

Figure 23: sample notification display

~~**Figure 10: sample notification display**~~

Note: The error codes should be displayed as a three digit decimal number. Refer to Table 41~~Table 41~~~~Table 41~~ for an overview of possible error codes.

Table 41: status / error codes

Status / Error	value$_{(h)}$	comment
Success	0x00	
NotSupported	0x03	
DeviceTimeError	0x0B	
InvalidDomain	0x0D	
DomainFull	0x0E	
ForceInteractiveChannel	0x11	
ForceOobChannel	0x12	
Reserved for future use	0x11-0xFF	

NotSupported indicates the Device made a request for a feature currently not supported by the RI.

DeviceTimeError indicates that Rights Issuer request a Device to set the Device DRM Time with a new value and report the time drift to the Rights Issuer.

InvalidDomain indicates that the request was invalid due to an unrecognized Domain Identifier.

DomainFull indicates that no more Devices are allowed to join the Domain.

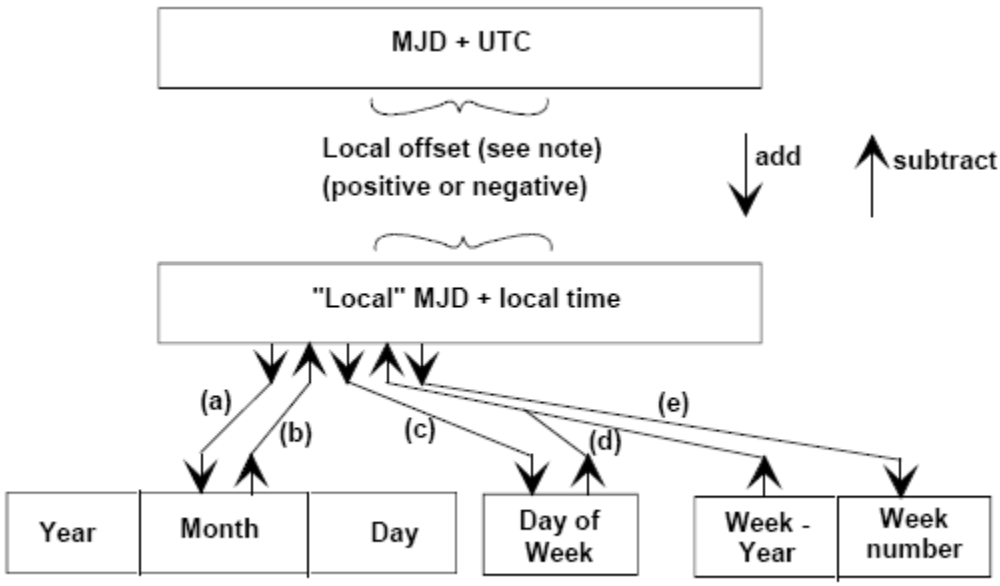
ForceInteractiveChannel indicates that the RI forces a mixed mode device to exclusively use it's interactive channel and not it's OOB channel.

ForceOobChannel indicates that the RI forces a mixed mode device to exclusively use it's OOB channel and not it's interactive channel.

A.4 Conversion between time and date conventions

(please note: this text is a slightly changed versionfrom ETSI EN 300 468 V1.6.1 The version in ETSI EN 300 468 V1.6.1 has a byte alignment problem caused by the time offset polarity field. The version in this text maintains byte alignment everywhere)

The types of conversion which may be required are summarized in Figure 24~~Figure 24~~~~Figure 24~~.



NOTE: Offsets are positive for Longitudes East of Greenwich and negative for Longitudes West of Greenwich.

Figure 24: Conversion routes between Modified Julian Date (MJD) and Co-ordinated Universal Time (UTC)

The conversion between MJD + UTC and the "local" MJD + local time is simply a matter of adding or subtracting the local offset. This process may, of course, involve a "carry" or "borrow" from the UTC affecting the MJD. The other five conversion routes shown on the diagram are detailed in the formulas below:

Symbols used:

D	Day of month from 1 to 31
int	Integer part, ignoring remainder
K, L ,M', W, Y'	Intermediate variables
M	Month from January (= 1) to December (= 12)
MJD	Modified Julian Date
MN	Week number according to ISO 2015 [21]
mod 7	Remainder (0-6) after dividing integer by 7
UTC	Universal Time, Co-ordinated
WD	Day of week from Monday (= 1) to Sunday (= 7)
WY	"Week number" Year from 1900
×	Multiplication
Y	Year from 1900 (e.g. for 2003, Y = 103)

a) To find Y, M, D from MJD

Y' = int [(MJD - 15 078,2) / 365,25]

M' = int { [MJD - 14 956,1 - int (Y' × 365,25)] / 30,6001 }

D = MJD - 14 956 - int (Y' × 365,25) - int (M' × 30,6001)

If M' = 14 or M' = 15, then K = 1; else K = 0

Y = Y' + K

M = M' - 1 - K × 12

b) To find MJD from Y, M, D

If M = 1 or M = 2, then L = 1; else L = 0

MJD = 14 956 + D + int [(Y - L) × 365,25] + int [(M + 1 + L × 12) × 30,6001]

c) To find WD from MJD

WD = [(MJD + 2) mod 7] + 1

d) To find MJD from WY, WN, WD

MJD = 15 012 + WD + 7 × { WN + int [(WY × 1 461 / 28) + 0,41] }

e) To find WY, WN from MJD

W = int [(MJD / 7) - 2 144,64]

WY = int [(W × 28 / 1 461) - 0,0079]

WN = W - int [(WY × 1 461 / 28) + 0,41]

EXAMPLE: MJD = 45 218 W = 4 315

Y = (19)82 WY = (19)82

M = 9 (September) N = 36

D = 6 WD = 1 (Monday)

NOTE: These formulas are applicable between the inclusive dates 1900 March 1 to 2100 February 28.

A.4.1 Local time offset

This 16-bit field contains the current offset time from UTC in the range between –12 hours and +13 hours at the area which is indicated by the combination of country_code and country_region_id in advance. These 16 bits are coded as 4 digits in 4-bit BCD in the order hour tens, hour, minute tens, and minutes.

The positive or negative offset from the UTC is indicated with the 1 bit local_time_offset_polarity. If this bit is set to “0” the polarity is positive and the local time is advanced to UTC. (Usually east direction from Greenwich). If this bit is set to “1” the

polarity is negative and the local time is behind UTC. Please note that the local_time_offset_polarity is represented by the first bit of the first nibble representing the hour tens field. The first nibble of the local_time_offset is therefore encoded as follows:

Table 42: Local time offset coding

local_time_offset_polarity	offset hour tens	first nibble
0 (i.e. "+")	0	0000
0 (i.e. "+")	1	0001
1 (i.e. "-")	0	1000
1 (i.e. "-")	1	1001

A.5 RSA signatures under PKCS#1

RSA signatures are made as described by the implementation guidelines of [PKCS #1] v2.1: RSA Cryptography Standard, *RSA Laboratories, June 14, 2002.*

The scheme is RSA + SHA1. There are two choices described in the [PKCS#1] as they are RSASSA-PSS and RSASSA-PKCS1-v1_5

Since OMA DRM 2.0 is used for interactive mode of operation and uses RSASSA-PSS, this specification will also use RSASSA-PSS to sign the binary messages for broadcast mode of operation.

A.6 C-style types

Following abbreviated types are used in the document:

type name	description	remark
bslbf	bit serial leftmost bit first	
mjdutc	modified julian date UTC	
uimsbf	unsigned integer most significant bit first	

All fields marked as reserved for future use SHALL be set to the value 0, when not used.

All fields marked as reserved SHALL be set to value 0, and never to any other value.

A.7 Tag Length Format for keyset_block

A.7.1 Syntax definition

A Tag Length Format (TLF) is defined to identify the keyset_items in the keyset_block. A keyset_item is identified by following syntax:

<tag> [optional <clarifier>] <length> <keyset_item>

Following values are defined and SHALL be used:

tag values:

This is a 4 bit field (bslbf) indicating the tag that uniquely identifies the keyset item.

Table 43: defined tag values

Keyset_item	Tag (b)	remark
UGK	0000	
SGK	0001	
UDK	0010	
UDF	0011	
LDK	0100	
SLDF	0101	shortform_domain_id
LLDF	0110	
RIAK	0111	
TDK	1000	
reserved for future use	1001-1111	not used in this version of the spec

Note:

- The keyset items SHALL be included in the order of the table above.
- The keyset SHALL include only one instance of the following keys: UGK, UDK, UDF, RIAK and TDK.
- If included the BGKs (8 or 9) SHALL follow in fashion BGK1..n.
- The keyset MAY include zero or more domain sets (LDK, SLDF, LLDF). If included the SLDF SHALL follow the LDK it belongs to, followed by the optional LLDF that belongs to the aforementioned SLDF.

clarifier (optional):

This is a 10 bit field (bslbf) can be used to indicate the following possible values:

- in case the preceding <tag> value indicates a SGK, this field represents the position of a SGK in the Fiat Naor tree.
- in case the preceding <tag> value indicates a LLDF this field represents the length on the LLDF.

describing the use of the clarifier field for position of SGK:

If keyset_item == 001 (i.e. SGK) then the optional field “clarifier” SHALL indicate the position of the SGK as a node in the [FIAT NOAR] tree. When m = groupsize, then n = 2 log (m), where n is number of BGKs in tree. Possible positions for the BGKs in the tree are $2^{(n+1)}$ -1 . Therefore parameter “position” is expressed with 10 bits to express 1023 nodes in a tree. First MSB left will be used as binary indicator to indicate if the SGK position is a node (0, zero) or a leaf (1, one). Bit positions 2..10 (from left to right LSB) are used in binary format as an indication of the node and leaf position. Nodes and leafs SHALL be numbered according to following Figure 25Figure 25Figure 25:

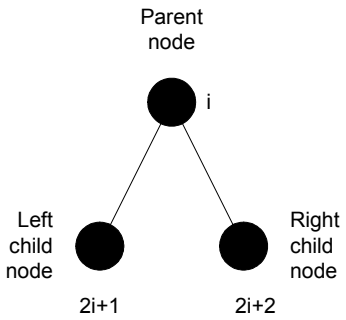


Figure 25: node numbering

Key:

The root key R is numbered zero. Node keys NK are sequentially numbered per "level" in a breadth-first manner from left to right, starting from the root node with number 0

describing the use of the clarifier for length of LLDF:

If LLDF is included the optional field "clarifier" describes the variable length of the LLDF in bits, as described in A.7.3.

length values:

This is a 3 bit field (bslbf) indicating the length of a keyset item.

Table 111110: defined length values

(key)length prescriber	Length (b)	remark
128 bit AES	000	
192 bit AES	001	
256 bit AES	010	
5 byte Eurocrypt	011	
6 byte	100	SLDF
reserved for future use	101-111	not used in this version of the specification

Note: In case of the LLDF there is no extra length field, since the length value is indicated by the clarifier.

A.7.2 TLF examples

E.g.1: A 5 byte Eurocrypt address implementing the UDF will be coded like:

```
<0011> <011> <UDF>
```

E.g.2: A 48 bits SLDF address will be coded like:

```
<0101> <100> <SLDF>
```

E.g.3: A LLDF address of 105 bytes will be coded like:

```
<0110> <1101001000> <LLDF>
```

E.g.4: A 128 but AES key implementing the UGK will be coded like:

```
<0000> <000> <UGK>
```

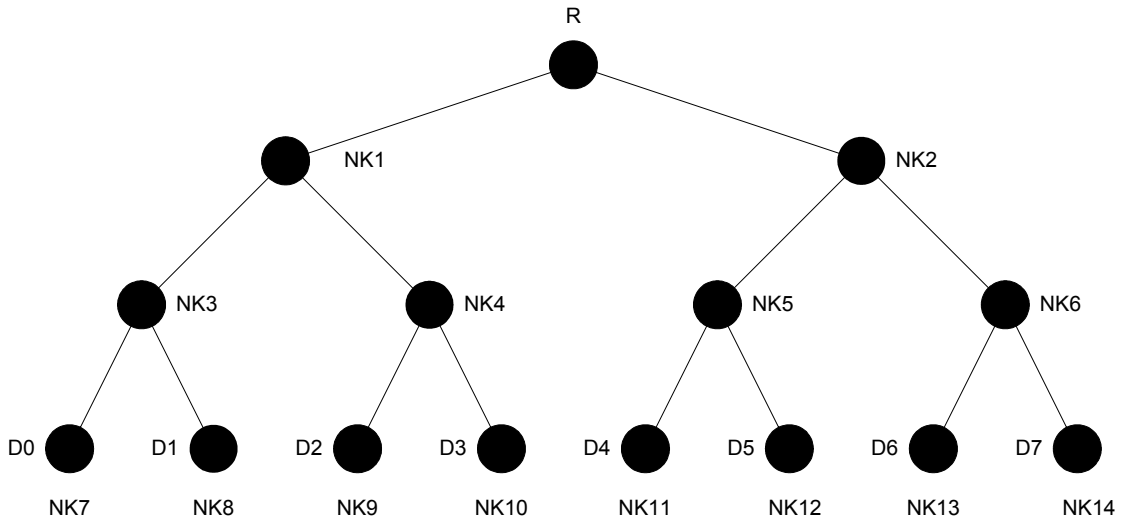


Figure 26: sample tree with correct node and device numbering.

E.g.5: A 128 bit AES key implementing the SGK on node position NK5 in Figure 26Figure 26Figure 26 will be coded like:

```
<0001> <0000000101> <000> <SGK>
```

E.g.6: A 128 bit AES key implementing the SGK on node position NK7 (i.e. D0) in Figure 26Figure 26Figure 26 will be coded like:

```
<0001> <1000000001> <000> <SGK>
```

A.7.3 LLDF syntax

In OMA DRM 2.0 the domain ID can be 1 to 17 characters (any) followed by 3 digit characters.

The string that forms the identifier is encoded normally in ROAP messages using UTF-8 [RFC 3629]. UTF-8 character encoding for ASCII characters is 'efficient' with 1 byte per character. On the other hand, there are characters that are encoded using 6 bytes (Asian languages).

The 17 XML UTF-8 characters are translated into bytes as follows:

Longest OMA DRM 2.0 domain identifier encoded as bytes is 6*17+3 bytes = 105 bytes.

Shortest domain identifier is 4 bytes.

A.8 Message_tag overview

The messages that are defined in this specification SHALL use following message_tag values:

Table 44: message_tag overview

message name	message_tag	described in section
domain_registration_response()	0x02	{EN: TBD}6.4.3.1.3
domain_update_response()	0x03	{EN: TBD}6.4.3.2.2
re_register_msg()	0x11	{EN: TBD}6.2.5.1.2
update_ri_certificate_msg()	0x12	{EN: TBD}6.2.2.1
update_drmtime_msg()	0x13	{EN: TBD}6.2.3.1.2
update_contact_number_msg()	0x14	{EN: TBD}6.2.4.1.2
join_domain_msg()	0x15	{EN: TBD}6.4.3.3
leave_domain_msg()	0x17	{EN: TBD}6.4.3.4
BCRO	0x20	{EN: TBD}7.2.1
token_delivery_response()	0x30	{EN: TBD}6.3.4.1.2

A.9 Authentication

A.9.1 Authentication for IPsec

IPsec can be used with authentication. In case of authentication with IPsec the authentication data will carry the TAS. The authentication mechanism will create the TAK from the TAS.

To obtain the encrypted traffic key material from the KSM the encrypted traffic key material is decrypted with the SEK or PEK:

$$TAS = D\{SEK\}(traffic\_key\_material)$$

or

$$TAS = D\{PEK\}(traffic\_key\_material)$$

~~or~~

~~$TAS = D\{PEK\}(traffic\_key\_material)$~~

~~T~~

The authentication key is generated from the authentication seed:

$$TAK = f_{auth}\{TAS\}(CONSTANT\_KSM)$$

where:

```
CONSTANT_KSM = 0x04040404040404040404040404040404 (120 bit)
```

Refer to A.9.4 for details on f-auth.

The TAK is used in the MAC generation / verification of the IPsec data. Refer to [RFC 2406] for details.

A.9.2 Authentication for KSMs

A key stream message can contain two MAC fields. The programme MAC field and the service MAC field. If only one MAC field would be used, the authentication key could only be renewed when both SEK and PEK change at the same time. Having two MAC fields and two authentication keys makes it possible to authenticate the message and check for its integrity while only having one key set. The service authentication key (SAK) and the programme authentication key (PAK) will be derived from the service authentication seed and the programme authentication seed respectively which are transmitted together with the encryption keys in the ROs (How this is carried in the BCRO and ICRO in explained in subsequent sections). A service

RO will contain service encryption and authentication keys (SEAK) and a programme RO will contain programme encryption and authentication keys (PEAK).

To obtain the PAS or TAS from the BCRO the SEAK/PEAK is decrypted with the DEK:

$$SAS = LSB_{128}(D\{DEK\}(SEAK))$$

$$PAS = LSB_{128}(D\{DEK\}(PEAK))$$

The authentication key is generated from the authentication seed:

$$SAK = f_{auth}\{SAS\}(CONSTANT\_SAK)$$

$$PAK = f_{auth}\{PAS\}(CONSTANT\_PAK)$$

where :

```
CONSTANT_SAK = 0x02020202020202020202020202020202 (120 bit)

CONSTANT_PAK = 0x01010101010101010101010101010101 (120 bit)
```

Refer to A.9.4 for details on f-auth.

The SAK or PAK is used in the MAC generation / verification of the KSM. The algorithm used to calculate the MAC field is HMAC-SHA1-96 according to [FIPS 198] and [RFC2104], using authentication keys of 160 bit in both cases.

A.9.2.1 Transport of SEAK and PEAK in OMA DRM 2.0 Rights Objects

The encryption keys and authentication keys (SEAK and PEAK) are transported in a ICRO by concatenating the encryption key and the authentication seed and then protecting the resulting field with AES_wrap [AES_WRAP].

encrypted_service_encryption_authentication_key =

$$E\{REK\}(SEAK) = AES\_wrap\{REK\}(SEK << 128) \,||\, SAS)$$

where SEK and SAS are both an AES key of 128 bits.

and encrypted_programme_encryption_authentication_key =

$$E\{REK\}(PEAK) = AES\_wrap\{REK\}(PEK << 128) \,||\, PAS)$$

where PEK and PAS are both an AES key of 128 bits.

A.9.2.2 Transport of SEAK and PEAK in BCROs

The encryption keys and authentication keys (SEAK and PEAK) are transported in a BCRO by concatenating the encryption key and the authentication seed and then protecting the resulting field with AES CBC.

encrypted_service_encryption_authentication_key =

$$E\{DEK\}(SEAK) = AES\_CBC\{DEK\}(SEK << 128) \,||\, SAS)$$

where SEK and SAS are both an AES key of 128 bits.

and encrypted_programme_encryption_authentication_key =

$$E\{DEK\}(PEAK) = AES\_CBC\{DEK\}(PEK << 128) \,||\, PAS)$$

where PEK and PAS are both an AES key of 128 bits.

A.9.3 Authentication of BCROs

BCROs contain one MAC field which is used to authenticate the message and to protect the integrity of the message.

The authentication key is generated from the RIAK:

$$BAK = f_{auth}\{RIAK\}(CONSTANT\_BCRO)$$

where:

```
CONSTANT_BCRO= 0x030303030303030303030303030303 (120 bit)
```

Note: To obtain the RIAK the device needs to have been equipped with a valid keyset. Refer to section 6.1.3 for details.

Refer to A.9.4 for details on f-auth.

The BAK is used in the MAC generation / verification of the BCRO. The algorithm used to calculate the MAC field is HMAC-SHA1-96 according to [FIPS 198] and [RFC2104], using a authentication key of 160 bit.

A.9.4 General authentication mechanism

The function F-auth consists of several steps:

1. Denote by PRF{key}(text) as the AES-XCBC-MAC-PRF with output blocksize 128 bits as defined by IPsec WG in IETF. Please note:
   - Refer to [RFC 3566] for the AES-XCBC-MAC-PRF based key generation function.
   - Refer to [RFC 3664] for the requirement NOT to truncate the generated key material.
2. Apply the generated input key according to ideas of IKEv2 to generate authentication key. Define a key generator function f-kg{key}(constant). Keying material will always be derived as the output of the negotiated PRF algorithm.. PRF^+ describes the function that outputs a pseudo-random stream of n blocks based on the inputs to a PRF as follows:

   $$T1 = AES\_XCBC\_MAC\_PRF\{AS\}(CONSTANT \,||\, 0x01)$$

   $$T2 = AES\_XCBC\_MAC\_PRF\{AS\}(T1 \,||\, CONSTANT \,||\, 0x02)$$

   ....

   $$Tn = AES\_XCBC\_MAC\_PRF\{AS\}(T1 \,||\, CONSTANT \,||\, n)$$

   where *AS* is the appropriate authentication seed (be it TAS, PAS, SAS or RIAK) and *CONSTANT* is the appropriate constant as described in preceding sections. The amount of blocks to derive is defined by the amount of key material needed, i.e. n is the amount of needed key bits divided by 128 and rounded up.

   This means that if 160 bits were needed then PRF^{+}~~*~~() would be computed as:

   $$T1 \,||\, T2 = PRF^{+}\{K\}(S)$$

3. The 160 bit authentication key is taken from the generated key material as follows:

$$AK = MSB_{160}(T1 \| T2)$$

The generated authentication key is applied as described in preceding sections.

A.9.5 Authentication of token delivery response messages

Token delivery response messages contain one MAC field which is used to authenticate the message and to protect the integrity of the message.

The authentication key is generated from the RIAK:

$$TDRMAK = f_{auth}\{RIAK\}(CONSTANT\_TDRM)$$

where:

```
CONSTANT_TDRM= 0x04040404040404040404040404040404 (120 bit)
```

Note: To obtain the RIAK the device needs have been equipped with a valid keyset. Refer to 6.1.3 for details.

Refer to A.9.4 for details on f-auth.

The TDRMAK is used in the MAC generation / verification of the token delivery response message. The algorithm used to calculate the MAC field is HMAC-SHA1-96 according to [FIPS 198] and [RFC2104], using a authentication key of 160 bit.

A.10 Authentication of the tokens_consumed field in the token consumption data

Devices SHALL authenticate the tokens_consumed field and the device_nonce of the token consumption report, see section 5.1.2[*1)], in the way specified in this section. The hash function used here is one of the four secure hash functions from [SCHNEIER], page 449 (the upper left one in figure 18.9).

The maximum amount of tokens that can be reported as consumed is 9999. The amount of tokens, with the before mentioned restriction, is represented with a 14 bit uimsbf number and called tokens_consumed. The value of device_nonce can be in value between 0 and 9 and is represented as a 4 bit uimsbf number. The 14 bit number tokens_consumed is right concatenated with the 4 bit number device_nonce. The resulting 18 bit number is right padded with 0x1 and right padded again with 109 binary zeroes (so 2^{109}). The resulting 128 bit number is used as the input for a single AES block. The Report Authentication Key, as obtained with the token delivery response message, see section **Error! Reference source not found.**, is used as the key input for the AES block. The 128-bit output of the AES block is EXOR-ed with the 128-bit input of the AES block. The left-most 43 bits of the result of this EXOR operation are taken as the report_authentication_code. The 13 digit decimal representation of these 43 bits, including any leading zeroes is used as the report_authentication_code in the token consumption report. See also the next figure.

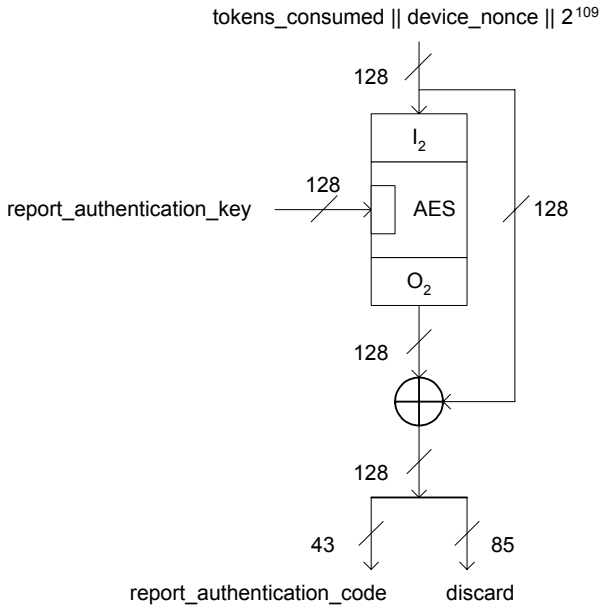


Figure 27: computation of the report_authentication_code

A.11 Management of tokens by RIs and devices

A.11.1 Token management by RIs

There are two business models for the use of tokens.

The first business model is that all tokens ordered by the user are paid for by the user. These tokens are pre-paid tokens. The second business model is that a user orders tokens, but only wants to pay for the ones he actually consumes. These tokens are post-paid tokens.

The tools to support these two business models are the token delivery response message, see section 6.3.2 and the token reporting protocol, see section 6.3.3. The next sections describe how these tools can be used by an RI to support the above two business models and how one can switch from one business model to the other.

A.11.1.1 Pre-paid token business model

Setting the token_reporting_flag in the token delivery response message to 0x0 will signal to the device that it does not now nor in the future have to report anymore on the consumption of any of the tokens received so far from this RI.

Therefore in the pre-paid token business model, where the user has agreed to be billed for the delivery of the tokens and their consumption need not be reported, the RI will set the token_reporting_flag to 0x0.

Tokens that are delivered from an RI to a device with a token delivery response message which token_reporting_flag has been set to 0x0 can be called pre-paid tokens.

A.11.1.2 Post-paid token business model

Setting the token_reporting_flag in the token delivery response message to 0x1 will signal to the device that it SHALL report on the consumption of these tokens[14].

Therefore in the post-paid token business model, where the user has to be billed for the actual consumption of the tokens and their actual consumption SHALL be reported by the device, the RI will set the token_reporting_flag to 0x1.

Tokens that are delivered from an RI to a device with a token delivery response message which token_reporting_flag has been set to 0x1 can be called post-paid tokens.

In the post-paid token business model, the RI can limit its risk, by making sure that a device at all times only contains post-paid tokens up to a certain maximum, the so called credit-limit. Furthermore, the RI can set a date/time limit in the device after which the device is not allowed to consume post-paid tokens any more. This can be done as follows

1. The RI sends in the first token delivery response message a number of tokens equal to the credit-limit. Furthermore, the RI sets the token_reporting_flag in the token delivery response message to 0x1 and sets the latest_consumption_time to a suitable date/time.
2. The RI waits for the reception of a token consumption message.
3. If the RI receives a token consumption message, it SHALL check the authenticity of the tokens_consumed field. If the authentication fails, go to step 2, otherwise continue with step 4.
4. For reasons explained in section ~~Error! Reference source not found.~~A.11.1.3, if the reported number of consumed tokens is higher than the credit-limit, the RI SHALL assume that only a number of post-paid tokens equal to the credit-limit have been consumed by the device.
5. The RI bills the user for the amount of post-paid tokens consumed with a maximum equal to the credit-limit.
6. The RI sends a token delivery response message a number of tokens equal to the amount of post-paid tokens consumed with a maximum equal to the credit-limit. Furthermore, the RI sets the token_reporting_flag in the token delivery response message to 0x1 and sets the latest_consumption_time to a suitable date/time.
7. Go to step 2.

Note that in the above, the use of the response_flag, device_nonce, earliest_reporting_time, latest_reporting_time and other fields has not been included.

Note further that the RI can force the creation of a token consumption message by sending a token delivery response message with its status field set to “TokenConsumptionMessageError” or to “NoTokenConsumptionMessage”.

A.11.1.3 Switching from the pre-paid token business model to the post-paid token business model

When at a certain point of time, the user asks the RI to switch from the use of pre-paid tokens to the use of post-paid tokens and the RI agrees, the RI starts at step 1 in the previous section. The device will report the actual consumption of tokens that will delivered to it in step 1 and in all steps 6. However, at the time of executing step 1, the device MAY still have some pre-paid tokens. Based on the implementation of the device, these pre-paid tokens MAY also be reported as consumed by the device, see A.11.2. Because the RI knows that a device never holds more post-paid tokens than the credit limit, the RI SHALL assume that at most an amount of tokens equal to credit_limit have been consumed by the device. Hence step 4 in section A.11.2.

[14] Note that although a broadcast device can only display the token consumption message to the user and must rely on the user to report this message to the RI, the wording in this section is as if the device does the reporting.

A.11.1.4 Switching from the post-paid token business model to the pre-paid token business model

When at a certain point of time, the user asks the RI to switch from the use of post-paid tokens to the use of pre-paid tokens, and the RI agrees, the RI has a few options

One option is that the RI bills the user for the amount of post-paid tokens that were left in the device at the time of the last token consumption message. These tokens have in effect then become pre-paid tokens. The RI will set the token_reporting_flag in the next token delivery response message to 0x0 and set the value of token_quantity to zero or to the amount of tokens that the user wished to purchase in addition to the amount of post-paid tokens left in the device (encrypting token_quantity yields the encrypted_token_quantity field).

Another option, useful e.g. when the previous option turns out to be expensive, is that the RI performs the actions described in section A.11.1.5 for clearing the post-paid tokens left in the device. After clearing the post-paid tokens, the RI can start sending pre-paid tokens to the device if the user wishes to purchase these.

A.11.1.5 Stopping the post-paid token business model

When at a certain point of time, the user informs the RI that he does no longer wish to use post-paid tokens, not even the ones that are still in his device, the RI can do the following. The RI sends the device a token delivery response message with:

- the value of token_quantity set to zero (encrypting token_quantity yields the encrypted_token_quantity field), or set the token_quantity_flag to 0x0,
- the token_reporting_flag field set to 0x1,
- the latest_token_consumption_time set to a date/time in the past,
- the status field to “NoTokenConsumptionMessage”.

This forces the device to generate a token consumption report. Using this, the RI determines how many post-paid tokens are still in the device. The RI sends the device a token delivery response message with:

- the value of token_quantity set to minus the amount of post-paid tokens left in the device (encrypting token_quantity yields the encrypted_token_quantity field),
- the token_reporting_flag field set to 0x1,
- the latest_token_consumption_time set to a date/time in the past,
- the status field to “Success”.

The above message MAY be repeated several times. After reception of this token delivery message, the device will have no post-paid tokens left. Any remaining pre-paid tokens can still be consumed by the device.

A.11.2 Token management by devices

Each RI context in a device SHALL at a minimum contain:

- a token purse, which is incremented with the tokens received from the RI and which is decremented with the amount of tokens required for each requested consumption of metered protected content from the corresponding RI. If a decrement would yield an accumulator value of less than zero, the device SHALL deny the requested consumption of metered protected content from the corresponding RI.
- a token consumption accumulator, which initially starts at zero.
- the token purse initially starts at zero.

Whenever a device receives from an RI a token delivery response message that has its token_reporting_flag set to 0x0, the device sets the token consumption accumulator associated with the RI to zero and SHALL consider all tokens in the token purse associated with the RI as pre-paid tokens.

In case of the reception of a (series of) token delivery response message with the token_reporting_flag set to 0x1, there are 2 possible implementations of token consumption registration.

A device with a simple implementation of token consumption registration would do the following:

1. Whenever tokens are required and available for consumption, then in addition to decrementing the token purse associated with the RI, the device also increments the token consumption accumulator associated with the RI with the same amount.
2. When a device receives a token delivery response message with status "Success" and a token_reporting_flag set to 0x1, the device SHALL schedule the creation of a token consumption report at a suitable time, keeping in mind the value in the field latest_consumption_time and possibly the values in the fields earliest_reporting_time and latest_reporting_time.

   If the response_flag was set to 0x1, the device SHALL decrement the token consumption accumulator associated with the RI with the amount of tokens reported in the token consumption report that this token delivery response message was a response to. The device MAY delete that token consumption report and all others sent before that token consumption report was sent

   The device SHALL increment the token purse associated with the RI with the number of tokens indicated in the encrypted_token_quantity field of this token delivery response message.
3. When a device receives a token delivery response message with status "TokenConsumptionMessageError" or with status "NoTokenConsumptionMessage", the device SHALL immediately create a token consumption report.
4. When a device creates a token consumption report, it uses a device_nonce which is one higher than the previously used device_nonce. If the previously used device_nonce was 9, the device uses 0 as the next device_nonce.
5. When a device creates a token consumption report, it uses the value of the token consumption accumulator associated with the RI as the amount of tokens to report as consumed.
6. Token consumption reports SHALL be stored by the device, at least until the device receives a token delivery response message indicating that they have been successfully been processed by the RI or indicating that later created token delivery messages have been successfully been processed by the RI
7. The device SHALL stop with the execution of the above actions stop when it receives a token delivery response message with the token_reporting_flag set to 0x0. The device SHALL then set the token consumption accumulator associated with the RI to zero and MAY delete all created token consumption reports.

The device actions above imply that the RI will consider the first credit_limit tokens reported as consumed to be post-paid tokens, even though the device might still have had pre-paid tokens. Furthermore, if the current date/time is past the date/time set in the latest_token_consumption_time field, a device is not allowed to use tokens any more. Since a device according to the above rules does not keep track separately of the pre-paid tokens, it cannot use tokens anymore even though the device might still have had pre-paid tokens.

With a slightly more complex implementation, the above two disadvantages can be solved. The device can then first consume all pre-paid tokens before consuming any post-paid tokens. To this end, the device would implement two token purses per RI, a pre-paid token purse and a post-paid token purse for tokens that have been delivered to the device with token delivery response messages with the token_reporting_flag set to 0x1. The device can first consume all tokens in the pre-paid token purse. Consumption of tokens from the pre-paid token purse will not influence the value of the token consumption accumulator and this consumption will therefore not be reported by the device. Whenever the device consumes tokens from the other token purse, the token consumption accumulator SHALL be incremented with the same amount. A device according to this implementation, upon the reception of a token delivery response message with the token_reporting_flag set to 0x0, SHALL increment the pre-paid token purse with the tokens in the post-paid token purse, SHALL set the post-paid token purse as well as the token consumption accumulator to zero.

A.12 Confidentiality in the Subscriber Group Concept

A.12.1 Exponential Scheme

As there are 2^n subsets of a group of n devices, a very inefficient way of implementing the scheme is to generate 2^n distinct keys. Each device would be provided with the keys associated with all the subsets that include that device.

Group size	Number of subsets	Number of keys per device
1	2	1
2	4	2
4	16	8
8	256	128
16	65536	32768
32	4294967296	2147483648

This is for all practical purposes completely unusable.

A.12.2 Linear Scheme

An easy optimisation of the grossly impractical scheme is to generate an exclusion key unique per device part of the group. Each device is given all exclusion keys, except its own exclusion key. For any subset of the group that is to be allowed to access content, one can define the complement subset. If all the exclusion keys of the devices in the complement subset are used in a key derivation function, then only those devices in the complement subset cannot compute all the key material required: they lack the key associated with themselves.

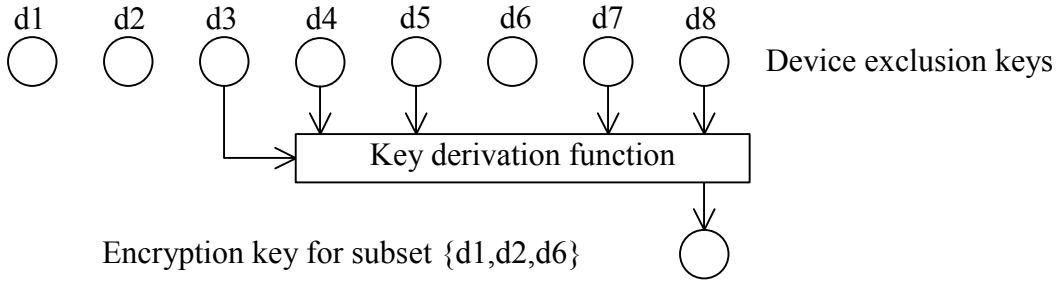


Figure 28: Derivation of an encryption key associated with a subset of the group

The figure shows the derivation of an encryption key for the subset {d1, d2, d6}. A simple derivation function used as example here is the bitwise XOR. Each of the devices from the complement subset {d3, d4, d5, d7, d8} will find that its key is used in this derivation. Consequently neither of the devices from the complement subset can compute the encryption key. For example, device d4 cannot compute the required:

d3 XOR d4 XOR d5 XOR d7 XOR d8

because it only knows d1, d2, d3, d5, d6, d7 and d8.

The size of the key material to be distributed now scales linear with the size of the group. This is a big improvement over the exponential scaling of the naïve approach.

Group size	Number of subsets	Number of keys per device
1	2	0
2	4	1

4	16	3
8	256	7
16	65536	15
32	4294967296	31
64	1.84×10^{19}	63
128	3.40×10^{38}	127
256	1.16×10^{77}	255
512	1.34×10^{154}	511
1024	1.80×10^{308}	1023

This is a great improvement, and can make the scheme already practical for modest group sizes.

A.12.3 Logarithmic Scheme

In [Broadcast Encryption, Advances in Cryptology - CRYPTO '93 Proceedings, Lecture Notes in Computer Science, Vol. 773, 1994, pp. 480–491, A. Fiat, M. Noar] the authors provide a scheme of hierarchical key derivations. Under this scheme, each device is provided key material that allows on-demand computing of the keys associated with all other devices in the group, except itself. The following picture shows schematically how this operates:

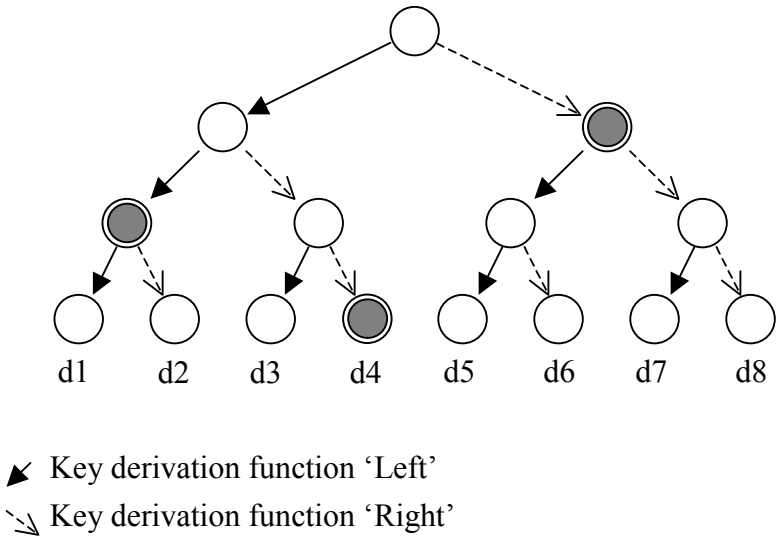


Figure 29: Fiat-Naor key derivation scheme

The figure shows the application of two similar, but different, key derivation algorithms. From a single key, two child keys can be derived using these two distinct functions. A tree hierarchy of keys can thus be formed. The complete tree is determined completely by the two key derivation functions and the single root key.

This scheme allows an efficient version of the linear scheme. Instead of distributing all keys (except its own) to a device, now only a few keys from the tree need to be distributed to each device. It can be shown that instead of n-1 keys, now it is sufficient to distribute $\log_2 n$ keys to each device.

Group size (*n* devices)	Total number of keys in the group		Number of keys per device	
	Linear scheme n **x** $(n-1)$	**Logarithmic scheme** n x $\log_2 n$	**Linear scheme** $(n-1)$	**Logarithmic scheme** $\log_2 n$
1	0	0	0	0
2	2	2	1	1
4	12	8	3	2
8	56	24	7	3
16	240	64	15	4
32	992	160	31	5
64	4032	384	63	6
128	16256	896	127	7
256	65280	2048	255	8
512	261632	4608	511	9
1024	1047552	10240	1023	10
…				
1048576	1.10×10^{12}	20971520	1048575	20

A practical limit to the subscriber group size is given by the need to communicate which subset of the group is selected to access particular content. This is typically done with a bitvector, indicating which devices are included in the subset. For each communication to a specific subset, such a bitvector of *n* bits length must be added in order for the devices to determine the used encryption key.

The BCRO format restricts the size of the subscriber group to 256 or 512 devices.

It must be noted that if the subset of devices allowed to access content is the whole group, then the derivation of the content encryption key fails, because there is no device key at all to include in the key derivation algorithm. To address this issue, one can provide all devices with one additional key special key, to be used when the whole group is addressed.

Appendix B. Change History (Informative)

B.1 Approved Version History

Reference	Date	Description
n/a	n/a	No prior version –or- No previous version within OMA

B.2 Draft/Candidate Version V1_0 History

Document Identifier	Date	Sections	Description
Draft Versions OMA-DRM-XBS-V1_0	21 Feb 2005	n/a	First draft outline based on input to the joined committees (BAC-DLDRM and BAC-BCAST): OMA-BCAST-2005-0048-Joint-BCAST-DRM-Task-Workplan as well as discussions and contributions to the email reflector (prioritisation of work items).
	15 Mar 2005	6.3	OMA-BCAST-2005-0100R01-token-based-metering (approved at the Chicago BCAST/DLDRM joint meeting).
	17 Mar 2005	6.1 & 6.2 7	OMA-DLDRM-2005-0064-Broadcast-Rights-Object (approved in conference call 17 mar 2005) OMA-DLDRM-2005-0071R01-subscriber-group-addressing (approved in conference call 17 mar 2005)
	8 Apr 2005	5.1 5.1 5.1	OMA-DLDRM-2005-0085R01-offline-notification-of-detailed-device-data (approved in conference call 6 apr 2005) OMA-DLDRM-2005-0086-Push-binary-Device-Registration-data (approved in conference call 6 apr 2005) OMA-DLDRM-2005-0087R01-offline-notification-of-short-device-data (approved in conference call 6 apr 2005)
	11 May 2005	5.1.4, 7.2.3, 7.3.5 & 7.3.6	OMA-DLDRM-2005-0100R01-Broadcast-Extensions-Device-Registration (approved in Singapore BCAST/DLDRM joint meeting)
	21 Jun 2005	10 6.3, 7 9 6.3 8.3	OMA-BCAST-2005-0094R04-PDCF-adaptation-for-Traffic-Encryption-Key-stream OMA-BCAST-2005-0100R03-token-based-metering-specification-text OMA-DLDRM-2005-0098R03-Broadcast-Extensions-Key-Stream-Handling OMA-DLDRM-2005-0114R01-Broadcast-Extension-Key-Stream-Authentication-Key-Transfer OMA-DLDRM-2005-0169-Broadcast-encryption-key-derivation-functions
	11 Aug 2005	Many sections changed. Many sections moved to appendix A.	OMA-DLDRM-2005-0213-offline-notification-of-short-device-data OMA-DLDRM-2005-0214R01-update-ri-certificate OMA-DLDRM-2005-0215R01-update-DRM-time-via-broadcast OMA-DLDRM-2005-0216-update-contact-numbers-for-out-of-band-notification OMA-DLDRM-2005-0217-force-broadcast-reregistration OMA-DLDRM-2005-0220-domains-for-broadcast OMA-DLDRM-2005-0221-Token-handling-using-broadcast-channel OMA-DLDRM-2005-0224-RI-Services-for-broadcast OMA-DLDRM-2005-0229-authentication-for-broadcast OMA-DLDRM-2005-0232-BCRO_update OMA-DLDRM-2005-0243-BCRO-message-tag

Document Identifier	Date	Sections	Description
	10 Sep 2005	6.1.1 A.2	OMA-DLDRM-2005-0211R02-offline-notification-of-detailed-device-data
		6.1.3 A.4	OMA-DLDRM-2005-0212R02-push-device-registration-data-to-device-during-broadcast-registration
		A.13	OMA-DLDRM-2005-0223R02-OCSP-grace_-broadcast-
		7.2.1 7.2.2	OMA-DLDRM-2005-0254-BCRO-optimisations

Appendix C. Static Conformance Requirements (Normative)

The notation used in this appendix is specified in [IOPPROC].