SMPTE ST 2071-1:2014
Revision of
SMPTE ST 2071-1:2012

SMPTE STANDARD

Media Device Control Framework (MDCF)

Table of Contents

Page

Foreword 3
Intellectual Property 3
Introduction 3
1 Scope 5
2 Conformance Notation 5
3 Normative References 6
4 Identity 6
4.1 Media Device Control Resource Name 6
4.2 Uniform Device Name (UDN) 8
4.3 Uniform Service Name (USN) 8
4.4 Uniform Media Name (UMN) 9
4.5 Uniform Capability Name (UCN) 10
4.6 Namespace Names 11
5 Directories 12
5.1 Registries 12
5.2 Load Balancing and Fault Tolerance 12
6 Signals 13
6.1 Polling 13
6.2 Asynchronous Callback 13
7 Capability-Based Programming 13
7.1 Capability Interfaces 14
8 Services 15
8.1 Service Factory 15
8.2 Service Directory 15
8.3 Service Registry 15
8.4 Service Templates 15
9 Devices 16
9.1 Device Directory 16
9.2 Device Registry 16
9.3 Device Hierarchy 16
10 Modes 17

Copyright © 2014 by THE SOCIETY OF
MOTION PICTURE AND TELEVISION ENGINEERS
3 Barker Avenue, White Plains, NY 10601
(914) 761-1100

Approved
May 13, 2014

11 Media ... 17
11.1 Media Assets ... 17
11.2 Material Assets ... 17
11.3 Media Files ... 18
11.4 Media Instances ... 18
11.5 Media Containers ... 18
11.6 Media Bundles ... 18
11.7 Media Pointers and Segments ... 18
11.8 Media Directory ... 18
12 Query Expressions and Query Syntax ... 19
12.1 Attribute Designation ... 20
13 Delegation of Control ... 21
13.1 Acquiring a Session ... 21
13.2 Exclusive Locking ... 21
13.3 Requesting a Session ... 22
13.4 Requesting an Exclusive Lock ... 22
13.5 Administration of Sessions and Locks ... 23
14 Authentication / Authorization ... 24
14.1 Authentication ... 24
14.2 Authentication Sequence ... 24
14.3 Security Layer ... 25
14.4 Authentication Servers ... 25
14.5 Permissions ... 25
15 Data and Operation Model ... 26
15.1 Identity ... 26
15.2 Status and Events ... 29
15.3 Service Framework ... 33
15.4 Device Framework ... 37
15.5 Session and Lock Management (Delegation of Control) ... 41
15.6 Modes Framework ... 46
15.7 Media Framework ... 48
15.8 Data Types ... 53
15.9 Querying Expression Syntax Object Notation ... 57
15.10 Security Framework ... 61
Annex A Bibliography (Informative) ... 66
Annex B Glossary (Normative) ... 67
Annex C Complete MDCF UML® (Normative) ... 69
Annex D MDCF IDL (Informative) ... 70

Foreword

SMPTE (the Society of Motion Picture and Television Engineers) is an internationally-recognized standards developing organization. Headquartered and incorporated in the United States of America, SMPTE has members in over 80 countries on six continents. SMPTE's Engineering Documents, including Standards, Recommended Practices, and Engineering Guidelines, are prepared by SMPTE's Technology Committees. Participation in these Committees is open to all with a bona fide interest in their work. SMPTE cooperates closely with other standards-developing organizations, including ISO, IEC and ITU.

SMPTE Engineering Documents are drafted in accordance with the rules given in its Standards Operations Manual.

SMPTE ST 2071-1 was prepared by Technology Committee 34CS.

Intellectual Property

At the time of publication no notice had been received by SMPTE claiming patent rights essential to the implementation of this Engineering Document. However, attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. SMPTE shall not be held responsible for identifying any or all such patent rights.

Introduction

This section is entirely informative and does not form an integral part of this Engineering Document.

Since the inception of media devices there has been a continual need for a standardized means of controlling them. This need led to the initial creation of protocols such as de-facto, manufacturer based, serial RS-422 control on a 9-pin “D” connector, and later VDCP. However, unfortunately as technologies advanced and media devices became attached to Internet Protocol networks, these methods were replaced with proprietary solutions. This Media Device Control suite of standards is intended to address this issue and deliver the same level of interoperability as its predecessors, using Internet Protocol, with provisions for extensibility and adaptability. Both device and media control standardization are included in this document. This suite of standards presents media in a fashion similar to that of a POSIX file system, allowing media to be searched and manipulated without regard to its physical location.

This document contains the specification of the core Media Device Control Framework (MDCF) and is Part 1 of a series of documents that define the complete Media Device Control over IP specification. All subsequent documents describe applications of or extensions to this framework, such as the wire protocols and/or additional device interfaces.

The diagrams below depict how a client interacts with the MDC system to play media. Figure 1 – Using MDC Directories to Play Media illustrates the flow a client would follow to search for devices and media, while Figure 2 – Directly Connecting to the Play device depicts a client that has implicit knowledge about a device and accesses the device directly, without the use of the directories.

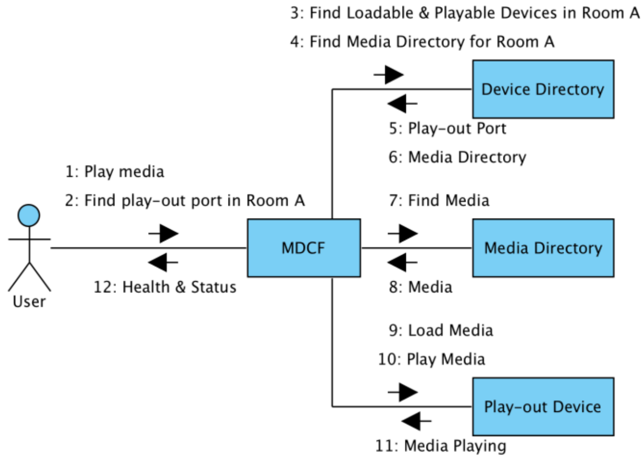

Figure 1 – Using MDC Directories to Play Media

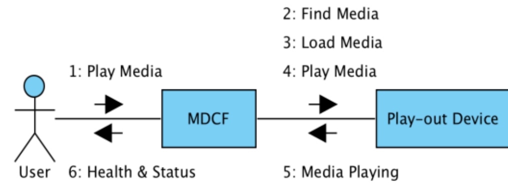

Figure 2 – Directly Connecting to the Play Device

1 Scope

This document is Part 1 of a series of documents that specify the concepts, data structures and operations required to control modern media devices. This document presents a platform agnostic model that can in turn be adapted to any protocol, platform and/or architecture, for the purpose of machine level control of media devices on Internet Protocol networks. Further documents in this series will supply protocol and platform specific adaptations of this model.

The Media Device Control (MDC) suite of standards addresses the low level, atomic operations needed to control media devices over the Internet Protocol, in a deterministic, low-latency manner. MDC is designed to bridge the gap between workflow level interfaces and the physical hardware and is intended for use on private networks, with sufficient bandwidth and bounded latency. While the control of devices over the Internet may be inherently supported, it is not recommended for low-latency applications, due to the unpredictable nature of public, general-purpose, networks.

2 Conformance Notation

Normative text is text that describes elements of the design that are indispensable or contains the conformance language keywords: "shall", "should", or "may". Informative text is text that is potentially helpful to the user, but not indispensable, and can be removed, changed, or added editorially without affecting interoperability. Informative text does not contain any conformance keywords.

All text in this document is, by default, normative, except: the Introduction, any section explicitly labeled as "Informative" or individual paragraphs that start with "Note:”

The keywords "shall" and "shall not" indicate requirements strictly to be followed in order to conform to the document and from which no deviation is permitted.

The keywords, "should" and "should not" indicate that, among several possibilities, one is recommended as particularly suitable, without mentioning or excluding others; or that a certain course of action is preferred but not necessarily required; or that (in the negative form) a certain possibility or course of action is deprecated but not prohibited.

The keywords "may" and "need not" indicate courses of action permissible within the limits of the document.

The keyword “reserved” indicates a provision that is not defined at this time, shall not be used, and may be defined in the future. The keyword “forbidden” indicates “reserved” and in addition indicates that the provision will never be defined in the future.

A conformant implementation according to this document is one that includes all mandatory provisions ("shall") and, if implemented, all recommended provisions ("should") as described. A conformant implementation need not implement optional provisions ("may") and need not implement them as described.

Unless otherwise specified, the order of precedence of the types of normative information in this document shall be as follows: Normative prose shall be the authoritative definition; Tables shall be next; followed by formal languages; then figures; and then any other language forms.

3 Normative References

The following standards contain provisions that, through reference in this text, constitute provisions of this standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this standard are encouraged to investigate the possibility of applying the most recent edition of the standards indicated below.

IETF RFC 3986, Uniform Resource Identifiers (URI): Generic Syntax

IETF RFC 2141, Uniform Resource Name (URN): URN Syntax

IETF RFC 1737, Functional Requirements for Uniform Resource Names

IETF RFC 1738, Uniform Resource Locator (URL)

IETF RFC 2046, Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types

IETF RFC 5234, Augmented BNF for Syntax Specifications: ABNF

ISO/IEC 19505-1,Object Management Group® Unified Modeling Language® (UML®), Infrastructure

ISO/IEC 19505-2,Object Management Group® Unified Modeling Language® (UML®), Superstructure

ISO/IEC 8601:2004, Date Elements and Interchange Formats — Information Interchange — Representation of Dates and Times

SMPTE ST 330:2011, Unique Material Identifier (UMID)

SMPTE RP 205:2009, Application of Unique Material Identifiers in Production and Broadcast Environments

4 Identity

The unique identification of resources is a cornerstone to any network based device or media control system. Each resource within the system shall be addressed with a persistent identifier and those identifiers shall provide enough information so that each resource is uniquely identified. The Media Device Control (MDC) Framework defines an identity scheme that is based on the Uniform Resource Name (URN) as defined by the Internet Engineering Task Force's RFC 1737 and RFC 2141.

4.1 Media Device Control Resource Name

Media Device Control Resource Names are Uniform Resource Names (URNs) defined by the MDCF to persistently and portably identify specific system resources, such as devices, namespaces and media. All Media Device Control Resource Names shall share the same format and may be used to wrap existing identity schemes or to create new ones. Media Device Control Resource Names shall not imply the availability of the identified resource, but shall be used in conjunction with directories to identify, locate, access, and control the identified resources.

The following ABNF grammar defines the Media Device Control Resource Name syntax. Please refer to RFC 5234 for the definition of the ABNF language.

```
;start ABNF notation
NAMESPACE ":" NAME_TYPE ":" [SCOPE] ":" [IDENTITY_ATTRIBUTES]

NAMESPACE = "urn:smpte"
NAME_TYPE = 1*(DIGIT / ALPHA)
SCOPE = NAME *("." NAME)
IDENTITY_ATTRIBUTES = ATTRIBUTE *(("," / ";") ATTRIBUTE)
ATTRIBUTE = NAME ["=" VALUE]
NAME = 1*(DIGIT / ALPHA / "+" / "-" / "_" / SP)
VALUE = [DQUOTE] 1*(SP / "!" / %23-%2B / %2D-%3A / "<" / %3E-%7E / ESCAPE_SEQUENCE) [DQUOTE]
ESCAPE_SEQUENCE = "%" 2HEXDIG
;end ABNF notation
```

4.1.1 Namespace

Media Device Control Resource Names shall be assigned to and arranged by namespace. The namespace shall be the text of the Media Device Control Resource Name following the URN scheme ("urn:"), up to and including the last colon (":"). Namespaces are used to help depict the scope, functional arrangement and relationship of resources within the MDCF and group resources by system, sub-system and/or purpose. Each namespace shall have a parent namespace and may have zero or more child namespaces, forming a genealogy or hierarchy of namespaces. The namespace hierarchy is separate from but related to the physical arrangement of the system, reflecting the functional arrangement of the system resources and their ability to interact.

Some fundamental assumptions are made regarding the assignment and management of namespaces in the MDCF.

1. Namespaces are assigned to sub-systems, with each sub-system having its own namespace.
2. Namespaces are assigned by workflow, purpose and/or usage.
3. Resources in the same namespace share the same purpose and work together to fulfill that purpose.
4. Resources in the same namespace can interact and connect directly with one another.

Given the previous assumptions, the following rules apply.

1. If the purpose or workflow of a resource changes, the namespace of that resource shall change.
2. If a resource is moved to another sub-system, the namespace of that resource shall change.
3. Changes to the namespace assignment of a resource shall change the identity of that resource within the MDCF, even if the vendor specific identity of that resource remains the same.

4.1.1.1 Scope

A Scope shall be a collection of resources, arranged in a logical fashion based on the vendor, workflow and/or purpose of the resources and may contain the location information, if the distances between the resources is large enough to impact the latency or deterministic nature of the network communications. Each Media Device Control Resource Name shall contain a Scope depicted by a period ["."] delimited list of name parts defined in a manner that is meaningful to both humans and the MDCF. The period delimitation was chosen to simplify the parsing of the Scope by differentiating it from the rest of the URN namespace.

Example Scopes

```
company
company.east
company.east.ingest
company.east.ingest.vendor
company.east.ingest.vendor.1
```

4.1.2 Identity Attributes

The MDCF identity scheme shall provide a space for vendor specific information, known as identity attributes. The identity attributes shall be appended to the namespace of the Media Device Control Resource Name as a comma (",") or semi-colon (";") delimited list of name/value pairs, with the name of the identity attribute and the value being separated by an equals ("=") sign. The identity attributes allow Media Device Control Resource Names to wrap existing identity schemes without the need for external mapping mechanisms. There are no restrictions to the contents of an identity attribute, provided those contents can be represented as a string of characters.

4.2 Uniform Device Name (UDN)

The Uniform Device Name (UDN) shall be a Media Device Control Resource Name of type “udn”, with “udn” being a new sub-namespace of the SMPTE URN namespace (“urn:smpte”) that uniquely identifies devices within the MDCF.

Example UDNs

urn:smpte:udn:Subsystem.Name:attribute1=value;attribute2=value
urn:smpte:udn:company.ingest.1:
urn:smpte:udn:company.ingest.1:id=12345
urn:smpte:udn:company.ingest.1:server=12345;channel=1

The following ABNF grammar defines the Uniform Device Name syntax. Please refer to RFC 5234 for the definition of the ABNF language.

```
;start ABNF notation
NAMESPACE “:” NAME_TYPE “:” [SCOPE] “:” [IDENTITY_ATTRIBUTES]

NAMESPACE = “urn:smpte”
NAME_TYPE = “udn”
SCOPE = NAME *(“.” NAME)
IDENTITY_ATTRIBUTES = ATTRIBUTE *((“,” / “;”) ATTRIBUTE)
ATTRIBUTE = NAME [“=” VALUE]
NAME = 1*(DIGIT / ALPHA / “+” / “-“ / “_” / SP)
VALUE = [DQUOTE] 1*(SP / “!” / %23-%2B / %2D-%3A / “<” / %3E-%7E) [DQUOTE]
ESCAPE_SEQUENCE = “%” 2HEXDIG
;end ABNF notation
```

4.3 Uniform Service Name (USN)

The Uniform Service Name (USN) shall be a Media Device Control Resource Name of type “usn”, with “usn” being a new sub-namespace of the SMPTE URN namespace (“urn:smpte”) that uniquely identifies Services within the MDCF.

4.3.1 Type Identity Attribute

The type identity attribute shall be a required identity attributed of the USN and shall be used to specify whether the USN represents a service template or a service instance. The type identity attribute shall be denoted with the name “type” and its value shall specify one of the enumerated types listed in Table 1.

Table 1 – USN Types

USN Type	Description
template	Indicates that the USN represents a Service Template
instance	Indicates that the USN represents a Service Instance

Example USNs

urn:smpte:usn:Subsystem.Name:type=type;attribute1=value;attribute2=value
urn:smpte:usn:company.task.1:
urn:smpte:usn:company.task.1:type=instance;uuid=12345678-9abc-def0-1234-56789abcdef0
urn:smpte:usn:company.task.1:type=template;uuid=12345678-9abc-def0-1234-56789abcdef0

The following ABNF grammar defines the Uniform Service Name syntax. Please refer to RFC 5234 for the definition of the ABNF language.

```
;start ABNF notation
NAMESPACE ":" NAME_TYPE ":" [SCOPE] ":" [IDENTITY_ATTRIBUTES]

NAMESPACE = "urn:smpte"
NAME_TYPE = "usn"
SCOPE = NAME *("." NAME)
IDENTITY_ATTRIBUTES = TYPE *(("," / ";") ATTRIBUTE)
TYPE = "type=" VALUE
ATTRIBUTE = NAME ["=" VALUE]
NAME = 1*(DIGIT / ALPHA / "+" / "-" / "_" / SP)
VALUE = [DQUOTE] 1*(SP / "!" / %23-%2B / %2D-%3A / "<" / %3E-%7E) [DQUOTE]
ESCAPE_SEQUENCE = "%" 2HEXDIG
;end ABNF notation
```

4.4 Uniform Media Name (UMN)

The Uniform Media Name (UMN) shall be a Media Device Control Resource Name of type "umn", with "umn" being a new sub-namespace of the SMPTE URN namespace ("urn:smpte") that uniquely identifies media within the MDCF. UMNs shall contain a type identity attribute, identified by the name "type", that indicates the type of media represented by the UMN and may contain a Media Identifier (MID) identity attribute, identified by the name "mid", that depicts the media's association to other media within the system. The UMN may also contain a Material Identifier (MAID) identity attribute, identified by the name "maid", that depicts the media's association to other material within the system

Example UMNs

urn:smpte:umn:Uniform.Namespace:type=media_type;mid=value;mid=value;attribute =value
urn:smpte:umn:company.location.ingest.1:
urn:smpte:umn:company.location.nearline:type=instance;mid=[*MID*];maid=[*MAID*]
urn:smpte:umn:company.location.nearline:type=instance;mid=[*MID*];maid=[*MAID*];path=/Dir/file1.mxf
urn:smpte:umn:company.location.database:type=asset;mid=[*MID*]

The following ABNF grammar defines the Uniform Media Name syntax. Please refer to RFC 5234 for the definition of the ABNF language.

```
;start ABNF notation
NAMESPACE ":" NAME_TYPE ":" [SCOPE] ":" [IDENTITY_ATTRIBUTES]

NAMESPACE = "urn:smpte"
NAME_TYPE = "umn"
SCOPE = NAME *("." NAME)
IDENTITY_ATTRIBUTES = TYPE *(("," / ";") (ATTRIBUTE / MID / MAID))
TYPE = "type=" VALUE
ATTRIBUTE = NAME ["=" VALUE]
NAME = 1*(DIGIT / ALPHA / "+" / "-" / "_" / SP)
VALUE = [DQUOTE] 1*(SP / "!" / %23-%2B / %2D-%3A / "<" / %3E-%7E) [DQUOTE]
ESCAPE_SEQUENCE = "%" 2HEXDIG

MID = "mid=" VALUE
MAID = "maid=" VALUE
;end ABNF notation
```

4.4.1 Type Identity Attribute

The type identity attribute shall be a required identity attributed of the UMN and shall be used to specify the type of the media represented by the UMN. The type identity attribute shall be denoted with the name "type" and its value shall specify one of the enumerated media types listed in Table 2.

Table 2 – Media Types

Media Type	Description
media_asset	Indicates that the media is a Media Asset.
material_asset	Indicates that the media is a Material Asset.
media_container	Indicates that the media is a Media Container
media_file	Indicates that the media is a Media File
media_instance	Indicates that the media is a Media Instance
media_bundle	Indicates that the media is a Media Bundle

4.4.2 Media Identifier (MID)

The Media Identifier (MID) shall identify unique collections of media (pictures and sounds), regardless of the format or quality of the content. The MID shall not uniquely identify material, instances or files, but shall identify the logical concept of a unique collection of media and shall facilitate the grouping of like media, such as the association of proxy/browse media to its broadcast and contribution quality counterparts. The MID identity attribute shall be denoted with the name “mid” and shall be present for UMNs identifying Media Assets and all of the Media Asset sub-types.

4.4.3 Material Identifier (MAID)

The Material Identifier (MAID) shall be a locally created, locally managed identifier that identifies a unique collection of audio-visual material. The MAID shall not uniquely identify media instances or files, but shall identify the grouping of like instances or files. One or more media instances may share the same MAID, provided that each instance contains “identical audio-visual essence”, where “identical audio-visual essence” refers to essence that is pixel-for-pixel identical during base-band play out. The MAID is identical in purpose and function to the SMPTE Unique Material Identifier (SMPTE UMID) and a SMPTE UMID may be used as a MAID. Please refer to SMPTE ST 330 and SMPTE RP 205 for a detailed description of the SMPTE UMID, its generation, depiction, and for a detailed explanation of a Material Identifier. The MAID identity attribute shall be denoted with the name “maid” and shall be present for UMNs identifying Material Assets and all of the Material Asset sub-types.

4.5 Uniform Capability Name (UCN)

The Uniform Capability Name (UCN) shall be a Uniform Resource Name (URN) of type “smpte:ucn”, with “ucn” being a new sub-namespace of the SMPTE URN namespace (“urn:smpte”), that uniquely identifies Capability Interfaces. The UCN shall contain a name and a version and should be assigned an interface namespace. The UCN version may also specify one or more levels of revision.

In many programming languages, namespaces provide context for identifiers and/or names. The interface namespace field of the UCN shall indicate the namespace of the Capability Interface identified by the UCN. The interface name shall be unique within the context of the assigned interface namespace. An interface namespace may be vendor specific or it may be defined by a Standards Development Organization (SDO), such as SMPTE. Each Capability Interface shall be assigned a “practically” globally unique UCN.

Example UCNs

urn:smpte:ucn:org.smpte.st2071:playable_v1
urn:smpte:ucn:org.smpte.st2071:playable_v1.0.0
urn:smpte:ucn:com.some_company.playout:playable_v1.0

The following ABNF grammar defines the Uniform Capability Name syntax. Please refer to RFC 5234 for the definition of the ABNF language.

```
;start ABNF notation
# UCN ABNF Syntax
NAMESPACE ":" NAME_TYPE ":" [INTERFACE_NAMESPACE] ":" [INTERFACE "_v" VERSION *("." REVISION)]

NAMESPACE = "urn:smpte"
NAME_TYPE = "ucn"
INTERFACE_NAMESPACE = NAME *("." NAME)
INTERFACE = NAME
VERSION = 1*DIGIT
REVISION = 1*DIGIT
NAME = 1*(DIGIT / ALPHA / "+" / "-" / "_" / SP)
;end ABNF notation
```

4.6 Namespace Names

Namespace names shall be Media Device Control Resource Names that are comprised only of the URN scheme ["urn:"] and the URN namespace; no additional content shall appear after the last colon (":"). Namespace names shall be used by the MDCF to construct the namespace hierarchy and to facilitate the routing of requests to the appropriate sub-systems or directories. Please note that the namespace name of any Media Device Control Resource Name can be determined by stripping the identity attributes from the name.

The ABNF grammar that defines the Namespace Name syntax is outlined below. Please refer to RFC 5234 for the definition of the ABNF language.

```
;start ABNF notation
NAMESPACE ":" NAME_TYPE ":" [SCOPE] ":"

NAMESPACE = "urn:smpte"
NAME_TYPE = ("umn" / "udn" / "usn" / "ucn")
SCOPE = NAME *("." NAME)
NAME = 1*(DIGIT / ALPHA / "+" / "-" / "_" / SP)
;end ABNF notation
```

Example Namespace UCNs
urn:smpte:ucn:company:
urn:smpte:ucn:company.ingest:

Example Namespace UDNs
urn:smpte:udn:company:
urn:smpte:udn:company.ingest.1:
urn:smpte:udn:company.east.ingest.1:

Example Namespace UMNs
urn:smpte:umn:company:
urn:smpte:umn:company.ingest:1:
urn:smpte:umn:company.east.ingest.1:

Example Namespace USNs
urn:smpte:usn:company:
urn:smpte:usn:company.ingest:1:
urn:smpte:usn:company.east.ingest.1:

4.6.1 Root Namespace Names

Root namespaces shall be the root most namespaces afforded to the identity scheme. There shall be a root namespace name for each of the Media Device Control Resource Name types (UCN, UDN, USN, and UMN) and each namespace shall descend from the root namespace of its respective type. Root namespace names shall be depicted as namespace names with no Scope value, containing an empty Scope string and therefore shall end with a double colon "::". The root namespace name for the Uniform Device Name (UDN) shall be "urn:smpte:udn::", the root namespace name for the Uniform Service Name (USN) shall be "urn:smpte:usn::", the root namespace name for the Uniform Capability Name (UCN) shall be "urn:smpte:ucn::", and the root namespace name for the Uniform Media Name (UMN) shall be "urn:smpte:umn::".

4.6.2 Determining the Parent Namespace Name

Each namespace descends from a parent namespace, up to the root namespace. To determine the parent namespace, the text beginning with the last period of the Scope, up to the last colon [":"] of the namespace name shall be removed. If no period ["."] is present within the Scope, the entire Scope shall be removed, leaving the root namespace. The pseudo code describing the algorithm used to extract the parent namespace name from any given Media Device Control Resource Name is depicted below.

```
rn = "urn:smpte:udn:scope.sub_scope:attr1=a,attr2=b,attr3=c";
namespace_index = rn.indexOf(0, ':') + 1;
type_index = rn.indexOf(namespace_index, ':') + 1;
scope_index = rn.indexOf(type_index, ':') + 1;
attribute_index = rn.indexOf(scope_index, ':') + 1;
scheme = rn.substring(0, namespace_index);  // "urn:"
urn_namespace = rn.substring(namespace_index, type_index);  // "smpte:"
name_type = rn.substring(type_index, scope_index); // "udn:", "usn:", or "umn:"
scope = rn.substring(scope_index, rn.indexOf(scope_index, ':'));
parent_scope = scope.substring(0, scope.lastIndexOf('.'));
parent_namespace = scheme + urn_namespace + name_type + parent_scope + ":";
```

5 Directories

Directories are software services that facilitate the searching and lookup of systemic resources. Directories may represent resources as a doubly linked hierarchical tree, known as a resource hierarchy. Directories expose the operations necessary to lookup, list, and search for the resources they contain. Directories may act as simple wrappers around existing vendor systems, converting the Media Device Control Protocol into the proprietary protocol understood by the vendor system or they may be self-contained registries, managing the resources and resource hierarchy internally. Directories may also aggregate multiple child directories into a single, consolidated view, acting as a centralized view into a distributed repository.

5.1 Registries

Registries facilitate the manipulation of the resources and the resource hierarchy through the process of dynamic registration and de-registration. Resources are added to the registry through a process known as binding and once bound, a resource may be unbound. Unbinding resources removes all references and information pertaining to the unbound resource from the registry. Resources may also be marked as offline, indicating that they are still present within the registry, but may be temporarily unavailable or are not ready for operation. Registries shall retain all references and information pertaining to bound resources until those resources are unbound.

5.2 Load Balancing and Fault Tolerance

Load balancing and fault tolerance are facilitated through the use of redundant directories that represent intersecting sets of namespaces. The means by which changes to the resources and/or the resource hierarchy are communicated across directory instances is determined by the implementation and is out of the scope for this document. To prevent inconsistent results/return values, the changes to the resources and/or the resource hierarchy shall not be made available to operation executions or attribute reads that are executing before or at the time the changes are made. The changes to the resources and/or the resource hierarchy shall be made

available to operation executions and attribute reads that are executed after the changes have been successfully committed to the load-balanced set. To prevent data inconsistencies, changes should be committed to a load-balanced set in a manner conducive to:

1. A change request R_x to modify resources and/or the resource hierarchy is received by directory D_x.
   a. R_x is ignored if it has already been applied to or has already been received by D_x.
2. The changes in R_x are applied to D_x, but are not available to operation executions and attribute reads.
3. Directory D_x broadcasts the change request R_x, as R_{x+1}, to each directory within the load-balanced set to which D_x belongs, excluding those directories that have already received R_x.
   a. Each directory D_{x+1} to which R_{x+1} was broadcast begins execution at step 1.
4. The changes in R_x are committed to the directory D_x upon the successful committing of the changes R_{x+1} to the directories D_{x+1}, making the changes available to operation executions and attribute reads.
5. Directory D_x broadcasts one or more signals to all registered listeners notifying them of changes to the resources and/or resource hierarchy contained within D_x.

6 Signals

Signals are one-way, asynchronous, communications between one or more nodes, used to notify one another of changes to their internal state. Signals always have a point of origin, known as the sender, and may have zero or more recipients, known as listeners. Signals are used to broadcast packets of information to interested parties in an asynchronous fashion, commonly referred to as messages or events. Listeners are not required to confirm the receipt of such messages or events, but guaranteed delivery can be specified using an acknowledgement-based protocol or by the wire-level transport protocol used to transmit the signals.

6.1 Polling

Polling is the process by which a client application, or device, executes an operation on an event generating device, or service, to retrieve the events queued for that client. Polling is most effective if communications between the sender and the receiver are limited to a single direction, such as when a firewall or router is present within the network communications path.

6.2 Asynchronous Callback

An Asynchronous Callback is when an event sender executes an operation on a remote listener to deliver events asynchronously. The Asynchronous Callback method of event delivery is more efficient and scalable than the Polling method. However, listener must implement a remotely executable service and bi-directional communications must be permitted between the sender and the listener, with the sender being able to open connections on and/or send unsolicited packets to the listener.

7 Capability-Based Programming

Capability-based programming is a concept that is closely related to Interface-based programming; that both extends and restricts the basic principles of Interface-based programming to better facilitate the requirements of device control and the “Internet of Things” (IoT). Capability-based programming enhances modularity by encouraging interfaces to represent only the minimal behavior required to implement a concise feature or capability, e.g., “Read Sensor Integer”, “Write Sensor Integer”, “Read State”, “Play”, “Pause”, or “Stop”. Devices, services, and objects expose themselves to the network using a set of independently accessible capability interfaces, without a single interface that consolidates the capability interfaces into a single, aggregated, view. This interface independence allows devices, services, and objects to dynamically change their characteristics based on their state, mode, and environmental factors, such as the presence of other devices, services, or objects within the network. Capability-base programming allows for complex behaviors to be modeled through the use of smaller building blocks, in much the same way as building materials are used to construct buildings.

The following list enumerates the interface requirements of Capability-based programming:

1. A “practically” globally unique identifier shall uniquely identify each interface.
2. Each interface shall specify the authoritative source that maintains the definition of the interface. The authoritative source may be specified as part of the interface’s unique identity.
3. Each interface endpoint shall be independently accessible from all other interfaces exposed by the device, service, or object.
4. Each interface should define a contract of behavior for a concise feature, function, or capability; implementing the minimal set operations, attributes, and signals required to implement a concise feature or capability.
5. The documentation and programmatic artifacts for each interface should be made available and obtainable, over the Internet, using the interface’s unique identity.
6. Interfaces meeting the interface requirements, above, shall be referred to as capability interfaces.

The following list enumerates the Service, device, and object requirements of Capability-based programming:

1. A “practically” globally unique identifier shall uniquely identify each device, service, and object.
2. Devices, services, and objects shall implement one or more capability interfaces.
3. Devices, services, and objects shall only be accessed via their capability interfaces.
4. Devices, services, and objects shall provide a means by which the set of exposed capability interfaces can be identified, iterated, and/or listed.
5. Devices, services, and objects may change the capability interfaces they expose, on the fly, at runtime, to support different “modes of operation.”

7.1 Capability Interfaces

MDC Capability Interfaces, hereafter referred to as “Capability Interfaces”, are capability interfaces used within the confines of the MDCF. Each Capability Interface shall be assigned a unique Uniform Capability Name (UCN) and the UCN shall depict the name, version, revision, and namespace in which the interface is defined and managed. Capability Interfaces shall be independently accessible from other Capability Interfaces, with clients being able to be directly access each Capability Interface independent from any other Capability Interface exposed by the implementing device or service.

7.1.1 Attributes

Attributes are read-only, named, properties that reflect the state of the software service or device implementing the Capability Interface to which the attributes are bound. Accessing or reading attributes shall not alter the state of the software service or device and reading their value should be lightweight, requiring minimal system resources (CPU cycles, memory, network bandwidth, etc.). Attributes in this context should not be confused with the “Attributes” attribute associated to the Device and Service Capability Interfaces, which contain named values associated to the service or device instance.

7.1.2 Operations

Operations are commands bound to an interface that perform actions and may alter the state of the software service or device implementing the Capability Interface to which the operations are bound. Operations shall define the actions that the Capability Interface can perform and can be characterized as procedures, functions, methods or subroutines. Operations shall accept zero or more inputs, known as parameters, and may produce zero or one output, known as a result or return value.

7.1.3 Inheritance

A Capability Interface may extend the behavior of zero or more other Capability Interfaces. These Capability Interfaces are said to “inherit” the behavior of the Capability Interfaces that they extend and the Capability Interfaces that are extended are commonly referred to as “parent” interfaces. The inheritance model allows for a hierarchical relationship of behavior in which the child most interfaces implement the full behavior of their entire set of parent interfaces, plus the behavior they themselves define.

This can be expressed mathematically as:

$$A = A \cup B \cup B_P \cup C$$

Where A is the child most interface, B and C are interfaces extended by A, and B inherits from interface B_P.

8 Services

A Service is a software service that exposes one or more Capability Interfaces. The Service Capability Interface represents a software service within the MDCF and shall contain the information pertaining to the identity, type, attributes, and capabilities exposed by a software service. Each Service may be discoverable via the Service Directories representing the namespace in which the Service instance is bound and shall be discoverable by any means in which the Service Factory, that created the Service instance, is discoverable. Upon its destruction, see Service Factory below, a Service shall be removed from all Service Directories and shall no longer be discoverable by any means.

8.1 Service Factory

A Service Factory is a software service that manages the life cycle of other software services. The Service Factory Capability Interface represents a Service Factory within the MDCF and should be used to create and destroy Service instances. A Service Factory shall contain the list of Service Templates describing the Service types which it can create and shall be able to destroy / terminate any and all Service instances in which it has created. A Service Factory shall expose Service instances it creates via one or more Service Directories and shall make the Service instance it has created discoverable by any mean in which the Service Factory itself is discoverable.

8.2 Service Directory

The Service Directory is a type of directory that exposes Services. It is one of the main entry points defined by the MDCF and represents executing services made available by the MDCF. Each Service Directory manages and represents a set of namespaces, their child namespaces, and all of the Services assigned to those namespaces. Since Service Directories are also Services themselves, a Service Directory may contain references to other Service Directories. Service Directories should be used to translate Uniform Service Names into URLs, providing the means by which the exposed Service instances may be accessed.

8.3 Service Registry

The Service Registry is a Capability Interface that allows Service instances to be bound and unbound dynamically to one or more Service Directories, at run-time. Service Registries may be assigned to one or more Service Directories, allowing for the simplified management of the Service instances across load-balanced sets of Service Directories. Service Factories should utilize Service Registries for the binding and unbinding of Service instances, if the Service Factory and the Service Directory by which the Service instances are created and exposed are not the same software service instance or device.

8.4 Service Templates

The Service Templates describes the capabilities and attributes describing distinct types of software services. Service Templates are analogous to classes within the Object Oriented Programing paradigm, with the addition of static attributes that describe the characteristics of the Service Template that cannot be depicted by Service behavioral depiction (capabilities) alone. A Service Template shall contain attributes and a list of Capability

Interfaces that are implemented by Services instances of that type and each Service Template shall be assigned a unique USN. The USN assigned to the Service Template shall specify the name, the Scope, and the version of that Service Template and the USN shall specify a “type” identity attribute with a value of “template”. The attributes associated to the Service Template and the attributes associated to the Capability Interfaces defined within the Service Template shall be applicable to all Service instances of that type. The Capability Interfaces associated to a Service Template shall not specify URL values, as the Service Templates do not describe Service instances and therefore do not have anything that can be accessed.

9 Devices

A Device is a software service that represents logical or physical hardware. A unique Uniform Device Name (UDN) shall identify each Device within the MDCF and each Device shall have one or more URLs providing the information needed to connect to the Device interface. Each Device shall implement one or more Capability Interfaces and each Capability Interface shall having a dedicated list of URLs, providing the information needed to connect to the Capability Interface, independently of any other interface exposed by the Device. Attributes shall be associated to the Device and each Capability Interface assigned to the Device. These Device and Capability Interface attributes shall describe specific characteristics that constrain the Device and/or its Capabilities (e.g. the number of supported audio channels, the location name, the workflow name). A Device or Capability Interface may be accessed directly, without the use of a Device Directory, if the client has prior knowledge of one or more of the Device or Capability Interface URLs. The Device interface allows clients to discovery the Capability Interfaces supported by the Device and its attributes, name, and additional methods by which to connect to the Device interface (URLs). Information pertaining to the resource hierarchy, beyond the Scope specified in the UDN, shall be acquired from a Device Directory.

9.1 Device Directory

The Device Directory is a type of directory that exposes Devices. It is one of the main entry points defined by the MDCF and represents the devices within the MDCF as a hierarchical tree of devices. Each Device Directory manages and represents a set of namespaces, their child namespaces and all of the Devices assigned to those namespaces. Since the Device Directory is a Capability Interface, Device Directories may also contain references to other Device Directories, forming a hierarchy of Device Directories. The Device Directory is similar in function and purpose to a DNS or LDAP server, acting as a point of discovery and lookup. Device Directories should be used to translate Uniform Device Names into URLs for accessing the Devices they contain.

9.2 Device Registry

A Device Registry is a Capability Interface that allows Devices to be bound and unbound, at run-time, to one or more Device Directories . Device Directories that allow for this dynamic binding and unbinding of Devices shall implement the Device Registry Capability Interface. Device Registries may be used to create aggregate Device Directories that consolidate a number of directories into a single consolidated view.

9.3 Device Hierarchy

Device Directories represent the Devices within the MDCF as a doubly linked hierarchical tree, where each Device may be assigned to a parent and may be the parent of zero or more child devices. The device hierarchy shall be arranged in a fashion that represents the perceived or logical view of the underlying system(s), with the devices capable of directly interoperating with one another being assigned the same namespace or sharing the same namespace parentage. As a rule, devices should be able to directly connect to their direct parent, their parent's siblings, their siblings and their direct children.

The following list of rules applies to the hierarchy of devices within a Device Directory.

- Each Device shall be in the same namespace or a child namespace of its parent’s namespace. Devices shall extend or share the namespace of their parent.
- Each Device Directory shall represent one or more namespaces, processing requests for those namespaces.

- All Devices within the Device Directory shall share or extend one of the namespaces in which the Device Directory represents.
- Aggregate Device Directories shall delegate requests to the child Device Directory that most closely matches the namespace that is specified within the request.

9.3.1 Namespaces and Scope

Device Directories shall arrange the Devices contained within them in a hierarchy representative of the namespace hierarchy specified by the UDNs that identify the Devices, with Devices residing in the namespace of or a child namespace of their parent Device. Devices within a Device Directory shall be assigned to one of the namespaces or one of the child namespaces assigned to the Device Directory.

10 Modes

A Mode is an operational state that defines the Capabilities and behavior of a Device or Service. Each Device or Service may have multiple modes available, but shall only be in one active mode at a time. Each mode can drastically change the Capabilities and behavior of the Device and/or Service. The MDCF allows Devices and Services to support multiple modes of operation using the Mode Support Capability Interface. The Mode Support Capability Interface allows a Device and/or a Service to have a base set of Capability Interfaces that shall be available regardless of the active mode set for the Device or Service and a list of modes, with each mode containing a list of additional Capability Interfaces that are exposed by the Device or Service when that mode is activated. The mode shall be changed using the operations exposed by the Mode Support Capability Interface. The Capability Interfaces exposed by a Device or a Service shall be the union set of the list of base Capability Interfaces plus the Capability Interfaces defined for the active mode.

Supported Capability Interfaces = Base Capability Interfaces ***U*** *Active Mode Capability Interfaces*

11 Media

The MDCF defines media as audio-visual material and its ancillary information. Media can be audio-video clips, audio recordings, press releases, transcripts, captions, subtitles or any other descriptive data associated to an event that occurred in space-time; Media can also be a form of storage that contains other media. The MDCF defines six distinct media types, the Media Container, the Media Asset, the Material Asset, the Media File, the Media Instance and the Media Bundle, all extending from the base Media type.

11.1 Media Assets

Media Assets are the logical representation of unique sets of media, regardless of format or quality; stated specifically in terms of audio-visual media, Media Assets are unique collections of pictures and sound, regardless of the format or the quality of the audio-visual elements. Each Media Asset shall be assigned a unique Media Identifier (MID) and each media entity that shares the same MID shall contain the same set of media (pictures and sounds), although the format and quality of the contained media may be differ.

11.2 Material Assets

Material Assets are the logical representation of unique sets of media that are of the same quality; stated specifically in terms of audio-visual material, Material Assets are unique collections of pictures and sound that look and sound “identical” when presented or played out. Each Material Asset shall be assigned a unique Material Identifier (MAID) and each media resource sharing the same MAID shall contain “identical audio-visual content”, frames and/or samples; where “identical audio-visual content” is audio-visual material that is visually and audibly identical when presented or played out. Please refer to SMPTE ST 330 and SMPTE RP 205 for a detailed description of the SMPTE UMID and its uses for a detailed explanation of unique Material Identity.

11.3 Media Files

Media Files are physical instances of Media that do not have a temporal component (timeline) or duration. They can be press releases, still images, graphics, documents, or any virtual or physical file containing media.

11.4 Media Instances

Media Instances are Media Files that are physical manifestations of Material Assets and thus have a temporal component (timeline) and duration. A Media Instance is typically a file, but media systems may have an internal storage representation that does not produce, nor store files, until the media is egressed from that system. For this reason, Media Instances shall not specify URLs for media access or identity and the URLs shall be generated at the time the Media Instance is egressed from the system.

11.5 Media Containers

Media Containers are physical and logical entities that contain media. Media Containers can be directories on a file system or they can be play lists stored in a database. Media Containers represent the location in which media can be found, much like directories in the POSIX file system specification.

11.6 Media Bundles

Media Bundles are Media Containers that contain other Media to produce a Media Instance that is built from, and described by, the Media it contains. Media Bundles having their own temporal component (timeline) and duration that is an aggregate of their contents. For example, a POSIX directory that contains numerous media instances that are different segments of the same show can be represented as a Media Bundle. A Media Bundle can also be used to represent this POSIX directory compressed or packaged into a single file.

11.7 Media Pointers and Segments

Media is regularly composed of subsets of other media. The MDCF defines two data structures that are used to represent subsets of a Media. These structures are the Media Pointer and Media Segment.

11.7.1 Media Pointers

Media Pointers are data structures that represent subsets of media. Media Pointers shall specify the source Media, in and out point offsets, the offset types (i.e. Second, Microsecond, Byte, Frame), and the track within the source Media that describe the complete Media subset represented by the pointer. Media Pointers should be used primarily as input parameters for operations that manipulate Media.

11.7.2 Media Segments

Media Segments are data structures that fully describe subsets of Media. The Media Segment shall be an extension of the Media Pointer and the Media type and shall be used to describe the Composition of Media. The Media Segment shall define the information required, above and beyond, that defined in the Media Pointer and Media type to fully describe a section of Media, including the in and out point date-time, the format, the track name and the type for the represented section of Media. Media Segments shall be used primarily as return values from operations and attributes, but may be used as input parameters, provided the Media Segment was first queried from the MDCF. Multiple Media Segments may be used to describe the same section of a Composition timeline, for example, if a section of the Composition timeline contains essence from multiple sources or has multiple identities.

11.8 Media Directory

The Media Directory shall be the type of directory that exposes media; media can be in the form of a multimedia clip, file, directory, press release, text document, picture, graphic or even a storage partition, such as a hard disk or network attached storage array. The Media Directory shall expose the operations required to manipulate, search and list media within the system, including the ability to find all of the media instances that contain a specific subset of audiovisual material. The Media Directory is one of the main entry points defined by the MDCF and can be used in conjunction with or independently of a Device Directory. Like Device Directories, Media

Directories shall be bound to namespaces and all media within the Media Directory shall reside within one of those namespaces or one of the child namespaces

12 Query Expressions and Query Syntax

One of the primary purposes of Directories is to provide the ability to discover or search for Queryable resources. To facilitate this requirement, a simple object oriented query expression syntax shall be defined which allows for basic Boolean expressions, such as AND, OR, LESS THAN, GREATER THAN, EQUALS, MATCHES, CONTAINS and NOT. Additional domain specific operations shall also be included to facilitate searches for Devices and Services that implement specific behaviors and Media that contains subsets of other media.

Table 3 describes the syntactic elements for the Query Expression. The UML describing the Query Expression syntax object notation is defined in Diagram 15-8: Query Syntax Object Notation UML.

Table 3 – Query Expression Syntax

Expression	Description
NOT	Syntax: NOT(*expr1*) Boolean NOT operation. Inverts *expr1*, equates to TRUE if *expr1* is FALSE.
AND	Syntax: AND(*expr1, expr2*) Boolean AND operation. Equates to TRUE if both *expr1* and *expr2* are TRUE.
OR	Syntax: OR(*expr1, expr2*) Boolean OR operation. Equates to TRUE if either *expr1* or *expr2* are TRUE.
EQUALS	Syntax: EQUALS(*field, value*) Boolean EQUALS operation. Equates to TRUE if the value contained in the specified *field* is equal to the value specified in the *value* parameter.
MATCHES	Syntax: MATCHES(*field, regexp*) Regular expression MATCHES operation. Equates to TRUE if the value contained in the specified *field* matches the regular expression in the *regexp* parameter. The regular expression syntax shall be conformant to the extended regular expression grammar and rules defined in POSIX IEEE Std. 1003.1-2008-Base Definitions.
LESS_THAN	Syntax: LESS_THAN(*field, value*) Numeric LESS_THAN operation. Equates to TRUE if the value contained in the specified *field* is less than the value specified in the *value* parameter.
GREATER_THAN	Syntax: GREATER_THAN(*field, value*) Numeric GREATER_THAN operation. Equates to TRUE if the value contained in the specified *field* is greater than the value specified in the *value* parameter.
CONTAINS	Syntax: CONTAINS(*pointer*) Media centric CONTAINS operation. Equates to TRUE if the tested Media contains the essence referred to by the *pointer* parameter.

Expression	Description
IMPLEMENTS	Syntax: IMPLEMENTS([*mode*,] *capabilities*) Device and Service centric IMPLEMENTS operation. Equates to TRUE if the mode and capability signature for the tested device is a superset of the mode and capability signature specified by the *mode* and *capabilities* parameters. The *mode* parameter is optional and if specified, the device shall implement a mode that is a superset of the mode specified in the *mode* parameter. The name specified for the mode parameter may contain a regular expression conformant to the extended regular expression grammar and rules defined in POSIX IEEE Std. 1003.1-2008-Base Definitions. The capabilities parameter applies to the base capability interfaces defined for the device, base capability interfaces being the capability interfaces that are not associated to a mode.
PAGE	Syntax: PAGE(page_size, offset) Pagination information applied to the result set generated by the query expression. The *page_size* parameter indicates the number of items to include in the resulting page and the *offset* parameter specifies the numerical index value of the first item of the paged result set, starting at 1.
SORT_BY	Syntax: SORT_BY(field, descending) Sorting rules applied to the result set generated by the query expression. The *field* parameter specifies the name of the field containing the values to use while sorting the result set and the *descending* parameter indicates the direction of the sort. If *descending* is TRUE, then the result set is sorted in reverse alphabetical order, otherwise the result set is sorted in alphabetical order.

12.1 Attribute Designation

Query Expressions may be equated to the values contained within an object attribute. The Query Expression syntax provides a parameter by the name of *field* that allows for attributes to be specified by name. The attributes indicated by the field parameter may also contain attributes; the use of the dot notation “*attribute.child*” shall allow for these child attributes to be referenced.

12.1.1 Referencing Collections, Lists and Maps

Attributes may also be arrays, collections, lists or Maps of values. For these cases the simple dot notation of attribute designation is not sufficient to identify specific values stored within the attribute. To resolve this a square bracket notation “*attribute[key or index]*” shall be defined that allows for the specification of values within such collections, using either the numerical index of the value, 0 being the index of the first value in the collection, or the named key of the value, as is the case with the Map data type.

The ABNF grammar that defines the attribute designation syntax is outlined below. Please refer to RFC 5234 for the definition of the ABNF language.

```
;start ABNF notation
FIELD = NAME_PART *(“.” NAME_PART)
NAME_PART = NAME *1(“[“ (QUOTE NAME QUOTE) / DIGIT) “]”)
NAME = 1*(DIGIT / ALPHA / “-“ / “_” / SP)
QUOTE = (“\”” / “\’”)
;end ABNF notation
```

Example attribute designators:
Name
Capabilities[1]
UDN.Namespace
UMN.Attributes[“MID”]

13 Delegation of Control

Some devices are limited in the number of concurrent sessions they can support. The MDCF defines interfaces that allow clients to obtain sessions from a device in order to manage the physical resources of the device. Delegation of Control is the name assigned the process of managing these resources and the physical constraints of the device. Delegation of Control provides a workflow and the operations required to manage device control sessions and exclusive device locks. Each device shall manage its own sessions and locks.

Devices that require session management cannot have their operations executed by any client that does not belong to a session. However, since device attributes do not alter the state of a device nor does reading the attributes significantly impact the device, device attribute reads may be available to clients that have not acquired a session.

13.1 Acquiring a Session

To acquire a session the client shall send the device an acquisition request. If the device can support the session, a new session shall be created and a unique identity shall be assigned to that session. After the session is acquired the client can freely execute operations against that device. When a client is finished with the session it shall release the session. There shall be a corresponding release for every successful acquisition of a session. Figure 3 illustrates the sequence of events for the acquisition of a session.

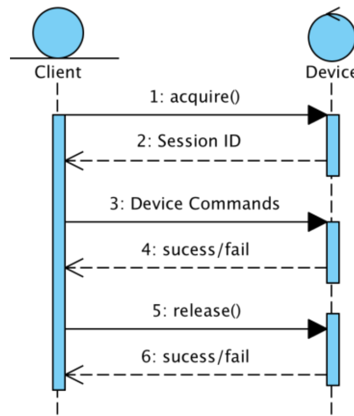

Figure 3 – Acquiring a Session

13.2 Exclusive Locking

A client may desire to acquire a device exclusively, guaranteeing that the client holds the only session to the device and is the only client that can alter the state of the device. This is called an exclusive lock and the MDCF provides a mechanism for the client to request and acquire an exclusive lock. The process of locking a device is similar to acquiring a session; however, when a device is exclusively locked no other clients may acquire sessions, locks or execute operations on the device until the exclusive lock is released. Figure 4 illustrates the sequence of events for the exclusive locking of a device.

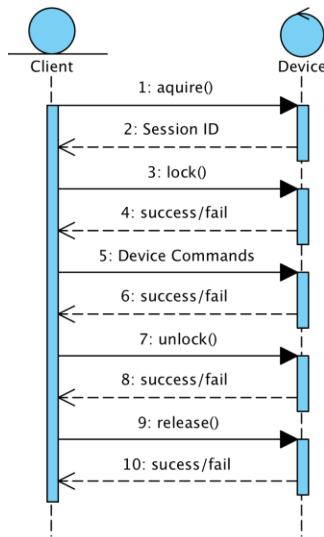

Figure 4 – Exclusive Locking

13.3 Requesting a Session

If a device runs out of sessions a client may opt to request a session from another client. To do so the requestor shall list the sessions from the device and select a session to request. A request is then sent to the session holder and the session holder may opt to accept or reject the request. If the request is not accepted or rejected in an amount of time configured for the device or if there are communication errors between the device and the session holder, the request shall automatically be accepted. Figure 5 illustrates the sequence of events to request a session from another client.

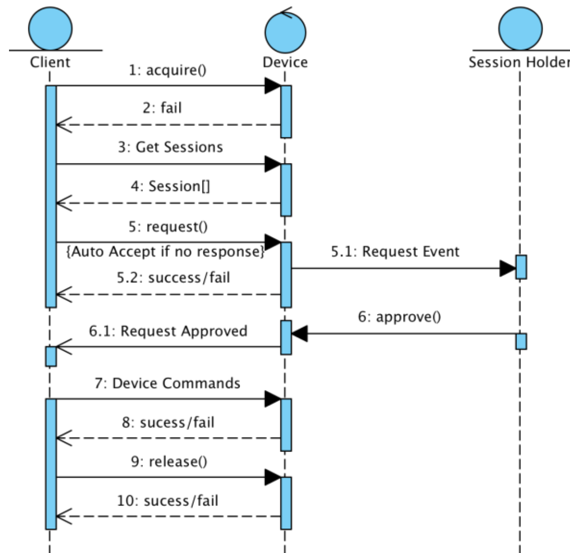

Figure 5 – Requesting a Session

13.4 Requesting an Exclusive Lock

If a client wishes an exclusive lock, but there are other client sessions or if another client has the device locked, the client may opt to request the lock from the current lock or session holders. To do so the client shall send a request for the lock to each session holder or the current lock holder. If all of the requests are accepted, the requestor is granted the exclusive lock. If a request is not accepted or rejected within an amount of time

configured for the device or if there are communication errors between the device and the session or lock holder, the request shall automatically be accepted for that session or lock. Figure 6 illustrates the sequence of events to request an exclusive lock, when other clients have session on that device.

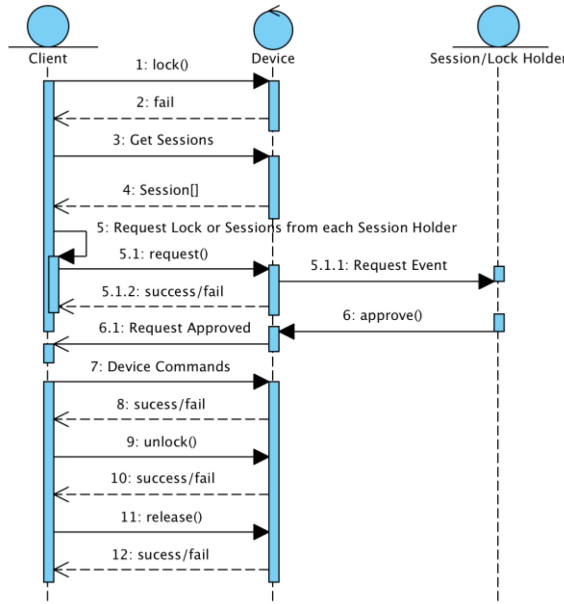

Figure 6 – Requesting an Exclusive Lock

13.5 Administration of Sessions and Locks

Device administrators manage sessions; device administrators are clients deemed to have escalated permissions on particular device or set of devices. The configuration and determination of administrators is determined by the device and outside the scope of this specification. Administrators have the ability to force acquire or force release any session or lock on a device. When a session or lock is released by any client other than the holder, an event is sent to the client notifying them that they no longer hold the session or lock and subsequent requests for that the released session or lock shall fail to execute. Figure 7 illustrates the sequence of events for forcing the release of sessions and lock.

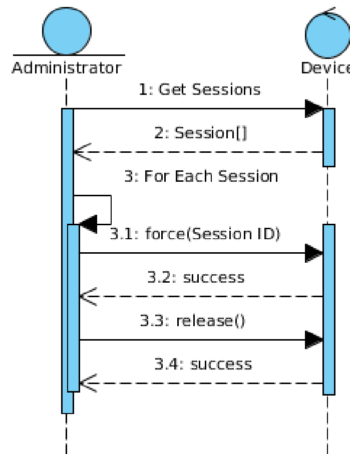

Figure 7 – Administration of Sessions and Locks

14 Authentication / Authorization

The MDCF defines a Simple Authentication and Security Layer (SASL) compliant and an OAuth compatible, role-based security framework. Please refer to RFC 4422, RFC 4752, RFC 5849, RFC 6749 and RFC 6750 for detailed information pertaining to SASL and OAuth. For greater security, the security layer is designed to facilitate rotating authorization and authentication tokens. The security framework is an optional component of the MDCF for device implementations, but is required for client implementations. If implemented, each secured resource shall at a minimum implement the Authorizer Capability Interface, while the Authenticator Capability Interface may be implemented by the resource or by a third party authentication server. The security framework defines a client as a Subject, with each Subject containing zero or more security credentials to identify that Subject. Each security credential is represented as a Principal, where a Principal is comprised of a unique identity within the specified realm and a list of zero or more security tokens. The security tokens are used by the Authenticators and Authorizers to prove that the Subject is whom they claim to be. Security tokens may be user names and passwords, digital certificates, or any other form of credential that is compliant with SASL and/or OAUTH. To ensure the greatest flexibility and level of security the following rules shall apply.

1. Clients shall authenticate if an error is raised indicating that Authentication is required.
2. Clients shall be authorized if an error is raised indicating that Authorization is required.
3. Clients shall use the Subject returned by the most recent request for the next request.
4. Clients shall re-authorize if any of the Security Tokens associated to the Subject have expired.

14.1 Authentication

Resources may support multiple authentication methods and allow Subjects to re-authenticate during a session, without invalidating the current session. When a Subject re-authenticates, the previous session and security layer are replaced with the new session and security layer. When the session is logged out, the original session is restored and a new security layer shall be negotiated. If re-authentication attempts fail, the original session and security layer remain unchanged, until the authentication failure threshold is reached, at which time all sessions and security layers for that Subject on that device are terminated (For example, 3 failed attempts to authenticate will cause the original session to be terminated).

14.2 Authentication Sequence

The Authentication sequence is a “client goes first” SASL mechanism, as defined by RFC 4422 Section 3.0. Below is an illustration of the authentication sequence, with ***C*** indicating the authentication client and ***A*** indicates the destination resource that implements authentication/authorization services.

1. Client ***C*** connects to the destination resource ***A***.
2. ***C*** issues an authorization request Rq_1 to ***A***, providing a Subject S_1 with the required Principals and Security Tokens populated, e.g. username/password, shared secret/private key, or digital certificate.
   a. ***A*** extracts Subject S_1 from request Rq_1.
   b. ***A*** validates the Identity and Security Tokens for each Principal provided by S_1.
   c. If Subject S_1 is successfully authenticated
      i. ***A*** duplicates S_1 as a new Subject S_2 and replaces its Security Tokens with authentication tokens.
      ii. ***A*** returns S_2 to ***C*** in response Rs_1.
   d. If Subject S_1 fails to authenticated
      i. ***A*** raises an error and returns the error to ***C*** in response Rs_1.
3. ***C*** stores Subject S_2.
4. ***C*** issues an authorization request Rq_2 to ***A*** using Subject S_2.
   a. ***A*** extracts Subject S_2 from request Rq_2.
   b. ***A*** validates the authentication tokens in S_2
   c. If Subject S_2 is successfully authorized
      i. ***A*** duplicates S_2 as a new Subject S_X
         1. If the authorization tokens in S_X require changing, ***A*** replaces them with updated tokens.
      ii. ***A*** returns S_X to ***C*** in response Rs_2.
   d. If Subject S_2 fails to authenticated
      i. ***A*** raises an error and returns the error to ***C*** in response Rs_2.

5. ***C*** stores Subject S_X.
6. Repeat Steps 4 and 5 to re-authorize expired Security Tokens, using S_X in place of S_2.

14.3 Security Layer

A security layer is a cryptographic layer over the base wire protocol that provides for the integrity and security of the wire level communications. Details pertaining to the implementation of the security layer are platform and protocol specific and out of the scope of this document.

14.4 Authentication Servers

The MDCF security framework allows for the use of external authentication servers. Authentication servers shall implement both of the Authenticator and Authorizer interfaces, while devices that implement the Authorizer interface, but do not implement the Authenticator interface, shall use the Security Tokens provided by the authentication servers for the establishment of the security layers. Devices that implement both the Authenticator and Authorizer interfaces may use the Security Tokens provided by authentication servers to establish security layers. The following outline defines the process that shall be used in the establishment of security layers when authentication servers are present:

1. Client ***C*** wishes to establish a security layer with destination resource ***R***.
2. ***C*** authenticates with the authentication server ***A***, using Section 14.2 Steps 1-5, specifying the URI that identifies the destination resource ***R*** for the authorization in Section 14.2 Step 4.
3. ***C*** stores Subject S_X from Section 14.2 Step 4.
4. ***C*** connects to ***R***.
5. ***C*** issues an authorization request Rq_3 to ***R*** using Subject S_X.
   a. ***R*** extracts Subject S_X from request Rq_3.
   b. ***R*** validates the authentication tokens in S_X
   c. If Subject S_X is successfully authorized
      i. ***R*** duplicates S_X as a new Subject S_{X+1}
         1. If the authorization tokens in S_{X+1} require changing, ***R*** replaces them with updated tokens.
      ii. ***R*** returns S_{X+1} to ***C*** in response Rs_3.
   d. If Subject S_X fails to authenticated
      i. ***R*** raises an error and returns the error to ***C*** in response Rs_3.
6. ***C*** stores Subject S_{X+1}.
7. Repeat Steps 4 and 5 to re-authorize expired Security Tokens, using S_{X+1} as S_X.

14.5 Permissions

Authentication permissions can be assigned to a resource, as a whole, or to specific Capability Interfaces on a resource. Different roles may be required to execute different interfaces.

15 Data and Operation Model

15.1 Identity

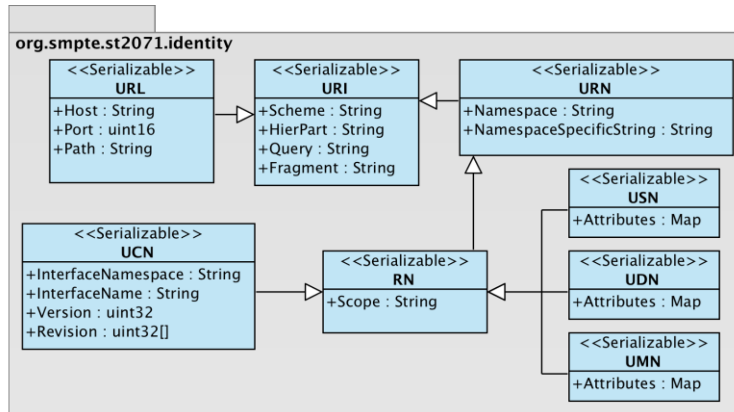

Figure 8 – Identity UML Diagram

Figure 8 illustrates the Identity model used by the MDCF. The Identifiers are Universal Resource Names (URNs) that are wrappers around the underlying system's Identity scheme and include routing information.

Namespace: org.smpte.st2071.identity

15.1.1 URI

The Uniform Resource Identifier (URI) as defined by IETF RFC 3986. A URI is a compact sequence of characters that identifiers an abstract or physical resource.

Attributes:

Name	Data Type	Description
Scheme	String	The Scheme of the URI.
HierPart	String	The hierarchical information contained within the URI, e.g. the Path specification.
Query	String	The specification of parameters, following the first question mark '?'.
Fragment	String	The specification of a sub-section within the resource referenced by the URI, following the first hash '#'.

15.1.2 URL

The Uniform Resource Locator (URL) as defined by IETF RFC 1738. A URL shall be an extension of the URI that provides the information required to access the identified resource. The attributes and operations available to the URL are platform specific and not in scope for this specification.

Attributes:

Name	Data Type	Description
Host	String	The host name or IP address specified in the URL.
Port	uint16	The port number specified in the URL.
Path	String	The path specified in the URL.

15.1.3 URN

The Uniform Resource Name (URN) as defined by IETF RFC 2141. A URN shall be an extension of the URI that identifies, but does not imply the availability of a logical or physical resource. URNs are intended to be persistent and location independent. The attributes and operations available to the URN are platform specific and not in scope for this specification.

Attributes:

Name	Data Type	Description
Namespace	String	The URN Namespace. The text between the first colon ':' and the last colon.
NamespaceSpecificString	String	The value after the Namespace. Namespace specific.

15.1.4 RN

The Resource Name (RN) shall be an extension of the URN that is used to identify resources within the MDCF. The RN shall indicate the Scope in which the identified resource resides.

Attributes:

Name	Data Type	Description
Scope	String	The Scope in which the identified resource resides.

15.1.5 UMN

The Uniform Media Name (UMN) shall be an extension of the Resource Name (RN) that identifies media in all its forms, logical or physical. The UMN shall indicate the media's type, the namespace that identifies the system in which the media resides, and when relevant, the media's MID and/or MAID.

Attributes:

Name	Data Type	Description
Attributes	Map	List of named attributes.

Attribute Keys:

Key	Data Type	Description
type	String	The media's type. Shall contain one of the values specified in Table *2 – Media* Types.
mid	String	The Media Identifier for the Media. Shall only be present for UMNs identifying Media of types media_asset, material_asset, media_instance, and media_bundle.
maid	String	The Material Identifier for the Media. Shall only be present for UMNs identifying Media of types material_asset, media_instance, and media_bundle.

15.1.6 UDN

The Uniform Device Name (UDN) shall be an extension of the Resource Name (RN) that identifies Devices. The UDN shall inherit the Scope and Attributes from the Resource Name (RN).

Attributes:

Name	Data Type	Description
Attributes	Map	List of named attributes.

15.1.7 USN

The Uniform Service Name (USN) shall be an extension of the Resource Name (RN) that identifies Services. The USN shall inherit the Scope and Attributes from the Resource Name (RN).

Attributes:

Name	Data Type	Description
Attributes	Map	List of named attributes.

Attribute Keys:

Name	Data Type	Description
type	String	The USN type. Shall contain one of the values specified in Table 1 – USN Types.

15.1.8 UCN

The Uniform Capability Name (UCN) shall be an extension of the Resource Name (RN) that globally and uniquely identifies Capability Interfaces. The UCN shall indicate the namespace, name, version, and revision.

Attributes:

Name	Data Type	Description
InterfaceNamespace	String	The namespace in which the interface is defined.
InterfaceName	String	The name of the interface.
Version	uint32	The major version number for the interface.
Revision	Uint32[]	The list of revision numbers for the interface.

15.2 Status and Events

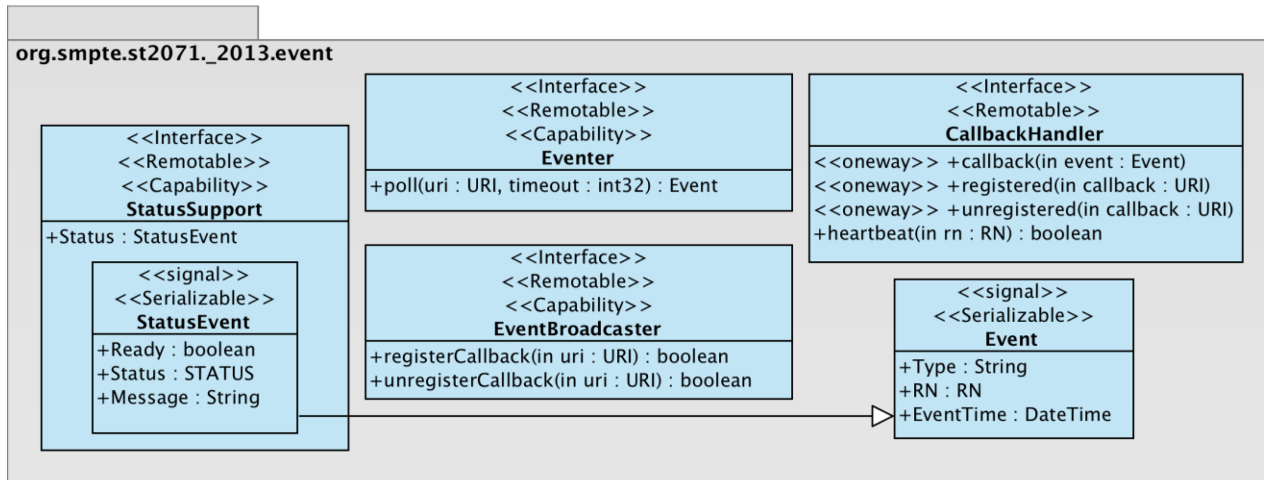

Figure 9 – Status and Events UML Diagram

Figure 9 depicts the complete UML for the MDCF representation of signaling / events.

Namespace: org.smpte.st2071.event

15.2.1 Event

The Event shall be a Serializable signal interface that is transmitted by an event broadcaster to notify listeners of a change in the event broadcaster's state.

Attributes:

Name	Data Type	Description
Type	String	The event type. A name associated to the event type. Indicates what type of state change occurred.
RN	RN	The RN that identifies the originating event broadcaster.
EventTime	DateTime	The date and time that the event was created.

15.2.2 Eventer

The Eventer shall be a Capability Interface that provides the operations required by a client to request synchronous notification of changes to the state or status of a Device or Service. The Eventer interface may be used by a client to "pull" events from the device or service through a process known as polling. Polling is the process by which a client executes an operation on an event generator and that operation shall return the next Event in a sequence of events queued for the client. If no events are available for the client, the polling operation may wait a specified amount of time before returning nothing or NULL, but shall return immediately if an event becomes or is available for the client. A URI shall identify each client and each client URI shall have a dedicated priority queue. A queue associated to a URI shall only contain events for client presenting that URI. A client may register more than one URI and multiple clients may share the same URI, for redundancy and load balancing.

UCN: urn:smpte:ucn:org.smpte.st2071.signals:eventer_v1.0

Operations:

Return Data Type	Operation / Description
Event	**poll(uri : URI, timeout : int32)** Asks the event broadcaster for the next event in the event queue for the listener identified by *uri.* If no events are available, The operation shall wait *timeout* number of milliseconds or until an event becomes available, whichever occurs first before returning. If no event is available the operation returns nothing or NULL. **Exceptions** SecurityException

15.2.3 EventBroadcaster

The EventBroadcaster shall be the Capability Interface that provides the operations required by a client to request asynchronous notifications when the state or status of a device or service changes. The client may use the EventBroadcaster Capability Interface to register a CallbackHandler. As events become available, the EventBroadcaster shall execution the “callback” operation on the client CallbackHandler interface located at the absolute URI provided during the CallbackHandler registration, passing to it the Event as the only parameter. The client shall implement the CallbackHandler interface and the URI provided in the EventBroadcaster “registerCallback” operation shall be an absolute URI, providing the information required for the EventBroadcaster to connect to and execute the “callback” operation on that CallbackHandler. Before the EventBroadcaster registers the CallbackHandler it shall validate that the provided URI is valid. The EventBroadcaster shall attempt to execute the “registered” operation on the CallbackHandler located at the provided URI. If the execution of the “registered” operation fails, the EventBroadcaster shall not register the CallbackHandler specified by the URI and the “registerCallback” operation shall raise a “Client Unreachable” error to indicate the failure. Each CallbackHandler URI shall have a dedicated priority queue, and that queue shall only contain events for the CallbackHandler represented by that URI. A CallbackHandler may register more than one URI and multiple CallbackHandlers may share the same URI, for redundancy and load balancing.

UCN: urn:smpte:ucn:org.smpte.st2071.signals:event_broadcaster_v1.0

Exceptions:

Name	Description
ClientUnreachable	Indicates that client cannot be reached via the communication path.

Operations:

Return Data Type	Operation / Description
boolean	**registerCallback(uri : URI)** Registers a CallbackHandler with the EventBroadcaster. Whenever an event occurs the event is broadcast to the registered callback handlers. The *uri* specified shall be absolute and contain the information required for to connect to the CallbackHandler. This operation shall generate a "registered" operation execution on the CallbackHandler to ensure that the CallbackHandler is reachable before successfully registering the CallbackHandler. If the "registered" operation execution fails, a "Client Unreachable" error shall be raised to indicate the error. This operation shall return TRUE if the callback handler registration is successful and FALSE if the registration fails for any reason not indicated by an error. **Exceptions** ClientUnreachable, SecurityException
boolean	**unregisterCallback(uri : URI)** Unregisters the CallbackHandler indicated by the *uri* from the EventBroadcaster. When the CallbackHandler is unregistered, the EventBroadcaster shall execute the "unregistered" operation on the CallbackHandler. The CallbackHandler shall be unregistered from the EventBroadcaster regardless of the results of the operation execution. **Exceptions** SecurityException

15.2.4 CallbackHandler

The CallbackHandler shall be a client interface, implemented by a client and executed by an EventBroadcaster to asynchronously deliver events and status signals.

UCN: urn:smpte:ucn:org.smpte.st2071.signals:callback_handler_v1.0

Operations:

Return Data Type	Operation / Description
oneway	**callback(event : Event)** The callback operation is executed by an EventBroadcaster to deliver an Event to the CallbackHandler.
oneway	**registered(callback : URI)** The registered operation is executed by the EventBroadcaster whenever the CallbackHandler is registered to the EventBroadcaster.
oneway	**unregistered(callback : URI)** The unregistered operation is executed by the EventBroadcaster whenever the CallbackHandler is unregistered from the EventBroadcaster.

boolean	**heartbeat(rn : RN)** Executed by the EventBroadcaster periodically to check the health of the CallbackHandler and the EventBroadcaster shall provide its Resource Name as the parameter *rn*. The CallbackHandler shall return TRUE if it still wishes to receive events from the EventBroadcaster, otherwise FALSE shall be returned. If the EventBroadcaster continually receives network errors while executing this operation, the EventBroadcaster shall unregister the CallbackHandler, executing the "unregistered" operation on the CallbackHandler.

15.2.5 StatusEvent

The StatusEvent shall be a Serializable signal interface that is an extension of the Event and indicates the status of a Device or Service. StatusEvent may be a reflection of the Device or Service state or it may be an indication of health. Many event types defined in the MDCF extend the StatusEvent to reflect the general health and state of the Device or Service.

Attributes:

Name	Data Type	Description
Ready	boolean	A flag indicating if the device or service is ready to receive commands.
Status	STATUS	The STATUS of the device or service. OK, WARNING or ERROR.
Message	String	A message further describing the current status.

15.2.6 StatusSupport

StatusSupport shall be a Capability Interface that provides the attributes required to query the status of a device or service. Status may be used to reflect the many states of a device or service and to communicate which action the device or service is currently performing. For example, a tape deck may indicate that it is ejecting a tape and everything is working as expected or it may indicate that the tape is jammed. Another example would be a storage device warning its clients that it is running low on storage space.

UCN: urn:smpte:ucn:org.smpte.st2071.signals:status_support _v1.0

Attributes:

Name	Data Type	Description
Status	StatusEvent	The current status of the device.

15.3 The Service Framework

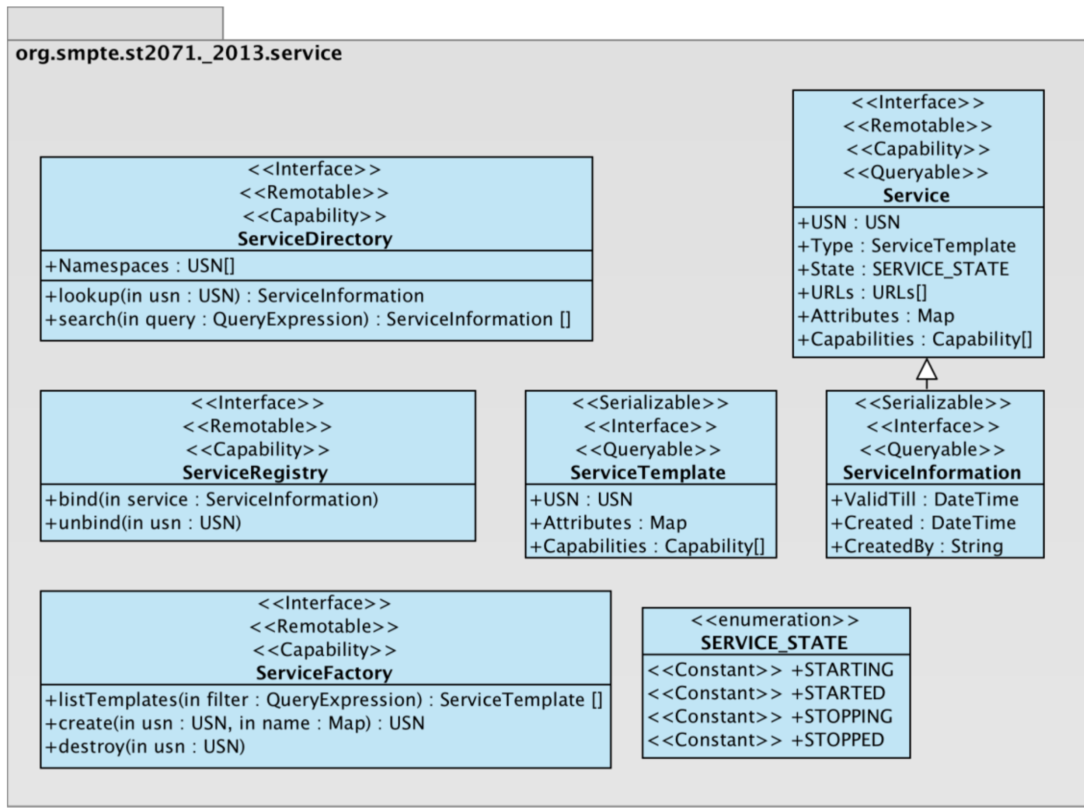

Figure 10 – Service Framework UML Diagram

Figure 10 depicts the complete UML for the MDCF representation of software services and the supporting constructs.

Namespace: org.smpte.st2071.service

15.3.1 Service

The Service interface shall be a remotely executable interface that defines the most fundamental attributes of a software service and shall provide a list of the Capability Interfaces exposed by the service.

UCN: urn:smpte:ucn:org.smpte.st2071:service_v1.0

Attributes:

Name	Data Type	Description
USN	USN	The USN that uniquely identifies the Service Instance.
Type	ServiceTemplate	The Service Template that describes the Service type.
State	SERVICE_STATE	The runtime state of the Service.
URLs	URL[]	The list of URLs to be used to access the service.
Attributes	Map	A Map of attributes describing characteristics of the Service.
Capabilities	Capability[]	The list of Capability Interfaces supported by the Service.

15.3.2 ServiceInformation

The Service Information shall be a Serializable interface that provides the means in which the information describing a Service is transmitted between the Service Factory and the client. It shall extend the Service interface and, therefore, it shall contain all of the information represented by the Service interface, but shall also include a ValidTill data element, which shall indicate at what date and time the Service Information is no longer valid.

Attributes:

Name	Data Type	Description
USN	USN	The USN that uniquely identifies the Service Instance.
Type	ServiceTemplate	The Service Template that describes the Service type.
State	SERVICE_STATE	The runtime state of the Service.
URLs	URL[]	The list of URLs to be used to access the service.
Attributes	Map	A Map of attributes describing characteristics of the Service.
Capabilities	Capability[]	The list of Capability Interfaces supported by the Service.
Created	DateTime	The date and time when the Service was first created.
CreatedBy	String	The name of the Subject that created the Service.
ValidTill	DateTime	The date and time at which the information is no longer valid.

15.3.3 ServiceTemplate

The Service Template shall describe the Capabilities and attributes of a particular type of service. A USN shall uniquely identify each Service Template and the USN identifying the Service Template shall have a “type” identity attribute containing the value “template”.

Attributes:

Name	Data Type	Description
USN	USN	The USN that uniquely identifies the Service Template.
Attributes	Map	A Map of attributes describing the characteristics of the Service Template.
Capabilities	Capability[]	The list of Capability Interfaces supported by the Service Template and implemented by instances of that Service Template.

15.3.4 SERVICE_STATE

The SERVICE_STATE enumeration is used to indicate the run time state of a Service.

Enumeration:

Name	Description
STARTING	Indicates that the value is a simple Boolean. TRUE or FALSE.
STARTED	Indicates that the value is a string of characters. The termination of the string depends on the implemented wire protocol and platform and shall be detailed in subsequent documents.
STOPPING	Indicates that the value is a positive or negative whole number.
STOPPED	Indicates that the value is a positive or negative decimal number.

15.3.5 ServiceDirectory

The ServiceDirectory Capability Interface shall provide the attributes and operations by which Services can be located.

UCN: urn:smpte:ucn:org.smpte.st2071.service_directory_v1.0

Exceptions:

Name	Description
ServiceException	Indicates that a fatal error has occurred.
ServiceNotFound	Indicates that the Service cannot be found or does not exist.

Attributes:

Name	Data Type	Description
Namespaces	USN[]	The list of namespaces represented by the factory.

Operations:

Return Data Type	Operation / Description
ServiceInformation	**lookup(usn : USN)** Returns the Service Information for the identified Service instance. **Exceptions** ServiceException, ServiceNotFound, SecurityException
ServiceInformation[]	**search(query : QueryExpression)** Returns a filtered list of Service Information objects that represent the available Service Instances. If *query* is NULL, all of the Service instances are returned. **Exceptions** InvalidQuery, ServiceException, SecurityException

15.3.6 ServiceRegistry

The Service Registry Capability Interface shall provide operations for the registration and de-registration of Services within a Service Directory.

UCN: urn:smpte:ucn:org.smpte.st2071:service_registry_v1.0

Exceptions:

Name	Description
ServiceNotBound	Indicates that the Service could not be bound.
ServiceAlreadyBound	Indicates that the Service is already bound.
ServiceNotUnbound	Indicates that the Service could not be unbound.

Operations:

Return Data Type	Operation / Description
n/a	**bind(service : ServiceInformation)** Registers the specified *service* to the Service Directory implementing the Service Registry Capability Interface. **Exceptions** ServiceNotBound, ServiceAlreadyBound, SecurityException
n/a	**unbind(usn : USN)** Unregisters the service identified by *usn* from the Service Directory implementing the Service Registry Capability Interface. **Exceptions** ServiceNotFound, ServiceNotUnbound, SecurityException

15.3.7 ServiceFactory

The ServiceFactory Capability Interface shall provide the attributes and operations by which Services can be created and destroyed.

UCN: urn:smpte:ucn:org.smpte.st2071.service_factory_v1.0

Operations:

Return Data Type	Operation / Description
ServiceTemplate[]	**listTemplates(filter : QueryExpression)** Returns a filtered list of the supported Service Templates. If *filter* is NULL, all of the supported Service Templates are returned. **Exceptions** InvalidQuery, ServiceException, SecurityException
USN	**create (usn: USN, parameters : Map)** Called by a client to create a new Service instance. The parameters shall contain the input parameters required to construct the Service instance. **Exceptions** ServiceException, SecurityException
n/a	**destroy(usn : USN)** Called by a client to stop and destroy the identified Service instance. **Exceptions** ServiceException, ServiceNotFound, SecurityException

15.4 Device Framework

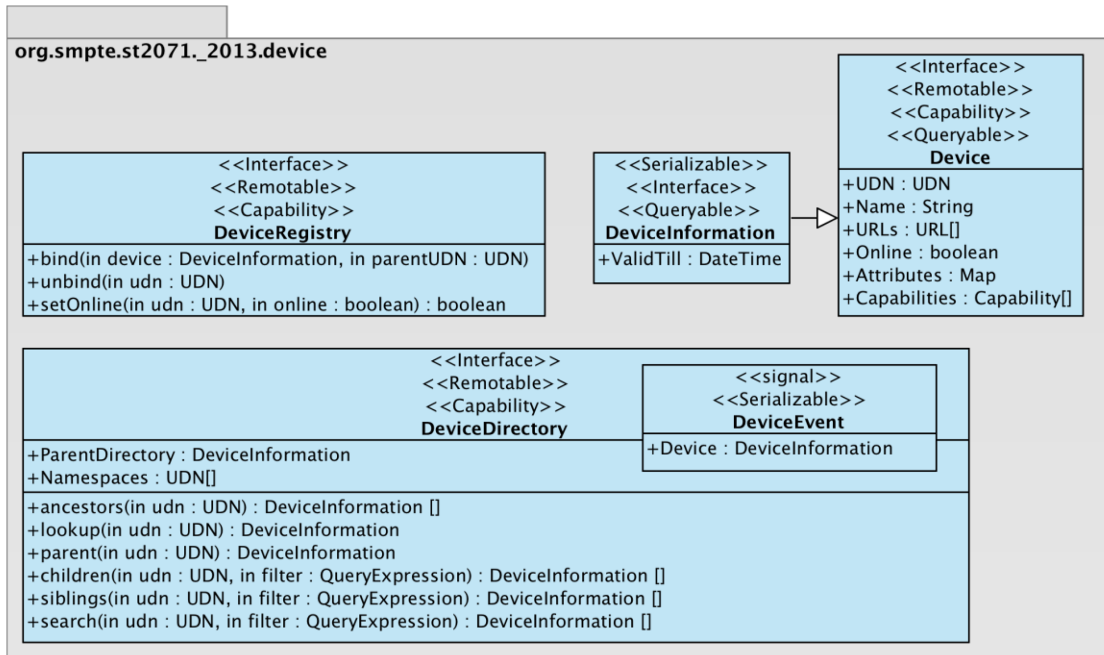

Figure 11 – Device Framework UML Diagram

Figure 11 defines the complete UML for the MDCF device representation and the constructs required to support the devices.

Namespace: org.smpte.st2071.device

15.4.1 DeviceDirectory

The Device Directory shall be the Capability Interface that provides the operations to list, search and access Devices. As with all Capability Interfaces any device may implement the Device Directory interface.

UCN: urn:smpte:ucn:org.smpte.st2071:device_directory_v1.0

Exceptions:

Name	Description
DeviceNotFound	Indicates that the Device could not be bound.

Attributes:

Name	Data Type	Description
ParentDeviceDirectory	DeviceInformation	The device information for the parent Device Directory.
Namespaces	UDN[]	The list of namespaces governed by this Device Directory.

Operations:

Return Data Type	Operation / Description
DeviceInformation[]	**ancestors(udn : UDN)** Returns the list of ancestors/lineage for *udn.* **Exceptions** DeviceNotFound, SecurityException
DeviceInformation[]	**children(udn : UDN, QueryExpression<Device> filter)** Returns the list of child devices for *udn* filtered by *filter.* **Exceptions** DeviceNotFound, InvalidQuery, SecurityException
DeviceInformation	**lookup(udn : UDN)** Returns the device information for *udn.* **Exceptions** DeviceNotFound, SecurityException
DeviceInformation	**parent(udn : UDN)** Returns the device information for *udn*'s parent device. **Exceptions** DeviceNotFound, SecurityException
DeviceInformation[]	**search(udn : UDN, QueryExpression<Device> filter)** Recursively searches the children of *udn* for any devices that match *filter.* **Exceptions** DeviceNotFound, InvalidQuery, SecurityException
DeviceInformation[]	**siblings(udn : UDN, QueryExpression<Device> filter)** Returns the list of sibling devices for *udn* filtered by *filter.* **Exceptions** DeviceNotFound, InvalidQuery, SecurityException

Events:

Data Type	Event Type	Description
DeviceEvent	Device Bound	A Device has been bound to the directory.
DeviceEvent	Device Unbound	A Device has been unbound from the directory.
DeviceEvent	Device Online	A Device has come online.
DeviceEvent	Device Offline	A Device has been taken offline.

15.4.2 DeviceEvent

The Device Event shall be a Serializable signal interface that in an extension of the StatusEvent and is transmitted by a Device when an asynchronous event occurs. Each Capability Interface may define a unique set of Device Events that may be broadcast by the device.

Attributes:

Name	Data Type	Description
Device	DeviceInformation	The device information for Device that the event applies to.

15.4.3 DeviceRegistry

The Device Registry Capability Interface shall be an extension of the Device Directory that provides operations for the registration and de-registration of Devices.

UCN: urn:smpte:ucn:org.smpte.st2071:device_registry_v1.0

Exceptions:

Name	Description
DeviceNotBound	Indicates that the Device could not be bound.
DeviceAlreadyBound	Indicates that the Device is already bound.
DeviceNotUnbound	Indicates that the Device could not be unbound.

Operations:

Return Data Type	Operation / Description
n/a	**bind(device : DeviceInformation, parentUDN : UDN)** Registers a Device to the Device Hierarchy as a child of the Device identified by the *parentUDN*,. **Exceptions** DeviceNotFound, DeviceNotBound, DeviceAlreadyBound, SecurityException
n/a	**unbind(udn : UDN)** Unregisters the device identified by *udn.* **Exceptions** DeviceNotFound, DeviceNotUnbound, SecurityException
boolean	**setOnline(udn : UDN, online : boolean)** Specifies whether the device identified by *udn* is online or offline. Returns the actual online/offline state of the device. TRUE indicates the device is online. **Exceptions** DeviceNotFound, SecurityException

15.4.4 Device

The Device interface shall be a remotely executable interface that defines the most fundamental attributes of a Device and shall provide a list of the Capability Interfaces exposed by the device.

UCN: urn:smpte:ucn:org.smpte.st2071:device_v1.0

Attributes:

Name	Data Type	Description
UDN	UDN	The Uniform Device Name of the device.
URLs	URL[]	The list of URLs to be used to access the device.
Name	String	The human readable name of the device.
Online	boolean	Indicates whether the device is capable of accepting commands. TRUE indicates the device is available.
Attributes	Map	A Map of attributes that describe the characteristics of the Device.
Capabilities	Capability[]	The list of Capability Interfaces exposed by the device.

15.4.5 DeviceInformation

The Device Information shall be a Serializable interface that provides the means in which the information describing a Device is transmitted between the Device Directory and the client. It shall extend the Device interface and therefore it shall contain all of the information represented by the Device interface, but shall also include a ValidTill data element, which shall indicate at what date and time the Device Information is no longer valid.

Attributes:

Name	Data Type	Description
UDN	UDN	The Uniform Device Name of the device.
URLs	URL[]	The list of URLs to be used to access the device.
Name	String	The human readable name of the device.
Online	boolean	Indicates whether the device is capable of accepting commands. TRUE indicates the device is available.
Attributes	Map	A Map of attributes that describes the characteristics of the Device.
Capabilities	Capability[]	The list of Capability Interfaces exposed by the device.
ValidTill	DateTime	The date and time that the device information is no longer valid.

15.5 Session and Lock Management (Delegation of Control)

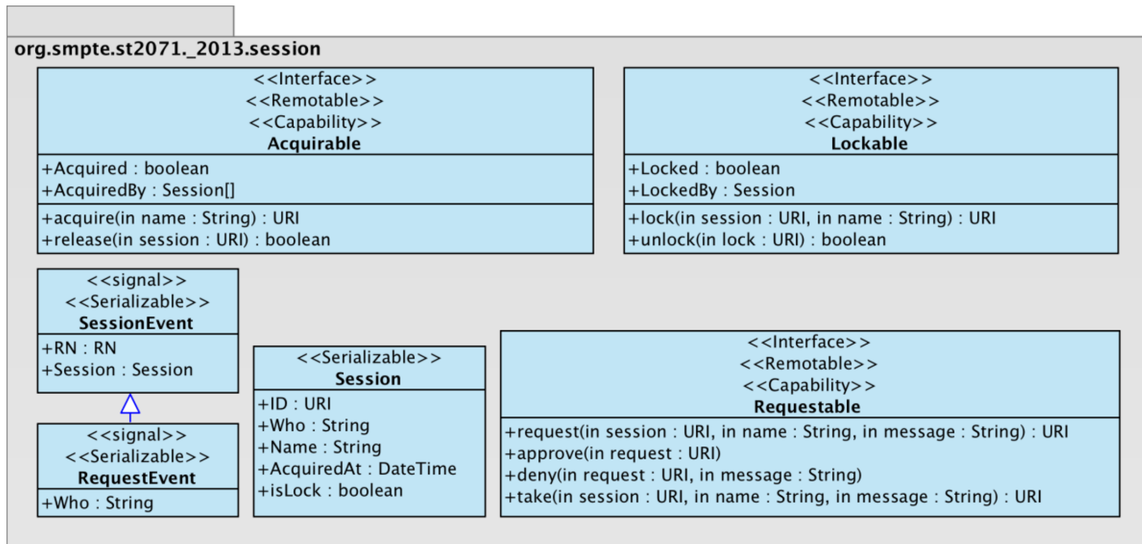

Figure 12 – Session and Lock Management

Figure 12 defines the complete UML for the MDCF constructs for session and lock management, also known as delegation of control.

Namespace: org.smpte.st2071.session

15.5.1 Session

The Session shall be a Serializable data structure that describes a session or client connection. The session shall contain a unique session identifier, represented as a URI, a human readable name, describing the session, the name of the client that established the session, and the date and time in which the session was first established.

Attributes:

Name	Data Type	Description
ID	URI	The unique identification of the session.
Name	String	The human readable name assigned to the session.
Who	String	The name of the client that acquired the session.
AcquiredAt	DateTime	The date and time that the session was acquired.
IsLock	Boolean	Indicates if the session is a lock.

15.5.2 SessionEvent

The SessionEvent shall be a Serializable signal that is an extension of the StatusEvent and is transmitted by a resource when a change to the session or a lock state of a resource changes. The SessionEvent shall be

generated by the various methods defined in the Aquirable and Lockable Capability Interfaces and shall be used to notify clients of a session/lock has been established or that a session/lock has been relinquished.

Attributes:

Name	Data Type	Description
Session	Session	Specifies the session that the event applies to. Which session is in this attribute depends on the event type. Which session is contained in the event is described for each RequestEvent "Type" defined for the Acquirable and Lockable interfaces.

15.5.3 RequestEvent

The RequestEvent shall be a Serializable signal that in an extension of the SessionEvent and is transmitted by a resource when a client is requesting a session or a lock from a resource. The RequestEvent shall be generated by the various methods defined in the Requestable Capability Interface and shall be used to notify clients of a session/lock request, that their session/lock has been taken by another client, or that their request was approved or denied.

Attributes:

Name	Data Type	Description
Who	String	The name of the client who is requesting the session.

15.5.4 Acquirable

Acquirable resources are resources that require the client to acquire a session before executing operations. Resources shall be acquirable if they require session management. The NotAcquired exception shall be added to all operations, for all Capability Interfaces, exposed by the Resource implementing the Acquirable interface.

UCN: urn:smpte:ucn:org.smpte.st2071:acquirable_v1.0

Exceptions:

Name	Description
NotAcquired	Indicates that resource must be acquired.
NameInUse	Indicates that the specified name is already in use by another session.
SessionNotFound	Indicates that the session cannot be found.
TooManySessions	Indicates that the resource cannot support any more sessions.
SessionTaken	Indicates that the session has been taken. See Requestable.take().

Attributes:

Name	Data Type	Description
Acquired	boolean	Indicates if the resource is acquired by at least 1 session.
AcquiredBy	Session[]	The list of sessions held by the resource.

Operations:

Return Data Type	Operation / Description
URI	**acquire(name :String)** Attempts to acquire a new session with the given *name*. If successful the URI identifying the new session is returned. **Exceptions** NotAcquired, Locked, TooManySessions, NameInUse, SecurityException
boolean	**release(session : URI)** Releases the session specified by *session*, if the caller has permissions to release the session or is the session holder. **Exceptions** NotAcquired, Locked, SessionNotFound, SecurityException

Events:

Data Type	Event Type	Description
SessionEvent	Acquired	A session has been acquired.
SessionEvent	Released	A session has been released.

15.5.5 Lockable

Lockable resources are resources that allow the client to acquire an exclusive lock of the resource implementing the Lockable interface. Locks shall prevent clients that do not hold the lock from executing operations or changing the state of the resource. The Locked and NotLocked exceptions specified for the Lockable interface shall be added to all operations, for all Capability Interfaces, exposed by the resource implementing the Lockable interface.

UCN: urn:smpte:ucn:org.smpte.st2071:lockable_v1.0

Exceptions:

Name	Description
Locked	Indicates that resource is locked.
NotLocked	Indicates that the resource is not locked and a lock is required to perform the attempted operation or that an attempt to lock the resource has failed.
LockNotFound	Indicates that the lock cannot be found.
LockTaken	Indicates that the lock has been taken. See Requestable.take().

Attributes:

Name	Data Type	Description
Locked	boolean	Indicates if the resource is locked.
LockedBy	Session	The session currently holding the exclusive lock.

Operations:

Return Data Type	Operation / Description
URI	**lock(session : URI, name : String)** Attempts to convert the session identified by *session* to an exclusive lock with the specified *name*. If successful the URI identifying the lock is returned, otherwise a NotLocked exception is raised. **Exceptions** SessionNotFound, NotLocked, NotAcquired, SecurityException
boolean	**unlock(lock : URI)** Releases the lock for the session specified by *lock*, allowing other sessions and locks to be established. If successful, TRUE is returned. **Exceptions** LockNotFound, NotLocked, NotAcquired, SecurityException

Events:

Data Type	Event Type	Description
SessionEvent	Locked	A Device has been exclusively locked.
SessionEvent	Unlocked	A Device has been unlocked.

15.5.6 Requestable

Acquirable devices are devices that require the client to acquire a session before executing operations. Devices shall be acquirable if the require session management. The exceptions specified for the Acquirable interface shall be added for all operation and attribute accesses for all Capability Interfaces exposed by the Device implementing the Acquirable interface.

UCN: urn:smpte:ucn:org.smpte.st2071:requestable_v1.0

Exceptions:

Name	Description
RequestNotFound	Indicates that the request cannot be found.
RequestExpired	Indicates that the request has expired and is no longer valid.
RequestDenied	Indicated that the request was denied.

Operations:

Return Data Type	Operation / Description
URI	**request(session : URI, name : String, message : String)** Called by a client to request a session from another client. The *session* specifies the session that the requestor wishes to take and the name specifies the new *name* to assign to the session. The message is sent to the current session holder as informative text, conveying the reason for the request. **Exceptions** SessionNotFound, RequestDenied, NotAcquired, SecurityException
n/a	**approve(request : URI)** Called by a client to approve a session request, relinquishing the session specified by *request.* **Exceptions** RequestNotFound, RequestExpired, SecurityException
n/a	**deny(request : URI, message : String)** Called by a client to reject a session request for the specified session *request.* The *message* is provided so that the session holder may indicate why the request was rejected. **Exceptions** RequestNotFound, RequestExpired, SecurityException
URI	**take(session : URI, name : String, message : String)** Called by a device administrator to forcefully take the session or lock identifier by *session.* The new session or lock is assigned the *name* and the *message* is sent to the former session holder as informative text describing the reason for the forced acquisition. **Exceptions** SessionNotFound, LockNotFound, NotAcquired, SecurityException

Events:

Data Type	Event Type	Description
RequestEvent	RequestAcquire	A client is requesting a session from a session holder. The Session attribute is set to the session that is being requested.
RequestEvent	RequestLock	A client is requesting an exclusive lock on the resource. The Session attribute is set to the session that is requesting the lock.
RequestEvent	Approved	The request for the session or lock has been approved. The Session attribute is set to the session that was requested and has been relinquished.
RequestEvent	Denied	The request for the session or lock has been denied. The Session attribute is set to the session that was requested.
RequestEvent	SessionTaken	The system or a system administrator has taken the session. The Session attribute is set to the session that was taken.
RequestEvent	LockTaken	The system or a system administrator has taken the lock. The Session attribute is set to the session that held the lock.

15.6 Modes Framework

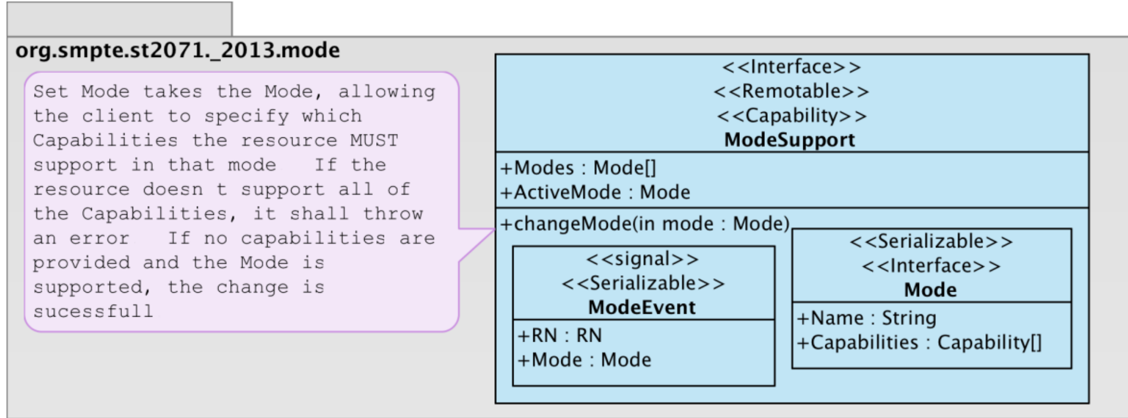

Figure 13 – Modes Framework UML Diagram

Figure 13 depicts the complete UML for the MDCF representation of software service and device modes.

Namespace: org.smpte.st2071.mode

15.6.1 Mode

The Mode shall be a Serializable interface that provides the means in which the information describing a mode of operation is transmitted between a Service, Device, and a client. The Mode interface describes a mode of operation as a list of Capabilities with an associated, human readable, name. A Service or Device may support zero or more modes of operation, but shall have only one mode active at any given point in time. When a mode is active, the Service or Device implementing the mode shall support the Capability Interfaces specified by the active mode and the Capability Interfaces listed by the Service or Device "Capabilities" attribute. Therefore, the Capability Interfaces supported by a Service or a Device shall be the union set of the Capability Interfaces listing in the Capabilities attribute of the Service or Device and the Capabilities attribute of the active mode.

Attributes:

Name	Data Type	Description
Name	String	The human readable name of the mode.
Capabilities	Capability[]	The list of additional Capability Interfaces supported by the Device or Service when this mode is active.

15.6.2 ModeEvent

The ModeEvent shall be a Serializable signal interface that in an extension of the StatusEvent and is transmitted by a Service or Device when the active mode has changed. The ModeEvent shall specify the active mode of the Service or Device when the Service or Device completes a transition into a new active mode.

Attributes:

Name	Data Type	Description
RN	RN	The Resource Name of the Device or Service.
Mode	Mode	Specifies the new active mode for the Service or Device.

15.6.3 ModeSupport

The ModeSupport Capability Interface shall provide the operations and attributes necessary for Services or Devices to support multiple modes of operation. A Service or Device implementing the ModeSupport Capability Interface shall have more than one mode of operation; each mode changing the aggregate list of Capability Interfaces exposed by that Service or Device. The Capability Interfaces specified in the Service or Device "Capabilities" attribute shall always be available, regardless of the active mode.

UCN: urn:smpte:ucn:org.smpte.st2071:mode_support_v1.0

Exceptions:

Name	Description
InvalidMode	Indicates that specified mode is not supported by the resource.
ModeException	Indicates that a generic failure occurred.

Attributes:

Name	Data Type	Description
Modes	Mode[]	Contains the list of modes that the resource supports.
ActiveMode	Mode	Indicates the mode that is currently active.

Operations:

Return Data Type	Operation / Description
Mode	**changeMode(mode : Mode)** Changes the ActiveMode of the resource to the specified *mode*. Upon success the operation returns the new ActiveMode. **Exceptions** NotLocked, Locked, InvalidMode, ModeException, SecurityException

Events:

Data Type	Event Type	Description
ModeEvent	Mode Changed	The resource has changed modes.

15.7 Media Framework

Figure 14 – Media Framework UML Diagram

Figure 14 defines the complete UML for the MDCF representation of media and the supporting constructs.

Namespace: org.smpte.st2071.media

15.7.1 MediaDirectory

The Media Directory is the Capability Interface that provides the operations to list, search, create, update, delete and determine the composition of Media. As with all Capability Interfaces any device may implement the Media Directory interface.

UCN: urn:smpte:ucn:org.smpte.st2071:media_directory_v1.0

Exceptions:

Name	Description
MediaCreationFailed	Indicates that the attempt to create new media failed.
MediaDeletionFailed	Indicates that the attempt to delete existing media failed.
MediaUpdateFailed	Indicates that the attempt to update existing media failed.
MediaNotFound	Indicates that the specified media could not be found.

Attributes:

Name	Data Type	Description
MediaContainer	MediaContainer	The Media Container that is represented by this Media Directory. Similar to the root directory of a POSIX file system.

Operations:

Return Data Type	Operation / Description
Media	**create(media : Media, pointers : MediaPointer[])** Creates a new piece of Media using the metadata specified in the *media* parameter with the essence specified by the *pointers* and returns the Media representation for the newly created Media. The type of the *media* parameter instructs the Media Directory as to which type of Media it is to create. **Exceptions** MediaCreationFailed, MediaNotFound, SecurityException
Media	**delete(umn : UMN)** Deletes the Media identified by the *umn*, returning the Media representation for the deleted Media as it was before the deletion. **Exceptions** MediaNotFound, MediaDeletionFailed, SecurityException
Media[]	**list(container : UMN, filter : QueryExpression<Media>)** Returns the list of Media that resides in the Media Container specified by the *umn* parameter, if it matches the *filter* expression. If the *filter* is not provided, all Media in the container is returned. **Exceptions** MediaNotFound, InvalidQuery, SecurityException
Media	**lookup(umn : UMN)** Returns the Media identified by the *umn.* **Exceptions** MediaNotFound, SecurityException
MediaAsset	**lookupAsset(mid : String)** Returns the Media Asset identified by the *mid.* **Exceptions** MediaNotFound, SecurityException
Media[]	**search(container : UMN, QueryExpression<Media> filter)** Recursively searches the Media Container identified by *umn* and all of its descendants for all Media that matches the *filter.* **Exceptions** MediaNotFound, InvalidQuery, SecurityException
Media	**update(media : Media)** Updates the specified Media with the metadata provided in the *media* parameter, returning the Media representation of the media after the update. **Exceptions** MediaNotFound, MediaUpdateFailed, SecurityException

Events:

Data Type	Event Type	Description
MediaEvent	Media Created	A new piece of Media has been created.
MediaEvent	Media Deleted	A piece of Media has been deleted.
MediaEvent	Media Updated	A piece of Media has been updated.

15.7.2 MediaAccess

The Media Access Capability Interface provides the operations required to determine the URLs that can be used to access a piece of media.

UCN: urn:smpte:ucn:org.smpte.st2071:media_access_v1.0

Operations:

Return Data Type	Operation / Description
URL[]	**lookupURLs(umn : UMN)** Returns the zero or more URLs that can be used to access the media represented by the provided *umn*. **Exceptions** MediaNotFound, SecurityException

15.7.3 MediaEvent

The Media Event is a Serializable signal object that is an extension of the StatusEvent and is transmitted by a Media Directory when an asynchronous event occurs.

Attributes:

Name	Data Type	Description
Media	Media	The media representation for the Media that the event applies to.

15.7.4 Media

The Media is a Serializable interface that provides the means in which the information describing a piece of Media is transmitted between the Media Directory and the client.

Attributes:

Name	Data Type	Description
UMN	UMN	The UMN that identifies the Media.
Name	String	The human readable name of the Media.
Location	UMN	The UMN that identifies the Media Container in which this Media resides.
Created	DateTime	The date and time in which the Media was first created.
Modified	DateTime	The data and time in which the Media was last updated.
Metadata	Map	A Map of name / value pairs.

15.7.4.1 MediaAsset

The Media Asset is an extension of the Media type that provides the means in which the information describing a Media Asset is transmitted between the Media Directory and the client.

Attributes:

Name	Data Type	Description
Duration	DateTime	The duration of the Media Asset.
Composition	MediaSegment[]	The list of Media Segments describing the contents of the Media Asset.

15.7.4.2 MaterialAsset

The Material Asset is an extension of the Media Asset type that provides the means in which the information describing a Material Asset is transmitted between the Media Directory and the client.

Attributes:

Name	Data Type	Description
Duration	FramedTime	The duration of the Material Asset.

15.7.4.3 MediaContainer

The Media Container is an extension of the Media type that provides the means in which the information describing Media Containers is transmitted between the Media Directory and the client.

Attributes:

Name	Data Type	Description
UDN	UDN	The UDN identifying the media access device. The device identified by this UDN shall, at a minimum implement the Media Access Capability Interface

15.7.4.4 MediaFile

The Media File is an extension of the Media type that provides the means in which the information describing a Media File is transmitted between the Media Directory and the client.

Attributes:

Name	Data Type	Description
MIMEType	String	The MIME type of the Media File.
Size	uint64	The size of the Media File in bytes.

15.7.4.5 MediaInstance

The Media Instance is an extension of the Media File and Material Asset types that provides the means by which the information describing a Media Instance is transmitted between the Media Directory and the client.

15.7.4.6 MediaBundle

The Media Bundle is an extension of the Media Instance and Media Container types that provides the means in which the information describing a Media Bundle is transmitted between the Media Directory and the client. The Media Bundle inherits the attributes from both the Media Instance and the Media Container, but does not define any additional attributes.

15.7.4.7 MediaPointer

The Media Pointer is a Serializable interface that provides the means in which the information describing a Media Pointer is transmitted between the Media Directory and the client.

Attributes:

Name	Data Type	Description
Source	UMN	The source or originating piece of Media.
OffsetType	OFFSET_TYPE	Indicates the offsets type.
InpointOffset	uint64	The in point offset in units indicated by the offset type.
OutpointOffset	uint64	The out point offset in units indicated by the offset type.
Track	Int16	The track index.

15.7.4.8 MediaSegment

The Media Segment is an extension of the Media base type and the Media Pointer that provides the means in which the information describing a segment of Media is transmitted between the Media Directory and the client.

Attributes:

Name	Data Type	Description
Inpoint	DateTime	The DateTime or FramedTime of the in point offset.
Outpoint	DateTime	The DateTime or FramedTime of the out point offset.
Format	URI	The format of the Media essence.
TrackName	String	The name of the track
TrackType	TRACK_TYPE	The type of track. e.g. audio, video, data

15.7.5 TRACK_TYPE

The TRACK_TYPE enumeration is used to indicate the type of track that is represented by a MediaSegment.

Enumeration:

Name	Description
AUDIO	Indicates that the track is audio.
VIDEO	Indicates that the track is video.
TIMECODE	Indicates that the track is a data track containing timecode.
TEXT	Indicates that the track is a data track containing text. e.g. CC
DATA	Indicates that the track is a generic data track.

15.8 Data Types

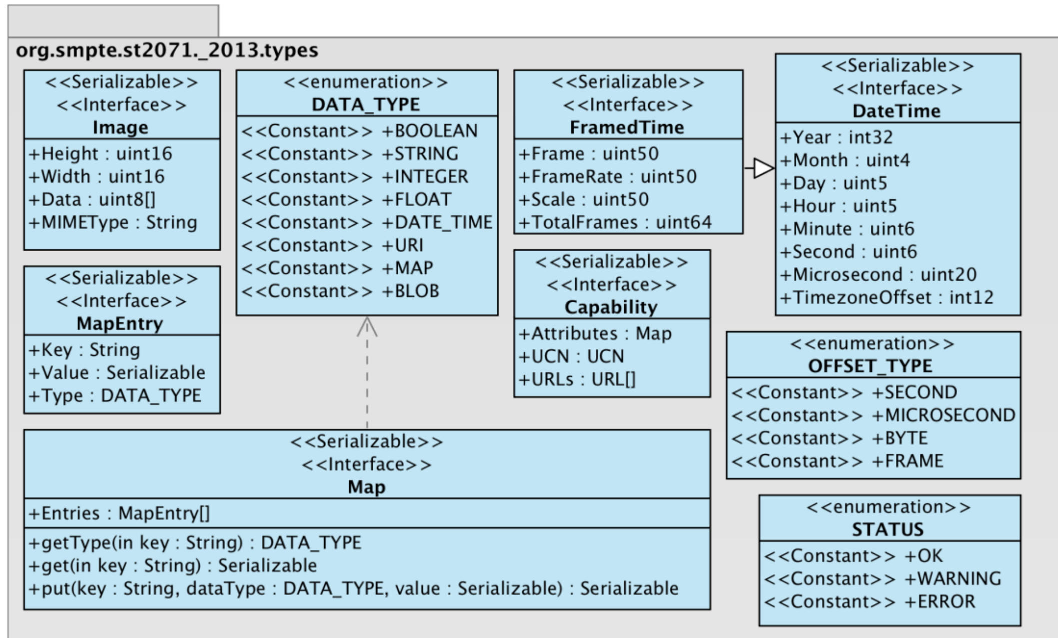

Figure 15 – Data Types UML Diagram

Figure 15 defines a collection of reusable data types that are used throughout the MDCF UML diagrams.

Namespace: org.smpte.st2071.types

15.8.1 Capability

The Capability shall be a Serializable interface that describes a Capability Interface exposed. The Capability shall contain a UCN that uniquely identifies the Capability Interface, a list of URLs used to access the Capability Interface, and a Map of attributes that describes the characteristics of the Capability Interface. Each Capability Interface shall be independently addressed, allowing for direct access to the Capability Interface independently from other Capability Interfaces.

Attributes:

Name	Data Type	Description
Attributes	Map	A Map of attributes that describes the characteristics of the Capability.
UCN	UCN	The unique identifier for the supported Capability Interface.
URLs	URL[]	The list of URLs to be used to access the Capability Interface.

15.8.2 STATUS

The STATUS enumeration is used to indicate the operational status of a resource. The values indicate whether there is an error condition and the severity of that condition.

Enumeration:

Name	Description
OK	Indicates that there are no warnings or errors.
WARNING	Indicates that there are warnings. Warnings are conditions that should be brought to the attention of the client, but do not prevent operation. For example, a low amount of storage space would be a warning, until a client attempted to allocate more space than is available.
ERROR	Indicates that there is an error condition. An error condition is any condition that prevents an operation from executing or a fault in the system.

15.8.3 DATA_TYPE

The DATA_TYPE enumeration is used to indicate that data type of the value stored within a Map. The DATA_TYPE indicates how the value should be interpreted and processed.

Enumeration:

Name	Description
BOOLEAN	Indicates that the value is a Boolean value of TRUE or FALSE.
STRING	Indicates that the value is a string of characters.
INTEGER	Indicates that the value is a positive or negative whole number.
FLOAT	Indicates that the value is a positive or negative decimal number.
DATE_TIME	Indicates that the value is of type DateTime.
URI	Indicates that the value is a string of characters that is formatted in accordance to the rules specified in RFC 3986: Uniform Resource Identifier (URI) General Syntax.
MAP	Indicates that the value is a Map.
BLOB	Indicates that the value is a Binary Long Object, represented as a sequence of uint8 values. Equivalent to uint8[].

15.8.4 OFFSET_TYPE

The OFFSET_TYPE enumeration is used to indicate that type of the offset value stored within a Media Pointer. The OFFSET_TYPE indicates how the value should be interpreted and processed.

Enumeration:

Name	Description
SECOND	Indicates that the value is in seconds.
MICROSECOND	Indicates that the value is in microseconds.
BYTE	Indicates that the value is in bytes.
FRAME	Indicates that the value is in frames.

15.8.5 Map

The Map is a general data structure used to store key/value pairs. Each key can only occur once within the Map. The format by which the values are represented within the Map structure is protocol dependent. Therefore, each protocol adaptation of the MDCF shall specify the means by which the Map structure is transmitted and how each DATA_TYPE is encoded within the Map structure for transmission.

Exceptions:

Name	Description
KeyNotFound	Indicates that the specified key could not be found in the Map.

Attributes:

Name	Data Type	Description
Entries	MapEntry[]	The list of keys stored within the Map.

Operations:

Return Data Type	Operation / Description
DATA_TYPE	**getType(key : String)** Returns the value from the DATA_TYPE enumeration that indicates the data type of the value for the specified *key*. **Exceptions** KeyNotFound
Serializable	**get(key : String)** Returns the value of the specified *key*. **Exceptions** KeyNotFound
Serializable	**put(key : String, dataType : DATA_TYPE, value : Serializable)** Sets the specified *key* to the provided *value* as the data type specified in the *dataType* parameter. Returns the *value* as it will be presented using the *get(key)* operation.

15.8.5.1 MapEntry

The Map is a general data structure used to store key/value pairs. Each key can only occur once within the Map. The format by which the values are represented within the Map structure is protocol dependent. Therefore, each protocol adaptation of the MDCF shall specify the means by which the Map structure is transmitted and how each DATA_TYPE is encoded within the Map structure for transmission.

Attributes:

Name	Data Type	Description
Key	String	The unique key that identifies the Map Entry with a Map.
Type	DATA_TYPE	The data type of the value stored within this Map Entry.
Value	*Protocol Specific*	The value of the Map Entry, encoded in accordance to the DATA_TYPE specified in the Type attribute.

15.8.6 DateTime

The DateTime data type represents a Gregorian date and time value, modeling the pertinent date and time fields as specified within ISO 8601-2004.

Attributes:

Name	Data Type	Description
Year	int32	Indicates the year, in accordance to ISO 8601-2004.
Month	uint4	Indicates the month within the year
Day	uint5	Indicates the day within the month.
Hour	uint5	Indicates the hour within the day.
Minute	uint6	Indicates the minute within the hour.
Second	uint6	Indicates the second within the minute.
Microsecond	uint20	Indicates the microsecond within the second.
TimezoneOffset	int12	Indicates the Time Zone offset in minutes.

15.8.7 FramedTime

The FramedTime data type is an extension of the DateTime type that adds support for framed time. Framed time is the division of time into increments that are not standard units of time (e.g., Seconds, Microseconds, Milliseconds). The duration of each frame is determined by dividing the FrameRate by the Scale ($FrameRate \div Scale$). When a FramedTime is used to represent duration, the TotalFrames field shall contain the total number of frames representing that duration of time. When the FramedTime is used to represent a point in time, such as the time of day, the TotalFrames value shall be set to 0.

Attributes:

Name	Data Type	Description
Frame	uint50	Indicates the fractional second value in frames.
FrameRate	uint50	The frame rate.
Scale	uint50	The time scale. The frame duration is determined by dividing the FrameRate by the Scale ($FrameRate \div Scale$).
TotalFrames	uint64	Contains the total number of frames for the represented duration.

15.8.8 Image

The Image data type represents a graphical image and all the attributes required to describe its interpretation.

Attributes:

Name	Data Type	Description
Height	uint16	The height of the image in pixels.
Width	uint16	The width of the image in pixels.
Data	uint8[]	The data blob containing the image data. The MIME type indicates the format of the data.
MIMEType	String	The MIME type of encoding used to compress the image.

15.9 Querying Expression Syntax Object Notation

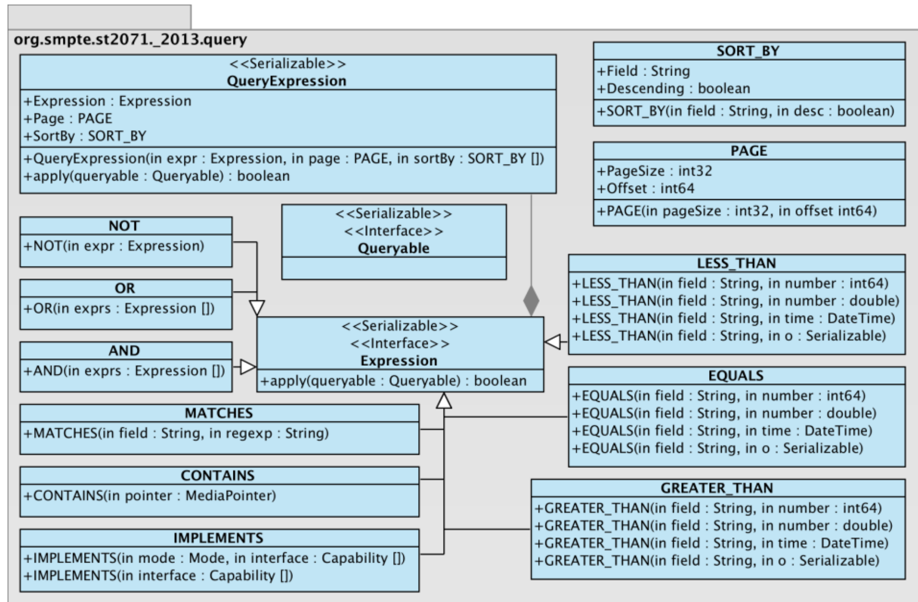

Figure 16 – Query Syntax Object Notation UML Diagram

Figure 16 defines the object notation for the query expression syntax.

Namespace: org.smpte.st2071.query

15.9.1 QueryExpression

The QueryExpression data type represents a complete query expression and is used by the MDC Framework to filter Queryable resources from result sets.

Exceptions:

Name	Description
InvalidQuery	Indicates that the specified QueryExpression is not valid or contains errors.

Constructors:

Constructor
QueryExpression(expr : Expression, page : PAGE, sortBy : SORT_BY[]) **Exceptions** InvalidQuery

Attributes:

Name	Data Type	Description
Expression	Expression	The query Expression.
Page	PAGE	The pagination information applicable to the query results.
SortBy	SORT_BY[]	The sort order applicable to the query results.

Operations:

Return Data Type	Operation / Description
boolean	**apply(queryable : Queryable)** Returns TRUE if the provided *media* matches the query expression.

15.9.2 Queryable

The Queryable interface shall denote interfaces and data types that can be queried or filtered using the Query Expression Object Oriented Query Notation.

15.9.3 Expression

The Expression interface describes the behavior of the Expression data types. The Expression interface and its implementers make up the MDC object oriented query language and are arranged in a hierarchy, to create a complete query expression. All Expression data types shall implement the Expression interface.

Operations:

Return Data Type	Operation / Description
boolean	**apply(queryable : Queryable)** Applies the Expression to the provided *queryable* resource and returns TRUE if queryable matches the represented Expression. Returns TRUE if the provided *queryable* matches the query expression.

15.9.4 AND

The Boolean AND operator shall be used to combine two or more query expressions into one Boolean expression and shall implement the Expression interface. All of the provided query expressions shall equate to TRUE in order for the AND expression to equate to TRUE.

Constructors:

Constructor
AND(expressions : QueryExpression[])

15.9.5 OR

The Boolean OR operator shall be used to combine two or more query expressions into one boolean expression and shall implement the Expression interface. If any of the provided query expression are TRUE the OR expression shall equate to TRUE.

Constructors:

Constructor
OR(expressions : QueryExpression[])

15.9.6 NOT

The Boolean NOT unary operator shall be used to invert provided query expression, returning TRUE if the provided expression equates to FALSE. The NOT operator shall implement the Expression interface.

Constructors:

Constructor
NOT(expr : QueryExpression)

15.9.7 LESS_THAN

The LESS_THAN operator shall be used to determine if the value contained within a data field is less than the provided value and shall implement the Expression interface. It can be applied to numerical values, dates and times or other Serializable data types.

Constructors:

Constructor
LESS_THAN(field : String, value : int64)
LESS_THAN(field : String, value : double)
LESS_THAN(field : String, value : DateTime)
LESS_THAN(field : String, value : Serializable)

15.9.8 GREATER_THAN

The GREATER_THAN operator shall be used to determine if the value contained within a data field is greater than the provided value and shall implement the Expression interface. It can be applied to numerical values, dates and times or other Serializable data types.

Constructors:

Constructor
GREATER_THAN(field : String, value : int64)
GREATER_THAN(field : String, value : double)
GREATER_THAN(field : String, value : DateTime)
GREATER_THAN(field : String, value : Serializable)

15.9.9 EQUALS

The EQUALS operator shall be used to determine if the value in the specified data field is equal to the provided value and shall implement the Expression interface. It can be applied to numerical values, dates and times or other Serializable data types.

Constructors:

Constructor
EQUALS(field : String, value : int64)
EQUALS (field : String, value : double)
EQUALS (field : String, value : DateTime)
EQUALS (field : String, value : Serializable)

15.9.10 MATCHES

The MATCHES operator shall be used to match a data field against the provided regular expression and shall implement the Expression interface. The regular expression shall be specified using the POSIX Regular Expression syntax as defined by the POSIX specification.

Constructors:

Constructor
MATCHES(field : String, regexp : String)

15.9.11 CONTAINS

The CONTAINS operator shall be used to determine if the Media contains the specified media element, expressed as a MediaPointer and shall implement the Expression interface. The media element can be expressed using any of the data types extending from MediaPointer and for CONTAINS to evaluate to TRUE, the MediaPointer’s inpoint and outpoint values shall specify a valid subset of one or more media elements that exist for the Media. CONTAINS shall be true if the provided MediaPointer and the Media contain an intersecting set.

Constructors:

Constructor
CONTAINS(pointer : MediaPointer)

15.9.12 IMPLEMENTS

The IMPLEMENTS operator shall be an extension of the Expression interface that tests if a device implements a list of Capability Interfaces and/or supports a mode that implements a list of Capability Interfaces. In order for IMPLEMENTS to equate to TRUE, the evaluated device shall list all of the provided Capability Interfaces, support the specified mode, and the specified mode shall list all of the Capability Interfaces specified for that mode.

Constructors:

Constructor
IMPLEMENTS(mode : Mode, interfaces : Capability[])
IMPLEMENTS(interfaces : Capability[])

15.9.13 PAGE

The PAGE operator shall specify the paging information for a query expression, allowing the result set to be divided into a series of smaller pages. The sort order of the query expression is applied to the result set before the result set is divided into pages and thus, the offset of a specific result can change as the sort order changes.

Constructors:

Constructor
PAGE(pageSize : int 32, offset : int64)

Attributes:

Name	Data Type	Description
PageSize	int32	The number of items to include in the results page.
Offset	int64	The numerical index of the first item in the result page. The *offset* shall start at 1 and increment by 1 for each item in the source list that matches the query expression for the applied sort order.

15.9.14 SORT_BY

The SORT_BY operator shall be used to specify the order in which the result set is to be sorted. The natural sorting order for the data type contained within the specified field shall be used.

Constructors:

Constructor
SORT_BY(field : String, descending : boolean)

Attributes:

Name	Data Type	Description
Field	String	The name of the field containing the data to sort.
Descending	Boolean	The sort direction. If TRUE the sort is reverse natural order, if FALSE the sort is in natural order.

15.10 Security Framework

Figure 17 – Security Framework UML Diagram

Figure 17 defines the roles based security model used by the MDCF.

Namespace: org.smpte.st2071.security

15.10.1 Error Conditions

The MDCF Security Framework provides a number of exceptions that are added globally to all operations and attribute accesses for all Capability Interfaces. These exceptions are raised in addition to the exceptions defined for the operations and attributes defined in the Capability Interfaces.

Exceptions:

Name	Description
AuthenticationRequired	Indicates that authentication is required to access the resource.
AuthenticationFailed	Indicates that the authentication attempt failed.
AuthorizationRequired	Indicates that authorization is required to access the resource.
AuthorizationFailed	Indicates that the authorization attempt failed.
AuthenticationError	Indicates that some error occurred during authentication.
AuthorizationError	Indicates that some error occurred during authorization.
AuthenticationAborted	Indicates that the authentication sequence was aborted.
AuthorizationAborted	Indicates that the authorization sequence was aborted.
SecurityLayerExpired	Indicates that the Security Layer or the encryption key has expired.

15.10.2 Principal

The Principal represents a specific set of login credentials, such as a username with a password or an identity with a digital certificate. The values stored within the Principal are implementation specific. Clients may possess more than one Principal, for more than one form of authentication. For example a client may have a user name and password, but may also have a digital certificate. The Security Tokens contained within the Principal may be changed during authentication, changing the authentication identity to an authorization identity to be used in subsequent calls to authorize access to a systemic resource.

Attributes:

Name	Data Type	Description
Realm	String	The SASL realm or scope applicable to the Principal. The realm can be an Internet domain or any collections of characters used to indicate the scope of the authoritative source.
Identifier	String	The string of characters that uniquely identifies the Principal within the specified realm.
Roles	Role[]	The list of Roles assigned to the Principal.
Tokens	SecurityToken[]	The list of security tokens. Can contain simple username/password credentials or digital certificates.

15.10.3 Subject

The Subject represents a collection of authentication/authorization information for a single entity. The Subject can possess zero or more Principals, each Principal containing a unique set of authentication/authorization credentials, such as a user name with a password or an identity with a digital certificate.

Attributes:

Name	Data Type	Description
Name	String	The human readable name of the Subject.
Principals	Principal[]	The list of Principals (authentication credentials or authorization credentials) possessed by the Subject.

15.10.4 SecurityToken

A SecurityToken is a temporary piece of information that is created by the client or by the Authenticator to carry implementation specific security information. The SecurityToken is used to transmit the authentication credentials to the Authenticator and in turn the Authenticator can create a new set of SecurityTokens that contain the authorization credentials used by the client to authorize against a systemic resources. SecurityTokens are temporary by nature and should not be stored persistently. Every measure should be taken to protect the SecurityToken while it is in memory and to securely erase it when it is discarded.

Attributes:

Name	Data Type	Description
Mechanism	String	The name assigned to the SASL mechanism. Defines the format of the Data. For example “PLAIN” if the value is a plaintext password, “CRAM-MD5” or “DIGEST-MD5” for MD5 password hashes and “SASL-GSSAPI” for GSSAPI authentication tokens.
Data	uint8[]	The raw data depicting the value, formatted in the means described by the format field.
ValidTill	DateTime	The Data and Time until which the SecurityToken is Valid. Used for Token Expiry and Rotation.

15.10.5 Permission

Permissions are granular representations of access control rights. The Permission data type is used to indicate which Roles have the ability to perform which functions.

Attributes:

Name	Data Type	Description
Roles	Role[]	The list of Roles that the client shall possess for this permission to apply.
Allow	boolean	Specifies if the permission rule is an Allow rule or a Deny rule. Allow rules indicate that if the Subject possesses all of the Roles they are allowed access to the resource. Deny rules indicate that if the Subject possesses all of the Roles they are denied access to the resource.
Permissions	PERMISSION_TYPE[]	The list of permissions that is applicable to this permission rule.
Resource	URI	A URI specifying the URI of the resource that the permission rule applies to.

15.10.6 Role

A Role is a security group used to assign Permissions to clients. Permissions are assigned to one or more Roles and a client shall be assigned to all of the Roles listed for a specific Permission in order for that Permission to apply.

Attributes:

Name	Data Type	Description
Parent	Role	The parent Role. Roles can form a hierarchy, with each child Role inheriting the permissions of its parent. A Role with a parent shall extend the parents permissions, overriding or extending the permissions.
Name	String	The unique human readable name assigned to the Role.

15.10.7 PERMISSION_TYPE

The PERMISSION_TYPE enumeration provides the valid permissions for the MDCF.

Enumeration:

Name	Description
READ	Indicates that the Role can read from the resource.
WRITE	Indicates that the Role can write to the resource.
EXECUTE	Indicates that the Role can execute the resource or operations on the resource.
DELETE	Indicates that the Role can delete resource.
ADMINISTER	Indicates that the Role can administer the resource.

15.10.8 Authorizer

The Authorizer is a Capability Interface that defines the operations required to authorize an authenticated client with a device. The client must have received a digital certificate from an Authenticator before attempting to authorize itself with a device. Once authorized the client may begin executing commands for the current communications session.

UCN: urn:smpte:ucn:org.smpte.st2071:authorizer_v1.0

Attributes:

Name	Data Type	Description
Mechanisms	String[]	Provides a list of all of the supported authorization mechanisms supported by the Authorizer.

Operations:

Return Data Type	Operation / Description
Subject	**authorize(subject : Subject, uri : URI)** Authorizes a session for the resource specified by the URI. Returns a new Subject if the specified *subject* has the necessary permissions to access the resource and raises an Authorization Failed exception if the specified *subject* does not. Authorize shall be called after the client has authenticated with an Authenticator, but before any commands are executed. The *subject* provided shall be the Subject returned from an Authenticator. The Authenticator that was used to authenticate the Subject does not need to be the same device as the Authorizer. **Exceptions** AuthorizationFailed, AuthorizationError, AuthorizationAborted, InvalidSubject

15.10.9 Authenticator

The Authenticator is a Capability Interface that defines the operations required to authenticate with a device implementing the MDCF. The Authenticator is used for the initial login of a client and shall provide the client with a digital certificate that the client can later use to authorize itself with a device.

UCN: urn:smpte:ucn:org.smpte.st2071:authenticator_v1.0

Attributes:

Name	Data Type	Description
Mechanisms	String[]	Provides a list of all of the supported authentication mechanisms supported by the Authenticator

Operations:

Return Data Type	Operation / Description
Subject	**authenticate(subject : Subject)** Authenticates the *subject* with the MDCF or Device. The Authenticator may alter the Principals within then Subject, removing the specified criteria and replacing them with security tokens that can be used with an Authorizer. For example, if the Authenticator implements Kerberos, the provided Principal will be replaced with a Kerberos authorization certificate. **Exceptions** AuthenticationFailed, AuthenticationError, AuthenticationAborted, InvalidSubject
Subject	**logout(subject : Subject)** Logs the subject out of the system, invalidating any system resources, sessions or certificates reserved for the Subject. **Exceptions** NotAuthenticated, InvalidSubject

Annex A Bibliography (Informative)

[POSIX] IEEE Std. 1003.1-2008, Standard for Information Technology – Portable Operating System Interface (POSIX) Base Specification, Issue 7

[RFC 2276] IETF RFC 2276, Architecture Principles of Uniform Resource Name Resolution.

[UPnP] UPnP™ Forum, Universal Plug and Play (UPnP™) – Device Architecture 1.1.

[ISO/IEC 19500-1], Object Management Group Common Object Request Broker Architecture (CORBA®), Interfaces

[ISO/IEC 19500-3], Object Management Group Common Object Request Broker Architecture (CORBA®), Components

Annex B Glossary (Normative)

A

ABNF: See the definition for Augmented Backus Normal Form.

API: See the definition for Application Programming Interface.

Application Programming Interface: The specification of how software components should interact with one another.

Atomic: The smallest unit of something, indivisible and irreducible, e.g., the smallest unit of "control."

Attribute: A named value that depicts some portion of the state of an device, object, or service.

Augmented Backus Normal Form: An extension of the Backus-Naur Form used primarily for the definition of formal systems of language used as bi-directional communication protocols, defined by the IETF RFC 5234.

B

Backus-Naur Form: A metalanguage used to describe the syntax of languages used in computing.

Backus Normal Form: See the definition for Backus-Naur Form.

C

Capability: A small concise feature of a device, service, or object, the smallest unit of control.

Capability Interface: An interface that defines a Capability, e.g, Read Value, Write Value, Pausable, or Playable.

Class: In Object Oriented Programming, a class is a construct that is a blueprint describing the state and behavior of an entity represented in software.

Clip: A piece of media, whole or in part. In common language the term "Clip" is overloaded and is used to refer to either, or both, the media instance and the media asset.

D

Device: A controllable hardware or software resource used in the creation, storage, and manipulation of media, e.g., a media server, VTR, audio/video router, camera, character generator, or transcoder.

Device Directory: A directory of devices and their hierarchical arrangement.

Digital Media: The creative convergence of digital arts, science, technology and business for human expression, communication, social interaction and education.

Directory: A repository of information represented in a logical hierarchical order that supports searching, listing and lookup.

E

Entity: Something that exists by itself; something that is separate from other things.

Event: See the definition for Signal

I

IDL: See the definition for Interface Definition Language.

Interface: An interface is an abstract data type that describes the behavior of a software component. The behavior may be represented as attributes, operations, and/or signals.A software component is said to "implement" an interface if it behaves in accordance to that interface, containing all of the attributes, operations, and signals defined by that interface.

Interface Definition Language: A specification language that defines the interface implemented by a software component.

Internet of Things: While still in its infancy, the Internet of Things is a network infrastructure linking physical and virtual objects, utilizing the existing and evolving Internet and network developments. Each object is uniquely identifiable, network connected, discoverable, and interoperates with other objects within the network.

IoT: See the definition for Internet of Things.

M

MDCF: Media Device Control Framework.

Media: Aural and/or visual representations and ancillary information about them such as transcripts, captions, subtitles, and descriptive data.

Media Access Device: A device that provides the Capability Interfaces to stream, transfer and/or manipulate the essence of a piece of media, e.g. a device implementing a Capability Interface that translates UMNs into URLs.

Media Directory: A directory of media and its hierarchical arrangement.

Metalanguage: A language or symbols used to make statements about or describe another language.

O

Object: In Object Oriented Programming, an object is an instantiation of a class or an ephemeral compilation of the state and behaviors of an entity.

OMG® IDL: The Interface Definition Language defined by the Object Management Group (OMG®) to describe CORBA® interfaces.

P

Pointer: A data structure that refers to or points to something else.

POSIX: The Portable Operating System Interface; a family of standards defined by the IEEE for maintaining compatibility between operating systems.

S

Segment: A part of, or subset, of a whole entity.

Serializable: Serializable interfaces define simple data structures used to transmit data between computer processes or over the network. State information is marshaled for inter-process transmission and reconstituted by the receiving process, re-creating the original software component and state.

Server: A physical or logical superset of media devices, including storage, that is capable of acquiring, storing, and/or distributing media.

Signal: An asynchronous notification sent by a device or software service to zero or more listeners notifying those listeners of changes to the state of that device or software service.

Status: The indication of the state and/or error condition of a device, service, or object, e.g., stopped, paused, panning, pan complete, new source displayed, transform initiated, and transform complete.

Storage Device: A device that holds or contains media, e.g., solid-state memory or spinning disk drives.

U

URI: Uniform Resource Identifier as defined by RFC 3986.

URL: Uniform Resource Locator as defined by RFC 1738.

URN: Uniform Resource Name as defined by RFC 1737 and RFC 2141.

Annex C Complete MDCF UML® (Normative)

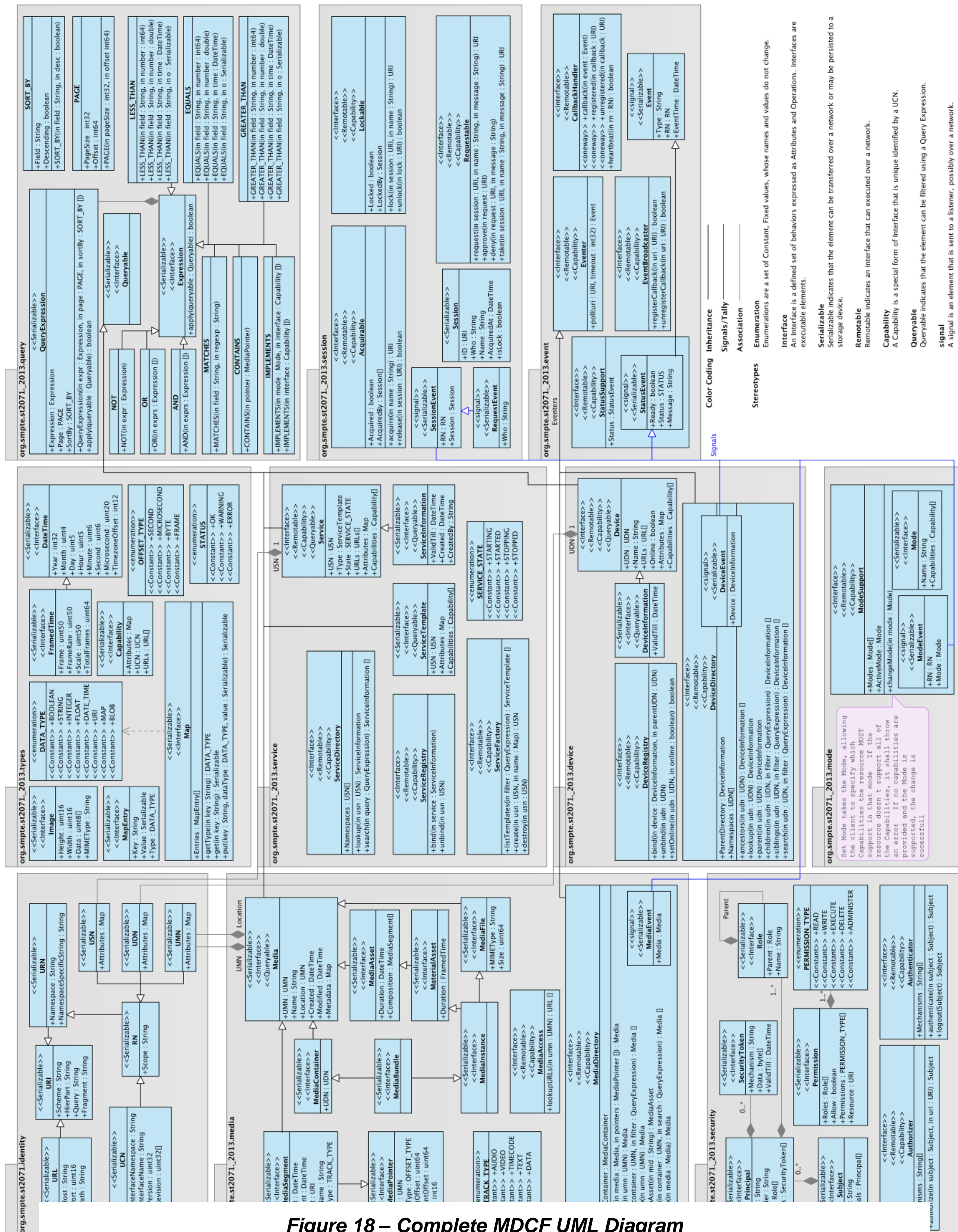

Figure 18 – Complete MDCF UML Diagram

Annex D MDCF Core IDL (Informative)

The Media Device Control Framework (MDCF) must be applied to a specific platform and/or wire level protocol in order to produce an interoperable Media Device Control protocol. This annex defines a sample CORBA® 2.3 Interface Definition Language (IDL) application of the MDCF. These mappings, in conjunctions with the CORBA 2.3 specification provide for a complete interoperable Media Device Control protocol. The following rules dictate how CORBA IDL is constructed from the UML® design of the MDCF. These rules apply to the core framework and any additional Capability Interfaces that are defined.

1. Classes defined with the *Serializable* stereotype or the *Serializable* and *interface* stereotypes are to be defined using the IDL *valuetype* directive.

   Ex.
   ```
   valuetype SomeSerializable {}
   ```

2. Classes defined with the *Remotable* stereotype or the *Remotable* and *interface* stereotypes are to be defined using the IDL *interface* directive.

   Ex.
   ```
   interface SomeRemotable {}
   ```

3. Classes defined with the *Capability* stereotype are Capability Interfaces and are to be defined using the IDL interface directive and must have a constant UCN instance variable by the name ["UCN"] defined.

   Ex.
   ```
   const org::smpte::tc34cs::mdcp::identity::UCN UCN = "urn:smpte:ucn:some_interface_v1.0";
   ```

4. Class *Attributes* are to be defined as IDL *readonly attribute* types using the smallest IDL data type that can contain the full size specified for the UML data type.

   Ex.
   ```
   readonly attribute SomeAttribute
   ```

5. UML® character and string types are represented using wide characters and strings. (wchar, wstring)

6. Class *Operations* are to be defined as IDL operations with public scope.

   a. Class Operation parameters are to be defined using the *in* parameter attribute.

   Ex.
   ```
   void someOperation(in param1, in param2) {}
   ```

7. The UML® inheritance must be reflected using the inheritance model defined for IDL.

8. Constructors specified in UML are to be implemented as IDL *factory* operations with the name ["init"].

9. Class attributes, operations or parameters that are defined as an array are to be defined as an IDL *sequence* type with a suitable name type.

   Ex.
   ```
   typedef sequence<wstring> Strings;
   ```

```
typedef sequence<unsigned long> UnsignedLongs;
typedef sequence<wstring> Strings;
typedef octet bits128[16];
typedef sequence<octet> RawData;

module org { module smpte { module st2071 { module _2014 {
module types
{
    valuetype Map;
};

module query
{
    interface Queryable
    {
    };

    valuetype QueryExpression;

    exception InvalidQuery
    {
        Queryable Queryable;
        QueryExpression QueryExpression;
        wstring message;
    };

    valuetype PAGE
    {
        readonly attribute long pageSize;
        readonly attribute long offset;

        factory init(in long pageSize, in long offset);
    };

    valuetype SORT_BY
    {
        readonly attribute wstring field;
        readonly attribute boolean descending;

        factory init(in wstring field, in boolean descending);
    };

    typedef sequence<SORT_BY> SORT_BYs;

    interface Expression
    {
        boolean apply(in Queryable queryable);
    };

    valuetype QueryExpression
    {
        readonly attribute PAGE page;
        readonly attribute SORT_BYs sortBy;
        readonly attribute Expression expr;

        factory init(in Expression expr, in PAGE page, in SORT_BYs sortBy);

        boolean apply(in Queryable queryable);
    };
};


module identity
{
    valuetype URI
    {
        readonly attribute wstring Scheme;
        readonly attribute wstring HierPart;
        readonly attribute wstring Query;
        readonly attribute wstring Fragment;
```

```
        factory init(in wstring text);
    };

    valuetype URL supports URI
    {
        readonly attribute wstring Host;
        readonly attribute unsigned short Port;
        readonly attribute wstring Path;

        factory init(in wstring text);
    };

    valuetype URN supports URI
    {
        readonly attribute wstring Namespace;
        readonly attribute wstring NamespaceSpecificString;

        factory init(in wstring text);
    };

    valuetype RN supports URN
    {
        readonly attribute wstring Scope;
    };

    valuetype UCN supports RN
    {
        readonly attribute wstring InterfaceNamespace;
        readonly attribute wstring InterfaceName;
        readonly attribute unsigned long Version;
        readonly attribute UnsignedLongs Revision;

        factory init(in wstring text);
    };

    valuetype UDN supports RN
    {
        readonly attribute org::smpte::st2071::_2014::types::Map Attributes;

        factory init(in wstring text);
    };

    valuetype USN supports RN
    {
        readonly attribute org::smpte::st2071::_2014::types::Map Attributes;

        factory init(in wstring text);
    };

    valuetype UMN supports RN
    {
        readonly attribute org::smpte::st2071::_2014::types::Map Attributes;

        factory init(in wstring text);
    };

    typedef sequence<URI> URIs;
    typedef sequence<URL> URLs;
    typedef sequence<URN> URNs;
    typedef sequence<UCN> UCNs;
    typedef sequence<UDN> UDNs;
    typedef sequence<USN> USNs;
    typedef sequence<UMN> UMNs;
};


module types
{
    enum DATA_TYPE
    {
        _BOOLEAN,
```

```
    _STRING,
    _INTEGER,
    _FLOAT,
    _DATE_TIME,
    _URI,
    _MAP,
    _BLOB
};

enum STATUS
{
    _OK,
    _WARNING,
    _ERROR
};

enum OFFSET_TYPE
{
    _SECOND,
    _MICROSECOND,
    _BYTE,
    _FRAME
};

exception KeyNotFound
{
    wstring key;
};

valuetype MapEntry
{
    readonly attribute wstring key;
    readonly attribute DATA_TYPE type;
    readonly attribute any value;
};

typedef sequence<MapEntry> Entries;

valuetype Map
{
    readonly attribute Entries Entries;

    DATA_TYPE getType(in wstring key);

    any get(in wstring key)
    raises(KeyNotFound);

    any put(in wstring key, in DATA_TYPE type, in any value)
    raises(KeyNotFound);
};

valuetype Image
{
    readonly attribute unsigned short Height;
    readonly attribute unsigned short Width;
    readonly attribute wstring MIMEType;
    readonly attribute RawData Data;
};

valuetype DateTime
{
    readonly attribute long Year;
    readonly attribute unsigned short Month;
    readonly attribute unsigned short Day;
    readonly attribute unsigned short Hour;
    readonly attribute unsigned short Minute;
    readonly attribute unsigned short Second;
    readonly attribute unsigned long Microsecond;
    readonly attribute short TimezoneOffset;
};
```

```
        valuetype FramedTime supports DateTime
        {
            readonly attribute unsigned long long Frame;
            readonly attribute unsigned long long FrameRate;
            readonly attribute unsigned long long Scale;
            readonly attribute unsigned long long TotalFrames;
        };

        valuetype Capability
        {
            readonly attribute Map Attributes;
            readonly attribute org::smpte::st2071::_2014::identity::UCN UCN;
            readonly attribute org::smpte::st2071::_2014::identity::URLs URLs;
        };

        typedef sequence<Capability> Capabilities;
    };


    module security
    {
        valuetype SecurityToken
        {
            readonly attribute wstring Mechanism;
            readonly attribute RawData Data;
            readonly attribute org::smpte::st2071::_2014::types::DateTime ValidTill;
        };

        valuetype Role
        {
            readonly attribute wstring Name;
            readonly attribute Role Parent;
        };

        typedef sequence<SecurityToken> SecurityTokens;
        typedef sequence<Role> Roles;

        valuetype Principal
        {
            readonly attribute wstring Realm;
            readonly attribute wstring Identifier;
            readonly attribute Roles Roles;
            readonly attribute SecurityTokens Tokens;
        };

        typedef sequence<Principal> Principals;

        valuetype Subject
        {
            readonly attribute wstring Name;
            readonly attribute Principals Principals;
        };

        enum PERMISSION_TYPE
        {
            _READ,
            _WRITE,
            _EXECUTE,
            _DELETE,
            _ADMINISTER
        };

        typedef sequence<PERMISSION_TYPE> Permissions;

        valuetype Permission
        {
            readonly attribute Roles Roles;
            readonly attribute boolean Allow;
            readonly attribute Permissions PERMISSION_TYPE;
            readonly attribute org::smpte::st2071::_2014::identity::URI Resource;
        };
```

```
    exception SecurityException
    {
        enum EXCEPTION_TYPE
        {
            _AUTHENTICATION,
            _AUTHORIZATION,
            _SECURITY_LAYER
        } Type;

        enum EXCEPTION_STATUS
        {
            _REQUIRED,
            _FAILED,
            _ABORTED,
            _ERROR,
            _EXPIRED
        } Status;

        org::smpte::st2071::_2014::identity::URI Resource;
        Subject Subject;
    };

    interface Authorizer
    {
        // UCN = "urn:smpte:ucn:org.smpte.st2071:authorizer_v1.0";

        readonly attribute Strings Mechanisms;

        Subject authorize(in Subject subject, in org::smpte::st2071::_2014::identity::URI uri)
        raises(SecurityException);
    };

    interface Authenticator
    {
        // UCN = "urn:smpte:ucn:org.smpte.st2071:authenticator_v1.0";

        readonly attribute Strings Mechanisms;

        Subject authenticate(in Subject subject)
        raises(SecurityException);

        Subject logout(in Subject subject)
        raises(SecurityException);
    };
};


module event
{
    valuetype Event
    {
        readonly attribute wstring Type;
        readonly attribute org::smpte::st2071::_2014::identity::RN RN;
        readonly attribute org::smpte::st2071::_2014::types::DateTime EventTime;
    };

    valuetype StatusEvent supports Event
    {
        readonly attribute boolean Ready;
        readonly attribute org::smpte::st2071::_2014::types::STATUS Status;
        readonly attribute wstring Message;
    };

    interface StatusSupport
    {
        // UCN = "urn:smpte:ucn:org.smpte.st2071:event_support_v1.0";

        readonly attribute StatusEvent Status;
        // raises (org::smpte::st2071::_2014::security::SecurityException);
    };
```

```
        interface CallbackHandler
        {
            oneway void callback(in Event event);

            oneway void registered(in org::smpte::st2071::_2014::identity::URI callback);

            oneway void unregistered(in org::smpte::st2071::_2014::identity::URI callback);

            boolean heartbeat(in org::smpte::st2071::_2014::identity::UDN eventBroadcaster);
        };

        interface Eventer
        {
            // UCN = "urn:smpte:ucn:org.smpte.st2071:event:eventer_v1.0";

            Event poll(in org::smpte::st2071::_2014::identity::URI uri,
                       in unsigned long timeout)
            raises(org::smpte::st2071::_2014::security::SecurityException);
        };

        interface EventBroadcaster
        {
            // UCN = "urn:smpte:ucn:org.smpte.st2071:event:event_broadcaster_v1.0";

            boolean registerCallback(in org::smpte::st2071::_2014::identity::URI uri)
            raises(org::smpte::st2071::_2014::security::SecurityException);

            boolean unregisterCallback(in org::smpte::st2071::_2014::identity::URI uri)
            raises(org::smpte::st2071::_2014::security::SecurityException);
        };
    };


    module mode
    {
        valuetype Mode
        {
            readonly attribute wstring Name;
            readonly attribute org::smpte::st2071::_2014::types::Capabilities Capabilities;
        };

        typedef sequence<Mode> Modes;

        exception InvalidMode
        {
            Mode Mode;
            wstring Message;
        };

        exception ModeException
        {
            Mode Mode;
            wstring Message;
        };

        interface ModeSupport
        {
            // UCN = "urn:smpte:ucn:org.smpte.st2071:mode_support_v1.0";

            readonly attribute Modes Modes;
            // raises (org::smpte::st2071::_2014::security::SecurityException);
            readonly attribute Mode ActiveMode;
            // raises (org::smpte::st2071::_2014::security::SecurityException);

            void changeMode(in Mode mode)
            raises (InvalidMode, ModeException,
                    org::smpte::st2071::_2014::security::SecurityException);
        };
```

```
    /*
     * Mode Events are sent by device that support multiple modes of operation.
     *
     * Valid Device Type Values
     *     'ModeChanged' : Indicates that the device's mode has changed.
     */
    valuetype ModeEvent supports org::smpte::st2071::_2014::event::StatusEvent
    {
        readonly attribute Mode Mode;
    };
};


module device
{
    interface Device : org::smpte::st2071::_2014::query::Queryable
    {
        // UCN = "urn:smpte:ucn:org.smpte.st2071:device_v1.0";

        readonly attribute org::smpte::st2071::_2014::identity::UDN UDN;
        // raises (org::smpte::st2071::_2014::security::SecurityException);
        readonly attribute org::smpte::st2071::_2014::identity::URLs URLs;
        // raises (org::smpte::st2071::_2014::security::SecurityException);
        readonly attribute wstring Name;
        // raises (org::smpte::st2071::_2014::security::SecurityException);
        readonly attribute boolean Online;
        // raises (org::smpte::st2071::_2014::security::SecurityException);
        readonly attribute org::smpte::st2071::_2014::types::Map Attributes;
        // raises (org::smpte::st2071::_2014::security::SecurityException);
        readonly attribute org::smpte::st2071::_2014::types::Capabilities Capabilities;
        // raises (org::smpte::st2071::_2014::security::SecurityException);
    };

    valuetype DeviceInformation supports Device
    {
        readonly attribute org::smpte::st2071::_2014::types::DateTime ValidTill;
    };

    typedef sequence<DeviceInformation> DeviceInformations;

    exception DeviceNotFound
    {
        org::smpte::st2071::_2014::identity::UDN UDN;
        wstring Message;
    };

    exception DeviceNotBound
    {
        org::smpte::st2071::_2014::identity::UDN UDN;
        org::smpte::st2071::_2014::identity::UDN parentUDN;
        org::smpte::st2071::_2014::identity::URLs urls;
        wstring name;
        org::smpte::st2071::_2014::types::Capabilities capabilities;
        wstring Message;
    };

    exception DeviceException
    {
        org::smpte::st2071::_2014::identity::UDN UDN;
        wstring Message;
    };

    exception DeviceNotUnbound
    {
        org::smpte::st2071::_2014::identity::UDN UDN;
        org::smpte::st2071::_2014::identity::UDN parentUDN;
        org::smpte::st2071::_2014::identity::URLs urls;
        wstring name;
        org::smpte::st2071::_2014::types::Capabilities capabilities;
        wstring Message;
    };
```

```
exception DeviceAlreadyBound
{
    org::smpte::st2071::_2014::identity::UDN UDN;
    wstring Message;
};

interface DeviceDirectory
{
    // UCN = "urn:smpte:ucn:org.smpte.st2071:device_directory_v1.0";

    readonly attribute DeviceInformation ParentDeviceDirectory;
    // raises (org::smpte::st2071::_2014::security::SecurityException);
    readonly attribute org::smpte::st2071::_2014::identity::UDNs Namespaces;
    // raises (org::smpte::st2071::_2014::security::SecurityException);

    DeviceInformations ancestors(in org::smpte::st2071::_2014::identity::UDN udn)
    raises (DeviceNotFound, org::smpte::st2071::_2014::security::SecurityException);

    DeviceInformation lookup(in org::smpte::st2071::_2014::identity::UDN udn)
    raises (DeviceNotFound, org::smpte::st2071::_2014::security::SecurityException);

    DeviceInformation parent(in org::smpte::st2071::_2014::identity::UDN udn)
    raises (DeviceNotFound, org::smpte::st2071::_2014::security::SecurityException);

    DeviceInformations children(in org::smpte::st2071::_2014::identity::UDN udn,
                                in org::smpte::st2071::_2014::query::QueryExpression filter)
    raises (DeviceNotFound, org::smpte::st2071::_2014::query::InvalidQuery,
            org::smpte::st2071::_2014::security::SecurityException);

    DeviceInformations siblings(in org::smpte::st2071::_2014::identity::UDN udn,
                                in org::smpte::st2071::_2014::query::QueryExpression filter)
    raises (DeviceNotFound, org::smpte::st2071::_2014::query::InvalidQuery,
            org::smpte::st2071::_2014::security::SecurityException);

    DeviceInformations search(in org::smpte::st2071::_2014::identity::UDN udn,
                              in org::smpte::st2071::_2014::query::QueryExpression filter)
    raises (DeviceNotFound, org::smpte::st2071::_2014::query::InvalidQuery,
            org::smpte::st2071::_2014::security::SecurityException);
};

interface DeviceRegistry
{
    // UCN = "urn:smpte:ucn:org.smpte.st2071:device_registry_v1.0";

    void bind(in org::smpte::st2071::_2014::identity::UDN udn,
              in org::smpte::st2071::_2014::identity::UDN parentUDN,
              in org::smpte::st2071::_2014::identity::URLs urls,
              in wstring name,
              in org::smpte::st2071::_2014::types::Capabilities capabilities)
    raises (DeviceNotFound, DeviceNotBound, DeviceAlreadyBound,
            org::smpte::st2071::_2014::security::SecurityException);

    void unbind(in org::smpte::st2071::_2014::identity::UDN udn)
    raises (DeviceNotFound, DeviceNotUnbound,
            org::smpte::st2071::_2014::security::SecurityException);

    boolean setOnline(in org::smpte::st2071::_2014::identity::UDN udn, in boolean online)
    raises (DeviceNotFound, org::smpte::st2071::_2014::security::SecurityException);
};

/*
 * Device Events are sent by all device types.
 *
 * Valid Device Type Values
 *     'Acquired' : Indicates that a device has been acquired.
 *     'Released' : Indicates that a device has been released.
 *     'Bound'    : Indicates that a device has been bound to the registry.
 *     'Unbound'  : Indicates that a device has been unbound to the registry.
 *     'Online'   : Indicates that a device has been set online.
 *     'Offline'  : Indicates that a device has been set offline.
```

```
     *     'Locked'   : Indicates that a device has been locked.
     *     'Unlocked' : Indicates that a device has been unlocked.
     */
    valuetype DeviceEvent supports org::smpte::st2071::_2014::event::StatusEvent
    {
        readonly attribute DeviceInformation Device;
    };
};


module session
{
    valuetype Session
    {
        readonly attribute org::smpte::st2071::_2014::identity::URI ID;
        readonly attribute wstring Who;
        readonly attribute wstring Name;
        readonly attribute org::smpte::st2071::_2014::types::DateTime AcquiredAt;
        readonly attribute boolean IsLock;
    };

    typedef sequence<Session> Sessions;

    exception NotAcquired
    {
        org::smpte::st2071::_2014::identity::RN RN;
        wstring Message;
    };

    exception NotLocked
    {
        org::smpte::st2071::_2014::identity::RN RN;
        wstring Message;
    };

    exception Locked
    {
        org::smpte::st2071::_2014::identity::RN RN;
        wstring Message;
    };

    exception TooManySessions
    {
        org::smpte::st2071::_2014::identity::RN RN;
        wstring Message;
    };

    exception NameInUse
    {
        org::smpte::st2071::_2014::identity::RN RN;
        wstring Message;
    };

    exception SessionNotFound
    {
        org::smpte::st2071::_2014::identity::RN RN;
        org::smpte::st2071::_2014::identity::URI ID;
        wstring Message;
    };

    exception RequestNotFound
    {
        org::smpte::st2071::_2014::identity::RN RN;
        org::smpte::st2071::_2014::identity::URI ID;
        wstring Message;
    };

    exception RequestExpired
    {
        org::smpte::st2071::_2014::identity::RN RN;
        org::smpte::st2071::_2014::identity::URI ID;
```

```
    wstring Message;
};

exception RequestDenied
{
    org::smpte::st2071::_2014::identity::RN RN;
    org::smpte::st2071::_2014::identity::URI ID;
    wstring Message;
};

interface Acquirable
{
    // UCN = "urn:smpte:ucn:org.smpte.st2071:acquirable_v1.0";

    readonly attribute boolean Acquired;
    // raises (org::smpte::st2071::_2014::security::SecurityException);
    readonly attribute Sessions AcquiredBy;
    // raises (org::smpte::st2071::_2014::security::SecurityException);

    org::smpte::st2071::_2014::identity::URI acquire(in wstring name)
    raises (NotAcquired, Locked, TooManySessions, NameInUse,
            org::smpte::st2071::_2014::security::SecurityException);

    boolean release(in org::smpte::st2071::_2014::identity::URI session)
    raises (NotAcquired, Locked, SessionNotFound,
            org::smpte::st2071::_2014::security::SecurityException);
};

interface Lockable
{
    // UCN = "urn:smpte:ucn:org.smpte.st2071:lockable_v1.0";

    readonly attribute boolean Locked;
    // raises (org::smpte::st2071::_2014::security::SecurityException);
    readonly attribute Session LockedBy;
    // raises (org::smpte::st2071::_2014::security::SecurityException);

    org::smpte::st2071::_2014::identity::URI lock(
                       in org::smpte::st2071::_2014::identity::URI session, in wstring name)
    raises(SessionNotFound, NotAcquired, NotLocked,
           org::smpte::st2071::_2014::security::SecurityException);

    boolean unlock(in org::smpte::st2071::_2014::identity::URI lock)
    raises (SessionNotFound, org::smpte::st2071::_2014::security::SecurityException);
};

interface Requestable
{
    // UCN = "urn:smpte:ucn:org.smpte.st2071:requestable_v1.0";

    org::smpte::st2071::_2014::identity::URI request(
                                    in org::smpte::st2071::_2014::identity::URI session,
                                    in wstring name, in wstring message)
    raises(SessionNotFound, RequestDenied, NotAcquired,
           org::smpte::st2071::_2014::security::SecurityException);

    void approve(in org::smpte::st2071::_2014::identity::URI request)
    raises(RequestNotFound, RequestExpired,
           org::smpte::st2071::_2014::security::SecurityException);

    void deny(in org::smpte::st2071::_2014::identity::URI request, in wstring message)
    raises(RequestNotFound, RequestExpired,
           org::smpte::st2071::_2014::security::SecurityException);

    org::smpte::st2071::_2014::identity::URI take(
                                    in org::smpte::st2071::_2014::identity::URI session,
                                    in wstring name, in wstring message)
    raises(SessionNotFound, NotAcquired,
           org::smpte::st2071::_2014::security::SecurityException);
};
```

```
    valuetype SessionEvent supports org::smpte::st2071::_2014::event::StatusEvent
    {
        readonly attribute Session Session;
    };

    /*
     * Valid Event Type Values
     *     'RequestAcquire' : Indicates that a client or device is requesting the specified
     *                        session.
     *     'RequestLock'    : Indicates that a client or device is requesting an exclusive lock
     *                        of the device.
     *     'Approved'       : Indicates that the request was approved.
     *     'Denied'         : Indicates that the request was denied.
     *     'SessionTaken'   : Indicates that the specified session was taken by the system or
     *                        an administrator.
     *     'LockTaken'      : Indicates that the lock for the specified session was taken by the
     *                        system or an administrator.
     */
    valuetype RequestEvent supports SessionEvent
    {
        readonly attribute wstring Who;
    };
};


module service
{
    valuetype ServiceTemplate supports org::smpte::st2071::_2014::query::Queryable
    {
        readonly attribute org::smpte::st2071::_2014::identity::USN USN;
        // raises (org::smpte::st2071::_2014::security::SecurityException);
        readonly attribute org::smpte::st2071::_2014::types::Map Attributes;
        // raises (org::smpte::st2071::_2014::security::SecurityException);
        readonly attribute org::smpte::st2071::_2014::types::Capabilities Capabilities;
        // raises (org::smpte::st2071::_2014::security::SecurityException);
    };

    typedef sequence<ServiceTemplate> ServiceTemplates;

    interface Service : org::smpte::st2071::_2014::query::Queryable
    {
        // UCN = "urn:smpte:ucn:org.smpte.st2071:service_v1.0";

        readonly attribute org::smpte::st2071::_2014::identity::USN USN;
        // raises (org::smpte::st2071::_2014::security::SecurityException);
        readonly attribute ServiceTemplate Type;
        // raises (org::smpte::st2071::_2014::security::SecurityException);
        readonly attribute org::smpte::st2071::_2014::identity::URLs URLs;
        // raises (org::smpte::st2071::_2014::security::SecurityException);
        readonly attribute org::smpte::st2071::_2014::types::Map Attributes;
        // raises (org::smpte::st2071::_2014::security::SecurityException);
        readonly attribute org::smpte::st2071::_2014::types::Capabilities Capabilities;
        // raises (org::smpte::st2071::_2014::security::SecurityException);
    };

    valuetype ServiceInformation supports Service
    {
        readonly attribute org::smpte::st2071::_2014::types::DateTime Created;
        // raises (org::smpte::st2071::_2014::security::SecurityException);
        readonly attribute wstring CreatedBy;
        // raises (org::smpte::st2071::_2014::security::SecurityException);
        readonly attribute org::smpte::st2071::_2014::types::DateTime ValidTill;
    };

    typedef sequence<ServiceInformation> ServiceInformations;

    exception ServiceNotFound
    {
        org::smpte::st2071::_2014::identity::USN USN;
        wstring Message;
    };
```

```
        exception ServiceNotBound
        {
            org::smpte::st2071::_2014::identity::USN USN;
            wstring Message;
        };

        exception ServiceAlreadyBound
        {
            org::smpte::st2071::_2014::identity::USN USN;
            wstring Message;
        };

        exception ServiceNotUnbound
        {
            org::smpte::st2071::_2014::identity::USN USN;
            wstring Message;
        };

        exception ServiceException
        {
            org::smpte::st2071::_2014::identity::USN USN;
            wstring Message;
        };

        interface ServiceDirectory
        {
            // UCN = "urn:smpte:ucn:org.smpte.st2071:service_directory_v1.0";

            readonly attribute org::smpte::st2071::_2014::identity::USNs Namespaces;

            // raises (org::smpte::st2071::_2014::security::SecurityException);
            void bind(in ServiceInformation service)
            raises(ServiceNotBound, ServiceAlreadyBound,
                   org::smpte::st2071::_2014::security::SecurityException);

            void unbind(in org::smpte::st2071::_2014::identity::USN usn)
            raises(ServiceNotFound, ServiceNotUnbound,
                   org::smpte::st2071::_2014::security::SecurityException);
        };

        interface ServiceRegistry
        {
            // UCN = "urn:smpte:ucn:org.smpte.st2071:service_registry_v1.0";

            ServiceInformation lookup(in org::smpte::st2071::_2014::identity::USN usn)
            raises(ServiceException, org::smpte::st2071::_2014::security::SecurityException);

            ServiceInformations search(in org::smpte::st2071::_2014::query::QueryExpression query)
            raises(ServiceException, org::smpte::st2071::_2014::security::SecurityException);
        };

        interface ServiceFactory
        {
            ServiceTemplates listTemplates(
                                 in org::smpte::st2071::_2014::query::QueryExpression filter)
            raises(ServiceException, org::smpte::st2071::_2014::security::SecurityException);

            org::smpte::st2071::_2014::identity::USN create(
                                   in org::smpte::st2071::_2014::identity::USN usn,
                                   in org::smpte::st2071::_2014::types::Map parameters)
            raises(ServiceException, org::smpte::st2071::_2014::security::SecurityException);

            org::smpte::st2071::_2014::identity::USN destroy(
                                   in org::smpte::st2071::_2014::identity::USN usn)
            raises(ServiceException, org::smpte::st2071::_2014::security::SecurityException);
        };
    };

    module media
```

```
{
    enum TRACK_TYPE
    {
        _AUDIO,
        _VIDEO,
        _TIMECODE,
        _TEXT,
        _DATA
    };

    valuetype Media supports org::smpte::st2071::_2014::query::Queryable
    {
        readonly attribute org::smpte::st2071::_2014::identity::UMN UMN;
        readonly attribute wstring Name;
        readonly attribute org::smpte::st2071::_2014::identity::UMN Location;
        readonly attribute org::smpte::st2071::_2014::types::DateTime Created;
        readonly attribute org::smpte::st2071::_2014::types::DateTime Modified;
        readonly attribute org::smpte::st2071::_2014::types::Map Metadata;
    };

    valuetype MediaPointer
    {
        readonly attribute org::smpte::st2071::_2014::identity::UMN Source;
        readonly attribute unsigned long long InpointOffset;
        readonly attribute unsigned long long OutpointOffset;
        readonly attribute org::smpte::st2071::_2014::types::OFFSET_TYPE OffsetType;
        readonly attribute short Track;
    };

    valuetype MediaSegment supports MediaPointer, Media
    {
        readonly attribute org::smpte::st2071::_2014::types::DateTime Inpoint;
        readonly attribute org::smpte::st2071::_2014::types::DateTime Outpoint;
        readonly attribute org::smpte::st2071::_2014::identity::URI Format;
        readonly attribute wstring TrackName;
        readonly attribute TRACK_TYPE TrackType;
    };

    typedef sequence<MediaPointer> MediaPointers;
    typedef sequence<MediaSegment> MediaSegments;

    valuetype MediaContainer supports Media
    {
        readonly attribute org::smpte::st2071::_2014::identity::UDN UDN;
    };

    valuetype MediaAsset supports Media
    {
        readonly attribute org::smpte::st2071::_2014::types::DateTime Duration;
        readonly attribute MediaSegments Composition;
    };

    valuetype MediaFile supports Media
    {
        readonly attribute wstring MIMEType;
        readonly attribute unsigned long long Size;
    };

    valuetype MediaInstance supports MediaFile, MediaAsset
    {
        // readonly attribute org::smpte::st2071::_2014::types::FramedTime Duration;
        // readonly attribute MediaFormatSegments Composition;
    };

    valuetype MediaBundle supports MediaContainer, MediaInstance
    {
    };

    exception MediaNotFound
    {
        org::smpte::st2071::_2014::identity::UMN UMN;
```

```
    wstring Message;
};

exception MediaCreationFailed
{
    Media Media;
    wstring Message;
};

exception MediaDeletionFailed
{
    Media Media;
    wstring Message;
};

exception MediaUpdateFailed
{
    Media Media;
    wstring Message;
};

typedef sequence<Media> MediaList;

interface MediaDirectory
{
    // UCN = "urn:smpte:ucn:org.smpte.st2071:media_directory_v1.0";

    readonly attribute MediaContainer MediaContainer;
    // raises (org::smpte::st2071::_2014::security::SecurityException);

    Media create(in Media media, in MediaPointers pointers)
    raises(MediaNotFound, MediaCreationFailed,
           org::smpte::st2071::_2014::security::SecurityException);

    Media delete(in org::smpte::st2071::_2014::identity::UMN umn)
    raises(MediaNotFound, MediaDeletionFailed,
           org::smpte::st2071::_2014::security::SecurityException);

    MediaList list(in org::smpte::st2071::_2014::identity::UMN container,
                   in org::smpte::st2071::_2014::query::QueryExpression filter)
    raises(MediaNotFound, org::smpte::st2071::_2014::query::InvalidQuery,
           org::smpte::st2071::_2014::security::SecurityException);

    Media lookup(in org::smpte::st2071::_2014::identity::UMN umn)
    raises(MediaNotFound, org::smpte::st2071::_2014::security::SecurityException);

    MediaAsset lookupAsset(in wstring mid)
    raises(MediaNotFound, org::smpte::st2071::_2014::security::SecurityException);

    MediaList search(in org::smpte::st2071::_2014::identity::UMN container,
                     in org::smpte::st2071::_2014::query::QueryExpression filter)
    raises(MediaNotFound, org::smpte::st2071::_2014::query::InvalidQuery,
           org::smpte::st2071::_2014::security::SecurityException);

    Media update(in Media media)
    raises(MediaNotFound, MediaUpdateFailed,
           org::smpte::st2071::_2014::security::SecurityException);
};

interface MediaAccess
{
    // UCN = "urn:smpte:ucn:org.smpte.st2071:media_access_v1.0";

    org::smpte::st2071::_2014::identity::URLs lookupURLs(
                                      in org::smpte::st2071::_2014::identity::UMN umn)
    raises(MediaNotFound,
           org::smpte::st2071::_2014::security::SecurityException);
};

/*
 * Media Events are sent by the Media Directory.
```

```
         *
         * Valid Device Type Values
         *     'Created'  : Indicates that a Media has been created.
         *     'Updated'  : Indicates that a Media has been updated.
         *     'Deleted'  : Indicates that a Media has been deleted.
         */
        valuetype MediaEvent supports org::smpte::st2071::_2014::event::StatusEvent
        {
            readonly attribute Media Media;
        };
    };

    module query
    {
        valuetype NOT supports Expression
        {
            factory init(in Expression expr);
        };

        valuetype OR supports Expression
        {
            factory init(in Expression expr1, in Expression expr2);
        };

        valuetype AND supports Expression
        {
            factory init(in Expression expr1, in Expression expr2);
        };

        valuetype MATCHES supports Expression
        {
            factory init(in wstring field, in wstring regexp);
        };

        valuetype CONTAINS supports Expression
        {
            factory init(in org::smpte::st2071::_2014::media::MediaPointer segment);
        };

        valuetype IMPLEMENTS supports Expression
        {
            factory init(in org::smpte::st2071::_2014::mode::Mode mode,
                         in org::smpte::st2071::_2014::types::Capabilities interfaces);
            factory init(in org::smpte::st2071::_2014::types::Capabilities interfaces);
        };

        valuetype LESS_THAN supports Expression
        {
            factory init(in wstring field, in long long number);
            factory init(in wstring field, in double number);
            factory init(in wstring field, in org::smpte::st2071::_2014::types::DateTime time);
            factory init(in wstring field, in any serializable);
        };

        valuetype GREATER_THAN supports Expression
        {
            factory init(in wstring field, in long long number);
            factory init(in wstring field, in double number);
            factory init(in wstring field, in org::smpte::st2071::_2014::types::DateTime time);
            factory init(in wstring field, in any serializable);
        };

        valuetype EQUALS supports Expression
        {
            factory init(in wstring field, in long long number);
            factory init(in wstring field, in double number);
            factory init(in wstring field, in org::smpte::st2071::_2014::types::DateTime time);
            factory init(in wstring field, in any serializable);
        };
    };
    };};};};
```