# SMPTE STANDARD

# Media Device Control — Part 1: Framework (MDCF)



## Table of Contents





Approved
October 8, 2012

## 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 Part XIII of its Operations Manual.

SMPTE ST 2071-1 was prepared by Technology Committee 34CS on Media Systems, Control and Services.

## Intellectual Property

At the time of publication no notice had been received by SMPTE claiming patent rights essential to the implementation of this Standard. 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. But, 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 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. The first diagram (Figure I-1) illustrates the flow a client would follow to search for devices and media, while the second diagram (Figure I-2) depicts a client that has implicit knowledge about a device and accesses the device directly, without the use of the directories.

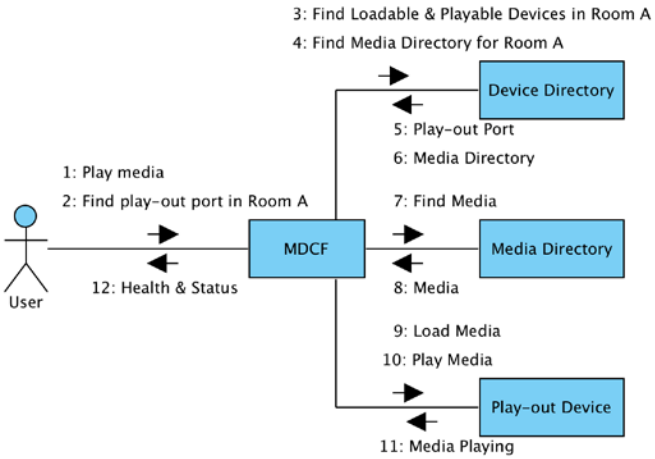



***Figure I-1 – Using the MDC Directories to Play Media***

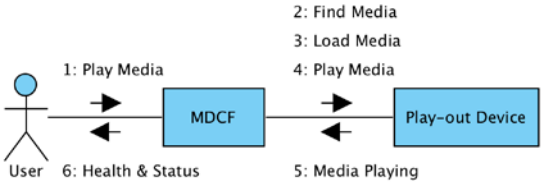



***Figure I-2 – Directly Connecting to the Play Device***

# 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. The MDC is designed to bridge the gap between workflow level interfaces and the physical hardware. MDC 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 due to the uncontrolled nature of public networks.

# 2 Conformance Notation

Normative text is text that describes elements of the design that are indispensable or text that contains the conformance language keywords: "shall," "should," or "may." Informative text is text that is potentially helpful to the user, but not indispensable, and that 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 recommended practice. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this recommended practice are encouraged to investigate the possibility of applying the most recent edition of the standards indicated below.

[RFC 1738] Internet Engineering Task Force (IETF) (1994, December). RFC 1738 — Uniform Resource Locator (URL) : Uniform Resource Locators (URL. http://www.ietf.org/rfc/rfc1738.txt

[RFC 2046] Internet Engineering Task Force (IETF) (1996, November). RFC 2046 — Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. http://www.ietf.org/rfc/rfc2046.txt

[RFC 2141] Internet Engineering Task Force (IETF) (1997, May). RFC 2141 — Uniform Resource Name (URN) : URN Syntax. http://www.ietf.org/rfc/rfc2141.txt

[RFC 3986] Internet Engineering Task Force (IETF) (2005, January). RFC 3986 — Uniform Resource Identifiers (URI) : Generic Syntax. http://www.ietf.org/rfc/rfc3986.txt

[RFC 5234] Internet Engineering Task Force (IETF) (2008, January). RFC 5234 — Augmented BNF for Syntax Specifications: ABNF. http://www.ietf.org/rfc/rfc5234.txt

[UML] The Object Management Group (OMG) (2007, February). UML Superstructure, v2.1.1. http://www.omg.org/cgi-bin/doc?formal/07-02-05

[UML] The Object Management Group (OMG) (2007, February). UML Infrastructure, v2.1.1. http://www.omg.org/cgi-bin/doc?formal/07-02-06

[IDL] The Object Management Group (OMG) (1998, December). Corba, v2.3. http://www.omg.org/spec/CORBA/2.3/

# 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. The identity scheme shall be extensible and backward compatible, providing mechanisms to simplify integration with existing vendor systems and proprietary identity schemes.

## 4.1 Media Device Control Resource Name

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

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
SCHEME NAMESPACE ATTRIBUTE_MAP
SCHEME = “urn:”
NAMESPACE = “smpte:” NAME_TYPE “:” SCOPE “:”

NAME_TYPE = 1*(DIGIT / ALPHA)
SCOPE = NAME *(“.” NAME)
ATTRIBUTE_MAP = ATTRIBUTE *((“,” / “;”) ATTRIBUTE )
ATTRIBUTE = NAME [“=” VALUE]
NAME = 1*(DIGIT / ALPHA / “+” / “-“ / “_”)
VALUE = [DQUOTE] 1*(DIGIT / ALPHA / “+” / “-“ / “_” / ESCAPE_SEQUENCE / “&” / “@” / “$” / “!” / “/”
/ “(“ / “)”) [DQUOTE]
ESCAPE_SEQUENCE = “%” 2HEXDIG
;end ABNF notation
```

### 4.1.1 Namespace

Media Device Control Resource Names are assigned to and arranged by namespace. The namespace is 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 Media Device Control Framework and group resources by system, sub-system and/or purpose. Each namespace has 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 Media Device Control Framework.

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 will change.
2. If a resource is moved to another sub-system, the namespace of that resource will change.
3. Changes to the namespace assignment of a resource will change the identity of that resource within the Media Device Control Framework, even if the vendor specific identity of that resource remains the same.

#### 4.1.1.1 Scope

A Scope is 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 are 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 Media Device Control Framework. 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 Attributes

The Media Device Control Framework identity scheme shall provide a space for vendor specific information, known as attributes. The attributes are 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 attribute and the value being separated by an equals [“=”] sign. The 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 attribute, as long as those contents can be represented as a string of characters.

### 4.2 Uniform Device Name (UDN)

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

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
SCHEME NAMESPACE ATTRIBUTE_MAP
SCHEME = “urn:”
NAMESPACE = “smpte:” NAME_TYPE “:” SCOPE “:”

NAME_TYPE = “udn”
SCOPE = NAME *(“.” NAME)
ATTRIBUTE_MAP = ATTRIBUTE *((“,” / “;”) ATTRIBUTE)
ATTRIBUTE = NAME [“=” VALUE]
NAME = 1*(DIGIT / ALPHA / “+” / “-“ / “_”)
VALUE = [DQUOTE] 1*(DIGIT / ALPHA / “+” / “-“ / “_” / ESCAPE_SEQUENCE / “&” / “@” / “$” /
        “!” / “/” / “(“ / “)”) [DQUOTE]
ESCAPE_SEQUENCE = “%” 2HEXDIG
;end ABNF notation
```

### 4.3 Uniform Media Name (UMN)

The Uniform Media Name (UMN) is a Media Device Control Resource Name of type [“umn”], with [“umn:”] being a new sub-namespace of the [“urn:smpte:”] namespace that uniquely identifies media within the Media Device Control Framework. UMNs shall contain a type [“type”] attribute that indicates the type of media represented by the UMN and, where applicable, a Media Identifier (MID) [“mid”] attribute that depicts the media’s association to other media within the system.

Example UMNs
urn:smpte:umn:Uniform.Namespace:type=media_type;mid=value; attribute =value
urn:smpte:umn:company.location.ingest.1:
urn:smpte:umn:company.location.nearline:type=instance;mid=[*MID*]
urn:smpte:umn:company.location.nearline:type=instance;path=/sports/FCBB;file=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
SCHEME NAMESPACE ATTRIBUTE_MAP
SCHEME = “urn:”
NAMESPACE = “smpte:” NAME_TYPE “:” SCOPE “:”

NAME_TYPE = “umn”
SCOPE = NAME *(“.” NAME)
ATTRIBUTE_MAP = TYPE *((“,” / “;”) (ATTRIBUTE / MID))
ATTRIBUTE = NAME [“=” VALUE]
NAME = 1*(DIGIT / ALPHA / “+” / “-“ / “_”)
VALUE = [DQUOTE] 1*(DIGIT / ALPHA / “+” / “-“ / “_” / ESCAPE_SEQUENCE / “&” / “@” / “$” /
        “!” / “/” / “(“ / “)”) [DQUOTE]
ESCAPE_SEQUENCE = “%” 2HEXDIG

MID = "mid=" VALUE
TYPE = "type=" VALUE
;end ABNF notation
```

### 4.3.1 Type

The type attribute is a required attributed of the UMN and is used to specify the type of the media. The type attribute shall be denoted with the attribute name [“type”] and its value shall specify one of the enumerated media types enumerated in Table 1.

***Table 1* – Media *Types***

| Media Type | Description |
|---|---|
| media_asset | Indicates that the media is a Media 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 |

### 4.3.2 Material Identifier (MID)

The Material Identifier (MID) is a locally created, locally managed identifier that identifies a unique collection of audio-visual material. The MID does not uniquely identify media instances or files, but identifies the grouping of like instances. One or more media instances or files may share the same MID, provided that each instance or file contains “identical audio-visual essence”, where “identical audio-visual essence” refers to essence that is perceivably identical when viewed by a human observer at the time the essence is played out. The MID is identical in purpose and function to the SMPTE Unique Material Identifier (UMID) and a SMPTE UMID may be used as a MID. Please refer to SMPTE ST 330 for a detailed description of the SMPTE UMID, its generation, depiction, and for a detailed explanation of a Material Identifier. The MID attribute shall be denoted with the attribute name [“mid”] and shall be present for UMN identifying Media Assets and all of the Media Asset sub-types.

## 4.4 Uniform Capability Name (UCN)

The Uniform Capability Name (UCN) is a Uniform Resource Name (URN) of type [“ucn”], with [“ucn:”] being a new sub-namespace of the [“urn:smpte:”] namespace that uniquely identifies capability interfaces. The UCN shall contain the capability interface’s name, version, and revision and if the interface is a proprietary vendor interface, the UCN shall contain the name of the vendor. Each capability interface shall have a unique UCN that is used by Media Device Control clients to identify the interfaces implemented by a device.

Example UCNs
urn:smpte:ucn:playable_v1
urn:smpte:ucn:playable_v1.0.0
urn:smpte:ucn:some_company: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
UCN = UCN_TYPE [VENDOR_NAME “:”] INTERFACE  “_v” VERSION *(“.” REVISION)

UCN_TYPE = “urn:smpte:ucn:”
VENDOR_NAME = NAME
INTERFACE = NAME
VERSION = 1*DIGIT
REVISION = 1*DIGIT
NAME = 1*(DIGIT / ALPHA / “+” / “-“ / “_”)
;end ABNF notation
```

## 4.5 Namespace Names

Namespace names are Media Device Control Resource Names that are comprised only of the URN scheme [“urn:”] and the URN namespace; no additional content can appear after the last colon [“:”]. Namespace names are used by the Media Device Control Framework 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 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
SCHEME NAMESPACE
SCHEME = “urn:”
NAMESPACE = “smpte:” NAME_TYPE “:” [SCOPE] “:”

NAME_TYPE = (“umn” / “udn”)
SCOPE = NAME *(“.” NAME)
;end ABNF notation
```

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:

### 4.5.1 Root Namespace Names

Root namespaces are the root most namespaces afforded by the identity scheme. There is a root namespace name for each of the Media Device Control Resource Name types (UDN and UMN) and each namespace shall descend from the root namespace of its respective type. Root namespace names are depicted as namespace names that have no Scope, containing an empty Scope string and therefore ending with a double colon [“::”]. The root namespace name for the Uniform Device Name (UDN) shall be [“urn:smpte:udn::”] and the root namespace name for the Uniform Media Name (UMN) shall be [“urn:smpte:umn::”].

### 4.5.2 Determining the Parent Namespace Name

Each namespace descends from a parent namespace, to the root namespace. To determine the parent namespace, the text between the last period of the namespace name’s Scope and the last colon [“:”] of the namespace name are removed. If no period [“.”] is present within the Scope the entire Scope is 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:” 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 represent resources as a doubly linked hierarchical tree, known as the resource hierarchy. Directories expose the operations necessary to lookup, list, search, and manipulate the resource hierarchy and the resources contained therein. Directories may act 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 resource hierarchy and resources internally. Directories may also aggregate multiple child directories into a single, consolidated view, acting as a centralized repository.

## 5.1 Registries

Registries are directories that allow for the manipulation of the resource hierarchy through the dynamic registration and de-registration of resources. Resources are added to the registry through a process known as binding and once bound a resource can be unbound from the registry, removing all references and information pertaining to that resource from the registry. Resources can also be marked as offline to indicate that the resource is no longer available, but the registry will retain all references and information pertaining to the resource.

## 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 in which the resource hierarchy is shared across directory instances is determined by the implementation; however, each directory instance in a load-balanced set shall broadcast a signal indicating changes to the resource hierarchy to all registered listeners, upon the successful committing of those changes to the resource hierarchy. In order to prevent inconsistent results or return values, the changes to the resource hierarchy shall not be made available to operation executions or attribute reads that are executing at the time of the change. The changes to the resource hierarchy shall only be made available to new operation executions and attribute reads that are executed immediately after the successful commit of the changes to the resource hierarchy. The operational steps regarding a change to the resource hierarchy in a load-balanced set are indicated below.

1. A request is received to modify the resource hierarchy.
2. The modification is applied to the resource hierarchy of the directory receiving the request.
3. The modification request is broadcast to the remaining directories in the load-balanced set. The other directories start at step 1 and skip step 3.
4. The change is committed to the resource hierarchy of the directory receiving the change.
5. New operation executions and attribute reads reflect the change.
6. A signal indicating the change is broadcast to the all listeners registered with the directory.

# 6 Devices

Devices are software services that represent physical hardware or perform systemic functions. Devices are identified by unique Uniform Device Names and shall have one or more URLs providing the information required for connecting to the device. Each device shall implement one or more capability interfaces, each capability interface having its own list of URLs and being independently addressable from the device. The device shall expose a list of these capability interfaces. Attributes shall be associated with each supported capability interface and these attributes further describe the specific characteristics that constrain the device, such as the number of supported audio channels. A device may be accessed directly, without the use of a device directory, if the client is configured with one or more of the URLs for the device. The device can instruct the client on its supported interfaces, attributes, name and additional URLs via the Device interface. Information pertaining to the resource hierarchy and the relationship of the device within the resource hierarchy must be acquired from a device directory.

## 6.1 Capability Interfaces

Capability interfaces are “atomic” level interfaces that define the minimal set of operations, attributes, and signals required to perform a concise feature or function, also referred to as the smallest unit of control. Each capability interface shall be assigned a unique Uniform Capability Name (UCN), with the UCN depicting the name, version, revision and optionally the name of the vendor that created the capability interface. Devices shall implement one or more capability interfaces, each capability interface exposing a specific feature set. Capability interfaces shall be independently accessible, with the ability to be directly accessed by the client independently from the rest of the device interface.

### 6.1.1 Attributes

Attributes are read-only properties that reflect the state of the device implementing the capability interface to which the attributes are bound. Accessing or reading attributes shall not alter the state of the device and should be a lightweight operation, requiring minimal system resources (CPU cycles, memory, network bandwidth, etc.).

### 6.1.2 Operations

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

## 6.2 Signals

Signals are asynchronous events generated by a device to indicate that the state of the device has changed. Signals are broadcast to clients that have registered with the device to receive signals; such clients are referred to as listeners. Signals can be delivered to the listeners in two ways, polling or asynchronous callback. The interfaces used to deliver signals to the listeners are defined in the Device Status and Events section.

## 6.3 Device Modes

A device mode is an operational state that defines the capabilities and behavior of a device. Each device 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. The Media Device Control Framework allows devices to support multiple modes of operation using the Mode Support capability interface. The Mode Support capability interface allows a device to have a base set of capability interfaces that are available regardless of the active mode of the device and a list of modes, which each contain a list of capability interfaces exposed by the device when that mode is active. The mode is changed or set using the operations exposed by the Mode Support interface. The capability interfaces supported by a device 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* $\cup$ *Active Mode Capability Interfaces*

## 6.4 Device Directory

The Device Directory is a type of directory that exposes devices. It is one of the main entry points defined by the framework and represents the devices within the system as a hierarchical tree of devices. Each Device Directory manages and represents a list of namespaces, their child namespaces and all of the devices assigned to those namespaces. Since directories are also devices, the Device Directory can contain references to other Device Directories, forming a hierarchy of 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

translate Uniform Device Names into URLs that can be used by the client to connect to a device. These URLs may point directly to the device or they may point to some software service that translates the MDC requests into instructions understood by the device.

### 6.4.1 Device Registry

The Device Registry is a capability interface that allows devices to be bound and unbound to a Device Directory at run-time. Device Directories that allow for the dynamic binding and unbinding of devices at run-time must also implement the Device Registry capability interface. Device Registries are typically used to create aggregate Device Directories that consolidate a number of directories into a single view.

### 6.4.2 Device Hierarchy

Device Directories represent the devices within the system 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 (Scope). As a rule, devices can directly interoperate with their parents, their parent's siblings, their siblings and their direct children.

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

- Each device must be in the same namespace or a child namespace of its parent. Devices extend or share the scope of their parent.
- Each Device Directory represents one or more namespaces, processing request for those namespaces and the namespaces must extend or share the scope of Device Directory's UDN.
- All devices within the Device Directory must share or extend one of the namespaces the Device Directory represents.
- Aggregate Device Directories shall delegate requests to the child directory that represents the namespace that is the closet match to the namespace specified in the request's UDN.

### 6.4.3 Namespaces and Scope

Device Directories represent one or more namespaces and the namespaces represented by the Device Directory must extend the namespace defined in the UDN identifying the Device Directory. Device Directories also arrange devices by the namespace specified in the device's UDN. Devices within the Device Directory must be assigned to one of the namespaces represented by the Device Directory or one of the namespace's child namespaces.

## 7 Media

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

### 7.1 Media Assets

Media Assets are the logical view of a unique piece of media, whether a contiguous clip or any collection of media elements. Media Assets are a purely data representations of a unique collection of audio-visual elements, frames or samples. Each instance of a Media Asset is guaranteed to be visually and audibly identical when played out and possesses a unique Material Identifier (MID).

### 7.2 Media Files

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

### 7.3 Media Instances

Media Instances are physical manifestations of Media Assets. For any given Media Asset there can be zero or more Media Instances, each guaranteed to be visually and audibly identical when played out. A Media Instance is typically a file, but some media systems have an internal representation of a Media Instance that does not produce a file until the instance is egressed. For this reason Media Instances do not directly contain URLs, URLs for Media are generated at the time of egress by the capability interfaces designed to move or stream media.

### 7.4 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 a directory in a POSIX file system.

### 7.5 Media Bundles

A Media Bundle is a Media Container that has the characteristics of a Media Instance. For example, an MXF OP4a or a TAR file may be represented as a Media Bundle. Media Bundles are collections of Media Instances that relate with one another to produce a single audio-visual clip when played out or streamed.

### 7.6 Media Pointers and Segments

Media is regularly composed of pieces from other media. The Media Device Control Framework defines four data structures that are used to represent sub-sets of a Media. These structures are the Media Pointer, Media Segment, Media Format Pointer and the Media Segment Pointer. Each of these structures extends from the Media Pointer.

#### 7.6.1 Media Pointers

Media Pointers represent the in and out point offsets within a piece of media. The Media Format Pointer adds format information for the representation of the media’s composition.

#### 7.6.2 Media Segments

Media Segments are representations of the in and out points of a piece of media which include the in and out point date time or time code values. Media Segments are primarily used to reflect the composition time line for a piece of media, but may be used to specify the segment of media using the date time or time code values for the in and out points. The Media Format Segment adds format information to the Media Segment for a complete representation of the media composition, including in and outpoint offsets, time values, format and track.

### 7.7 Media Directory

The Media Directory is 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 exposes 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 framework and can be used in conjunction with or independently of a Device Directory. Like Device Directories, Media Directories are bound to namespaces and all media within the Media Directory must reside within one of those namespaces or one of the child namespaces

# 8 Query Expressions and Query Syntax

One of the primary purposes of the Device and Media Directories is to provide the ability to discover or search for devices and media. To facilitate this requirement a simple object based query expression syntax is defined which allows for basic Boolean expressions, such as AND, OR, LESS THAN, GREATER THAN, EQUALS, MATCHES, CONTAINS and NOT. Additional domain specific operations are also included to facilitate searches for devices that implement specific capabilities and media that contain specific audio-visual essence.

Table 2 describes the syntactic elements for the Query Expression. The UML describing the Query Expression syntax object notation is defined in Diagram 12.5 – Query Syntax Object Notation UML.

***Table 2 – Query Expression Syntax***

| Expression | Description |
|---|---|
| NOT | Syntax: NOT(*expr1*)<br><br>Boolean NOT operation. Inverts *expr1*, equates to TRUE if *expr1* is FALSE. |
| AND | Syntax: AND(*expr1, expr2*)<br><br>Boolean AND operation. Equates to TRUE if both *expr1* and *expr2* are TRUE. |
| OR | Syntax: OR(*expr1, expr2*)<br><br>Boolean OR operation. Equates to TRUE if either *expr1* or *expr2* are TRUE. |
| EQUALS | Syntax: EQUALS(*field, value*)<br><br>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*)<br><br>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*)<br><br>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*)<br><br>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*)<br><br>Media centric CONTAINS operation. Equates to TRUE if the tested Media contains the essence referred to by the *pointer* parameter. |
| IMPLEMENTS | Syntax: IMPLEMENTS([*mode*,] *capabilities*)<br><br>Device 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 must 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 |

| Expression | Description |
|---|---|
| | 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)<br><br>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)<br><br>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. |

## 8.1 Attribute Designation

Numerous Query Expressions must 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"* allows for these child attributes to be referenced.

### 8.1.1 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]*" is 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 / “-“ / “_”)
QUOTE = (“\”” / “\’”)
;end ABNF notation
```

Example attribute designators:

Name
Capabilities[1]
UDN.Namespace
UMN.Attributes[“MID”]

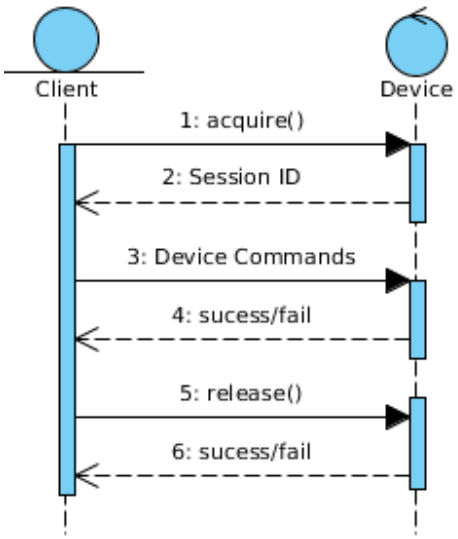
# 9 Delegation of Control

Some devices are limited in the number of concurrent sessions they can support. The Media Device Control Framework 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 possess a session. However, since device attributes do not alter the state of a device nor does reading them significantly impact the devices, device attributes shall be available at all times and clients are not require to acquire a session before reading them.

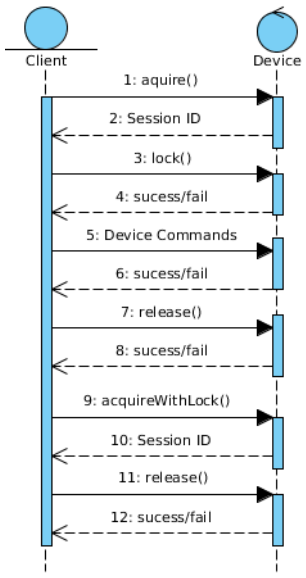
## 9.1 Acquiring a Device Session

To acquire a session the client must 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 must release the session. There shall be a corresponding release for every successful acquisition of a session.



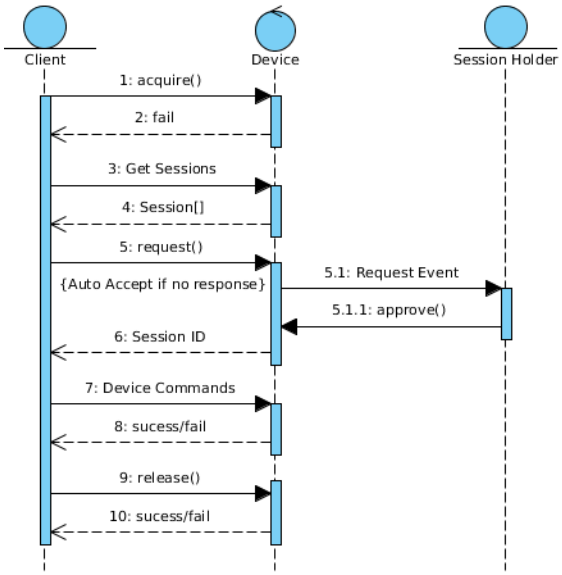

## 9.2 Exclusively Locking the Device

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 Media Device Control Framework 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.



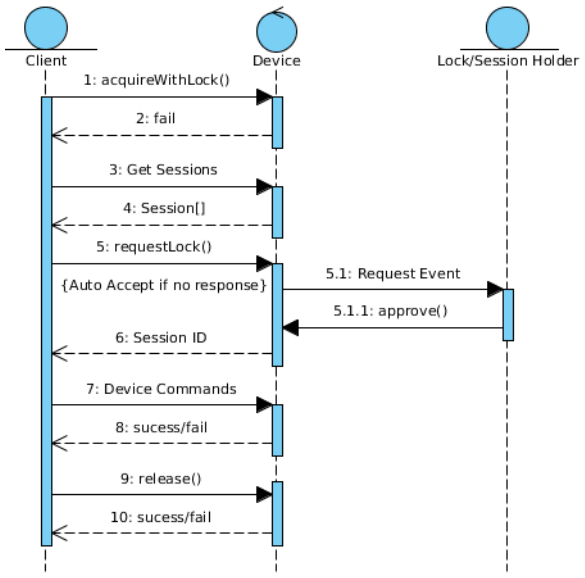

## 9.3 Requesting a Session from another Client

If a device runs out of sessions a client may opt to request a session from another client. To do so the requestor must list the sessions from the device and select a session to obtain. 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 will automatically be accepted for the requested session.



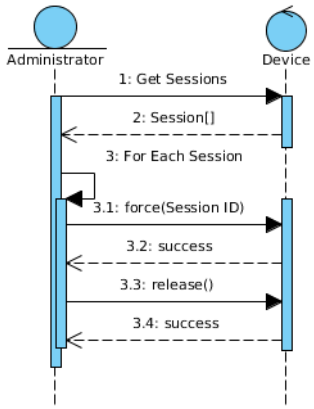

## 9.4 Requesting an Exclusive Lock

If a client wishes an exclusive lock, but there are other client sessions or if the device is locked by another client, the client may opt to request the lock from the current lock /session holders. To do so the client must send a request for the lock to each session or the current lock holder. If all of the requests are accepted the requestor is granted the exclusive lock. If any of the requests are not accepted or rejected within an amount of time configured for the device or if there are communication errors between the device and a session holder, the request will automatically be accepted for that session.




### 9.5 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 from that client will fail to execute.




## 10 Signals

Signals are one-way, asynchronous communications between one or more devices. Signals always have a point of origin or sender and may have zero or more recipients, commonly referred to a listeners. Signals are used to broadcast information to interested parties, asynchronously in the form of messages, commonly referred to as events. Listeners do not confirm the receipt of events; the guaranteed delivery of events is specified by the low-level wire transport protocol and is not within the scope of this document.

### 10.1 Polling

Polling is the process in which a client application or device executes an operation on the sending device, in order to retrieve the queued events. 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 in the network communications path.

### 10.2 Asynchronous Callback

An Asynchronous Callback is when the event sender executes an operation on the client to deliver events. In order to support Asynchronous Callbacks the client must implement some form of service that can be executed remotely and bi-directional communications must be permitted between the sending device and the receiving listener.

## 11 Authentication / Authorization

The Media Device Control Framework defines a Simple Authentication and Security Layer (SASL) compliant, role based security framework. Please refer to RFC 4422 and RFC 4752 for details regarding the Simple Authentication and Security Layer (SASL). The security framework is an optional component of the Media Device Control Framework for device implementations, but is required for client implementations. If implemented, each secured device must at a minimum implement the Authorizer capability interface, while the Authenticator capability interface may be implemented by the device or by a third party, such as an authentication server. The security framework defines a client as a Subject, with each Subject containing 0 or more security credentials to identify the Subject. Each of 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 Authenticator and Authorizer to prove that the Subject is in fact whom it claims to be. Security tokens may be user names and passwords, digital certificates or any other form of credential that is defined by SASL. It is possible to have a mixture of devices within the system, those that support the security framework and others that do not.

### 11.1 Authentication

Devices may support multiple authentication methods and allow Subjects to re-authenticate during a session, without logging out. 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 must 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 (Example: 3 failed attempts to authenticate as an administrator will cause the original session to be terminated).

### 11.2 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 lines beginning with [“C:”] indicating client transmissions and lines beginning with [“S:”] indicating server transmissions.

C: Request authentication + Initial credentials
S: Outcome of authentication with authentication tokens replaces with authorization tokens.

1. The client connects to the device.
2. The client uses the Authenticator capability interface to authenticate with the device, populating the Security Tokens with the identity and values for the authentication method. For example, username and password or identity and digital certificate.
3. The Authenticator iterates the Principals for the Provided Subject
    a. For each Principal the Authenticator validates the identity with the Security Token.
4. The Authenticator copies the supplied Subject and rewrites the Security Tokens with authentication tokens.
5. The client remembers the Subject returned by the Authenticator.

6. If the device implements the Authorizer capability interface, the client calls authorize using the Subject returned by the Authenticator.
7. The Authorizer returns the new Subject or raises an authorization failure.

### 11.3 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. If a security layer is implemented, the security layer shall begin on the $0^{th}$ byte of the request or response immediately following the response indicating the successful establishment of the security layer.

### 11.4 Authentication Servers

The security framework allows the system to have third party authentication servers. The authentication servers must implement both the Authenticator and Authorizer interfaces. In this case the devices within the system only need to implement the Authorizer interface.

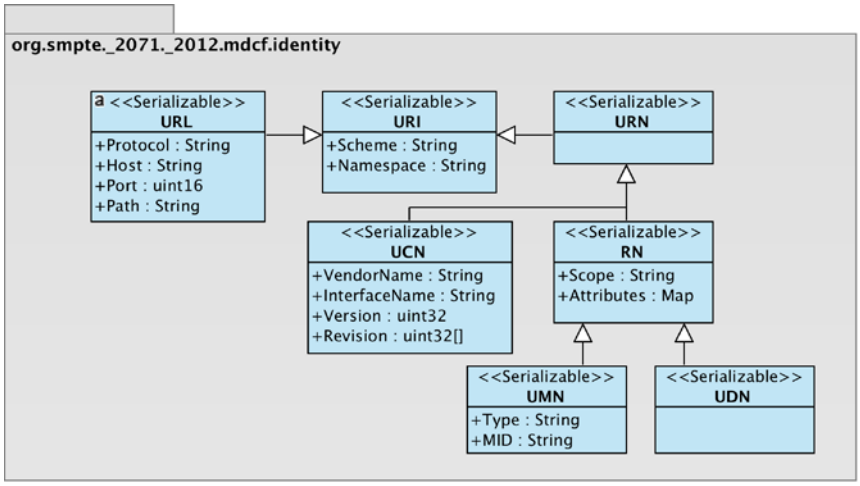
1. The client authenticates with authentication server.
2. The client authorizes with the authentication server, specifying the UDN of the destination device.
3. The client connects to the destination device.
4. The client authorizes with the device using the Subject returned by step 2, specifying the same UDN as in step 2.
5. The authorization succeeds or an error is raised indicating authorization failure.

### 11.5 Permissions

Authentication permissions can be assigned to a device, as a whole, or to specific capability interfaces on a device. Different roles may be required for different interfaces.

## 12 Data and Operation Model

### 12.1 Identity




**UML Diagram 12-1 – Identity UML**

The UML Diagram 12-1 – Identity ***UML*** illustrates the Identity model used by the Media Device Control Framework. The Identifiers are Universal Resource Names (URNs) that are wrappers around the underlying media systems Identity scheme and include routing information

### 12.1.1 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. The attributes and operations available to the URI are platform specific and not in scope for this specification.

**Attributes:**

| Name | Data Type | Description |
|---|---|---|
| Scheme | String | The Scheme of the URI. |
| Namespace | String | The Namespace of the URI. e.g. All the text between the first colon ':' and the the last colon ':'. |

### 12.1.2 URL

The Uniform Resource Locator (URL) as defined by IETF RFC 1738. A URL is 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 |
|---|---|---|
| Protocol | String | The Protocol specified in the URL. |
| 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. |

### 12.1.3 URN

The Uniform Resource Name (URN) as defined by IETF RFC 2141. A URN is 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.

### 12.1.4 RN

The Resource Name (RN) is an extension of the URN that identifies the Media Device Control Resource Name. The RN indicates the Scope and provides a list of named attributes (name/value pairs).

**Attributes:**

| Name | Data Type | Description |
|---|---|---|
| Scope | String | The Scope in which the identified resource resides. |
| Attributes | Map | Optional: List of named attributes. |

### 12.1.5 UMN

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

**Attributes:**

| Name | Data Type | Description |
|---|---|---|
| Type | String | The media's type. Must be one of the values defined in Table 1 – Media Types. |
| MID | String | The Material Identifier for the Media. Shall only be present for UMN identifying Media of types media_asset, media_instance, and the media_bundle. |

### 12.1.6 UDN

The Uniform Device Name (UDN) is an extension of the Resource Name (RN) that identifies devices. The UDN inherits the Scope and Attributes from the Resource Name (RN).

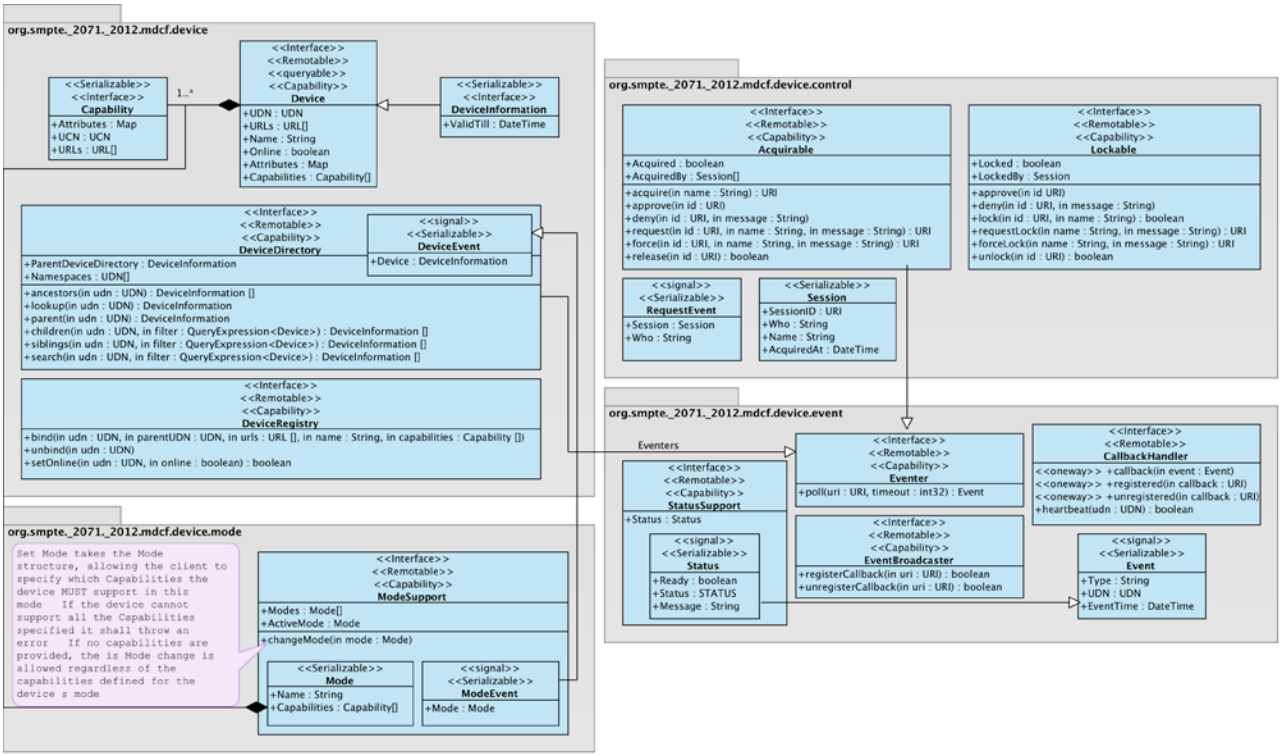
### 12.1.7 UCN

The Uniform Capability Name (UCN) is an extension of the URN that identifies Capability interfaces. The UCN indicates the name of the interface, version, revision and optionally can specify the name of the vendor, if the capability interface is a vendor specific API.

**Attributes:**

| Name | Data Type | Description |
|---|---|---|
| VendorName | String | The name of he vendor that defined the interface, if the interface is vendor specific and not standard. |
| 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. |

## 12.2 The Device Framework




**UML Diagram 12-2 – Device Framework UML**

The UML Diagram 12-2 – Device Framework ***UML*** defines the complete UML for the Media Device Control Framework device representation and the constructs required to support the devices.

### 12.2.1 DeviceDirectory

The Device Directory is 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:device_directory_v1.0

**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)**<br>Returns the list of ancestors/lineage for *udn.*<br>**Exceptions**<br>Device Not Found, SecurityException |
| DeviceInformation[] | **children(udn : UDN, QueryExpression<Device> filter)**<br>Returns the list of child devices for *udn* filtered by *filter.*<br>**Exceptions**<br>Device Not Found, Invalid Query, SecurityException |
| DeviceInformation | **lookup(udn : UDN)**<br>Returns the device information for *udn.*<br>**Exceptions**<br>Device Not Found, SecurityException |
| DeviceInformation | **parent(udn : UDN)**<br>Returns the device information for *udn*'s parent device.<br>**Exceptions**<br>Device Not Found, SecurityException |
| DeviceInformation[] | **search(udn : UDN, QueryExpression<Device> filter)**<br>Recursively searches the children of *udn* for any devices that match *filter.*<br>**Exceptions**<br>Device Not Found, Invalid Query, SecurityException |
| DeviceInformation[] | **siblings(udn : UDN, QueryExpression<Device> filter)**<br>Returns the list of sibling devices for *udn* filtered by *filter.*<br>**Exceptions**<br>Device Not Found, Invalid Query, 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. |

### 12.2.2 DeviceEvent

The Device Event is a Serializable signal interface that is transmitted by a Device when an asynchronous event occurs. 6 Each Capability interface defines 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. |

### 12.2.3 DeviceRegistry

The Device Registry Capability interface is an extension of the Device Directory that provides operations for the registration and de-registration of devices.

**UCN:** urn:smpte:ucn:device_registry_v1.0

**Operations:**

| Return Data Type | Operation / Description |
|---|---|
| n/a | **bind(udn : UDN, parentUDN : UDN, urls : URL[], name : String, capabilities : Capability[])**<br>Registers a device identified by *udn* with the *parentUDN*, the URLs *urls*, the *name* and the *capabilities.*<br><br>**Exceptions**<br>Device Not Found, Device Not Bound, Device Already Bound, SecurityException |
| n/a | **unbind(udn : UDN)**<br>Unregisters the device identified by *udn.*<br><br>**Exceptions**<br>Device Not Found, Device Not Unbound, SecurityException |
| boolean | **setOnline(udn : UDN, online : boolean)**<br>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.<br><br>**Exceptions**<br>Device Not Found, SecurityException |

### 12.2.4 Device

The Device interface is a remotely executable interface that defines the most fundamental attributes of a Device and provides a list of the capability interfaces defined for the device.

**UCN:** urn:smpte:ucn:device_v1.0

**Attributes:**

| Name | Data Type | Description |
|---|---|---|
| UDN | UDN | The Uniform Device Name of the device. |
| URLs | URL[] | The list of URLs that can 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 supported by the device. |

### 12.2.5 DeviceInformation

The Device Information is a Serializable interface that provides the means in which the information describing a device is transmitted between the Device Directory and the client. It contains all of the information represented in the Device interface, but also includes a ValidTill data element, which indicates 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 that can 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 supported by the device. |
| ValidTill | DateTime | The date and time that the device information is no longer valid. |

### 12.2.6 Capability

The capability interface is a Serializable interface that describes a device capability. The capability interface contains an identifier attribute that uniquely identifies a capability interface supported by the device and a Map of attributes describing the characteristics of the represented Capability. Each Capability interface can also be independently addressed and therefore a list of URLs is exposed for each Capability.

**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. |

### 12.2.7 Device Status and Events

#### 12.2.7.1 Event

The Event is a Serializable signal interface that is transmitted by a device to notify clients or other devices of a change in the device'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. |
| UDN | UDN | The UDN that identifies the originating device. |
| EventTime | DateTime | The date and time that the event was created. |

### 12.2.7.2 Eventer

The Eventer is a capability interface that provides the operations required for a client to request asynchronous notification of changes to a state or status from the event generator. The Eventer interface is used by the client to pull events from the event generator through the process known as polling. Polling is the process in which a client executes an operation on the event generator and the event generator will return the next Event queued for the client. If no events are available for the client, the operation execution may wait the specified amount of time before returning null, returning immediately when an event becomes available for the client. The client is registered using a URI and each client URL shall be provided a dedicated priority queue that only contains the events for that client URI. A client may register more than one URI and multiple client may share the same URI if the client is load balanced and the URI will resolve to each client in the load balanced set.

**UCN:** urn:smpte:ucn:eventer_v1.0

**Operations:**

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

### 12.2.7.3 EventBroadcaster

The EventBroadcaster is a capability interface that provides the operations required for a client to request asynchronous notification of changes to the state or status of an event broadcaster. The EventBroadcaster capability interface is used by the client to register a CallbackHandler interface that receives events from the EventBroadcaster. The EventBroadcaster initiates a *callback* operation execution on the client to broadcast the event to the client as the events become available. The client must implement the CallbackHandler interface and the *uri* provided in the EventBroadcaster *registerCallback* operation execution shall provide the information necessary for the EventBroadcaster to connect to the CallbackHandler interface on the client. Upon registration of a CallbackHandler the EventBroadcaster shall execute a *registered* operation on the CallbackHandler interface of the client. If the execution fails, the EventBroadcaster shall not register the CallbackHandler and shall raise a *Client Unreachable* exception indicating the failure. The client is registered using a URI and each client URI shall have a dedicated priority queue that shall only contains the events for that client URI. A client may register more than one URI and multiple clients may share the same URI, if the client is load balanced and the URI will resolve to each client in the load balanced set.

**UCN:** urn:smpte:ucn:event_broadcaster_v1.0

**Operations:**

| Return Data Type | Operation / Description |
|---|---|
| boolean | **registerCallback(uri : URI)**<br>Registers a callback with the event broadcaster. Whenever an event occurs the event is broadcast to the registered callback handlers. The *uri* specified must contain the information necessary for the event broadcaster to connect to the CallbackHandler interface on the client. This operation shall generate a *registered* operation execution on the client to ensure that the client is reachable before successfully registering the client. If the *registered* operation execution fails a *Client Unreachable* exception shall be thrown indicating the error. This operation shall return TRUE if the callback handler registration is successful and FALSE if the registration fails.<br><br>**Exceptions**<br>Client Unreachable, SecurityException |
| boolean | **unregisterCallback(uri : URI)**<br>Removes the callback handler declared by the *uri* from the event broadcaster. When the callback handler is unregistered, the event broadcaster shall call the *unregistered* operation on the client. Whether this operation execution is successful or not, the client shall be unregistered from the event broadcaster.<br><br>**Exceptions**<br>SecurityException |

#### 12.2.7.4 CallbackHandler

The CallbackHandler is a client interface, implemented by the client and executed by the event broadcaster to asynchronously deliver device events and status signals.

**UCN:** urn:smpte:ucn:callback_handler_v1.0

**Operations:**

| Return Data Type | Operation / Description |
|---|---|
| oneway | **callback(event : Event)**<br>The callback operation is called by the event source to deliver the event to the client. |
| oneway | **registered(callback : URI)**<br>Registered is executed whenever a new listener is registered to the device. |
| oneway | **unregistered(callback : URI)**<br>Unregistered is executed whenever a listener is unregistered or removed from the device. |

| | |
|---|---|
| boolean | **heartbeat(eventBroadcaster : UDN)**<br>Executed by the Event Broadcaster to check the health of the Callback Handler. The Callback Handler shall return TRUE if it still wishes to receive events from the Event Broadcaster, otherwise FALSE shall be returned. If the Event Broadcaster receives a network error while executing this operation, the Event Broadcaster shall unregister the Callback Handler, sending an unregistered event after doing so. |

#### 12.2.7.5 Status

The Status is a Serializable signal interface that indicates the status of the device. Status can be a reflection of the device state or it can be an indication of the device health. Many event types defined in the Media Device Control Framework extend the Status in order to reflect the general health or state of the device along with the message they are conveying.

**Attributes:**

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

#### 12.2.7.6 StatusSupport

The StatusSupport is a capability interface that provides the attributes required to query the status of a device. Status can be used to reflect the many states of a device and/or communicate to the client which action the device 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:status_support _v1.0

**Attributes:**

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

### 12.2.8 Delegation of Control

#### 12.2.8.1 Session

The Session is a Serializable interface that describes a device session or client connection. The session contains a unique 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 acquired. A URI identifier provides the flexibility to use any device specific or protocol specific mechanism for session identification.

**Attributes:**

| Name | Data Type | Description |
|---|---|---|
| SessionID | URI | The unique identification of the session. May be device, protocol specific or a UUID. |
| 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. |

#### 12.2.8.2 RequestEvent

The RequestEvent is a Serializable signal interface that is transmitted by a device when a client is requesting a session from or a lock of a device. The RequestEvent is generated by various methods defined in the Acquirable and Lockable capability interfaces and is used to notify clients of a request, that their session or lock has been taken or whether their request was approved or denied.

**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. |
| Who | String | The name of the client who is requesting the session. |

#### 12.2.8.3 Acquirable

Acquirable devices are devices that require the client to acquire a session before executing operations. Devices are not required to be acquirable, unless they require session management. The indicated exceptions are added for all operation and attribute accesses on all of the capability interfaces of the acquirable device.

**UCN:** urn:smpte:ucn:acquirable_v1.0

**Exceptions:**

| Name | Description |
|---|---|
| Not Acquired | Indicates that device must be acquired. |
| Session Terminated | Indicates that the session has been terminated by the system. See force(). |

**Attributes:**

| Name | Data Type | Description |
|---|---|---|
| Acquired | boolean | Indicates if the device is acquired. |
| AcquiredBy | Session[] | The list of session currently acquired for this device. |

**Operations:**

| Return Data Type | Operation / Description |
|---|---|
| URI | **acquire(name :String)**<br>Attempts to acquire a new session with the given *name*. If successful the URI identifying the new session is returned.<br><br>**Exceptions**<br>Device Not Acquired, Device Locked, Too Many Sessions, Name In Use, SecurityException |
| n/a | **approve(id : URI)**<br>Called by a client to approve a session request, relinquishing the session specified by *id.*<br><br>**Exceptions**<br>Request Not Found, Request Expired, SecurityException |
| n/a | **deny(id : URI, message : String)**<br>Called by a client to reject a session request for the specified session *id.* The *message* is provided so that the session holder may indicate why the request was rejected.<br><br>**Exceptions**<br>Request Not Found, Request Expired, SecurityException |
| URI | **request(id : URI, name : String, message : String)**<br>Called by a client to request a session from another client. The *id* 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.<br><br>**Exceptions**<br>Session Not Found, Request Denied, Device Not Acquired, SecurityException |
| URI | **force(id : URI, name : String, message : String)**<br>Called by a device administrator to forcefully take the session identifier by *id*. The new session is assigned the *name* and the *message* is sent to the former session holder as informative text describing the reason for the force acquisition.<br><br>**Exceptions**<br>Session Not Found, Device Not Acquired, SecurityException |
| boolean | **release(id : URI)**<br>Releases the session specified by *id*, if the caller is the current session holder.<br><br>**Exceptions**<br>Session Not Found, SecurityException |

**Events:**

| Data Type | Event Type | Description |
|---|---|---|
| DeviceEvent | Acquired | A Device session has been acquired. |
| DeviceEvent | Released | A Device session has been released. |
| RequestEvent | RequestAcquire | A client is requesting a Device 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 Device. 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. |

#### 12.2.8.4 Lockable

Lockable devices are devices that allow the client to acquire a lock on the device, which prevents other client from performing operations against or changing the state of the device. The indicated exceptions are added for all operation and attribute accesses on all of the lockable device's capability interfaces.

**UCN:** urn:smpte:ucn:lockable_v1.0

**Exceptions:**

| Name | Description |
|---|---|
| Device Locked | Indicates that device is locked. |
| Lock Terminated | Indicates that the session's lock has been terminated by the system. See forceLock(). |

**Attributes:**

| Name | Data Type | Description |
|---|---|---|
| Locked | boolean | Indicates if the device is acquired. |
| LockedBy | Session | The session currently locking this device. |

**Operations:**

| Return Data Type | Operation / Description |
|---|---|
| n/a | **approve(id : URI)**<br>Called by a client to approve a lock request, relinquishing the lock specified by *id.*<br><br>**Exceptions**<br>Request Not Found, Request Expired, SecurityException |
| n/a | **deny(id : URI, message : String)**<br>Called by a client to reject a lock request for the specified lock *id.* The *message* is provided so that the session/lock holder may indicate why the request was rejected.<br><br>**Exceptions**<br>Request Not Found, Request Expired, SecurityException |
| boolean | **lock(id : URI, name : String)**<br>Attempts to lock the device identified by *id* with the specified *name*. If successful a boolean TRUE is return.<br><br>**Exceptions**<br>Device Not Locked, Device Not Acquire, SecurityException |
| URI | **requestLock(name : String, message : String)**<br>Called by a client to request a lock on a device when others have the device locked or have sessions to the device. The *message* is sent to any clients that have sessions on the device as informative text specifying the reasons for the request. If successful a new session is created named with the specified *name* and the sessions URI is returned.<br><br>**Exceptions**<br>Session Not Found, Request Denied, Device Not Locked, SecurityException |
| URI | **forceLock(name : String, message : String)**<br>Called by an administrator to forcefully lock a device. The *message* is sent to any clients that have sessions on the device as informative text specifying the reasons for locking the device. If successful a new session is created named with the specified *name* and the sessions URI is returned.<br><br>**Exceptions**<br>Session Not Found, Device Not Locked, SecurityException |
| n/a | **unlock(id : URI)**<br>Unlocks the device, allowing other sessions and locks to be established.<br><br>**Exceptions**<br>Session Not Found, SecurityException |

**Events:**

| Data Type | Event Type | Description |
|---|---|---|
| DeviceEvent | Locked | A Device has been exclusively locked. |
| DeviceEvent | Unlocked | A Device has been unlocked. |

### 12.2.9 Device Modes

#### 12.2.9.1 Mode

The Mode is a Serializable interface that provides the means in which the information describing a mode of operation is transmitted between the device and the client. The Mode interface describes a mode of operation as a list of capability interfaces with an associated, human readable, name. A device can support zero or more modes of operation, but can have only one mode active at any given moment in time. When a mode is active, the device implementing the mode will support the capability interfaces specified for that mode along with the capability interfaces listed for the device. Therefore, the capability interfaces supported by a device is a union set of the capability interfaces listing in the Capabilities attribute of the device and the Capabilities attribute of the active mode of the device.

**Attributes:**

| Name | Data Type | Description |
|---|---|---|
| Name | String | The human readable name of the mode. |
| Capabilities | Capability | The list of Capability interfaces supported by this mode. When this mode is active, the device will support the capability interfaces specified by this mode and the capability interfaces listed for the device. |

#### 12.2.9.2 ModeEvent

The ModeEvent is a Serializable signal interface that is transmitted by a device's mode of operation changes.

**Attributes:**

| Name | Data Type | Description |
|---|---|---|
| Mode | Mode | Specifies the new active mode for the device. |

#### 12.2.9.3 ModeSupport

The ModeSupport capability interface provides the operations and attributes necessary for devices to support multiple modes of operation. A device implementing this capability interface shall have one or more modes of operation; each mode altering the capabilities of the device while it is in that mode. The device shall also have a base set of capability interfaces that are always available, regardless of the device's active mode.

**UCN:** urn:smpte:ucn:mode_support_v1.0

**Exceptions:**

| Name | Description |
|---|---|
| Invalid Mode | Indicates that device does not support the specified mode. |
| Mode Exception | Indicates a general error has occurred. |

**Attributes:**

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

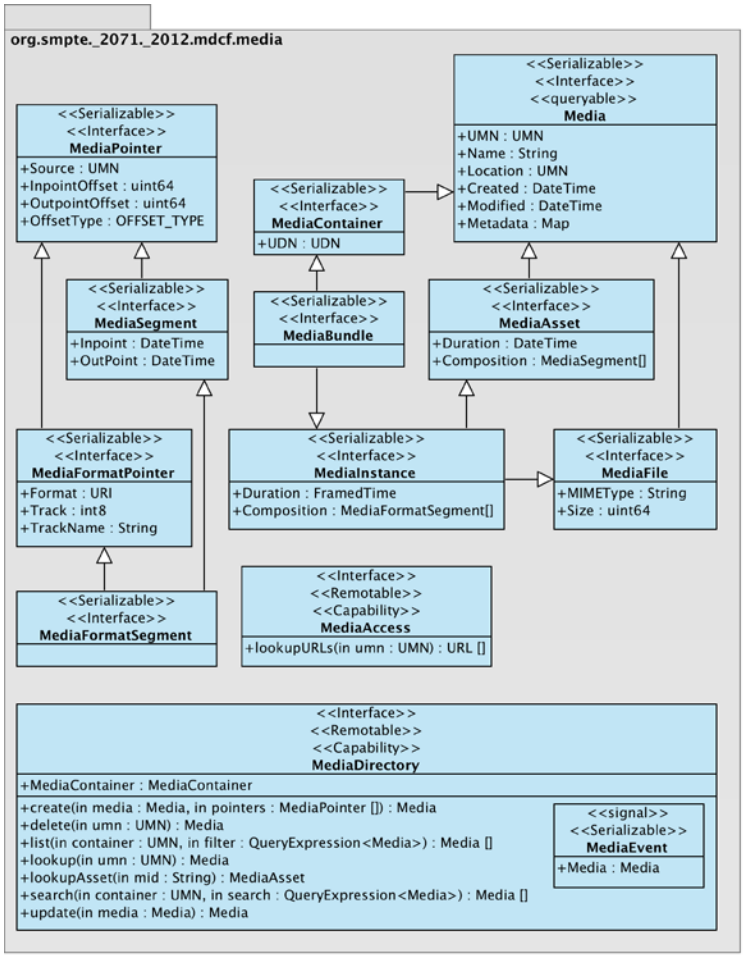
**Operations:**

| Return Data Type | Operation / Description |
|---|---|
| Mode | **changeMode(mode : Mode)**<br>Attempts to acquire a session with the given *name*, locking the device. If successful the URI identifying the new lock session is returned.<br><br>**Exceptions**<br>Device Not Acquired, Device Locked, Too Many Sessions, Name In Use, Device Not Locked, SecurityException |

**Events:**

| Data Type | Event Type | Description |
|---|---|---|
| ModeEvent | Mode Changed | The Device has changed modes. |

## 12.3 The Media Framework




**UML Diagram 12-3 – Media Framework UML**

The UML Diagram 12-3 – Media Framework ***UML*** defines the complete UML for the Media Device Control Framework representation of media and the supporting constructs.

### 12.3.1 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:media_directory_v1.0

**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[])**<br>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.<br><br>**Exceptions**<br>Media Creation Failed, Media Not Found, SecurityException |
| Media | **delete(umn : UMN)**<br>Deletes the Media identified by the *umn*, returning the Media representation for the deleted Media as it was before the deletion.<br><br>**Exceptions**<br>Media Not Found, Media Deletion Failed, SecurityException |
| Media[] | **list(container : UMN, filter : QueryExpression<Media>)**<br>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.<br><br>**Exceptions**<br>Media Not Found, Invalid Query, SecurityException |
| Media | **lookup(umn : UMN)**<br>Returns the Media identified by the *umn.*<br><br>**Exceptions**<br>Media Not Found, SecurityException |
| MediaAsset | **lookupAsset(mid : String)**<br>Returns the Media Asset identified by the *mid.*<br><br>**Exceptions**<br>Media Not Found, SecurityException |

| | |
|---|---|
| Media[] | **search(container : UMN, QueryExpression<Media> filter)**<br>Recursively searches the Media Container identified by *umn* and all of its descendants for all Media that matches the *filter.*<br><br>**Exceptions**<br>Media Not Found, Invalid Query, SecurityException |
| Media | **update(media : Media)**<br>Updates the specified Media with the metadata provided in the *media* parameter, returning the Media representation of the media after the update.<br><br>**Exceptions**<br>Media Not Found, Media Update Failed, 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. |

### 12.3.2 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:media_access_v1.0

**Operations:**

| Return Data Type | Operation / Description |
|---|---|
| URL[] | **lookupURLs(umn : UMN)**<br>Returns the zero or more URLs that can be used to access the media represented by the provided *umn*.<br><br>**Exceptions**<br>Media Not Found, SecurityException |

### 12.3.3 MediaEvent

The Media Event is a Serializable signal object that 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. |

### 12.3.4 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. |

#### 12.3.4.1 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. |

#### 12.3.4.2 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 must, at a minimum implement the Media Access capability interface. |

#### 12.3.4.3 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. |

#### 12.3.4.4 MediaInstance

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

**Attributes:**

| Name | Data Type | Description |
|---|---|---|
| Duration | FramedTime | The duration of the Media Instance in Framed Time (aka. time code). |
| Composition | MediaFormatSegment[] | The list of Media Format Segments describing the essence of the Media Instance. |

#### 12.3.4.5 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.

#### 12.3.4.6 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. |
| InpointOffset | uint64 | The inpoint offset in units indicated by the offset type. |
| OutpointOffset | uint64 | The outpoint offset in units indicated by the offset type. |
| OffsetType | OFFSET_TYPE | Indicates the offsets type. |

#### 12.3.4.7 MediaSegment

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

**Attributes:**

| Name | Data Type | Description |
|---|---|---|
| Inpoint | DateTime | The Date and Time or Framed Time of the inpoint offset. |
| Outpoint | DateTime | The Date and Time or Framed Time of the outpoint offset. |

#### 12.3.4.8 MediaFormatPointer

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

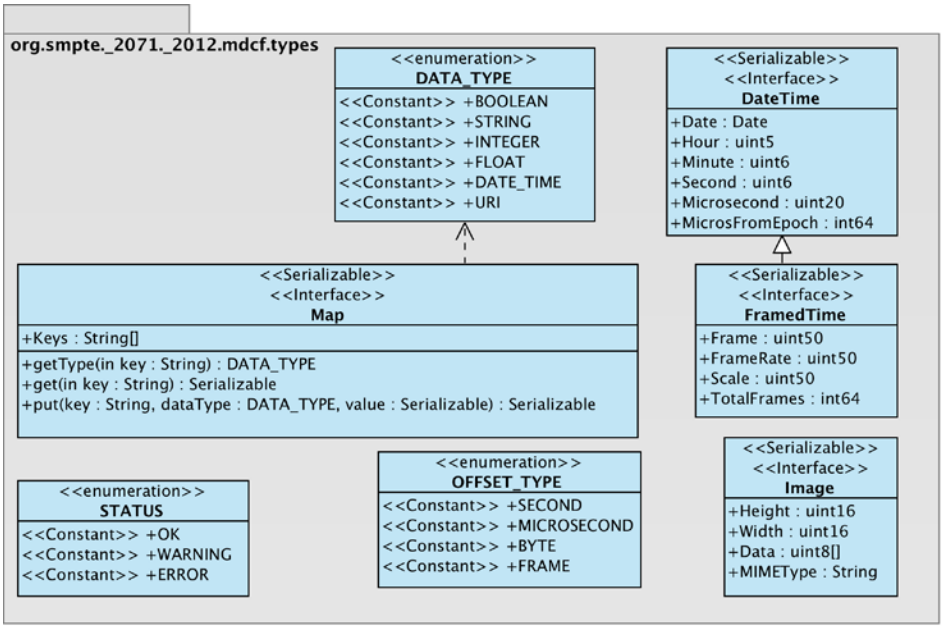
**Attributes:**

| Name | Data Type | Description |
|---|---|---|
| Format | URI | The media format of the essence. |
| Track | int8 | The track index. |
| TrackName | String | The name of the track |

#### 12.3.4.9 MediaFormatSegment

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

### 12.4 Data Types




**UML Diagram 12-4 – Data Types UML**

The UML Diagram 12-4 – Data Types ***UML*** defines a collection of reusable data types that are used throughout the Media Device Control Framework UML diagrams.

#### 12.4.1 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. |

### 12.4.2 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 simple Boolean. TRUE or FALSE. |
| STRING | Indicates that the value is a string of characters. The termination of the string depends on the implemented wire protocol and platform and will be detailed in subsequent documents. |
| 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. |

### 12.4.3 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. |

### 12.4.4 Map

The Map is a general data structure used to store key/value pairs. Each key can only occur once within the Map.

**Attributes:**

| Name | Data Type | Description |
|---|---|---|
| Keys | String[] | The list of keys stored within the Map. |

**Operations:**

| Return Data Type | Operation / Description |
|---|---|
| DATA_TYPE | **getType(key : String)**<br>Returns the value from the DATA_TYPE enumeration that indicates the data type of the value for the specified *key*.<br><br>**Exceptions**<br>Key Not Found |
| Serializable | **get(key : String)**<br>Returns the value of the specified *key*.<br><br>**Exceptions**<br>Key Not Found |
| Serializable | **put(key : String, dataType : DATA_TYPE, value : Serializable)**<br>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.<br><br>**Exceptions**<br>Duplicate Key |

### 12.4.5 DateTime

The DateTime data type represents a Gregorian date and time.

**Attributes:**

| Name | Data Type | Description |
|---|---|---|
| Date | Date | Indicates the date value of the DateTime. The exact format of the date data type is defined by the wire level protocol and platform chosen for the implementation and will be covered in subsequent documents. |
| Hour | uint5 | Indicates the hour value in the format 0 to 23. |
| Minute | uint6 | Indicates the minute value in the format 0 to 59. |
| Second | uint6 | Indicates the second value in the format 0 to 59. |
| Microsecond | uint20 | Indicates the microsecond within the second in the format 0 to 999999. |
| MicrosFromEpoch | uint64 | Indicates the microseconds since the Epoch. |

### 12.4.6 FramedTime

The FramedTime data type is an extension of the DateTime type that adds support for framed time. Framed time is the division of seconds into increments that are not standard time units (Microseconds, Milliseconds, etc…). Each of these slices of time is called a frame.

**Attributes:**

| Name | Data Type | Description |
|---|---|---|
| Frame | uint50 | The frame within the current second. |
| FrameRate | uint50 | The frame rate used to calculate the frames temporal vector. |
| Scale | uint50 | The scale used to calculate the frames temporal vector. That frames temporal vector is expressed as FrameRate divided by Scale (FrameRate/Scale). |
| TotalFrames | uint64 | The total number of frames represented by this FramedTime. |

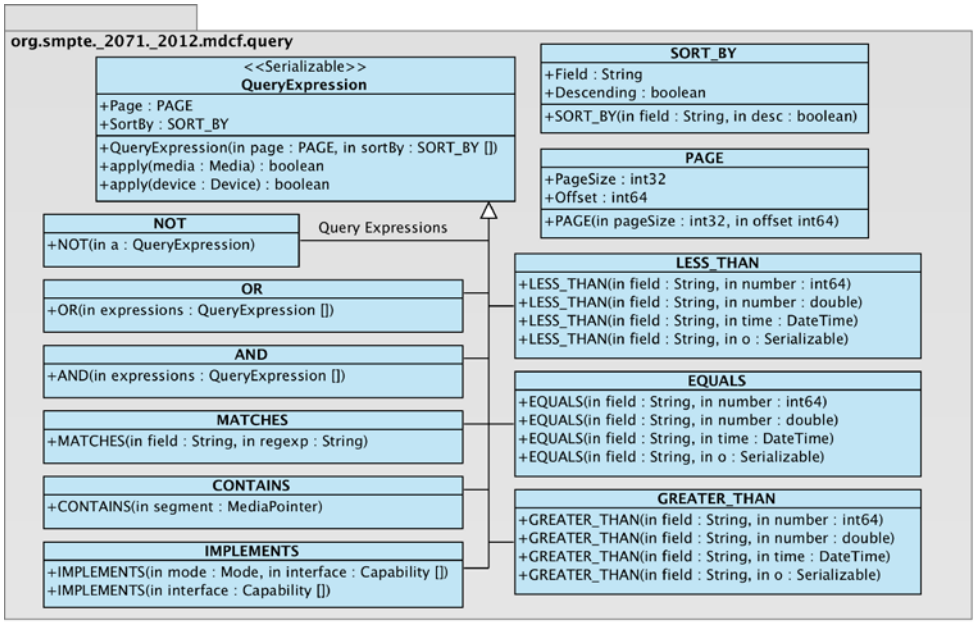
### 12.4.7 Image

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. |

## 12.5 Querying Expression Syntax Object Notation




**UML Diagram 12-5 – Query syntax Object Notation UML**

The UML Diagram 12-5 – Query syntax Object Notation ***UML*** defines the object notation for the query expression syntax.

### 12.5.1 QueryExpression

The QueryExpression is the base data type for all of the Query Expression objects that denote the object based query language. The query expression is applied to each of the resources to determine if the resource matches the expression. A resource is returned if it matches the query expression.

**Constructors:**

| Constructor |
|---|
| **QueryExpression(page : PAGE, sortBy : SORT_BY[])** |

**Attributes:**

| Name | Data Type | Description |
|---|---|---|
| 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(media : Media)**<br>Returns TRUE if the provided *media* matches the query expression. |
| boolean | **apply(device : Device)**<br>Returns TRUE if the provided *device* matches the query expression. |

### 12.5.2 AND

The Boolean AND operator is used to combine two or more query expressions into one boolean expression. All of the provided query expression must be TRUE in order for the AND expression to equate to TRUE.

**Constructors:**

| Constructor |
|---|
| **AND(expressions : QueryExpression[])** |

### 12.5.3 OR

The Boolean OR operator is used to combine two or more query expressions into one boolean expression. If any of the provided query expression are TRUE the OR expression will equate to TRUE.

**Constructors:**

| Constructor |
|---|
| **OR(expressions : QueryExpression[])** |

### 12.5.4 NOT

The Boolean NOT unary operator is used to invert provided query expression, returning TRUE if the provided expression equates to FALSE.

**Constructors:**

| Constructor |
|---|
| **NOT(expr : QueryExpression)** |

### 12.5.5 LESS_THAN

The LESS_THAN operator is used to determine if the value contained within a data field is less than the provided value. 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)** |

### 12.5.6 GREATER_THAN

The GREATER_THAN operator is used to determine if the value contained within a data field is greater than the provided value. 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)** |

### 12.5.7 EQUALS

The EQUALS operator is used to determine if the value in the specified data field is equal to the provided value. 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)** |

### 12.5.8 MATCHES

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

**Constructors:**

| Constructor |
|---|
| **MATCHES(field : String, regexp : String)** |

### 12.5.9 CONTAINS

The CONTAINS operator is used to determine if the Media contains the specified media element, expressed as a MediaPointer. 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 must specify a valid subset of one or more media elements that exist for the Media. CONTAINS will be true if the provided MediaPointer and the Media contain an intersecting set.

**Constructors:**

| Constructor |
|---|
| **CONTAINS(pointer : MediaPointer)** |

### 12.5.10 IMPLEMENTS

The IMPLEMENTS operator tests if a device implements a list of capability interfaces and/or supports a particular mode that implements the provided capability interfaces. In order for IMPLEMENTS to equate to TRUE the evaluated device must list all of the provided list the specified mode and support of the specified capability interfaces for that mode, list all of the specified base capability interfaces and contain all of the capability interfaces attributes.

**Constructors:**

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

### 12.5.11 PAGE

The PAGE operator specifies 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. |

### 12.5.12 SORT_BY

The SORT_BY operator is 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.

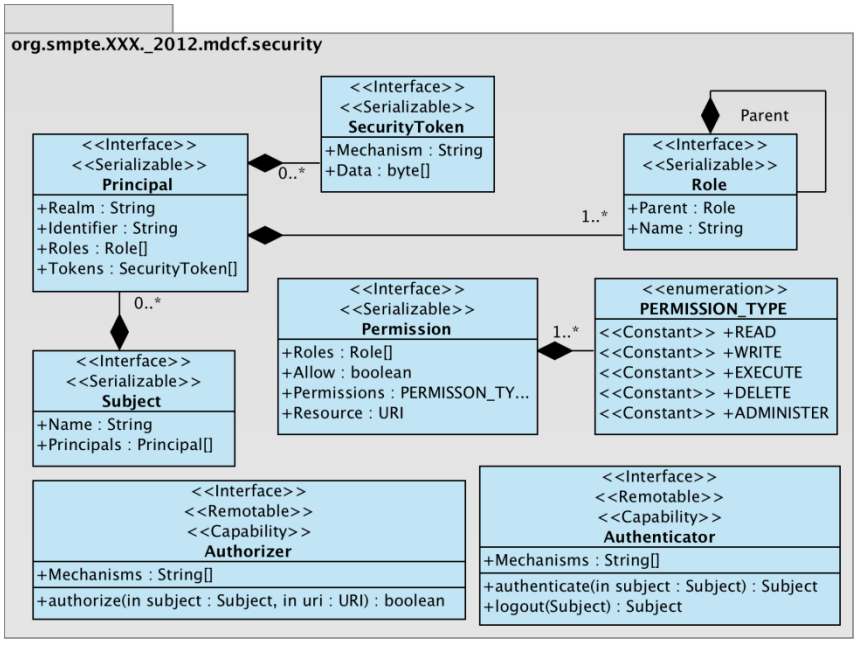
**Constructors:**

| 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. |

## 12.6 Security Framework




**UML Diagram 12-6 – Security Framework UML**

The UML Diagram 12-6 – Security Framework ***UML*** defines the roles based security model used by the Media Device Control Framework.

### 12.6.1 Error Conditions

The Media Device Control Framework 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 |
|---|---|
| Authentication Required | Indicates that authentication is required to access the resource. |
| Authentication Failed | Indicates that the authentication attempt failed. |
| Authorization Required | Indicates that authorization is required to access the resource. |
| Authorization Failed | Indicates that the authorization attempt failed. |
| Authentication Error | Indicates that some error occurred during authentication. |
| Authorization Error | Indicates that some error occurred during authorization. |
| Authentication Aborted | Indicates that the authentication sequence was aborted. |
| Authorization Aborted | Indicates that the authorization sequence was aborted. |
| Security Layer Expired | Indicates that the Security Layer or the encryption key has expired. |

### 12.6.2 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. |

### 12.6.3 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. |

### 12.6.4 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. |

### 12.6.5 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 must 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. |

### 12.6.6 Role

A Role is a security group used to assign Permissions. Permissions are assigned to one or more Roles and a client must be a part of 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 will extend the parents permissions, overriding or extending the permissions. |
| Name | String | The unique human readable name assigned to the Role. |

### 12.6.7 PERMISSION_TYPE

The PERMISSION_TYPE enumeration provides the valid permissions for the Media Device Control Framework.

**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. |

### 12.6.8 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: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 |
|---|---|
| boolean | **authorize(subject : Subject, uri : URI)**<br>Authorizes a session for the resource specified by the URI. Returns a new Subject if the *subject* has the necessary permissions to access the resource and FALSE if the subject does not. Authorize must be called after the client has authenticated with an Authenticator, but before any commands are executed. The *subject* provided must 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.<br><br>**Exceptions**<br>Authorization Failed, Authorization Error, Authorization Aborted, Invalid Subject |

### 12.6.9 Authenticator

The Authenticator is a capability interface that defines the operations required to authenticate with a device implementing the Media Device Control Framework. The Authenticator is used for the initial login of a client and will provide the client with a digital certificate that the client can later use to authorize itself with a device.

**UCN:** urn:smpte:ucn: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)**<br>Authenticates the *subject* with the Media Device Control Framework or Device. The Authenticator will most likely 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 Kerbrose authorization certificate.<br><br>**Exceptions**<br>Authentication Failed, Authentication Error, Authentication Aborted, Invalid Subject |
| Subject | **logout(subject : Subject)**<br>Logs the subject out of the system, invalidating any system resources, sessions or certificates reserved for the Subject.<br><br>**Exceptions**<br>Not Authenticated, Invalid Subject |

# Annex A Bibliography (Informative)

[POSIX] Institute of Electrical and Electronics Engineers (IEEE) (2008, December) Std 1003.1-2008 — Standard for Information Technology — Portable Operating System Interface (POSIX) Base Specification, Issue 7

[RFC 1737] Internet Engineering Task Force (IETF) (1994, December). RFC 1737 — Functional Requirements for Uniform Resource Names. http://www.ietf.org/rfc/rfc1737.txt

[RFC 2276] Internet Engineering Task Force (IETF) (1998, January). RFC 2276 — Architecture Principles of Uniform Resource Name Resolution. http://www.ietf.org/rfc/rfc2276.txt

[UPnP] UPnP™ Forum (2008, December). Universal Plug and Play — UPnP™ Device Architecture 1.1.

# Annex B Glossary (Normative)

## A

**ABNF**
The Augmented Backus Normal Form is a metalanguage based on the Backus-Naur Form used to define a formal system of language to be used as a communications protocol.

**API**
Application Programming Interface. The specification of how a software developer writing an application accesses the attributes, operations, state and behavior of a software program or library.

**Atomic**
The smallest unit of something, For example, the smallest unit of control.

**Attribute**
A read-only characteristic that depicts some portion of or the complete state of an interface.

## B

**Backus-Naur Form**
One of two main notation techniques used to describe the syntax of languages used in computing.

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

## C

**Capability**
See the definition for Capability Interface.

**Capability Interface**
The set of atomic level functions of a device. Examples include: Recordable, Panable, Switchable, Pausable, 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. Clip is used to refer to the media instances as well as media asset.

## D

**Device**
A controllable hardware or software resource used in the creation, storage or manipulation of media. Examples include: Video Server, VTR, Audio/Video Routers, Cameras, Graphics Devices, and Transcoders.

**Device Action**
Specific operations performed by media devices. Examples include: Play, Stop, Pan Left, Take.

**Device Attribute**
The characteristics of a particular media device. Examples include: number of audio channels, HD video type, and number of video inputs. Also see the definition for Attribute.

**Device Category**
The general function of a media device. Examples include: Record, Store, Playback, Generate, and Connect.

**Device Directory**
A repository of device information regarding device attributes and capabilities.

**Device Status**
Indication of the current state of a media device. Examples include: Stopped, Paused, Panning, Pan Complete, New Source Displayed, Transform initiated, Transform complete.

**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 has a real existence; a thing.

**Event**
See the definition for Signal

## I

**IDL**
See the definition for Interface Definition Language.

**Interface**
In Object Oriented Programming an interface is an abstract data type that describes the attributes and operations of a class or software component. Interfaces define the behavior of a class or software component. A class having all of the attributes and operations defined by an interface is said to “implement” that interface.

**Interface Definition Language**
A specification language that defines the interface exposed/implemented by a software service, software component or a class.

## M

**MDCF**
The 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 Asset**
The logical view of a unique piece of media whether a contiguous clip or any collection of media elements.

**Media Bundle**
Collections of Media Instances that relate with one another to produce a single audio-visual clip when played out or streamed.

**Media Container**
A logical or physical entity that holds media; equivalent to a POSIX file system directory, a list or a logical grouping.

**Media Directory**
A repository of descriptions of media that is available on storage capable devices.

**Media File**
A POSIX file system file that contains media.

**Media Instance**
A Media File that is also a particular occurrence of a media asset.

**Media Pointer**
A pointer that refers to a piece of or segment of Media.

**Media Segment**
A Media Pointer that contains additional temporal information pertaining to the in and out points.

**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**
Portable Operating System Interface. A family of standards by the IEEE for maintaining compatibility between operating systems.

## S

**Segment**
A part of or subset of a whole entity.

**Serializable**
Serializable interfaces are simple data structures used to transmit data between computer processes or over the network. The state information is marshaled for inter-process transmission and reconstituted by the receiving process to create an object that contains the original state.

**Server**
A superset of ports and storage that is capable of acquiring, storing and distributing media.

**Signal**
An asynchronous notification sent by a device to one or more clients notifying them that the state of the device has changed.

**Software Object**
See the definition for Object.

**Storage**
A device that holds media until it is needed later. For example, solid-state memory or spinning disk drives.

## U

**UCN**
A UCN or Uniform Capability Name is an identifier that uniquely identifies a capability interface.

**UDN**
A UDN or Uniform Device Name is an identifier that uniquely identifies a physical device or piece of hardware.

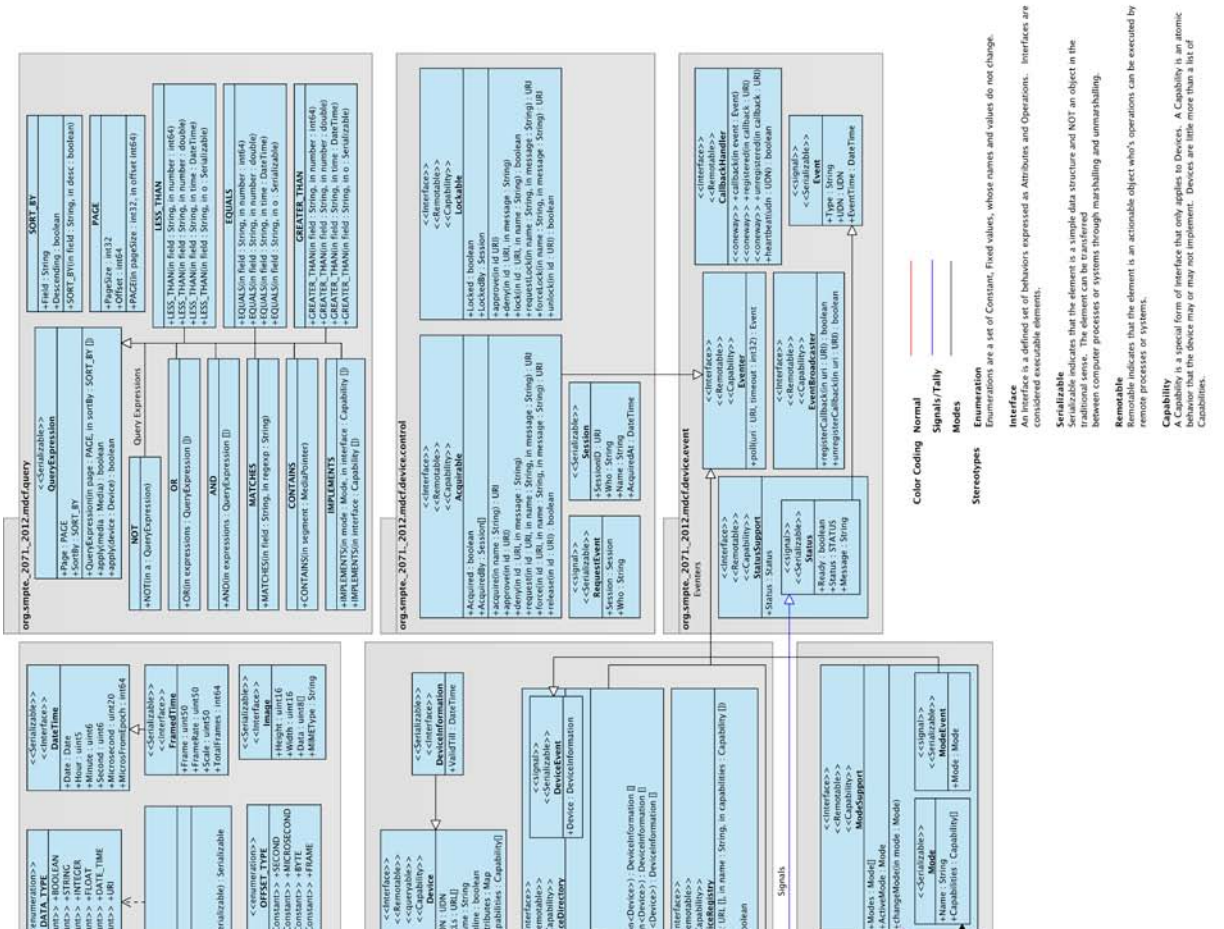
**UMN**
A UMN or Uniform Media Name is an identifier that uniquely identifies a piece of media.

**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 Media Device Control Framework Core UML (Normative)



# Annex D Media Device Control Framework Core IDL (Informative)

The Media Device Control Framework 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 the Corba 2.3 Interface Definition Language (IDL) application of the Media Device Control Framework. 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 Media Device Control Framework. 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 shall 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* shall 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* shall be defined as IDL operations with public scope.

   a. Class Operations parameters shall 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 shall be implemented as IDL *factory* operations with the name [“init”].

9. Class attributes, operations or parameters that are defined as an array shall 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 _2071 { module _2012 { module mdcf {
module types
{
    enum DATA_TYPE
    {
        _BOOLEAN,
        _STRING,
        _INTEGER,
        _FLOAT,
        _DATE_TIME,
        _URI
    };

    enum STATUS
    {
        _OK,
        _WARNING,
        _ERROR
    };

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

    valuetype Map
    {
        readonly attribute Strings Keys;

        DATA_TYPE getType(in wstring key);

        any get(in wstring key);

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

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

    valuetype Date
    {
        readonly attribute unsigned short Day;
        readonly attribute unsigned short Month;
        readonly attribute unsigned short Year;
    };

    valuetype DateTime
    {
        readonly attribute Date Date;
        readonly attribute unsigned short Hour;
        readonly attribute unsigned short Minute;
        readonly attribute unsigned short Second;
        readonly attribute unsigned long Microsecond;
        readonly attribute long MicrosFromEpoch;
    };

    valuetype FramedTime supports DateTime
    {
```

```
        readonly attribute unsigned long long Frame;
        readonly attribute unsigned long long FrameRate;
        readonly attribute unsigned long long Scale;
        readonly attribute long long TotalFrames;
    };
};


module identity
{
    valuetype URI
    {
        readonly attribute wstring Scheme;
        readonly attribute wstring Namespace;

        factory init(in wstring text);
    };

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

        factory init(in wstring text);
    };

    valuetype URN supports URI
    {
        factory init(in wstring text);
    };

    valuetype UCN supports URN
    {
        readonly attribute wstring VendorName;
        readonly attribute wstring InterfaceName;
        readonly attribute unsigned long Version;
        readonly attribute UnsignedLongs Revision;

        factory init(in wstring text);
    };

    valuetype RN supports URN
    {
        readonly attribute wstring Scope;
        readonly attribute org::smpte::_2071::_2012::mdcf::types::Map Attributes;
    };

    valuetype UDN supports RN
    {
        factory init(in wstring text);
    };

    valuetype UMN supports RN
    {
        readonly attribute wstring Type;
        readonly attribute wstring MID;

        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<UMN> UMNs;
};


module security
```

```
{
    valuetype SecurityToken
    {
        readonly attribute wstring Mechanism;
        readonly attribute RawData Data;
    };

    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::_2071::_2012::mdcf::identity::URI Resource;
    };

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

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

        org::smpte::_2071::_2012::mdcf::identity::URI Resource;
        Subject Subject;
```

```
    };

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

        readonly attribute Strings Mechanisms;

        boolean authorize(in Subject subject, in org::smpte::_2071::_2012::mdcf::identity::URI uri)
        raises(SecurityException);
    };

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

        readonly attribute Strings Mechanisms;

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

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


module query
{
    valuetype QueryExpression;

    valuetype Queryable
    {
    };

    exception InvalidQuery
    {
        QueryExpression filter;
        wstring Message;
    };
};


module device
{
    module event
    {
        valuetype Status
        {
            readonly attribute boolean Ready;
            readonly attribute org::smpte::_2071::_2012::mdcf::types::STATUS Status;
            readonly attribute wstring Message;
        };

        interface StatusSupport
        {
            // UCN = "urn:smpte:ucn:status_support_v1.0";

            readonly attribute Status Status;
            // raises (org::smpte::_2071::_2012::mdcf::security::SecurityException);
        };

        valuetype Event supports Status
        {
            readonly attribute wstring Type;
            readonly attribute org::smpte::_2071::_2012::mdcf::identity::UDN UDN;
            readonly attribute org::smpte::_2071::_2012::mdcf::types::DateTime EventTime;
        };

        interface CallbackHandler
        {
```

```
            oneway void callback(in Event event);

            oneway void registered(in org::smpte::_2071::_2012::mdcf::identity::URI callback);

            oneway void unregistered(in org::smpte::_2071::_2012::mdcf::identity::URI callback);

            boolean heartbeat(in org::smpte::_2071::_2012::mdcf::identity::UDN eventBroadcaster);
        };

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

            Event poll(in org::smpte::_2071::_2012::mdcf::identity::URI uri,
                       in unsigned long timeout)
            raises(org::smpte::_2071::_2012::mdcf::security::SecurityException);
        };

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

            boolean registerCallback(in org::smpte::_2071::_2012::mdcf::identity::URI uri)
            raises(org::smpte::_2071::_2012::mdcf::security::SecurityException);

            boolean unregisterCallback(in org::smpte::_2071::_2012::mdcf::identity::URI uri)
            raises(org::smpte::_2071::_2012::mdcf::security::SecurityException);
        };
    };

    valuetype Capability
    {
        readonly attribute org::smpte::_2071::_2012::mdcf::types::Map Attributes;
        readonly attribute org::smpte::_2071::_2012::mdcf::identity::UCN UCN;
        readonly attribute org::smpte::_2071::_2012::mdcf::identity::URLs URLs;
    };

    typedef sequence<Capability> Capabilities;

    interface Device : org::smpte::_2071::_2012::mdcf::query::Queryable
    {
        // UCN = "urn:smpte:ucn:device_v1.0";

        readonly attribute org::smpte::_2071::_2012::mdcf::identity::UDN UDN;
        // raises (org::smpte::_2071::_2012::mdcf::security::SecurityException);
        readonly attribute org::smpte::_2071::_2012::mdcf::identity::URLs URLs;
        // raises (org::smpte::_2071::_2012::mdcf::security::SecurityException);
        readonly attribute wstring Name;
        // raises (org::smpte::_2071::_2012::mdcf::security::SecurityException);
        readonly attribute boolean Online;
        // raises (org::smpte::_2071::_2012::mdcf::security::SecurityException);
        readonly attribute org::smpte::_2071::_2012::mdcf::types::Map Attributes;
        // raises (org::smpte::_2071::_2012::mdcf::security::SecurityException);
        readonly attribute Capabilities Capabilities;
        // raises (org::smpte::_2071::_2012::mdcf::security::SecurityException);
};

    valuetype DeviceInformation supports Device
    {
        readonly attribute org::smpte::_2071::_2012::mdcf::types::DateTime ValidTill;
    };

    typedef sequence<DeviceInformation> DeviceInformations;

    exception DeviceNotFound
    {
        org::smpte::_2071::_2012::mdcf::identity::UDN UDN;
        wstring Message;
    };

    exception DeviceNotBound
```

```
{
    org::smpte::_2071::_2012::mdcf::identity::UDN UDN;
    org::smpte::_2071::_2012::mdcf::identity::UDN parentUDN;
    org::smpte::_2071::_2012::mdcf::identity::URLs urls;
    wstring name;
    Capabilities capabilities;
    wstring Message;
};

exception DeviceException
{
    org::smpte::_2071::_2012::mdcf::identity::UDN UDN;
    wstring Message;
};

exception DeviceNotUnbound
{
    org::smpte::_2071::_2012::mdcf::identity::UDN UDN;
    org::smpte::_2071::_2012::mdcf::identity::UDN parentUDN;
    org::smpte::_2071::_2012::mdcf::identity::URLs urls;
    wstring name;
    Capabilities capabilities;
    wstring Message;
};

exception DeviceAlreadyBound
{
    org::smpte::_2071::_2012::mdcf::identity::UDN UDN;
    wstring Message;
};

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

    readonly attribute DeviceInformation ParentDeviceDirectory;
    // raises (org::smpte::_2071::_2012::mdcf::security::SecurityException);
    readonly attribute org::smpte::_2071::_2012::mdcf::identity::UDNs Namespaces;
    // raises (org::smpte::_2071::_2012::mdcf::security::SecurityException);

    DeviceInformations ancestors(in org::smpte::_2071::_2012::mdcf::identity::UDN udn)
    raises (DeviceNotFound, org::smpte::_2071::_2012::mdcf::security::SecurityException);

    DeviceInformation lookup(in org::smpte::_2071::_2012::mdcf::identity::UDN udn)
    raises (DeviceNotFound, org::smpte::_2071::_2012::mdcf::security::SecurityException);

    DeviceInformation parent(in org::smpte::_2071::_2012::mdcf::identity::UDN udn)
    raises (DeviceNotFound, org::smpte::_2071::_2012::mdcf::security::SecurityException);

    DeviceInformations children(in org::smpte::_2071::_2012::mdcf::identity::UDN udn,
                                in org::smpte::_2071::_2012::mdcf::query::QueryExpression filter)
    raises (DeviceNotFound, org::smpte::_2071::_2012::mdcf::query::InvalidQuery,
            org::smpte::_2071::_2012::mdcf::security::SecurityException);

    DeviceInformations siblings(in org::smpte::_2071::_2012::mdcf::identity::UDN udn,
                                in org::smpte::_2071::_2012::mdcf::query::QueryExpression filter)
    raises (DeviceNotFound, org::smpte::_2071::_2012::mdcf::query::InvalidQuery,
            org::smpte::_2071::_2012::mdcf::security::SecurityException);

    DeviceInformations search(in org::smpte::_2071::_2012::mdcf::identity::UDN udn,
                              in org::smpte::_2071::_2012::mdcf::query::QueryExpression filter)
    raises (DeviceNotFound, org::smpte::_2071::_2012::mdcf::query::InvalidQuery,
            org::smpte::_2071::_2012::mdcf::security::SecurityException);
};

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

    void bind(in org::smpte::_2071::_2012::mdcf::identity::UDN udn,
              in org::smpte::_2071::_2012::mdcf::identity::UDN parentUDN,
```

```
                in org::smpte::_2071::_2012::mdcf::identity::URLs urls,
                in wstring name, in Capabilities capabilities)
        raises (DeviceNotFound, DeviceNotBound, DeviceAlreadyBound,
                org::smpte::_2071::_2012::mdcf::security::SecurityException);

        void unbind(in org::smpte::_2071::_2012::mdcf::identity::UDN udn)
        raises (DeviceNotFound, DeviceNotUnbound,
                org::smpte::_2071::_2012::mdcf::security::SecurityException);

        boolean setOnline(in org::smpte::_2071::_2012::mdcf::identity::UDN udn, in boolean online)
        raises (DeviceNotFound, org::smpte::_2071::_2012::mdcf::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::_2071::_2012::mdcf::device::event::Event
    {
        readonly attribute DeviceInformation Device;
    };


    module mode
    {
        valuetype Mode
        {
            readonly attribute wstring Name;
            readonly attribute Capabilities Capabilities;
        };

        typedef sequence<Mode> Modes;

        exception InvalidMode
        {
            Mode Mode;
            wstring Message;
        };

        exception ModeException
        {
            Mode Mode;
            wstring Message;
        };

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

            readonly attribute Modes Modes;
            // raises (org::smpte::_2071::_2012::mdcf::security::SecurityException);
            readonly attribute Mode ActiveMode;
            // raises (org::smpte::_2071::_2012::mdcf::security::SecurityException);

            void changeMode(in Mode mode)
            raises (InvalidMode, ModeException,
                    org::smpte::_2071::_2012::mdcf::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::_2071::_2012::mdcf::device::DeviceEvent
    {
        readonly attribute Mode Mode;
    };
};


module control
{
    valuetype Session
    {
        readonly attribute org::smpte::_2071::_2012::mdcf::identity::URI SessionID;
        readonly attribute wstring Who;
        readonly attribute wstring Name;
        readonly attribute org::smpte::_2071::_2012::mdcf::types::DateTime AcquiredAt;
    };

    typedef sequence<Session> Sessions;

    exception DeviceNotAcquired
    {
        org::smpte::_2071::_2012::mdcf::identity::UDN UDN;
        wstring Message;
    };

    exception DeviceNotLocked
    {
        org::smpte::_2071::_2012::mdcf::identity::UDN UDN;
        wstring Message;
    };

    exception DeviceLocked
    {
        org::smpte::_2071::_2012::mdcf::identity::UDN UDN;
        wstring Message;
    };

    exception TooManySessions
    {
        org::smpte::_2071::_2012::mdcf::identity::UDN UDN;
        wstring Message;
    };

    exception NameInUse
    {
        org::smpte::_2071::_2012::mdcf::identity::UDN UDN;
        wstring Message;
    };

    exception SessionNotFound
    {
        org::smpte::_2071::_2012::mdcf::identity::UDN UDN;
        org::smpte::_2071::_2012::mdcf::identity::URI SessionID;
        wstring Message;
    };

    exception RequestNotFound
    {
        org::smpte::_2071::_2012::mdcf::identity::UDN UDN;
        org::smpte::_2071::_2012::mdcf::identity::URI SessionID;
        wstring Message;
    };

    exception RequestExpired
    {
        org::smpte::_2071::_2012::mdcf::identity::UDN UDN;
        org::smpte::_2071::_2012::mdcf::identity::URI SessionID;
        wstring Message;
```

```
};

exception RequestDenied
{
    org::smpte::_2071::_2012::mdcf::identity::UDN UDN;
    org::smpte::_2071::_2012::mdcf::identity::URI SessionID;
    wstring Message;
};

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

    readonly attribute boolean Acquired;
    // raises (org::smpte::_2071::_2012::mdcf::security::SecurityException);
    readonly attribute Sessions AcquiredBy;
    // raises (org::smpte::_2071::_2012::mdcf::security::SecurityException);

    org::smpte::_2071::_2012::mdcf::identity::URI acquire(in wstring name)
    raises(DeviceNotAcquired, DeviceLocked, TooManySessions, NameInUse,
           org::smpte::_2071::_2012::mdcf::security::SecurityException);

    void approve(in org::smpte::_2071::_2012::mdcf::identity::URI id)
    raises(RequestNotFound, RequestExpired,
           org::smpte::_2071::_2012::mdcf::security::SecurityException);

    void deny(in org::smpte::_2071::_2012::mdcf::identity::URI id, in wstring message)
    raises(RequestNotFound, RequestExpired,
           org::smpte::_2071::_2012::mdcf::security::SecurityException);

    org::smpte::_2071::_2012::mdcf::identity::URI request(
                                  in org::smpte::_2071::_2012::mdcf::identity::URI id,
                                  in wstring name, in wstring message)
    raises(SessionNotFound, RequestDenied, DeviceNotAcquired,
           org::smpte::_2071::_2012::mdcf::security::SecurityException);

    org::smpte::_2071::_2012::mdcf::identity::URI force(
                                  in org::smpte::_2071::_2012::mdcf::identity::URI id,
                                  in wstring name, in wstring message)
    raises(SessionNotFound, DeviceNotAcquired,
           org::smpte::_2071::_2012::mdcf::security::SecurityException);

    boolean release(in org::smpte::_2071::_2012::mdcf::identity::URI id)
    raises (SessionNotFound, org::smpte::_2071::_2012::mdcf::security::SecurityException);
};

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

    readonly attribute boolean Locked;
    // raises (org::smpte::_2071::_2012::mdcf::security::SecurityException);
    readonly attribute Session LockedBy;
    // raises (org::smpte::_2071::_2012::mdcf::security::SecurityException);

    void approve(in org::smpte::_2071::_2012::mdcf::identity::URI id)
    raises(RequestNotFound, RequestExpired,
           org::smpte::_2071::_2012::mdcf::security::SecurityException);

    void deny(in org::smpte::_2071::_2012::mdcf::identity::URI id, in wstring message)
    raises(RequestNotFound, RequestExpired,
           org::smpte::_2071::_2012::mdcf::security::SecurityException);

    boolean lock(in org::smpte::_2071::_2012::mdcf::identity::URI id, in string name)
    raises(SessionNotFound, DeviceNotAcquired, DeviceNotLocked,
           org::smpte::_2071::_2012::mdcf::security::SecurityException);

    org::smpte::_2071::_2012::mdcf::identity::URI requestLock(in wstring name,
                                                              in wstring message)
    raises(RequestDenied, DeviceNotLocked,
           org::smpte::_2071::_2012::mdcf::security::SecurityException);
```

```
            org::smpte::_2071::_2012::mdcf::identity::URI forceLock(in wstring name,
                                                                    in wstring message)
            raises(DeviceNotLocked,
                   org::smpte::_2071::_2012::mdcf::security::SecurityException);

            boolean unlock(in org::smpte::_2071::_2012::mdcf::identity::URI id)
            raises (SessionNotFound, org::smpte::_2071::_2012::mdcf::security::SecurityException);
        };

        /*
         * Mode Events are sent by device that support multiple modes of operation.
         *
         * Valid Device 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 org::smpte::_2071::_2012::mdcf::device::DeviceEvent
        {
            readonly attribute Session Session;
            readonly attribute wstring Who;
        };
    };
};


module media
{
    valuetype MediaPointer
    {
        readonly attribute org::smpte::_2071::_2012::mdcf::identity::UMN Source;
        readonly attribute unsigned long long InpointOffset;
        readonly attribute unsigned long long OutpointOffset;
        readonly attribute org::smpte::_2071::_2012::mdcf::types::OFFSET_TYPE OffsetType;
    };

    valuetype MediaSegment supports MediaPointer
    {
        readonly attribute org::smpte::_2071::_2012::mdcf::types::DateTime Inpoint;
        readonly attribute org::smpte::_2071::_2012::mdcf::types::DateTime Outpoint;
    };

    valuetype MediaFormatPointer supports MediaPointer
    {
        readonly attribute org::smpte::_2071::_2012::mdcf::identity::URI Format;
        readonly attribute short Track;
        readonly attribute wstring TrackName;
    };

    valuetype MediaFormatSegment supports MediaFormatPointer, MediaSegment
    {
    };

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

    valuetype Media supports org::smpte::_2071::_2012::mdcf::query::Queryable
    {
        readonly attribute org::smpte::_2071::_2012::mdcf::identity::UMN UMN;
        readonly attribute wstring Name;
        readonly attribute org::smpte::_2071::_2012::mdcf::identity::UMN Location;
        readonly attribute org::smpte::_2071::_2012::mdcf::types::DateTime Created;
```

```
    readonly attribute org::smpte::_2071::_2012::mdcf::types::DateTime Modified;
    readonly attribute org::smpte::_2071::_2012::mdcf::types::Map Metadata;
};

valuetype MediaContainer supports Media
{
    readonly attribute org::smpte::_2071::_2012::mdcf::identity::UDN UDN;
};

valuetype MediaAsset supports Media
{
    readonly attribute org::smpte::_2071::_2012::mdcf::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::_2071::_2012::mdcf::types::FramedTime Duration;
    // readonly attribute MediaFormatSegments Composition;
};

valuetype MediaBundle supports MediaContainer, MediaInstance
{
};

exception MediaNotFound
{
    org::smpte::_2071::_2012::mdcf::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:media_directory_v1.0";

    readonly attribute MediaContainer MediaContainer;
    // raises (org::smpte::_2071::_2012::mdcf::security::SecurityException);

    Media create(in Media media, in MediaPointers pointers)
    raises(MediaNotFound, MediaCreationFailed,
           org::smpte::_2071::_2012::mdcf::security::SecurityException);

    Media delete(in org::smpte::_2071::_2012::mdcf::identity::UMN umn)
    raises(MediaNotFound, MediaDeletionFailed,
           org::smpte::_2071::_2012::mdcf::security::SecurityException);
```

```
        MediaList list(in org::smpte::_2071::_2012::mdcf::identity::UMN container,
                       in org::smpte::_2071::_2012::mdcf::query::QueryExpression filter)
        raises(MediaNotFound, org::smpte::_2071::_2012::mdcf::query::InvalidQuery,
               org::smpte::_2071::_2012::mdcf::security::SecurityException);

        Media lookup(in org::smpte::_2071::_2012::mdcf::identity::UMN umn)
        raises(MediaNotFound, org::smpte::_2071::_2012::mdcf::security::SecurityException);

        MediaAsset lookupAsset(in wstring mid)
        raises(MediaNotFound, org::smpte::_2071::_2012::mdcf::security::SecurityException);

        MediaList search(in org::smpte::_2071::_2012::mdcf::identity::UMN container,
                         in org::smpte::_2071::_2012::mdcf::query::QueryExpression filter)
        raises(MediaNotFound, org::smpte::_2071::_2012::mdcf::query::InvalidQuery,
               org::smpte::_2071::_2012::mdcf::security::SecurityException);

        Media update(in Media media)
        raises(MediaNotFound, MediaUpdateFailed,
               org::smpte::_2071::_2012::mdcf::security::SecurityException);
    };

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

        org::smpte::_2071::_2012::mdcf::identity::URLs lookupURLs(
                                              in org::smpte::_2071::_2012::mdcf::identity::UMN umn)
        raises(MediaNotFound,
               org::smpte::_2071::_2012::mdcf::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::_2071::_2012::mdcf::device::event::Event
    {
        readonly attribute Media Media;
    };
};


module query
{
    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;

    valuetype QueryExpression
    {
        readonly attribute PAGE page;
        readonly attribute SORT_BYs sortBy;
```

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

    valuetype NOT supports QueryExpression
    {
        factory init(in QueryExpression expr);
    };

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

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

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

    valuetype CONTAINS supports QueryExpression
    {
        factory init(in org::smpte::_2071::_2012::mdcf::media::MediaPointer segment);
    };

    valuetype IMPLEMENTS supports QueryExpression
    {
        factory init(in org::smpte::_2071::_2012::mdcf::device::mode::Mode mode,
                     in org::smpte::_2071::_2012::mdcf::device::Capabilities interfaces);
        factory init(in org::smpte::_2071::_2012::mdcf::device::Capabilities interfaces);
    };

    valuetype LESS_THAN supports QueryExpression
    {
        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::_2071::_2012::mdcf::types::DateTime time);
        factory init(in wstring field, in any serializable);
    };

    valuetype GREATER_THAN supports QueryExpression
    {
        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::_2071::_2012::mdcf::types::DateTime time);
        factory init(in wstring field, in any serializable);
    };

    valuetype EQUALS supports QueryExpression
    {
        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::_2071::_2012::mdcf::types::DateTime time);
        factory init(in wstring field, in any serializable);
    };
};
};};};};};
```